

# Perfect Documentation Library



This library contains all the reference documentation and API reference-related material you need to run and use Perfect.

## Issues, Pull Requests, and Support

We have transitioned to using JIRA for dealing with all pull requests, bugs, and any other support-related issues. Therefore, the GitHub "issues" tab has been disabled.

If you find a bug, or have a helpful suggestion to improve this documentation, [please raise it](#).

Before you do, please view [a comprehensive list of all open issues](#).

## Contributing

We welcome all contributions to and recommendations for improving the Perfect documentation. We welcome contributions to Perfect's documentation. If you spot a typo, bug, or other errata, or have additions or suggestions to recommend, please [create a pull request or log an issue in JIRA](#).

We have [a system, written with Perfect](#), that picks up markdown files dynamically from this repository and incorporates them in an HTML docs site. This system will go live shortly.

## Table of Contents

---

- [Introduction](#)
- [Getting Started](#)
- [Getting Started From Scratch](#)
- [An HTTP and Web Services Primer](#)
- [Repository Layout](#)
- [Building with Swift Package Manager](#)
- [Configuring and Launching HTTPServer](#)
- [Handling Requests](#)
  - [Routing](#)
  - [HTTPRequest](#)
    - [Using Form Data](#)
    - [File Uploads](#)
  - [HTTPResponse](#)
  - [Request & Response Filters](#)
    - [Web Redirect Filters](#)
  - [Sessions](#)
    - [CSRF Security](#)
    - [CORS Security](#)
  - [Local Authentication modules](#)
  - [GSS-SPNEGO Security Feature](#)
  - [JSON](#)
  - [Static File Content](#)

- [Mustache](#)
- [Markdown](#)
- [HTTP Request Logging](#)
- [WebSockets](#)
- [Utilities](#)
  - [Bytes](#)
  - [File](#)
  - [Dir](#)
  - [Threading](#)
  - [Networking @kjessup](#)
  - [OAuth2 and Providers](#)
  - [UUID](#)
  - [SysProcess](#)
  - [Log](#)
  - [Log Files](#)
  - [Remote Logging](#)
  - [CURL](#)
  - [XML](#)
  - [INI](#)
  - [Zip](#)
  - [Crypto](#)
  - [SMTP](#)
  - [Google Analytics Measurement Protocol](#)
  - [Repeater](#)
- [Database Connectors](#)
  - [SQLite](#)
  - [MySQL](#)
  - [MariaDB](#)
  - [PostgreSQL](#)
  - [MongoDB](#)
    - [MongoDB Databases](#)
    - [MongoDB Collections](#)
    - [MongoDB Client](#)
    - [Working with BSON](#)
    - [GridFS](#)
  - [CouchDB](#)
  - [LDAP](#)
  - [Redis @kjessup](#)
  - [FileMaker](#)
- [iOS Notifications @kjessup](#)
- [Deployment](#)
  - [Ubuntu 15.10](#)
  - [Docker](#)
  - [Heroku](#)
  - [Azure](#)
  - [AWS](#)
  - [Linode](#)
  - [Digital Ocean](#)
- [Performance Profiling & Remote Monitoring](#)
  - [SysInfo](#)
  - [New Relic](#)
- [Message Queue & Cluster Control](#)
  - [Kafka](#)
  - [Mosquitto](#)

- [ZooKeeper \(Linux Only\)](#)
- Big Data / Artificial Intelligence & Machine Learning
  - [TensorFlow](#)
  - [Hadoop](#)
    - [HDFS](#)
    - [MapReduce Master](#)
    - [MapReduce History](#)
    - [YARN Node Manager](#)
    - [YARN Resource Manager](#)
- Platform specific Notes
  - [Ubuntu 16.04: Starting Services at System Boot](#)
- Language Extensions:
  - [Python](#)

## StORM, a Swift ORM

StORM is not distributed as a Perfect.org project; however, the Perfect libraries are integral to its operation, and some authors are common.

- [StORM, a Swift ORM](#)
  - [Introduction to StORM](#)
  - [Setting up a class](#)
  - [Saving, Retrieving and Deleting Rows](#)
  - [StORMCursor](#)
  - [Inserting rows](#)
  - [Updating rows](#)
  - Database Specific Implementations
    - [PostgresStORM](#)
    - [SQLiteStORM](#)
    - [MySQLStORM](#)
    - [Apache CouchDB](#)
    - [MongoDB](#)

## Perfect: Server-Side Swift

Swift 3.0 Platforms OS X | Linux License Apache docs 99% codebeat B Twitter @PerfectlySoft slack 82/2905

Perfect is a complete and powerful toolbox, framework, and application server for Linux, iOS, and macOS (OS X). It provides everything a Swift engineer needs for developing lightweight, maintainable, and scalable apps and other REST services entirely in the Swift programming language for both client-facing and server-side applications.

Built on a high-performance asynchronous networking engine, Perfect can also run on FastCGI, and it supports Secure Sockets Layer encryption. There are many other features including a suite of tools commonly required by internet servers such as WebSockets and iOS push notifications, but you are not limited to those options.

This guide is designed for developers at all levels of experience to get Perfect up and running quickly.

## Getting Started

Are you eager to get programming with Swift and Perfect? This guide will provide you with everything you need to know to run Perfect, and to create your first application.

After reading this guide, you will know:

- How to create and run an HTTP/HTTPS server and get Perfect up and running
- The prerequisite components you must install to run Perfect on either OS X or Ubuntu Linux
- How to build, test, and manage dependencies for Swift projects
- How to deploy Perfect in additional environments including Heroku, Amazon Web Services, Docker, Microsoft Azure, Google Cloud, IBM Bluemix CloudFoundry, and IBM Bluemix Docker

## Prerequisites

---

## Swift 3.0

After you have installed a Swift 3.0 toolchain from [Swift.org](https://swift.org), open up a terminal window and type `swift --version`

It will produce a message similar to this one:

```
Apple Swift version 3.0.1 (swiftlang-800.0.58.6 clang-800.0.42.1)
Target: x86_64-apple-macosx10.9
```

Make sure you are running the release version of Swift 3.0.1. Perfect will not compile successfully if you are running a version of Swift that is lower than 3.0.1.

You can find out which version of Swift you will need by looking in [the README of the main Perfect repo](#).

## OS X

Everything you need is already installed.

## Ubuntu Linux

Perfect runs in Ubuntu Linux 14.04, 15.10 and 16.04 environments. Perfect relies on OpenSSL, libssl-dev, and uuid-dev. To install these, in the terminal, type:

```
sudo apt-get install openssl libssl-dev uuid-dev
```

When building on Linux, OpenSSL 1.0.2+ is required for this package. On Ubuntu 14 or some Debian distributions you will need to update your OpenSSL before this package will build.

---

## Getting Started with Perfect

Now you're ready to build your first web application starter project.

### Build Starter Project

The following will clone and build an empty starter project. It will launch a local server that will run on port 8181 on your computer:

```
git clone https://github.com/PerfectlySoft/PerfectTemplate.git
cd PerfectTemplate
swift build
.build/debug/PerfectTemplate
```

You should see the following output:

```
Starting HTTP server on 0.0.0.0:8181 with document root ./webroot
```

The server is now running and waiting for connections. Access <http://localhost:8181/> to see the greeting. Hit "control-c" to terminate the server.

You can view the full source code for [PerfectTemplate](#).

## Xcode

Swift Package Manager (SPM) can generate an Xcode project which can run the PerfectTemplate server and provide full source code editing and debugging for your project. Enter the following in your terminal:

```
swift package generate-xcodeproj
```

Open the generated file "PerfectTemplate.xcodeproj". Ensure that you have selected the executable target and selected it to run on "My Mac". You can now run and debug the server directly in Xcode.

---

## Next Steps

These example snippets show how to accomplish several common tasks that one might need to do when developing a web or REST application. In all cases, the `request` and `response` variables refer, respectively, to the `HTTPRequest` and `HTTPResponse` objects which are given to your URL handlers.

Consult the [API reference](#) for more details.

### Get a Client Request Header

```
if let acceptEncoding = request.header(.acceptEncoding) {
    ...
}
```

## Client GET and POST Parameters

```
if let foo = request.param(name: "foo") {
    ...
}
if let foo = request.param(name: "foo", defaultValue: "default foo") {
    ...
}
let foos: [String] = request.params(named: "foo")
```

## Get the Current Request Path

```
let path = request.path
```

## Accessing the Server's Document Directory and Returning an Image File to the Client

```
let docRoot = request.documentRoot
do {
    let mrPebbles = File("\(docRoot)/mr_pebbles.jpg")
    let imageSize = mrPebbles.size
    let imageBytes = try mrPebbles.readSomeBytes(count: imageSize)
    response.setHeader(.contentType, value: MIMEType.forExtension("jpg"))
    response.setHeader(.contentLength, value: "\(imageBytes.count)")
    response.setBody(bytes: imageBytes)
} catch {
    response.status = .internalServerError
    response.setBody(string: "Error handling request: \(error)")
}
response.completed()
```

## Getting Client Cookies

```
for (cookieName, cookieValue) in request.cookies {
    ...
}
```

## Setting Client Cookies

```
let cookie = HTTPCookie(name: "cookie-name", value: "the value", domain: nil,
                        expires: .session, path: "/",
                        secure: false, httpOnly: false)
response.addCookie(cookie)
```

## Returning JSON Data to Client

```
response.setHeader(.contentType, value: "application/json")
let d: [String:Any] = ["a":1, "b":0.1, "c": true, "d":[2, 4, 5, 7, 8]]

do {
    try response.setBody(json: d)
} catch {
    //...
}
response.completed()
```

*This snippet uses built-in JSON encoding. Feel free to use the JSON encoder/decoder you prefer.*

## Redirecting the Client

```
response.status = .movedPermanently
response.setHeader(.location, value: "http://www.perfect.org/")
response.completed()
```

## Filtering and Handling 404 Errors in a Custom Manner

```
struct Filter404: HTTPResponseFilter {
    func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        callback(.continue)
    }

    func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        if case .notFound = response.status {
            response.bodyBytes.removeAll()
            response.setBody(string: "The file \(response.request.path) was not found.")
            response.setHeader(.contentType, value: "\(response.bodyBytes.count)")
            callback(.done)
        } else {
            callback(.continue)
        }
    }
}

try HTTPServer(documentRoot: webRoot)
    .setResponseFilters([(Filter404(), .high)])
    .start(port: 8181)
```

## Getting Started From Scratch

This guide will take you through the steps of settings up a simple HTTP server from scratch with Swift and Perfect.

## Prerequisites

### Swift 3.0

After you have installed a Swift 3.0 toolchain from [Swift.org](https://swift.org), open up a terminal window and type `swift --version`

It will produce a message similar to this one:

```
Apple Swift version 3.0.1 (swiftlang-800.0.58.6 clang-800.0.42.1)
Target: x86_64-apple-macosx10.9
```

Make sure you are running the release version of Swift 3.0.1. Perfect will not compile successfully if you are running a version of Swift that is lower than 3.0.1.

You can find out which version of Swift you will need by looking in [the README of the main Perfect repo](#).

### OS X

Everything you need is already installed.

### Ubuntu Linux

Perfect runs in Ubuntu Linux 14.04, 15.10 and 16.04 environments. Perfect relies on OpenSSL, libssl-dev, and uuid-dev. To install these, in the terminal, type:

```
sudo apt-get install openssl libssl-dev uuid-dev
```

## Getting Started with Perfect

Now that you're ready to build a simple web application from scratch, then go ahead and create a new folder where you will keep your project files:

```
mkdir MyAwesomeProject
cd MyAwesomeProject
```

As a good developer practice, make this folder a git repo:

```
git init
touch README.md
git add README.md
git commit -m "Initial commit"
```

It's also recommended to add a `.gitignore` similar to the contents of [this Swift .gitignore template from gitignore.io](https://github.com/PerfectlySoft/Perfect-HTTPServer/blob/master/.gitignore).

## Create the Swift Package

Now create a `Package.swift` file in the root of the repo with the following content. This is needed for the Swift Package Manager (SPM) to build the project.

```
import PackageDescription

let package = Package(
    name: "MyAwesomeProject",
    dependencies: [
        .Package(
            url: "https://github.com/PerfectlySoft/Perfect-HTTPServer.git",
            majorVersion: 2
        )
    ]
)
```

Next create a folder called `Sources` and create a `main.swift` in there:

```
mkdir Sources
echo 'print("Well hi there!")' >> Sources/main.swift
```

Now the project is ready to be built and run by running by the following two commands:

```
swift build
.build/debug/MyAwesomeProject
```

You should see the following output:

```
Well hi there!
```

## Setting up the server

Now that the Swift package is up and running, the next step is to implement the Perfect-HTTPServer. Open up the `Sources/main.swift` and change its content the following:

```

import PerfectLib
import PerfectHTTP
import PerfectHTTPServer

// Create HTTP server.
let server = HTTPServer()

// Register your own routes and handlers
var routes = Routes()
routes.add(method: .get, uri: "/", handler: {
    request, response in
    response.setHeader(.contentType, value: "text/html")
    response.appendBody(string: "<html><title>Hello, world!</title><body>Hello, world!</body></html>")
    response.completed()
})

// Add the routes to the server.
server.addRoutes(routes)

// Set a listen port of 8181
server.serverPort = 8181

do {
    // Launch the HTTP server.
    try server.start()
} catch PerfectError.networkError(let err, let msg) {
    print("Network error thrown: \(err) \(msg)")
}

```

Build and run the project again with:

```

swift build
.build/debug/MyAwesomeProject

```

The server is now running and waiting for connections. Access <http://localhost:8181/> to see the greeting. Hit "control-c" to terminate the server.

## Xcode

Swift Package Manager (SPM) can generate an Xcode project which can run the PerfectTemplate server and provide full source code editing and debugging for your project. Enter the following in your terminal:

```

swift package generate-xcodeproj

```

Open the generated file "PerfectTemplate.xcodeproj" and add the following to the "Library Search Paths" for the project (not just the target):

```

$(PROJECT_DIR) - Recursive

```

Ensure that you have selected the executable target and selected it to run on "My Mac". Also ensure that the correct Swift toolchain is selected. You can now run and debug the server directly in Xcode.

# An HTTP and Web Services Primer

Most of the web as we know it, and as applies to the common usage of server side development, is built upon only a few main technologies.

This document will attempt to provide an overview of some of these as they relate to Perfect, server-side Swift, and general application programming interface (API) development.

## What Is an API?

An API is a bridge between two systems. It generally accepts a standardized type and style of input, transforms it internally, and works with another system to perform an action or return information.

An example of an API is GitHub's API: most will be familiar with using GitHub's web interface, but they also have an API which allows applications such as task and issue management systems like [Jira](#) and continuous integration (CI) systems such as [Bamboo](#) to link directly to your repositories to **provide greater functionality than any one system could by itself**.

An API like these are generally built upon HTTP or HTTPS.

## HTTP and HTTPS

"HTTP" is the common acronym for "HyperText Transfer Protocol". It is a set of standards that allows users to publish multimedia documents on the internet and exchange information they find on webpages.

"HTTPS" is a protocol for secure, encrypted HTTP communications. Each end of the communication agrees on a trusted "key" that encrypts the information being transmitted, with the ability to decrypt it on receipt. The more complex the security, the harder it is for a third party to intercept and read it, and potentially change it.

When you access a website in your browser, it will likely be over HTTP, or if the "lock" shows it will be using HTTPS.

When an iOS or Android application accesses a backend server to get information such as the most recent weather report, it is using HTTP or, more likely, HTTPS.

## The General Form of an API

### Routes

An API consists of "routes", which are similar to directory/folder name paths through a file system to a document. Each route points to a different action you wish to perform. A "show me a list of users" route is different from a "create a new user" route.

In an API, these routes will often not exist as actual directories and documents, but as pointers to functions. The "directory structure" implied is usually a way of logically grouping functionality. For example:

```
/api/v1/users/list
/api/v1/users/detail
/api/v1/users/create
/api/v1/users/modify
/api/v1/users/delete

/api/v1/companies/list
/api/v1/companies/detail
/api/v1/companies/create
/api/v1/companies/modify
/api/v1/companies/delete
```

This example illustrates a typical "CRUD" system. "CRUD" means "Create, Read, Update, Delete". The difference between the two groups seems to only be the `users` versus `companies` part, but they will point to different functions.

Additionally, the "detail", "modify", and "delete" routes will include a unique identifier for the record. For example:

```
/api/v1/users/list
/api/v1/users/detail?id=5d096846-a000-43db-b6c5-a5883135d71d
/api/v1/users/create
/api/v1/users/modify?id=5d096846-a000-43db-b6c5-a5883135d71d
/api/v1/users/delete?id=5d096846-a000-43db-b6c5-a5883135d71d
```

The example above shows these routes passing to the server an id parameter that relates to a specific record. The list and create routes don't have an id because for those routes an id parameter is irrelevant.

In addition to routes, a request to each of these individual routes will include an HTTP "verb".

### HTTP Verbs

HTTP verbs are additional pieces of information that a web browser or mobile application client supplies with every request to a route. These verbs can give additional "context" to the API server as to the nature of the data being received.

Common HTTP verbs include:

#### GET

The GET method requests a specified route, and the only parameters passed from client to server are in the URL. Requests using GET should only retrieve data and have no other effect. Using a GET request to delete a database record is possible but it is not recommended.

#### POST

Normally used for sending info to create a new record in a database, a POST request is what normally gets submitted when you fill in a form on a website. The name-value pairs of the data are submitted in a POST request's "POST Body" and are read by the API server as discrete pairs.

## PATCH

PATCH requests are generally considered to be the same as a POST but they are used for updates.

## PUT

Used predominantly for file uploads, a PUT request will include a file in the body of the request sent to the server.

## DELETE

A DELETE request is the most descriptive of all the HTTP verbs. When sending a DELETE request you are instructing the API server to remove a specific resource. Usually some form of uniquely identifying information is included in the URL.

## Using HTTP Verbs and Routes to Simplify an API

When used together, these two components of a request can reduce the perceived complexity of an API.

Looking at the following structure with the HTTP verb followed by a URL, you will see that the process is much simpler and more specific:

```
GET    /api/v1/users
GET    /api/v1/users/5d096846-a000-43db-b6c5-a5883135d71d
POST   /api/v1/users
PATCH /api/v1/users/5d096846-a000-43db-b6c5-a5883135d71d
DELETE /api/v1/users/5d096846-a000-43db-b6c5-a5883135d71d
```

At first glance, all seem to be pointing to the same route: `/api/v1/users`. However, each route performs a different action. Similarly, the difference between the first two GET routes is the id appended as a URL component, not a parameter as shown in the earlier example. This command is usually achieved with a "wildcard" that is specified in the route setup.

The next step is to review the [Handling Requests](#) chapters in the Perfect documentation. These sections will outline how to implement routes pointing to functions and methods, how to access information passed to the API server from a frontend, and whether it be a web browser or a mobile application.

# Repository Layout

The Perfect framework has been divided into several repositories to make it easy for you to find, download, and install the components you need for your project:

## The Perfect Core Library

[Perfect](#) - This repository contains the core PerfectLib and will continue to be the main landing point for the project

## The Perfect Toolkit

There are many components in the main [Perfect Repo](#), <https://github.com/PerfectlySoft> that make up the comprehensive Perfect Toolkit. There are database drivers, utilities, session management, and authentication systems. All components are documented [here](#).

## The Perfect Template

[PerfectTemplate](#) - A simple starter project which compiles with the Swift Package Manager into a standalone executable HTTP server. This repository is ideal for starting on your own Perfect-based project

## The Perfect Documentation - Open Source

[PerfectDocs](#) - Contains all API reference-related material

## Perfect Examples

[PerfectExamples](#) - All the Perfect example projects and documentation

## StORM - A Swift ORM

[StORM is a Swift ORM](#), written in Perfect. The list of supported databases will continue to grow and mature.

# Perfect Servers

[A collection of standalone servers written in Perfect](#), ready for deployment (with a little configuration on your part!)# Building with Swift Package Manager

[Swift Package Manager](#) (SPM) is a command-line tool for building, testing, and managing dependencies for Swift projects. All of the components in Perfect are designed to build with SPM. If you create a project with Perfect, it will need to use SPM as well.

The best way to start a new Perfect project is to fork or clone the [PerfectTemplate](#). It will give you a very simple "Hello, World!" server message which you can edit and modify however you wish.

Before beginning, ensure you have read the [dependencies](#) document, and that you have a functioning Swift 3.0 toolchain for your platform.

The next step is to clone the template project. Open a new command-line terminal and change directory (cd) where you want the project to be cloned. Type the following into your terminal to download a directory called "PerfectTemplate":

```
git clone https://github.com/PerfectlySoft/PerfectTemplate.git
```

Within the Perfect Template, you will find two important file items:

- A *Sources* directory containing all of the Swift source files for Perfect
- An SPM manifest named "*Package.swift*" listing the dependencies this project requires

All SPM projects will have, at least, both a *Sources* directory and a *Package.swift* file. This project starts out with only one dependency: the [Perfect-HTTPServer](#) project.

## Dependencies in Package.swift

The PerfectTemplate *Package.swift* manifest file contains the following content:

```
import PackageDescription

let package = Package(
    name: "PerfectTemplate",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/PerfectlySoft/Perfect-HTTPServer.git",
            majorVersion: 2)
    ]
)
```

**Note:** The version presented above may differ from what you have on your terminal. We recommend you consult the actual repository for the most up-to-date content.

There are two important elements in the *Package.swift* file that you may wish to edit.

The first one is the **name** element. It indicates the name of the project, and thus, the name of the executable file which will be generated when the project is built.

The second element is the **dependencies** list. This element indicates all of the subprojects that your application is dependent upon. Each item in this array consists of a ".*Package*" with a repository URL and a version.

The example above indicates a wide range of versions so that the template will always grab the newest revision of the HTTPServer project. You may want to restrict your dependencies to specific stable versions. For example, if you want to only build against version 2 of the Perfect HTTPServer project, your ".*Package*" element may look like the following:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-HTTPServer.git", majorVersion: 2)
```

As your project grows and you add dependencies, you will put all of them in the **dependencies** list. SPM will automatically download the appropriate versions and compile them along with your project. All dependencies are downloaded into a *Packages* directory which SPM will automatically create. For example, if you wanted to use Mustache templates in your server, your *Package.swift* file might look like the following:

```
import PackageDescription

let package = Package(
    name: "PerfectTemplate",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/PerfectlySoft/Perfect-HTTPServer.git",
            majorVersion: 2),
        .Package(url: "https://github.com/PerfectlySoft/Perfect-Mustache.git",
            majorVersion: 2)
    ]
)
```

As you can see, the [Perfect-Mustache](#) project was added as a dependency. It provides Mustache template support for your Perfect server. Within your project code, you can now import “PerfectMustache”, and use the facilities it offers.

As your dependency list grows, you may want to manage the list differently. The following example includes all the Perfect repositories. The list of URLs is maintained separately, and they are mapped to the format required by the **dependencies** parameter.

```
import PackageDescription

let versions = majorVersion: 2
let urls = [
    "https://github.com/PerfectlySoft/Perfect-HTTPServer.git",
    "https://github.com/PerfectlySoft/Perfect-FastCGI.git",
    "https://github.com/PerfectlySoft/Perfect-CURL.git",
    "https://github.com/PerfectlySoft/Perfect-PostgreSQL.git",
    "https://github.com/PerfectlySoft/Perfect-SQLite.git",
    "https://github.com/PerfectlySoft/Perfect-Redis.git",
    "https://github.com/PerfectlySoft/Perfect-MySQL.git",
    "https://github.com/PerfectlySoft/Perfect-MongoDB.git",
    "https://github.com/PerfectlySoft/Perfect-WebSockets.git",
    "https://github.com/PerfectlySoft/Perfect-Notifications.git",
    "https://github.com/PerfectlySoft/Perfect-Mustache.git"
]

let package = Package(
    name: "PerfectTemplate",
    targets: [],
    dependencies: urls.map { .Package(url: $0, versions) }
)
```

## Building

SPM provides the following commands for building your project, and for cleaning up any build artifacts:

```
swift build
```

This command will download any dependencies if they haven't been already acquired and attempt to build the project. If the build is successful, then the resulting executable will be placed in the (hidden) `.build/debug/` directory. When building the PerfectTemplate project, you will see as the last line of SPM output:

Linking `.build/debug/PerfectTemplate` . Entering `.build/debug/PerfectTemplate` will run the server. By default, a debug version of the executable will be generated. To build a production ready release version, you would issue the command `swift build -c release` . This will place the resulting executable in the `.build/release/` directory.

```
swift build --clean
```

```
swift build --clean=dist
```

It can be useful to wipe out all intermediate data and do a fresh build. Providing the `--clean` argument will delete the `.build` directory, and permit a fresh build. Providing the `--clean=dist` argument will delete both the `.build` directory and the `Packages` directory. Doing so will re-download all project dependencies during the next build to ensure you have the latest version of a dependent project.

## Xcode Projects

SPM can generate an Xcode project based on your *Package.swift* file. This project will permit you to build and debug your application within Xcode. To generate the Xcode project, issue the following command:

```
swift package generate-xcodeproj
```

The command will generate the Xcode project file into the same directory. For example, issuing the command within the PerfectTemplate project directory will produce the message

```
generated: ./PerfectTemplate.xcodeproj .
```

**Note: It is not advised to edit or add files directly to this Xcode project.** If you add any further dependencies, or require later versions of any dependencies, you will need to regenerate this Xcode project. As a result, any modifications you have made will be overwritten.

## Additional Information

For more information on the Swift Package Manager, visit:

<https://swift.org/package-manager/>

# HTTPServer

This document describes the three methods by which you can launch new Perfect HTTP servers. These methods differ in their complexity and each caters to a different use case.

The first method is data driven whereby you provide either a Swift Dictionary or JSON file describing the servers you wish to launch. The second method describes the desired servers using Swift language constructs complete with the type checking and compile-time constraints provided by Swift. The third method permits you to instantiate an HTTPServer object and then procedurally configure each of the required properties before manually starting it.

HTTP servers are configured and started using the functions available in the `HTTPServer` namespace. A Perfect HTTP server consists of at least a name and a listen port, one or more handlers, and zero or more request or response filters. In addition, a secure HTTPS server will also have TLS related configuration information such as a certificate or key file path.

When starting servers you can choose to wait until the servers have terminated (which will generally not happen until the process is terminated) or receive `LaunchContext` objects for each server which permits them to be individually terminated and waited upon.

## HTTPServer Configuration Data

One or more Perfect HTTP servers can be configured and launched using structured configuration data. This includes setting elements such as the listen port and bind address but also permits pointing handlers to specific fuctions by name. This feature is required if loading server configuration data from a JSON file. In order to enable this functionality on Linux, you must build your SPM executable with an additional flag:

```
swift build -Xlinker --export-dynamic
```

This is only required on Linux and only if you are going to be using the configuration data system described in this section with JSON text files.

Call one of the static `HTTPServer.launch` functions with either a path to a JSON configuration file, a File object pointing to the configuration file or a Swift Dictionary<String:Any>.

The resulting configuration data will be used to launch one or more HTTP servers.

```
public extension HTTPServer {
    public static func launch(wait: Bool = true, configurationPath path: String) throws -> [LaunchContext]
    public static func launch(wait: Bool = true, configurationFile file: File) throws -> [LaunchContext]
    public static func launch(wait: Bool = true, configurationData data: [String:Any]) throws -> [LaunchContext]
}
```

The default value for the `wait` parameter indicates that the function should not return but should block until all servers have terminated or the process is killed. If `false` is given for `wait` then the returned array of `LaunchContext` objects can be used to monitor or terminate the individual servers. Most applications will want the function to wait and so the functions can be called without including the `wait` parameter.

```
do {
    try HTTPServer.launch(configurationPath: "/path/to/perfecthttp.json")
} catch {
    // handle critical failure
}
```

Note that the configuration file can be located or named whatever you'd like, but it should have the `.json` file extension. We may support other file formats in the future and ensuring that your configuration file has an extension which describes its content is important.

After it is decoded from JSON, at its top level, the configuration data should contain a "servers" key with a value that is an array of Dictionary<String:Any>. These dictionaries describe the servers which will be launched.

```
[
  "servers": [
    [...],
    [...],
    [...],
  ]
]
```

A simple example single server configuration dictionary might look as follows. Note that the keys and values in this example are all explained in the subsequent sections of this document.

```
[
  "servers": [
    [
      "name": "localhost",
      "port": 8080,
      "routes": [
        [
          "method": "get",
          "uri": "/*",
          "handler": PerfectHTTPServer.HTTPHandler.staticFiles,
          "documentRoot": "/path/to/webroot"
        ],
        [
          "methods": ["get", "post"],
          "uri": "/api/*",
          "handler": PerfectHTTPServer.HTTPHandler.redirect,
          "base": "http://other.server.ca"
        ]
      ]
    ]
  ]
]
```

The options available for servers are as follows:

### name:

This **required** string value is primarily used to identify the server and would generally be the same as the server's domain name. Applications may use the server name to construct URLs pointing back to the server.

It is permitted to use the same name for multiple servers. For example, you may have three servers on the same host all listening on different ports. These three servers could all have the same name.

Corresponding HTTPServer property: `HTTPServer.serverName`.

### port:

This **required** integer value indicates the port on which the server should listen. TCP ports range from 0 to 65535. Ports in the range of 0 to 1024 require root permissions to start. See the `runAs` option to indicate the user which the process should switch to after binding on such a port.

Corresponding HTTPServer property: `HTTPServer.serverPort`.

### address:

This **optional** String value should be a dotted IP address. This indicates the local address on which the server should bind. If not given, this value defaults to "0.0.0.0" which indicates that the server should bind on all available local IP addresses. Using "::" for this value will enable listening on all local IPv6 & IPv4 addresses.

Corresponding HTTPServer property: `HTTPServer.serverAddress`.

### routes:

This **optional** element should have a value that is an array of [String:Any] dictionaries. Each element of the array indicates a URI route or group of URI routes which map an incoming HTTP request to a handler. See [Routing](#) for specifics on Perfect's URI routing system.

Each [String:Any] dictionary in the "routes" array must be either a simple route with a handler or a group of routes.

If the dictionary has a "uri" and a "handler" key, then it is assumed to be a simple route. A group of yours must have at least a "children" key.

A simple route consists of zero or more HTTP methods, a URI and the name of a `RequestHandler` function or a function which returns a `RequestHandler`. The key names are: "method" or "methods", "uri", and "handler". The value for the "methods" key should be an array of strings. If no method values are provided then any HTTP method may trigger the handler.

A group of handlers consists of an optional "baseURI", an optional "handler", and a required array of "children" whose value must be `[[String:Any]]`. Each of the children in this array can in turn express either simple route handlers or further groups of routes. The optional "handler" function will be executed before the final request handler. Multiple handlers can be chained in this manner, some running earlier to set up certain state or to screen the incoming requests. Handlers used in this manner should call `response.next()` to indicate that the handler has finished executing and the next one can be called. If there are no further handlers then the request will be completed. If an intermediate handler determines that the request should go no further, it can call `response.completed()` and no further handlers will be executed.

Perfect comes with request handlers that take care of various common tasks such as redirecting clients or serving static, on-disk files. The following example defines a server which listens on port 8080 and has two handlers, one of which serves static files while the other redirects clients to a new URL.

```
[
  "servers": [
    [
      "name": "localhost",
      "port": 8080,
      "routes": [
        [
          "method": "get",
          "uri": "**",
          "handler": PerfectHTTPServer.HTTPHandler.staticFiles,
          "documentRoot": "/path/to/webroot"
        ],
        [
          "baseURI": "/api",
          "children": [
            [
              "methods": [ "get", "post" ],
              "uri": "**",
              "handler": PerfectHTTPServer.HTTPHandler.redirect,
              "base": "http://other.server.ca"
            ]
          ]
        ]
      ]
    ]
  ]
]
```

Corresponding HTTPServer property: `HTTPServer.addRoutes`.

## Adding Custom Request Handlers

While the built-in Perfect request handlers can be handy, most developers will want to add custom behaviour to their servers. The "handler" key values can point to your own functions which will each return the `RequestHandler` to be called when the route uri matches an incoming request.

It's important to note that the function names which you would enter into the configuration data are **static** functions which *return* the `RequestHandler` that will be subsequently used. These functions accept the current configuration data `[String:Any]` for the particular route in order to extract any available configuration data such as the `staticFiles` "documentRoot" described above.

Alternatively, if you do not need any of the available configuration data (for example, your handler requires no configuration) you can simply indicate the `RequestHandler` itself.

It's also vital that the name you provide be fully qualified. That is, it should include your Swift module name, the name of any interstitial nesting constructs such as struct or enum, and then the function name itself. These should all be separated by "." (periods). For example, you can see the static file handler is given as "PerfectHTTPServer.HTTPHandler.staticFiles". It resides in the module "PerfectHTTPServer", in an extension of the struct "HTTPHandler" and is named "staticFiles".

Note that if you are creating a configuration directly in Swift code as a dictionary then you do not have to quote the function names that you provide. The value for the "handler" (and subsequently the "filters" described later in this chapter) can be given as direct function references.

An example request handler generator which could be used in a server configuration follows.

```

public extension HTTPHandler {
    public static func staticFiles(data: [String:Any]) throws -> RequestHandler {
        let documentRoot = data["documentRoot"] as? String ?? "./webroot"
        let allowResponseFilters = data["allowResponseFilters"] as? Bool ?? false
        return {
            req, resp in
                StaticFileHandler(documentRoot: documentRoot, allowResponseFilters: allowResponseFilters)
                    .handleRequest(request: req, response: resp)
        }
    }
}

```

Note: the `HTTPHandler` struct is an abstract namespace defined in `PerfectHTTPServer`. It consists of only static request handler generators such as this.

Request handler generators are encouraged to `throw` when required configuration data is not provided by the user or if the data is invalid. Ensure that the Error you throw will provide a helpful message when it is converted to String. This will ensure that users see such configuration problems early so that they can be corrected. If the handler generator cannot return a valid `RequestHandler` then it should throw an error.

## filters:

Request filters can screen or manipulate incoming request data. For example, an authentication filter might check to see if a request has certain permissions, and if not, return an error to the client. Response filters do the same for outgoing data, having an opportunity to change response headers or body data. See [Request and Response Filters](#) for specifics on Perfect's request filtering system.

The value for the "filters" key is an array of dictionaries containing keys which describe each filter. The required keys for these dictionaries are "type", and "name". The possible values for the "type" key are "request" or "response", to indicate either a request or a response filter. A "priority" key can also be provided with a value of either "high", "medium", or "low". If a priority is not provided then the default value will be "high".

The following example adds two filters, one for requests and one for responses.

```

[
    "servers": [
        [
            "name": "localhost",
            "port": 8080,
            "routes": [
                [ "method": "get", "uri": "/*", "handler": "PerfectHTTPServer.HTTPHandler.staticFiles",
                  "documentRoot": "./webroot" ]
            ],
            "filters": [
                [
                    "type": "request",
                    "priority": "high",
                    "name": "PerfectHTTPServer.HTTPFilter.customReqFilter"
                ],
                [
                    "type": "response",
                    "priority": "high",
                    "name": "PerfectHTTPServer.HTTPFilter.custom404",
                    "path": "./webroot/404.html"
                ]
            ]
        ]
    ]
]

```

Filter names work in much the same way as route handlers do, however, the function signatures are different. A request filter generator function takes the `[String:Any]` containing the configuration data and returns a `HTTPRequestFilter` or a `HTTPResponseFilter` depending on the filter type.

```
// a request filter generator
public func customReqFilter(data: [String:Any]) throws -> HTTPRequestFilter {
    struct ReqFilter: HTTPRequestFilter {
        func filter(request: HTTPRequest, response: HTTPResponse, callback: (HTTPRequestFilterResult) -> ()) {
            callback(.continue(request, response))
        }
    }
    return ReqFilter()
}

// a response filter generator
public func custom404(data: [String:Any]) throws -> HTTPResponseFilter {
    guard let path = data["path"] as? String else {
        fatalError("HTTPFilter.custom404(data: [String:Any]) requires a value for key \"path\".")
    }
    struct Filter404: HTTPResponseFilter {
        let path: String
        func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
            if case .notFound = response.status {
                do {
                    response.setBody(string: try File(path).readString())
                } catch {
                    response.setBody(string: "An error occurred but I could not find the error file. \(response.status)")
                }
                response.setHeader(.contentType, value: "\(response.bodyBytes.count)")
            }
            return callback(.continue)
        }
        func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
            callback(.continue)
        }
    }
    return Filter404(path: path)
}
```

Corresponding HTTPServer properties: `HTTPServer.setRequestFilters` , `HTTPServer.setResponseFilters` .

## tlsConfig:

If a "tlsConfig" key is provided then a secure HTTPS server will be attempted. The value for the TLS config should be a dictionary containing the following required and optional keys/values:

- certPath - **required** String file path to the certificate file
- keyPath - optional String file path to the key file
- cipherList - optional array of ciphers that the server will support
- caCertPath - optional String file path to the CA cert file
- verifyMode - optional String indicating how the secure connections should be verified. The value should be one of:
  - none
  - peer
  - failIfNoPeerCert
  - clientOnce
  - peerWithFailIfNoPeerCert
  - peerClientOnce
  - peerWithFailIfNoPeerCertClientOnce

The default values for the cipher list can be obtained through the `TLSConfiguration.defaultCipherList` property.

## User Switching

After starting as root and binding the servers to the indicated ports (low, restricted ports such as 80, for example), it is recommended that the server process switch to a non-root operating system user. These users are generally given low or restricted permissions in order to prevent security attacks which could be perpetrated were the server running as root.

At the top level of your configuration data (as a sibling to the "servers" key), you can include a "runAs" key with a string value. This value indicates the name of the desired user. The process will switch to the user only after all servers have successfully bound their respective listen ports.

Only a server process which is started as root can switch users.

Corresponding HTTPServer function: `HTTPServer.runAs(_ user: String)` .

## HTTPServer.launch

There are several variants of the `HTTPServer.launch` functions which permit one or more servers to be started. These functions abstract the inner workings of the `HTTPServer` object and provide a more streamlined interface for server launching.

The simplest of these methods launches a single server with options:

```
public extension HTTPServer {
    public static func launch(wait: Bool = true, name: String, port: Int, routes: Routes,
        requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
        responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = []) throws -> LaunchContext
    public static func launch(wait: Bool = true, name: String, port: Int, routes: [Route],
        requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
        responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = []) throws -> LaunchContext
    public static func launch(wait: Bool = true, name: String, port: Int, documentRoot root: String,
        requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
        responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = []) throws -> LaunchContext
}
```

The remaining launch functions take one or more server descriptions, launches them and returns their `LaunchContext` objects.

```
public extension HTTPServer {
    public static func launch(wait: Bool = true, _ servers: [Server]) throws -> [LaunchContext]
    public static func launch(wait: Bool = true, _ server: Server, _ servers: Server...) throws -> [LaunchContext]
}
```

The `Server`, which describes the `HTTPServer` object that will eventually be launched, looks like so:

```
public extension HTTPServer {
    public struct Server {
        public init(name: String, address: String, port: Int, routes: Routes,
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = [])
        public init(tlsConfig: TLSConfiguration, name: String, address: String, port: Int, routes: Routes,
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = [])
        public init(name: String, port: Int, routes: Routes,
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = [])
        public init(tlsConfig: TLSConfiguration, name: String, port: Int, routes: Routes,
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = [])

        public static func server(name: String, port: Int, routes: Routes,
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = []) -> Server
        public static func server(name: String, port: Int, routes: [Route],
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = []) -> Server
        public static func server(name: String, port: Int, documentRoot root: String,
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = []) -> Server
        public static func secureServer(_ tlsConfig: TLSConfiguration, name: String, port: Int, routes: [Route],
            requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [],
            responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = []) -> Server
    }
}
```

The following examples show some common usages.

```
// start a single server serving static files
try HTTPServer.launch(name: "localhost", port: 8080, documentRoot: "/path/to/webroot")

// start two servers. have one serve static files and the other handle API requests
let apiRoutes = Route(method: .get, uri: "/foo/bar", handler: {
    req, resp in
    //do stuff
})
try HTTPServer.launch(
    .server(name: "localhost", port: 8080, documentRoot: "/path/to/webroot"),
    .server(name: "localhost", port: 8181, routes: [apiRoutes]))

// start a single server which handles API and static files
try HTTPServer.launch(name: "localhost", port: 8080, routes: [
    Route(method: .get, uri: "/foo/bar", handler: {
        req, resp in
        //do stuff
    }),
    Route(method: .get, uri: "/foo/bar", handler:
        HTTPHandler.staticFiles(documentRoot: "/path/to/webroot"))
])

let apiRoutes = Route(method: .get, uri: "/foo/bar", handler: {
    req, resp in
    //do stuff
})
// start a secure server
try HTTPServer.launch(.secureServer(TLSConfiguration(certPath: "/path/to/cert"), name: "localhost", port: 8080, routes: [apiRoutes]))
```

The `TLSConfiguration` struct configures the server for HTTPS and is defined as:

```
public struct TLSConfiguration {
    public init(certPath: String, keyPath: String? = nil,
        caCertPath: String? = nil, certVerifyMode: OpenSSLVerifyMode? = nil,
        cipherList: [String] = TLSConfiguration.defaultCipherList)
}
```

## LaunchContext

If `wait: false` is given to any of the `HTTPServer.launch` functions then one or more `LaunchContext` objects are returned. These objects permit each server's status to be checked and permit the server to be terminated.

```
public extension HTTPServer {
    public struct LaunchFailure: Error {
        let message: String
        let configuration: Server
    }

    public class LaunchContext {
        public var terminated: Bool
        public let server: Server
        public func terminate() -> LaunchContext
        public func wait(seconds: Double = Threading.noTimeout) throws -> Bool
    }
}
```

If a launched server fails because an error is thrown then that error will be translated and thrown when the `wait` function is called.

## HTTPServer Object

An `HTTPServer` object can be instantiated, configured and manually started.

```

public class HTTPServer {
    /// The directory in which web documents are sought.
    /// Setting the document root will add a default URL route which permits
    /// static files to be served from within.
    public var documentRoot: String
    /// The port on which the server is listening.
    public var serverPort: UInt16 = 0
    /// The local address on which the server is listening. The default of 0.0.0.0 indicates any address.
    public var serverAddress = "0.0.0.0"
    /// Switch to user after binding port
    public var runAsUser: String?

    /// The canonical server name.
    /// This is important if utilizing the `HTTPRequest.serverName` property.
    public var serverName = ""
    public var ssl: (sslCert: String, sslKey: String)?
    public var caCert: String?
    public var certVerifyMode: OpenSSLVerifyMode?
    public var cipherList: [String]
    /// Initialize the server object.
    public init()
    /// Add the Routes to this server.
    public func addRoutes(_ routes: Routes)
    /// Set the request filters. Each is provided along with its priority.
    /// The filters can be provided in any order. High priority filters will be sorted above lower priorities.
    /// Filters of equal priority will maintain the order given here.
    public func setRequestFilters(_ request: [(HTTPRequestFilter, HTTPFilterPriority)]) -> HTTPServer
    /// Set the response filters. Each is provided along with its priority.
    /// The filters can be provided in any order. High priority filters will be sorted above lower priorities.
    /// Filters of equal priority will maintain the order given here.
    public func setResponseFilters(_ response: [(HTTPResponseFilter, HTTPFilterPriority)]) -> HTTPServer
    /// Start the server. Does not return until the server terminates.
    public func start() throws
    /// Stop the server by closing the accepting TCP socket. Calling this will cause the server to break out of the otherwise blocking `start` function.
    public func stop()
}

```

## Handling Requests

As an internet server, Perfect's main function is to receive and respond to requests from clients. Perfect provides objects to represent the request and the response components, and it permits you to install handlers to control the resulting content generation.

Everything begins with the creation of the server object. The server object is configured and subsequently binds and listens for connections on a particular port. When a connection occurs, the server begins reading the request data. Once the request has been completely read, the server will pass the [request object](#) through any request filters.

These filters permit the incoming request to be modified. The server will then use the request's path URI and search the [routing](#) system for an appropriate handler. If a handler is found, it is given a chance to populate a [response object](#). Once the handler indicates that it is done responding, the response object is passed through any response filters. These filters permit the outgoing data to be modified. The resulting data is then pushed to the client, and the connection is either closed, or it can be reused as an HTTP persistent connection, a.k.a. HTTP keep-alive, for additional requests and responses.

Consult the following sections for more details on each specific phase and what can be accomplished during each:

- [Routing](#) - Describes the routing system and shows how to install URL handlers
- [HTTPRequest](#) - Provides details on the request object protocol
- [HTTPResponse](#) - Provides details on the response object protocol
- [Request & Response Filters](#) - Shows how to add filters and illustrates how they are useful

In addition, the following sections show how to use some of the pre-made, specialized handlers to accomplish specific tasks:

- [Static File Handler](#) - Describes how to serve static file content
- [Mustache](#) - Shows how to populate and serve Mustache template based content # Routing

Routing determines which handler receives a specific request. A handler is a routine, function, or method dedicated to receiving and acting on certain types of requests or signals. Requests are routed based on two pieces of information: the HTTP request method, and the request path. A route refers to an HTTP method, path, and handler combination. Routes are created and added to the server before it starts listening for requests. For example:

```

var routes = Routes()
routes.add(method: .get, uri: "/path/one") {
    request, response in
    response.setBody(string: "Handler was called")
    .completed()
}
server.addRoutes(routes)

```

Once the Perfect server receives a request, it will pass the request object through any registered request filters. These filters have a chance to modify the request object in ways which may affect the routing process, such as changing the request path. The server will then search for a route which matches the current request method and path. If a route is successfully found for the request, the server will deliver both the request and response objects to the found handler. If a route is not found for a request, the server will send a “404” or “Not Found” error response to the client.

## Creating Routes

The routing API is part of the [PerfectHTTP](#) project. Interacting with the routing system requires that you first `import PerfectHTTP`.

Before adding any route, you will need an appropriate handler function. Handler functions accept the request and response objects, and are expected to generate content for the response. They will indicate when they have completed a task. The typealias for a request handler is as follows:

```

// Function which receives request and response objects and generates content.
public typealias RequestHandler = (HTTPRequest, HTTPResponse) -> ()

```

Requests are considered active until a handler indicates that it has concluded. This is done by calling either the `HTTPResponse.next()` or the `HTTPResponse.completed()` function. Request handling in Perfect is fully asynchronous, so a handler function can return, spin off into new threads, or perform any other sort of asynchronous activity before calling either of these functions.

Request handlers can be chained, in that one request URI can identify multiple handlers along its path. The handlers will be executed in order, and each given a chance to either continue the request processing or halt it.

A request is considered to be active up until `HTTPResponse.completed()` is called. Calling `.next()` when there are no more handlers to execute is equivalent to calling `.completed()`. Once the response is marked as completed, the outgoing headers and body data, if any, will be sent to the client.

Individual uri handlers are added to a `Routes` object before they are added to the server. When a `Routes` object is created, one or more routes are added using its `add` functions. `Routes` provides the following functions:

```

public struct Routes {
    // Initialize with no baseUri.
    public init(handler: RequestHandler? = nil)
    // Initialize with a baseUri.
    public init(baseUri: String, handler: RequestHandler? = nil)
    // Add all the routes in the Routes object to this one.
    public mutating func add(routes: Routes)
    // Add the given method, uri and handler as a route.
    public mutating func add(method: HTTPMethod, uri: String, handler: RequestHandler)
    // Add the given method, uris and handler as a route.
    public mutating func add(method: HTTPMethod, uris: [String], handler: RequestHandler)
    // Add the given uri and handler as a route.
    // This will add the route got both GET and POST methods.
    public mutating func add(uri: String, handler: RequestHandler)
    // Add the given method, uris and handler as a route.
    // This will add the route got both GET and POST methods.
    public mutating func add(uris: [String], handler: RequestHandler)
    // Add one Route to this object.
    public mutating func add(_ route: Route)
}

```

A `Routes` object can be initialized with a baseURI. The baseURI will be prepended to any route added to the object. For example, one could initialize a `Routes` object for version one of an API and give it a baseURI of “/v1”. Every route added will be prefixed with /v1. `Routes` objects can also be added to other `Routes` objects, and each route therein will be prefixed in the same manner. The following example shows the creation of two sets of routes for two versions of an API. The second version differs in behavior in only one endpoint:

```

var routes = Routes()
// Create routes for version 1 API
var api = Routes()
api.add(method: .get, uri: "/call1", handler: { _, response in
    response.setBody(string: "API CALL 1")
    response.completed()
})
api.add(method: .get, uri: "/call2", handler: { _, response in
    response.setBody(string: "API CALL 2")
    response.completed()
})

// API version 1
var api1Routes = Routes(baseUrl: "/v1")
// API version 2
var api2Routes = Routes(baseUrl: "/v2")

// Add the main API calls to version 1
api1Routes.add(routes: api)
// Add the main API calls to version 2
api2Routes.add(routes: api)
// Update the call2 API
api2Routes.add(method: .get, uri: "/call2", handler: { _, response in
    response.setBody(string: "API v2 CALL 2")
    response.completed()
})

// Add both versions to the main server routes
routes.add(routes: api1Routes)
routes.add(routes: api2Routes)

```

A Routes object can also be given an optional handler. This handler will be called if that part of a path is matched against another subsequent handler.

For example, a "/v1" version of an API might enforce a particular authentication mechanism on the clients. The authentication handler could be added to the "/v1" portion of the api and as each actual endpoint is reached the authentication handler would be given a chance to evaluate the request before passing it down to the remaining handlers(s).

```

var routes = Routes(baseUrl: "/v1") {
    request, response in
    if authorized(request) {
        response.next()
    } else {
        response.completed(.unauthorized)
    }
}
routes.add(method: .get, uri: "/call1") {
    _, response in
    response.setBody(string: "API CALL 1").next()
}
routes.add(method: .get, uri: "/call2") {
    _, response in
    response.setBody(string: "API CALL 2").next()
}

```

Accessing either "/v1/call1" or "/v1/call2" will pass the request to the handler set on "/v1" routes.

Routes can be nested like this as deeply as you wish. Handlers set directly on Routes objects are considered non-terminal. That is, they can not be accessed directly by clients and will only be executed if the request goes on to match a handler which is terminal. Likewise, handlers which are terminal but lie on the path of a more complete match will not be executed. For example with a handler on "/v1/users" and on "/v1/users/foo", accessing "/v1/users/foo" will not execute "/v1/users".

## Adding Server Routes

Both Perfect-HTTPServer and Perfect-FastCGI support routing. Consult [HTTPServer]([Routing](#)) for details on how to apply routes to HTTP servers.

## Variables

Route URIs can also contain variable components. A variable component begins and ends with a set of curly brackets `{ }`. Within the brackets is the variable identifier. A variable identifier can consist of any character except the closing curly bracket `}`. Variable components work somewhat like single wildcards do in that they match any single literal path component value. The actual value of the URL component which is matched by the variable is saved and made available through the `HttpRequest.urlVariables` dictionary. This dictionary is of type `[String:String]`. URI variables are a good way to gather dynamic data from a request. For example, a URL might make a user management related request,

and include the user id as a component in the URL.

For example, when given the URI `/foo/{bar}/baz`, a request to the URL `/foo/123/baz` would match and place it in the `HttpRequest.urlVariables` dictionary with the value "123" under the key "bar".

## Wildcards

A wildcard, also referred to as a wild character, is a symbol used to replace or represent one or more characters. Beyond full literal URI paths, routes can contain wildcard segments.

Wildcards match any portion of a URI and can be used to route groups of URIs to a single handler. Wildcards consist of either one or two asterisks. A single asterisk can occur anywhere in a URI path as long as it represents one full component of the URI. A double asterisk, or trailing wildcard, can occur only at the end of a URI. Trailing wildcards match any remaining portion of a URI.

A route with the URI `/foo/*/baz` would match the both of these URLs:

```
/foo/123/baz
/foo/bar/baz
```

A route with the URI `/foo/**` would match all of the following URLs:

```
/foo/bar/baz
/foo
```

A route with the URI `/**` would match any request.

A trailing wildcard route will save the URI portion which is matched by the wildcard. It will place this path segment in the `HttpRequest.urlVariables` map under the key indicated by the global variable `routeTrailingWildcardKey`. For example, given the route URI `/foo/**` and a request URI `/foo/bar/baz`, the following snippet would be true:

```
request.urlVariables[routeTrailingWildcardKey] == "/bar/baz"
```

## Priority/Ordering

Because route URIs could potentially conflict, literal, wildcard and variable paths are checked in a specific order. Path types are checked in the following order:

1. Variable paths
2. Literal paths
3. Wildcard paths
4. Trailing wildcard paths are checked last

## Implicit Trailing Wildcard

When the `.documentRoot` property of the server is set, the server will automatically add a `/**` trailing wildcard route which will enable the serving of static content from the indicated directory. For example, setting the document root to `"/webroot"` would permit the server to deliver any files located within that directory.

## Further Information

For more information and examples of URL routing, see the [URL Routing](#) example application.

# HttpRequest

When handling a request, all client interaction is performed through `HttpRequest` and `HttpResponse` objects.

The `HttpRequest` object makes available all client headers, query parameters, POST body data, and other relevant information such as the client IP address and URL variables.

`HttpRequest` objects will handle parsing and decoding all "application/x-www-form-urlencoded", as well as "multipart/form-data" content type requests. It will make the data for any other content types available in a raw, unparsed form. When handling multipart form data, `HttpRequest` will automatically decode the data and create temporary files for any file uploads contained therein. These files will exist until the request ends after which they will be automatically deleted. In all of the sections below, the properties and functions are part of the `HttpRequest` protocol.

## Relevant Examples

- [Perfect-HttpRequestLogging](#)

## Metadata

HttpRequest provides several pieces of data which are not explicitly sent by the client. Information such as the client and server IP addresses, TCP ports, and document root fit into this category.

Client and server addresses are made available as tuples (a finite ordered list of elements) containing the IP addresses and respective ports of each:

```
/// The IP address and connecting port of the client.
var remoteAddress: (host: String, port: UInt16) { get }
/// The IP address and listening port for the server.
var serverAddress: (host: String, port: UInt16) { get }
```

When a server is created, you can set its canonical name, or CNAME. This can be useful in a variety of circumstances, such as when creating full links to your server. When a HttpRequest is created, the server will apply a CNAME to it. It is made available through the following property:

```
/// The canonical name for the server.
var serverName: String { get }
```

The server's document root is the directory from which static content is generally served. If you are not serving static content, then this document root may not exist. The document root is configured on the server before it begins accepting requests. Its value is transferred to the HttpRequest when it is created. When attempting to access static content such as a Mustache template, one would generally prefix all file paths with this document root value.

```
/// The server's document root from which static file content will generally be served.
var documentRoot: String { get }
```

## Request Line

An HttpRequest line consists of a method, path, query parameters, and an HTTP protocol identifier. An example HttpRequest line may appear as follows:

```
GET /path?q1=v1&q2=v2 HTTP/1.1
```

HttpRequest makes the parsed request line available through the properties below. The query parameters are presented as an array of name/value tuples in which all names and values have been URL decoded:

```
/// The HTTP request method.
var method: HTTPMethod { get set }
/// The request path.
var path: String { get set }
/// The parsed and decoded query/search arguments.
var queryParams: [(String, String)] { get }
/// The HTTP protocol version. For example (1, 0), (1, 1), (2, 0)
var protocolVersion: (Int, Int) { get }
```

During the routing process, the route URI may have consisted of URL variables, and these will have been parsed and made available as a dictionary:

```
/// Any URL variables acquired during routing the path to the request handler.
var urlVariables: [String:String] { get set }
```

An HttpRequest also makes the full request URI available. It will include the request path as well as any URL encoded query parameters:

```
/// Returns the full request URI.
var uri: String { get }
```

## Client Headers

Client request headers are made available either keyed by name or through an iterator permitting all header names and values to be accessed. HttpRequest will automatically parse and make available all HTTP cookie names and values. All possible request header names are represented in the enumeration type `HttpRequestHeader.Name`, which also includes a `.custom(name: String)` case for unaccounted header names.

It is possible to set client headers after the request has been read. This would be useful in, for example, HttpRequest filters as they may need to rewrite or add certain headers:

```

/// Returns the requested incoming header value.
func header(_ named: HTTPRequestHeader.Name) -> String?
/// Add a header to the response.
/// No check for duplicate or repeated headers will be made.
func addHeader(_ named: HTTPRequestHeader.Name, value: String)
/// Set the indicated header value.
/// If the header already exists then the existing value will be replaced.
func setHeader(_ named: HTTPRequestHeader.Name, value: String)
/// Provide access to all current header values.
var headers: AnyIterator<(HTTPRequestHeader.Name, String)> { get }

```

Cookies are made available through an array of name/value tuples.

```

/// Returns all the cookie name/value pairs parsed from the request.
var cookies: [(String, String)]

```

## GET and POST Parameters

For a detailed discussion of accessing GET and POST paramaters, see [Using Form Data](#).

### Body Data

For the content types "application/x-www-form-urlencoded" and "multipart/form-data", HTTPRequest will automatically parse and make the values available through the `postParams` or `postFileUploads` properties, respectively.

For a more detailed discussion of file upload handling, see [File Uploads](#). For more details on `postParams`, see [Using Form Data](#).

Request body data with other content types are not parsed and are made available either as raw bytes or as String data. For example, for a client submitting JSON data, one would want to access the body data as a String which would then be decoded into a useful value.

HTTPRequest makes body data available through the following properties:

```

/// POST body data as raw bytes.
/// If the POST content type is multipart/form-data then this will be nil.
var postBodyBytes: [UInt8]? { get set }
/// POST body data treated as UTF-8 bytes and decoded into a String, if possible.
/// If the POST content type is multipart/form-data then this will be nil.
var postBodyString: String? { get }

```

It's important to note that if the request has the "multipart/form-data" content type, then the `postBodyBytes` property will be nil. Otherwise, it will always contain the request body data regardless of the content type.

The `postBodyString` property will attempt to convert the body data from UTF-8 into a String. It will return nil if there is no body data or if the data could not successfully be converted from UTF-8.

In a REST application, there are several common HTTP "verbs" that are used. The most common of these are the "GET" and "POST" verbs.

The best-practice assignment of when to use each verb can vary between methodologies and is beyond the scope of this documentation.

An HTTP "GET" request only passes parameters in the URL:

```
http://www.example.com/page.html?message=Hello,%20World!
```

The "query parameters" in the above example are accessed using the `.queryParams` method:

```
let params = request.queryParams
```

While the above example only refers to a GET request, the `.queryParams` method applies to any HTTP request as they all can contain query parameters.

## POST Parameters

POST parameters, or params, are the standard method for passing complex data between browsers and other sources to APIs for creating or modifying content.

Perfect's HTTP libraries make it easy to access arrays of POST params or specific params.

To return all params (Query or POST) as a `[(String,String)]` array:

```
let params = request.params()
```

To return only POST params as a `[(String,String)]` array:

```
let params = request.postParams()
```

To return all params with a specific name such as multiple checkboxes, type:

```
let params = request.postParams(name: <String>)
```

This returns an array of strings: `[String]`

To return a specific parameter, as an optional `String?` :

```
let param = request.param(name: <String>)
```

When supplying a POST parameter in the `request` object is optional, it can be useful to specify a default value if one is not supplied. In this case, use the following syntax to return an optional `String?` :

```
let param = request.param(name: <String>, defaultValue: <string>)
```

## File Uploads

A special case of [using form data](#) is handling file uploads.

There are two main form encoding types:

- application/x-www-form-urlencoded (the default)
- multipart/form-data

When you wish to include file upload elements, you must choose multipart/form-data as your form's `enctype` (encoding) type.

All code used below can be seen in action as a complete example at <https://github.com/iamjono/perfect-file-uploads>.

An example HTML form containing the correct encoding and file input element might be represented like this:

```
<form
  method="POST"
  enctype="multipart/form-data"
  action="/upload">
  <input type="file" name="filetoupload">
  <br>
  <input type="submit">
</form>
```

## Receiving the File on the Server Side

Because the form is a POST method, we will handle the route with a `method: .post` :

```
var routes = Routes()
routes.add(
  method: .post,
  uri: "/upload",
  handler: handler)
server.addRoutes(routes)
```

Once the request has been offloaded to the `handler` we can:

```
// Grab the fileUploads array and see what's there
// If this POST was not multi-part, then this array will be empty

if let uploads = request.postFileUploads, uploads.count > 0 {
    // Create an array of dictionaries which will show what was uploaded
    var ary = [[String:Any]]()

    for upload in uploads {
        ary.append([
            "fieldName": upload.fieldName,
            "contentType": upload.contentType,
            "fileName": upload.fileName,
            "fileSize": upload.fileSize,
            "tmpFileName": upload.tmpFileName
        ])
    }
    values["files"] = ary
    values["count"] = ary.count
}
```

As demonstrated above, the file(s) uploaded are represented by the `request.postFileUploads` array, and the various properties such as `fileName`, `fileSize` and `tmpFileName` can be accessed from each array component.

**Note: The files uploaded are placed in a temporary directory. It is your responsibility to move them into the desired location.**

So let's create a directory to hold the uploaded files. This directory is outside of the webroot directory for security reasons:

```
// create uploads dir to store files
let fileDir = Dir(Dir.workingDir.path + "files")
do {
    try fileDir.create()
} catch {
    print(error)
}
```

Next, inside the `for upload in uploads` code block, we will create the action for the file to be moved:

```
// move file
let thisFile = File(upload.tmpFileName)
do {
    let _ = try thisFile.moveTo(path: fileDir.path + upload.fileName, overWrite: true)
} catch {
    print(error)
}
```

Now the uploaded files will move to the specified directory with the original filename restored.

For more information on file system manipulation, see the [Directory Operations](#) and [File Operations](#) chapters.

## HTTPResponse

When handling a request, all client interaction is performed through the provided `HTTPRequest` and `HTTPResponse` objects.

The `HTTPResponse` object contains all outgoing response data. It consists of the HTTP status code and message, the HTTP headers, and any response body data. `HTTPResponse` also contains the ability to stream or push chunks of response data to the client, and to complete or terminate the request.

In all of the sections below, unless otherwise noted, the properties and functions are part of the `HTTPResponse` protocol.

### Relevant Examples

- [Perfect-Cookie-Demo](#)
- [Perfect-HTTPRequestLogging](#)

### HTTP Status

The HTTP status indicates to the client whether or not the request was successful, if there was an error, or if it should take any other action. By default, the `HTTPResponse` object contains a 200 OK status. The status can be set to any other value if needed. HTTP status codes are represented by the `HTTPResponseStatus` enumeration (enum). This enum contains a case for each of the official status codes as well as a `.custom(code: Int, message: String)` case.

The response status is set with the following property:

```
/// The HTTP response status.
var status: HTTPResponseStatus { get set }
```

## Response Headers

Response headers can be retrieved, set, or iterated. Official/common header names are represented by the `HTTPHeader.Name` enum. This also contains a case for custom header names: `.custom(name: String)`.

```
/// Returns the requested outgoing header value.
func header(_ named: HTTPResponseHeader.Name) -> String?
/// Add a header to the outgoing response.
/// No check for duplicate or repeated headers will be made.
func addHeader(_ named: HTTPResponseHeader.Name, value: String)
/// Set the indicated header value.
/// If the header already exists then the existing value will be replaced.
func setHeader(_ named: HTTPResponseHeader.Name, value: String)
/// Provide access to all current header values.
var headers: AnyIterator<(HTTPResponseHeader.Name, String)> { get }
```

HTTPResponse provides higher level support for setting HTTP cookies. This is accomplished by creating a cookie object and adding it to the response object.

```
/// This bundles together the values which will be used to set a cookie in the outgoing response
public struct HTTPCookie {
    /// Cookie public initializer
    public init(name: String,
                value: String,
                domain: String?,
                expires: Expiration?,
                path: String?,
                secure: Bool?,
                httpOnly: Bool?)
}
```

Cookies are added to the HTTPResponse with the following function:

```
/// Add a cookie to the outgoing response.
func addCookie(_ cookie: HTTPCookie)
```

When a cookie is added it is properly formatted and added as a "Set-Cookie" header.

## Body Data

The response's current body data is exposed through the following property:

```
/// Body data waiting to be sent to the client.
/// This will be emptied after each chunk is sent.
var bodyBytes: [UInt8] { get set }
```

Data can be either directly added to this array, or can be added through one of the following convenience functions. These functions either set completely or append to the body bytes using either raw UInt8 bytes or String data. String data will be converted to UTF-8. The last function permits a [String:Any] dictionary to be converted into a JSON string.

```
/// Append data to the bodyBytes member.
func appendBody(bytes: [UInt8])
/// Append String data to the outgoing response.
/// All such data will be converted to a UTF-8 encoded [UInt8]
func appendBody(string: String)
/// Set the bodyBytes member, clearing out any existing data.
func setBody(bytes: [UInt8])
/// Set the String data of the outgoing response, clearing out any existing data.
/// All such data will be converted to a UTF-8 encoded [UInt8]
func setBody(string: String)
/// Encodes the Dictionary as a JSON string and converts that to a UTF-8 encoded [UInt8]
func setBody(json: [String:Any]) throws
```

When responding to a client, it is vital that a content-length header be included. When the HTTPResponse object begins sending the accumulated data to the client, it will check to see if a content length has been set. If it has not, the header will be set based on the count of bytes in the `bodyBytes` array.

The following function will push all current response headers and any body data:

```
/// Push all currently available headers and body data to the client.
/// May be called multiple times.
func push(callback: (Bool) -> ())
```

During most common requests, it is not necessary to call this method as the system will automatically flush all pending outgoing data when the request is completed. However, in some circumstances it may be desirable to have more direct control over this process.

For example, if one were to serve a very large file, it may not be practical to read the entire file content into memory and set the body bytes. Instead, one would set the response's content length to the size of the file and then, in chunks, read some of the file data, set the body bytes, and then call the `push` function repeatedly until all of the file has been sent. The `push` function will call the provided callback function parameter with a Boolean. This bool will be true if the content was successfully sent to the client. If the bool is false, then the request should be considered to have failed, and no further action should be taken.

## Streaming

In some cases, the content length of the response cannot be easily determined. For example, if one were streaming live video or audio content, then it may not be possible to set a content-length header. In such cases, set the `HTTPResponse` object into streaming mode. When using streaming mode, the response will be sent as HTTP-chunked encoding. When streaming, it is not required that the content length be set. Instead, add content to the body as usual, and call the `push` function. If the push succeeds, then the body data will be empty and ready for more data to be added. Continue calling `push` until the request has ended, or until push gives a `false` to the callback.

```
/// Indicate that the response should attempt to stream all outgoing data.
/// This is primarily used when the resulting content length can not be known.
var isStreaming: Bool { get set }
```

If streaming is to be used, it is required that `isStreaming` be set to true before any data is pushed to the client.

## Request Completion

**Important:** When a request has completed, it is **required** that the `HTTPResponse`'s `completed` function be called. This will ensure that all pending data is delivered to the client, and the underlying TCP connection will be either closed, or in the case of HTTP keep-alive, a new request will be read and processed.

```
/// Indicate that the request has completed.
/// Any currently available headers and body data will be pushed to the client.
/// No further request related activities should be performed after calling this.
func completed()
```

# Request and Response Filters

In addition to the regular request/response handler system, the Perfect server also provides a request and response filtering system. Any filters which are added to the server are called for each client request. When these filters run in turn, each are given a chance to modify either the request object before it is delivered to the handler, or the response object after the request has been marked as complete. Filters also have the option to terminate the current request.

Filters are added to the server along with a priority indicator. Priority levels can be either high, medium, or low. High-priority filters are executed before medium and low. Medium priorities are executed before any low-level filters.

Because filters are executed for every request, it is vital that they perform their tasks as quickly as possible so as to not hold up or delay request processing.

## Relevant Examples

- [Perfect-HttpRequestLogging](#)

## Request Filters

Request filters are called after the request has been fully read, but before the appropriate request handler has been located. This gives request filters an opportunity to modify the request before it is handled.

## Creating

Request filters must conform to the `HttpRequestFilter` protocol:

```

/// A filter which can be called to modify a HTTPRequest.
public protocol HTTPRequestFilter {
    /// Called once after the request has been read but before any handler is executed.
    func filter(request: HTTPRequest, response: HTTPResponse, callback: (HTTPRequestFilterResult) -> ())
}

```

When it comes time for the filter to run, its `filter` function will be called. The filter should perform any activities it needs, and then call the provided callback to indicate that it has completed its processing. The callback takes a value which indicates what the next step should be. This can indicate that the system should either continue with processing filters, stop processing request filters at the current priority level and proceed with delivering the request to a handler, or terminate the request entirely.

```

/// Result from one filter.
public enum HTTPRequestFilterResult {
    /// Continue with filtering.
    case `continue`(HTTPRequest, HTTPResponse)
    /// Halt and finalize the request. Handler is not run.
    case halt(HTTPRequest, HTTPResponse)
    /// Stop filtering and execute the request.
    /// No other filters at the current priority level will be executed.
    case execute(HTTPRequest, HTTPResponse)
}

```

Because the filter receives both the request and response objects and then delivers request and response objects in its `HTTPRequestFilterResult`, it's possible for a filter to entirely replace these objects if desired.

## Adding

Request filters are set directly on the server and given as an array of filter and priority tuples.

```

public class HTTPServer {
    public func setRequestFilters(_ request: [(HTTPRequestFilter, HTTPFilterPriority)]) -> HTTPServer
}

```

Calling this function sets the server's request filters. Each filter is provided along with its priority. The filters in the array parameter can be given in any order. The server will sort them appropriately, putting high-priority filters above those with lower priorities. Filters of equal priority will maintain the given order.

## Example

The following example is taken from a filter-related test case. It illustrates how to create and add filters, and shows how the filter priority levels interact.

```

var oneSet = false
var twoSet = false
var threeSet = false

struct Filter1: HTTPRequestFilter {
    func filter(request: HTTPRequest, response: HTTPResponse, callback: (HTTPRequestFilterResult) -> ()) {
        oneSet = true
        callback(.continue(request, response))
    }
}

struct Filter2: HTTPRequestFilter {
    func filter(request: HTTPRequest, response: HTTPResponse, callback: (HTTPRequestFilterResult) -> ()) {
        XCTAssert(oneSet)
        XCTAssert(!twoSet && !threeSet)
        twoSet = true
        callback(.execute(request, response))
    }
}

struct Filter3: HTTPRequestFilter {
    func filter(request: HTTPRequest, response: HTTPResponse, callback: (HTTPRequestFilterResult) -> ()) {
        XCTAssert(false, "This filter should be skipped")
        callback(.continue(request, response))
    }
}

struct Filter4: HTTPRequestFilter {
    func filter(request: HTTPRequest, response: HTTPResponse, callback: (HTTPRequestFilterResult) -> ()) {
        XCTAssert(oneSet && twoSet)
        XCTAssert(!threeSet)
        threeSet = true
        callback(.halt(request, response))
    }
}

var routes = Routes()
routes.add(method: .get, uri: "/", handler: {
    request, response in
    XCTAssert(false, "This handler should not execute")
    response.completed()
})

let requestFilters: [(HTTPRequestFilter, HTTPFilterPriority)] = [
    (Filter1(), HTTPFilterPriority.high),
    (Filter2(), HTTPFilterPriority.medium),
    (Filter3(), HTTPFilterPriority.medium),
    (Filter4(), HTTPFilterPriority.low)
]

let server = HTTPServer()
server.setRequestFilters(requestFilters)
server.serverPort = 8181
server.addRoutes(routes)
try server.start()

```

## Response Filters

Each response filter is executed once before response header data is sent to the client, and again for any subsequent chunk of body data. These filters can modify the outgoing response in any way they see fit, including adding or removing headers or rewriting body data.

## Creating

Response filters must conform to the `HTTPResponseFilter` protocol.

```

/// A filter which can be called to modify a HTTPResponse.
public protocol HTTPResponseFilter {
    /// Called once before headers are sent to the client.
    func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ())
    /// Called zero or more times for each bit of body data which is sent to the client.
    func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ())
}

```

When it comes time to send response headers, the `filterHeaders` function is called. This function should perform whatever tasks it needs on the provided `HTTPResponse` object, and then call the callback function. It should deliver unto the callback one of the `HTTPResponseFilterResult` values, which are defined as follows:

```
/// Response from one filter.
public enum HTTPResponseFilterResult {
    /// Continue with filtering.
    case `continue`
    /// Stop executing filters until the next push.
    case done
    /// Halt and close the request.
    case halt
}
```

These values indicate if the system should continue processing filters, stop executing filters until the next data push, or halt and terminate the request entirely.

When it comes time to send out one discrete chunk of data to the client, the filters' `filterBody` function is called. This function can inspect the outgoing data in the `HTTPResponse.bodyBytes` property, and potentially modify or replace the data. Since the headers have already been pushed out at this stage, any modifications to the header data will be ignored. Once a filter's body filtering has concluded, it should call the provided callback and deliver a `HTTPResponseFilterResult`. The meaning of these values is the same as for the `filterHeaders` function.

## Adding

Response filters are set directly on the server and given as an array of filter and priority tuples.

```
public class HTTPServer {
    public func setResponseFilters(_ response: [(HTTPResponseFilter, HTTPFilterPriority)]) -> HTTPServer
}
```

Calling this function sets the server's response filters. Each filter is provided along with its priority. The filters in the array parameter can be given in any order. The server will sort them appropriately, putting high-priority filters above those with lower priorities. Filters of equal priority will maintain the given order.

## Examples

The following example is taken from a filters test case. It illustrates how response filter priorities operate, and how response filters can modify outgoing headers and body data.

```

struct Filter1: HTTPResponseFilter {
    func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        response.setHeader(.custom(name: "X-Custom"), value: "Value")
        callback(.continue)
    }
    func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        callback(.continue)
    }
}
struct Filter2: HTTPResponseFilter {
    func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        callback(.continue)
    }
    func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        var b = response.bodyBytes
        b = b.map { $0 == 65 ? 97 : $0 }
        response.bodyBytes = b
        callback(.continue)
    }
}
struct Filter3: HTTPResponseFilter {
    func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        callback(.continue)
    }
    func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        var b = response.bodyBytes
        b = b.map { $0 == 66 ? 98 : $0 }
        response.bodyBytes = b
        callback(.done)
    }
}
struct Filter4: HTTPResponseFilter {
    func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        callback(.continue)
    }
    func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        XCTAssert(false, "This should not execute")
        callback(.done)
    }
}

var routes = Routes()
routes.add(method: .get, uri: "/", handler: {
    request, response in
    response.addHeader(.contentType, value: "text/plain")
    response.isStreaming = true
    response.setBody(string: "ABZ")
    response.push {
        _ in
        response.setBody(string: "ABZ")
        response.completed()
    }
})

let responseFilters: [(HTTPResponseFilter, HTTPFilterPriority)] = [
    (Filter1(), HTTPFilterPriority.high),
    (Filter2(), HTTPFilterPriority.medium),
    (Filter3(), HTTPFilterPriority.low),
    (Filter4(), HTTPFilterPriority.low)
]

let server = HTTPServer()
server.setResponseFilters(responseFilters)
server.serverPort = port
server.addRoutes(routes)
try server.start()

```

The example filters will add a X-Custom header and lowercase any A or B character in the body data. Note that the handler in this example sets the response to streaming mode, meaning that chunked encoding is used, and the body data is sent out in two discrete chunks.

## 404 Response Filter

A more useful example is posted below. This code will create and install a filter which monitors "404 not found" responses, and provides a custom message when it finds one.

```
struct Filter404: HTTPResponseFilter {
    func filterBody(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        callback(.continue)
    }

    func filterHeaders(response: HTTPResponse, callback: (HTTPResponseFilterResult) -> ()) {
        if case .notFound = response.status {
            response.setBody(string: "The file \(response.request.path) was not found.")
            response.setHeader(.contentType, value: "\(response.bodyBytes.count)")
            callback(.done)
        } else {
            callback(.continue)
        }
    }
}

let server = HTTPServer()
server.setResponseFilters([(Filter404(), .high)])
server.serverPort = 8181
try server.start()
```

## Web Redirects

The Perfect WebRedirects module will filter for specified routes (including trailing wildcard routes) and perform redirects as instructed if a match is found.

This can be important for maintaining SEO ranking in systems that have moved. For example, if moving from a static HTML site where `/about.html` is replaced with the new route `/about` and no valid redirect is in place, the site or system will lose SEO ranking.

A demo showing the usage, and working of the Perfect WebRedirects module can be found at [Perfect-WebRedirects-Demo](#).

## Including in your project

Import the dependency into your project by specifying it in your project's Package.swift file, or adding it via Perfect Assistant.

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-WebRedirects", majorVersion: 1),
```

Then in your `main.swift` file where you configure your web server, add it as an import, and add the filter:

```
import PerfectWebRedirects
```

Adding the filter:

```
// Add to the "filters" section of the config:
[
    "type": "request",
    "priority": "high",
    "name": WebRedirectsFilter.filterAPIRequest,
]
```

If you are also adding Request Logger filters, if the Web Redirects object is added second, directly after the RequestLogger filter, then both the original request (and its associated redirect code) and the new request will be logged correctly.

## Configuration file

The configuration for the routes is included in JSON files at `/config/redirect-rules/*.json` in the form:

```
{
  "/test/no": {
    "code": 302,
    "destination": "/test/yes"
  },

  "/test/no301": {
    "code": 301,
    "destination": "/test/yes"
  },

  "/test/wild/*": {
    "code": 302,
    "destination": "/test/wildyes"
  },

  "/test/wilder/*": {
    "code": 302,
    "destination": "/test/wilding*"
  }
}
```

Note that multiple JSON files can exist in this directory; all will be loaded the first time the filter is invoked.

The "key" is the matching route (the "old" file or route), and the "value" contains the HTTP code and new destination route to redirect to.  
# Sessions

Perfect includes session drivers for Redis, PostgreSQL, MySQL, SQLite3, CouchDB and MongoDB servers, as well as in-memory session storage for development purposes.

Session management is a foundational function for any web or application environment, and can provide linkage to authentication, transitional preference storage, and transactional data such as for a traditional shopping cart.

The general principle is that when a user visits a web site or system with a browser, the server assigns the user a "token" or "session id" and passes this value back to the client/browser in the form of a cookie or JSON data. This token is then included as either a cookie or Bearer Token with every subsequent request.

Sessions have an expiry time, usually in the form of an "idle timeout". This means that if a session has not been active for a set number of seconds, the session is considered expired and invalid.

The Perfect Sessions implementation stores the date and time each was created, when it was last "touched", and an idle time. On each access by the client/browser the session is "touched" and the idle time is reset. If the "last touched" plus "idle time" is less than the current date/time then the session has expired.

Each session has the following properties:

- **token** - the session id
- **userid** - an optionally stored user id string
- **created** - an integer representing the date/time created, in seconds
- **updated** - an integer representing the date/time last touched, in seconds
- **idle** - an integer representing the number of seconds the session can be idle before being considered expired
- **data** - a [String:Any] Array that is converted to JSON for storage. This is intended for storage of simple preference values.
- **ipaddress** - the IP Address (v4 or v6) that the session was first used on. Used for optional session verification.
- **useragent** - the User Agent string that the session was first used with. Used for optional session verification.
- **CSRF** - the [CSRF \(Cross Site Request Forgery\)](#) security configuration.
- **CORS** - the [CORS \(Cross Origin Resource Sharing\)](#) security configuration.

## Examples

Each of the modules has an associated example/demo. Much of the functionality described in this document can be observed in each of these examples.

- [In-Memory Sessions](#)
- [Redis Sessions](#)
- [PostgreSQL Sessions](#)
- [MySQL Sessions](#)
- [SQLite Sessions](#)
- [CouchDB Sessions](#)
- [MongoDB Sessions](#)

## Installation

---

If using the in-memory driver, import the base module by including the following in your project's Package.swift:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session.git", majorVersion: 1)
```

### Database-Specific Drivers

Redis:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-Redis.git", majorVersion: 1)
```

PostgreSQL:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-PostgreSQL.git", majorVersion: 1)
```

MySQL:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-MySQL.git", majorVersion: 1)
```

SQLite3:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-SQLite.git", majorVersion: 1)
```

CouchDB:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-CouchDB.git", majorVersion: 1)
```

MongoDB:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-MongoDB.git", majorVersion: 1)
```

## Configuration

---

The struct `SessionConfig` contains settings that can be customized to your own preference:

```
// The name of the session.
// This will also be the name of the cookie set in a browser.
SessionConfig.name = "PerfectSession"

// The "Idle" time for the session, in seconds.
// 86400 is one day.
SessionConfig.idle = 86400

// Optional cookie domain setting
SessionConfig.cookieDomain = "localhost"

// Optional setting to lock session to the initiating IP address. Default is false
SessionConfig.IPAddressLock = true

// Optional setting to lock session to the initiating user agent string. Default is false
SessionConfig.userAgentLock = true

// The interval at which stale sessions are purged from the database
SessionConfig.purgeInterval = 3600 // in seconds. Default is 1 hour.

// CouchDB-Specific
// The CouchDB database used to store sessions
SessionConfig.couchDatabase = "sessions"

// MongoDB-Specific
// The MongoDB collection used to store sessions
SessionConfig.mongoCollection = "sessions"
```

If you wish to change the SessionConfig values, you must set these before the Session Driver is defined.

The [CSRF \(Cross Site Request Forgery\)](#) security configuration and [CORS \(Cross Origin Resource Sharing\)](#) security configuration are discussed separately.

Note that for the Redis session driver there is no "timed event" via the `SessionConfig.purgeInterval` as the mechanism for expiry is handled directly via the `Expires` value added at session creation or update.

## IP Address and User Agent Locks

If the `SessionConfig.IPAddressLock` or `SessionConfig.userAgentLock` settings are true, then the session will be forcibly ended if the incoming information does not match that which was sent when the session was initiated.

This is a security measure to assist in preventing man-in-the-middle / session hijacking attacks.

Be aware that if a user has logged on through a WiFi network and transitions to a mobile or wired connection while the `SessionConfig.IPAddressLock` setting has been set to `true`, the user will be logged out of their session.

## Defining the Session Driver

Each Session Driver has its own implementation which is optimized for the storage option. Therefore you must set the session driver and HTTP filters before executing "server.start()".

In main.swift, after any SessionConfig changes, set the following:

```
// Instantiate the HTTPServer
let server = HTTPServer()

// Define the Session Driver
let sessionDriver = SessionMemoryDriver()

// Add the filters so the pre- and post- route actions are executed.
server.setRequestFilters([sessionDriver.requestFilter])
server.setResponseFilters([sessionDriver.responseFilter])
```

In Perfect, filters are analogous in some ways to the concept of "middleware" in other frameworks. The "request" filter will intercept the incoming request and extract the session token, attempt to load the session from storage, and make the session data available to the application. The response will be executed immediately before the response is returned to the client/browser and saves the session, and re-sends the cookie to the client/browser.

See "Database-Specific Options" below for storage-appropriate settings and Driver syntax.

## Accessing the Session Data

The `token`, `userid` and `data` properties of the session are exposed in the "request" object which is passed into all handlers. The `userid` and `data` properties can be read and written to during the scope of the handler, and automatically saved to storage in the response filter.

A sample handler can be seen in the example systems.

```
// Defining an "Index" handler
open static func indexHandlerGet(request: HTTPRequest, _ response: HTTPResponse) {
    // Random generator from TurnstileCrypto
    let rand = URandom()

    // Adding some random data to the session for demo purposes
    request.session.data[rand.secureToken] = rand.secureToken

    // For demo purposes, dumping all current session data as a JSON string.
    let dump = try? request.session.data.jsonEncodedString()

    // Some simple HTML that displays the session token/id, and the current data as JSON.
    let body = "<p>Your Session ID is: <code>\(request.session.token)</code></p><p>Session data: <code>\(dump)</code></p>"

    // Send the response back.
    response.setBody(string: body)
    response.completed()
}
```

To read the current session id:

```
request.session.token
```

To access the userid:

```
// read:
request.session.userid

// write:
request.session.userid = "MyString"
```

To access the data stored in the session:

```
// read:
request.session.data

// write:
request.session.data["keyString"] = "Value"
request.session.data["keyInteger"] = 1
request.session.data["keyBool"] = true

// reading a specific value
if let val = request.session.data["keyString"] as? String {
    let keyString = val
}
```

## Database-Specific options

### Redis

Importing the module, in Package.swift:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-Redis.git", majorVersion: 1)
```

Defining the connection to the PostgreSQL server:

```
RedisSessionConnector.host = "localhost"
RedisSessionConnector.port = 5432
RedisSessionConnector.password = "secret"
```

Defining the Session Driver:

```
let sessionDriver = SessionRedisDriver()
```

### PostgreSQL

Importing the module, in Package.swift:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-PostgreSQL.git", majorVersion: 1)
```

Defining the connection to the PostgreSQL server:

```
PostgresSessionConnector.host = "localhost"
PostgresSessionConnector.port = 5432
PostgresSessionConnector.username = "username"
PostgresSessionConnector.password = "secret"
PostgresSessionConnector.database = "mydatabase"
PostgresSessionConnector.table = "sessions"
```

Defining the Session Driver:

```
let sessionDriver = SessionPostgresDriver()
```

### MySQL

Importing the module, in Package.swift:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-MySQL.git", majorVersion: 1)
```

Defining the connection to the MySQL server:

```
MySQLSessionConnector.host = "localhost"
MySQLSessionConnector.port = 3306
MySQLSessionConnector.username = "username"
MySQLSessionConnector.password = "secret"
MySQLSessionConnector.database = "mydatabase"
MySQLSessionConnector.table = "sessions"
```

Defining the Session Driver:

```
let sessionDriver = SessionMySQLDriver()
```

## SQLite

Importing the module, in Package.swift:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-SQLite.git", majorVersion: 1)
```

Defining the connection to the SQLite server:

```
SQLiteConnector.db = "./SessionDB"
```

Defining the Session Driver:

```
let sessionDriver = SessionSQLiteDriver()
```

## CouchDB

Importing the module, in Package.swift:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-CouchDB.git", majorVersion: 1)
```

Defining the CouchDB database to use for session storage:

```
SessionConfig.couchDatabase = "perfectsessions"
```

Defining the connection to the CouchDB server:

```
CouchDBConnection.host = "localhost"
CouchDBConnection.username = "username"
CouchDBConnection.password = "secret"
```

Defining the Session Driver:

```
let sessionDriver = SessionCouchDBDriver()
```

## MongoDB

Importing the module, in Package.swift:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Session-MongoDB.git", majorVersion: 1)
```

Defining the MongoDB database to use for session storage:

```
SessionConfig.mongoCollection = "perfectsessions"
```

Defining the connection to the MongoDB server:

```
MongoDBConnection.host = "localhost"
MongoDBConnection.database = "perfect_testing"
```

Defining the Session Driver:

```
let sessionDriver = SessionMongoDBDriver()
```

# CSRF (Cross Site Request Forgery) Security

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. **CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.** With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. [1] - (OWASP)

CSRF as an attack vector is often overlooked, and represents a significant "chaos" factor unless the validation is handled at the highest level: the framework. This allows web application and API authors to have significant control over a vital layer of security.

The [Perfect Sessions](#) module includes support for CSRF configuration.

If you have included Perfect Sessions or any of its datasource-specific implementations in your Packages.swift file, you already have CSRF support.

## Relevant Examples

- [Perfect-Session-Memory-Demo](#)

## Configuration

An example CSRF Configuration might look like this:

```
SessionConfig.CSRF.checkState = true
SessionConfig.CSRF.failAction = .fail
SessionConfig.CSRF.checkHeaders = true
SessionConfig.CSRF.acceptableHostnames.append("http://www.example.com")
SessionConfig.CSRF.requireToken = true
```

### SessionConfig.CSRF.checkState

This is the "master switch". If enabled, CSRF will be enabled for all routes.

### SessionConfig.CSRF.failAction

This specifies the action to take if the CSRF validation fails. The possible options are:

- `.fail` - Execute an immediate halt. No further processing will be done, and an HTTP Status `406 Not Acceptable` is generated.
- `.log` - Processing will continue, however the event will be recorded in the log.
- `.none` - Processing will continue, no action is taken.

### SessionConfig.CSRF.acceptableHostnames

An array of host names that are compared in the following section for "origin" match acceptance.

### SessionConfig.CSRF.checkHeaders

If the `CORS.checkheader` is configured as `true`, origin and host headers are checked for validity.

- The `Origin`, `Referrer` or `X-Forwarded-For` headers must be populated ("origin").
- If the "origin" is specified in `SessionConfig.CSRF.acceptableHostnames`, the CSRF check will continue to the next phase and the following checks are skipped.
- The `Host` or `X-Forwarded-Host` header must be present ("host").
- The "host" and "origin" values must match exactly.

### SessionConfig.CSRF.requireToken

When set to true, this setting will enforce all POST requests to include a `"_csrf"` param, or if the content type header is "application/json" then an associated "X-CSRF-Token" header must be sent with the request. The content of the header or parameter should match the `request.session.data["csrf"]` value. This value is set automatically at session start.

### Session state

While not a configuration param, it is worth noting that if `SessionConfig.CSRF.checkState` is true, no POST request will be accepted if the session is "new". This is a deliberate position supported by security recommendations.

[1] - OWASP, Cross-Site Request Forgery (CSRF): [https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))## CORS (Cross Origin Resource Sharing) Security

Cross-Origin Resource Sharing (CORS) is an important part of the "open web", and as such, no framework is complete without enabling support for CORS.

Monsur Hossain from [html5rocks.com](#) introduces CORS very effectively:

APIs are the threads that let you stitch together a rich web experience. But this experience has a hard time translating to the browser, where the options for cross-domain requests are limited to techniques like JSON-P (which has limited use due to security concerns) or setting up a custom proxy (which can be a pain to set up and maintain).

Cross-Origin Resource Sharing (CORS) is a W3C spec that allows cross-domain communication from the browser. By building on top of the XMLHttpRequest object, CORS allows developers to work with the same idioms as same-domain requests.

The use-case for CORS is simple. Imagine the site `alice.com` has some data that the site `bob.com` wants to access. This type of request traditionally wouldn't be allowed under the browser's same origin policy. However, by supporting CORS requests, `alice.com` can add a few special response headers that allows `bob.com` to access the data.

As you can see from this example, CORS support requires coordination between both the server and client. Luckily, if you are a client-side developer you are shielded from most of these details. The rest of this article shows how clients can make cross-origin requests, and how servers can configure themselves to support CORS.

The [Perfect Sessions](#) module includes support for CORS configuration, enabling your API and assets to be available or secured in the way you wish.

If you have included Perfect Sessions or any of its datasource-specific implementations in your `Packages.swift` file, you already have CORS support; however, it is **off** by default.

## Relevant Examples

- [Perfect-Session-Memory-Demo](#)

## Configuration

```
// Enabled, true or false.
// Default is false.
SessionConfig.CORS.enabled = true

// Array of acceptable hostnames for incoming requests
// To enable CORS on all, have a single entry, *
SessionConfig.CORS.acceptableHostnames = ["*"]

// However if you wish to enable specific domains:
SessionConfig.CORS.acceptableHostnames.append("http://www.test-cors.org")

// Wildcards can also be used at the start or end of hosts
SessionConfig.CORS.acceptableHostnames.append("*.example.com")
SessionConfig.CORS.acceptableHostnames.append("http://www.domain.*")

// Array of acceptable methods
public var methods: [HTTPMethod] = [.get, .post, .put]

// An array of custom headers allowed
public var customHeaders = [String]()

// Access-Control-Allow-Credentials true/false.
// Standard CORS requests do not send or set any cookies by default.
// In order to include cookies as part of the request enable the client to do so by setting to true
public var withCredentials = false

// Max Age (seconds) of request / OPTION caching.
// Set to 0 for no caching (default)
public var maxAge = 3600
```

When a CORS request is submitted to the server, if there is no match then the CORS specific headers are not generated in the OPTIONS response, which will instruct the browser that it cannot accept the resource.

If the server determines that CORS headers should be generated, the following headers are sent with the response:

```
// An array of allowable HTTP Methods.
// In the case of the above configuration example:
Access-Control-Allow-Methods: GET, POST, PUT

// If the origin is acceptable the origin will be echoed back to the requester
// (even if configured with *)
Access-Control-Allow-Origin: http://www.test-cors.org

// If the server wishes cookies to be sent along with requests,
// this should return true
Access-Control-Allow-Credentials: true

// The length of time the OPTIONS request can be cached for.
Access-Control-Max-Age: 3600
```

An excellent resource for testing CORS entitlements and responses is available at <http://www.test-cors.org># Perfect Local Authentication

This package provides Local Authentication libraries for projects that require locally stored and handled authentication.

## Relevant Templates & Examples

A template application can be found at <https://github.com/PerfectlySoft/Perfect-Local-Auth-PostgreSQL-Template>, providing a fully functional starting point, as well as demonstrating the usage of the system.

## Adding to your project

Add this project as a dependency in your Package.swift file.

PostgreSQL Driver:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-LocalAuthentication-PostgreSQL.git", majorVersion: 1)
```

MySQL Driver:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-LocalAuthentication-MySQL.git", majorVersion: 1)
```

## Configuration

It is important to configure the following in main.swift to set up database and session configuration:

Import the required modules:

```
import PerfectSession
import PerfectSessionPostgreSQL // Or PerfectSessionMySQL
import PerfectCrypto
import LocalAuthentication
```

Initialize PerfectCrypto:

```
let _ = PerfectCrypto.isInitialized
```

Now set some defaults:

```
// Used in email communications
// The Base link to your system, such as http://www.example.com/
var baseURL = ""

// Configuration of Session
SessionConfig.name = "perfectSession" // <-- change
SessionConfig.idle = 86400
SessionConfig.cookieDomain = "localhost" //<-- change
SessionConfig.IPAddressLock = false
SessionConfig.userAgentLock = false
SessionConfig.CSRF.checkState = true
SessionConfig.CORS.enabled = true
SessionConfig.cookieSameSite = .lax
```

Detailed Session configuration documentation can be found at <https://www.perfect.org/docs/sessions.html>

The database and email configurations should be set as follows (if using JSON file config):

```
let opts = initializeSchema("./config/ApplicationConfiguration.json") // <-- loads base config like db and email configuration
httpPort = opts["httpPort"] as? Int ?? httpPort
baseURL = opts["baseURL"] as? String ?? baseURL
```

Otherwise, these will need to be set equivalent to one of these functions:

- PostgreSQL: <https://github.com/PerfectlySoft/Perfect-LocalAuthentication-PostgreSQL/blob/master/Sources/LocalAuthentication/Schema/InitializeSchema.swift>
- MySQL: <https://github.com/PerfectlySoft/Perfect-LocalAuthentication-MySQL/blob/master/Sources/LocalAuthentication/Schema/InitializeSchema.swift>

## Set the session driver

PostgreSQL:

```
let sessionDriver = SessionPostgresDriver()
```

MySQL:

```
let sessionDriver = SessionMySQLDriver()
```

## Request & Response Filters

The following two session filters need to be added to your server config:

PostgreSQL Driver:

```
// (where filter is a [[String: Any]] object)
filters.append(["type": "request", "priority": "high", "name": SessionPostgresFilter.filterAPIRequest])
filters.append(["type": "response", "priority": "high", "name": SessionPostgresFilter.filterAPIResponse])
```

MySQL Driver:

```
// (where filter is a [[String: Any]] object)
filters.append(["type": "request", "priority": "high", "name": SessionMySQLFilter.filterAPIRequest])
filters.append(["type": "response", "priority": "high", "name": SessionMySQLFilter.filterAPIResponse])
```

For example, see <https://github.com/PerfectlySoft/Perfect-Local-Auth-PostgreSQL-Template/blob/master/Sources/PerfectLocalAuthPostgreSQLTemplate/configuration/Filters.swift>

## Add routes for login, register etc

The following routes can be added as needed or customized to add login, logout, register:

```
// Login
routes.append(["method": "get", "uri": "/login", "handler": Handlers.login]) // simply a serving of the login GET
routes.append(["method": "post", "uri": "/login", "handler": LocalAuthWebHandlers.login])
routes.append(["method": "get", "uri": "/logout", "handler": LocalAuthWebHandlers.logout])

// Register
routes.append(["method": "get", "uri": "/register", "handler": LocalAuthWebHandlers.register])
routes.append(["method": "post", "uri": "/register", "handler": LocalAuthWebHandlers.registerPost])
routes.append(["method": "get", "uri": "/verifyAccount/{passvalidation}", "handler": LocalAuthWebHandlers.registerVerify])
routes.append(["method": "post", "uri": "/registrationCompletion", "handler": LocalAuthWebHandlers.registerCompletion])

// JSON
routes.append(["method": "get", "uri": "/api/v1/session", "handler": LocalAuthJSONHandlers.session])
routes.append(["method": "get", "uri": "/api/v1/logout", "handler": LocalAuthJSONHandlers.logout])
routes.append(["method": "post", "uri": "/api/v1/register", "handler": LocalAuthJSONHandlers.register])
routes.append(["method": "login", "uri": "/api/v1/login", "handler": LocalAuthJSONHandlers.login])
```

An example can be found at <https://github.com/PerfectlySoft/Perfect-Local-Auth-PostgreSQL-Template/blob/master/Sources/PerfectLocalAuthPostgreSQLTemplate/configuration/Routes.swift>

## Testing for authentication:

The user id can be accessed as follows:

```
request.session?.userid ?? ""
```

If a user id (i.e. logged in state) is required to access a page, code such as this could be used to detect and redirect:

```
let contextAuthenticated = !(request.session?.userid ?? "").isEmpty
if !contextAuthenticated { response.redirect(path: "/login") }
```# Perfect-SPNEGO [简体中文](README.zh_CN.md)
```

This project provides a general server library which provides SPNEGO mechanism.

### Before Start

Perfect SPNEGO is aiming on a general application of Server Side Swift, so it could be plugged into *\*ANY\** servers, such as HTTP / FTP / SSH, etc .

Although it supports Perfect HTTP server natively, it could be applied to any other Swift servers as well.

Before attaching to any actual server applications, please make sure your server has been already configured with Kerberos V5.

### Xcode Build Note

If you would like to use Xcode to build this project, please make sure to pass proper linker flags to the Swift Package Manager:

```
$ swift package -Xlinker -framework -Xlinker GSS generate-xcodeproj ``
```

## Linux Build Note

A special library called libkrb5-dev is required to build this project:

```
$ sudo apt-get install libkrb5-dev
```

If your server is a KDC, then you can skip this step, otherwise please install Kerberos V5 utilities:

```
$ sudo apt-get install krb5-user
```

## KDC Configuration

Configure the application server's /etc/krb5.conf to your KDC. The following sample configuration shows how to connect your application server to realm `KRB5.CA` under control of a KDC named `nut.krb5.ca` :

```
[realms]
KRB5.CA = {
    kdc = nut.krb5.ca
    admin_server = nut.krb5.ca
}
[domain_realm]
.krb5.ca = KRB5.CA
krb5.ca = KRB5.CA
```

## Prepare Kerberos Keys for Server

Contact to your KDC administrator to assign a `keytab` file to your application server.

Take example, *SUPPOSE ALL HOSTS BELOW REGISTERED ON THE SAME DNS SERVER*:

- KDC server: nut.krb5.ca
- Application server: coco.krb5.ca
- Application server type: HTTP

In such a case, KDC administrator shall login on `nut.krb5.ca` then perform following operation:

```
kadmin.local: addprinc -randkey HTTP/coco.krb5.ca@KRB5.CA
kadmin.local: ktadd -k /tmp/krb5.keytab HTTP/coco.krb5.ca@KRB5.CA
```

Then please ship this krb5.keytab file securely and install on your application server `coco.krb5.ca` and move to folder `/etc` , then grant sufficient permissions to your swift

application to access it.

## Quick Start

---

Add the following dependency to your project's Package.swift file:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-SPNEGO.git", majorVersion: 1)
```

Then import Perfect-SPNEGO to your source code:

```
import PerfectSPNEGO
```

## Connect to KDC

Use the key in your default keytab `/etc/krb5.keytab` file on application server to register your application server to the KDC.

```
let spnego = try Spnego("HTTP@coco.krb5.ca")
```

Please note that the host name and protocol type *MUST* match the records listed in the keytab file.

## Respond to A Spnego Challenge

Once initialized, object `spnego` could respond to the challenges. Take example, if a user is trying to connect to the application server as:

```
$ kinit rocky@KRB5.CA
$ curl --negotiate -u : http://coco.krb5.ca
```

In this case, the curl command would possibly send a base64 encoded challenge in the HTTP header:

```
> Authorization: Negotiate YIICnQYGKwYBBQUCoIIckTCCAo2gJzAlBgkqhkiG9xIBAgI ...
```

Once received such a challenge, you could apply this base64 string to the `spnego` object:

```
let (user, response) = try spnego.accept(base64Token: "YIICnQYGKwYBBQUCoIIckTCCAo2gJzAlBgkqhkiG9xIBAgI...")
```

If succeeded, the user would be "rocky@KRB5.CA". The variable `response` might be `nil` which indicates nothing is required to reply such a token, otherwise you should send this `response` back to the client.

Up till now, your application had already got the user information and request, then the application server might decide if this user could access the objective resource or not, according to your ACL (access control list) configuration.

## Relevant Examples

---

A good example to demonstrate how to use SPNEGO in a Perfect HTTP Server can be found from: \* [Perfect-Spnego-Demo](#)# JSON Converter

Perfect includes basic JSON encoding and decoding functionality. JSON encoding is provided through a series of extensions on many of the built-in Swift data types. Decoding is provided through an extension on the Swift String type.

It seems important to note that although Perfect provides this particular JSON encoding/decoding system, it is not required that your application uses it. Feel free to import your own favourite JSON-related functionality.

To use this system, first ensure that PerfectLib is imported:

```
import PerfectLib
```

## Encoding To JSON Data

You can convert any of the following types directly into JSON string data:

- String
- Int
- UInt
- Double
- Bool
- Array

- Dictionary
- Optional
- Custom classes which inherit from JSONConvertibleObject

Note that only Optionals which contain any of the above types are directly convertible. Optionals which are nil will output as a JSON "null".

To encode any of these values, call the `jsonEncodedString()` function which is provided as an extension on the objects. This function may potentially throw a `JSONConversionError.notConvertible` error.

Example:

```
let scoreArray: [String:Any] = ["1st Place": 300, "2nd Place": 230.45, "3rd Place": 150]
let encoded = try scoreArray.jsonEncodedString()
```

The result of the encoding would be the following String:

```
{"2nd Place":230.45,"1st Place":300,"3rd Place":150}
```

## Decoding JSON Data

String objects which contain JSON string data can be decoded by using the `jsonDecode()` function. This function can throw a `JSONConversionError.syntaxError` error if the String does not contain valid JSON data.

```
let encoded = "{\"2nd Place\":230.45,\"1st Place\":300,\"3rd Place\":150}"
let decoded = try encoded.jsonDecode() as? [String:Any]
```

Decoding the String will produce the following dictionary:

```
["2nd Place": 230.44999999999999, "1st Place": 300, "3rd Place": 150]
```

Though decoding a JSON string can produce any of the permitted values, it is most common to deal with JSON objects (dictionaries) or arrays. You will need to cast the resulting value to the expected type.

## Using the Decoded Data

Because decoded dictionaries or arrays are always of type `[String:Any]` or `[Any]`, respectively, you will need to cast the contained values to usable types. For example:

```
var firstPlace = 0
var secondPlace = 0.0
var thirdPlace = 0

let encoded = "{\"2nd Place\":230.45,\"1st Place\":300,\"3rd Place\":150}"
guard let decoded = try encoded.jsonDecode() as? [String:Any] else {
    return
}

for (key, value) in decoded {
    switch key {
    case "1st Place":
        firstPlace = value as! Int
    case "2nd Place":
        secondPlace = value as! Double
    case "3rd Place":
        thirdPlace = value as! Int
    default:
        break
    }
}

print("The top scores are: \r" + "First Place: " + "\(firstPlace)" + " Points\r" + "Second Place: " + "\(secondPlace)" + " Points\r" + "Third Place: " + "\(thirdPlace)" + " Points")
```

The output would be the following:

```
The top scores are:
First Place: 300 Points
Second Place: 230.45 Points
Third Place: 150 Points
```

## Decoding Empty Values from JSON Data

As JSON null values are untyped, the system will substitute a `JSONConvertibleNull` in place of all JSON nulls. Example:

```
let jsonString = "{\"1st Place\":300,\"4th place\":null,\"2nd Place\":230.45,\"3rd Place\":150}"

if let decoded = try jsonString.jsonDecode() as? [String:Any] {
    for (key, value) in decoded {
        if let value as? JSONConvertibleNull {
            print("The key \"\(key)\" had a null value")
        }
    }
}
```

The output would be:

```
The key "4th place" had a null value
```

## JSON Convertible Object

Perfect's JSON system provides the facilities for encoding and decoding custom classes. Any eligible class must inherit from the `JSONConvertibleObject` base class which is defined as follows:

```
/// Base for a custom object which can be converted to and from JSON.
public class JSONConvertibleObject: JSONConvertible {
    /// Default initializer.
    public init() {}
    /// Get the JSON keys/value.
    public func setJSONValues(_ values:[String:Any]) {}
    /// Set the object properties based on the JSON keys/values.
    public func getJSONValues() -> [String:Any] { return [String:Any]() }
    /// Encode the object into JSON text
    public func jsonEncodedString() throws -> String {
        return try self.getJSONValues().jsonEncodedString()
    }
}
```

Any object wishing to be JSON encoded/decoded must first register itself with the system. This registration should take place once when your application starts up. Call the `JSONDecoding.registerJSONDecodable` function to register your object. This function is defined as follows:

```
public class JSONDecoding {
    /// Function which returns a new instance of a custom object which will have its members set based on the JSON data.
    public typealias JSONConvertibleObjectCreator = () -> JSONConvertibleObject
    static public func registerJSONDecodable(name: String, creator: JSONConvertibleObjectCreator)
}
```

Registering an object requires a unique name which can be any string, provided it is unique. It also requires a "creator" function which returns a new instance of the object in question.

When the system encodes a `JSONConvertibleObject` it calls the object's `getJSONValues` function. This function should return a `[String:Any]` dictionary containing the names and values for any properties which should be encoded into the resulting JSON string. This dictionary **must** also contain a value identifying the object type. The value must match the name by which the object was originally registered. The dictionary key for the value is identified by the `JSONDecoding.objectIdentifierKey` property.

When the system decodes such an object, it will find the `JSONDecoding.objectIdentifierKey` value and look up the object creator which had been previously registered. It will create a new instance of the type by calling that function, and will then call the new object's `setJSONValues(_ values:[String:Any])` function. It will pass in a dictionary containing all of the deconverted values. These values will match those previously returned by the `getJSONValues` function when the object was first converted. Within the `setJSONValues` function, the object should retrieve all properties which it wants to reinstate.

The following example defines a custom `JSONConvertibleObject` and converts it to a JSON string. It then decodes the object and compares it to the original. Note that this example object calls the convenience function `getJSONValue`, which will pull a named value from the dictionary and permits providing a default value which will be returned if the dictionary does not contain the indicated key.

This example is split up into several sections.

Define the class:

```
class User: JSONConvertibleObject {
    static let registerName = "user"
    var firstName = ""
    var lastName = ""
    var age = 0
    override func setJSONValues(_ values: [String : Any]) {
        self.firstName = getJSONValue(named: "firstName", from: values, defaultValue: "")
        self.lastName = getJSONValue(named: "lastName", from: values, defaultValue: "")
        self.age = getJSONValue(named: "age", from: values, defaultValue: 0)
    }
    override func getJSONValues() -> [String : Any] {
        return [
            JSONDecoding.objectIdentifierKey: User.registerName,
            "firstName": firstName,
            "lastName": lastName,
            "age": age
        ]
    }
}
```

Register the class:

```
// do this once
JSONDecoding.registerJSONDecodable(name: User.registerName, creator: { return User() })
```

Encode the object:

```
let user = User()
user.firstName = "Donnie"
user.lastName = "Darko"
user.age = 17

let encoded = try user.jsonEncodedString()
```

The value of "encoded" will look as follows:

```
{"lastName": "Darko", "age": 17, "_jsonobjid": "user", "firstName": "Donnie"}
```

Decode the object:

```
guard let user2 = try encoded.jsonDecode() as? User else {
    return // error
}

// check the values
XCTAssert(user.firstName == user2.firstName)
XCTAssert(user.lastName == user2.lastName)
XCTAssert(user.age == user2.age)
```

Note that the `JSONDecoding.objectIdentifierKey` (the value of which is `"_jsonobjid"`) key/value pair that identifies the Swift object to be decoded must be in the incoming JSON string. If it is not present then the result of `jsonDecode` will be a regular Dictionary instead of the intended Swift object.

## JSON Conversion Error

As an object is converted to or from a JSON string, the process may throw a `JSONConversionError`. This is defined as follows:

```
/// An error occurring during JSON conversion.
public enum JSONConversionError: ErrorProtocol {
    /// The object did not support JSON conversion.
    case notConvertible(Any)
    /// A provided key was not a String.
    case invalidKey(Any)
    /// The JSON text contained a syntax error.
    case syntaxError
}
```

## Static File Content

As seen in the [Routing](#) chapter, Perfect is capable of complex dynamic routing. It is also capable of serving static content including HTML, images, CSS, and JavaScript.

Static file content serving is handled through the `StaticFileHandler` object. Once a request is given to an instance of this object, it will handle finding the indicated file or returning a 404 if it does not exist. `StaticFileHandler` also handles caching through use of the ETag header as well as byte range serving for very large files.

A `StaticFileHandler` object is initialized with a document root path parameter. This root path forms the prefix to which all file paths will be appended. The current HTTPRequest's `path` property will be used to indicate the path to the file which should be read and returned.

`StaticFileHandler` can be directly used by creating one in your web handler and calling its `handleRequest` method.

For example, a handler which simply returns the request file might look as follows:

```
{
  request, response in
  StaticFileHandler(documentRoot: request.documentRoot).handleRequest(request: request, response: response)
}
```

However, unless custom behaviour is required, it is not necessary to handle this manually. Setting the server's `documentRoot` property will automatically install a handler which will serve any files from the indicated directory. Setting the server's document root is analogous to the following snippet:

```
let dir = Dir(documentRoot)
if !dir.exists {
  try Dir(documentRoot).create()
}
routes.add(method: .get, uri: "**", handler: {
  request, response in
  StaticFileHandler(documentRoot: request.documentRoot).handleRequest(request: request, response: response)
})
```

The handler that gets installed will serve any files from the root or any sub-directories contained therein.

An example of the `documentRoot` property usage is found in the `PerfectTemplate`:

```

import PerfectLib
import PerfectHTTP
import PerfectHTTPServer

// Create HTTP server.
let server = HTTPServer()

// Register your own routes and handlers
var routes = Routes()
routes.add(method: .get, uri: "/", handler: {
    request, response in
        response.appendBody(string: "<html>...</html>")
        response.completed()
    }
)

// Add the routes to the server.
server.addRoutes(routes)

// Set a listen port of 8181
server.serverPort = 8181

// Set a document root.
// This is optional. If you do not want to serve
// static content then do not set this.
// Setting the document root will automatically add a
// static file handler for the route
server.documentRoot = "./webroot"

// Gather command line options and further configure the server.
// Run the server with --help to see the list of supported arguments.
// Command line arguments will supplant any of the values set above.
configureServer(server)

do {
    // Launch the HTTP server.
    try server.start()
} catch PerfectError.networkError(let err, let msg) {
    print("Network error thrown: \(err) \(msg)")
}

```

Note the `server.documentRoot = "./webroot"` line. It means that if there is a `styles.css` document in the specified webroot directory, then a request to the URI `/styles.css` will return that file to the browser.

The following example establishes a virtual documents path, serving all URIs which begin with `/files` from the physical directory `/var/www/htdocs`:

```

routes.add(method: .get, uri: "/files/**", handler: {
    request, response in

    // get the portion of the request path which was matched by the wildcard
    request.path = request.urlVariables[routeTrailingWildcardKey]

    // Initialize the StaticFileHandler with a documentRoot
    let handler = StaticFileHandler(documentRoot: "/var/www/htdocs")

    // trigger the handling of the request,
    // with our documentRoot and modified path set
    handler.handleRequest(request: request, response: response)
}
)

```

In the route example above, a request to `/files/foo.html` would return the corresponding file `/var/www/htdocs/foo.html`.

## Mustache Template Support

Mustache is a logic-less templating system. It permits you to use pre-written text files with placeholders that will be replaced at run-time with values particular to a given request.

For more general information on Mustache, consult the [mustache specification](#).

To use this module, add this project as a dependency in your `Package.swift` file.

```
.Package(
    url: "https://github.com/PerfectlySoft/Perfect-Mustache.git",
    majorVersion: 2
)
```

Then import the Mustache Module in your source code before using:

```
import PerfectMustache
```

Mustache templates can be used in either an HTTP server handler or standalone with no server.

## Mustache Server Handler

To use a mustache template as an HTTP response, you will need to create a handler object which conforms to `MustachePageHandler`. These handler objects generate the values which the template processor will use to produce its content.

```
/// A mustache handler, which should be passed to `mustacheRequest`, generates values to fill a mustache template
/// Call `context.extendValues(with: values)` one or more times and then
/// `context.requestCompleted(withCollector collector)` to complete the request and output the resulting content to the client.
public protocol MustachePageHandler {
    /// Called by the system when the handler needs to add values for the template.
    func extendValuesForResponse(context ctx: MustacheWebEvaluationContext, collector: MustacheEvaluationOutputCollector)
}
```

The template page handler, which you would implement, might look like the following:

```
struct TestHandler: MustachePageHandler { // all template handlers must inherit from PageHandler
    // This is the function which all handlers must impliment.
    // It is called by the system to allow the handler to return the set of values which will be used when populating the template.
    // - parameter context: The MustacheWebEvaluationContext which provides access to the HTTPRequest containing all the information pertaining
    // to the request
    // - parameter collector: The MustacheEvaluationOutputCollector which can be used to adjust the template output. For example a `defaultEncodingFunc` could be installed to change how outgoing values are encoded.
    func extendValuesForResponse(context ctx: MustacheWebEvaluationContext, collector: MustacheEvaluationOutputCollector) {
        var values = MustacheEvaluationContext.MapType()
        values["value"] = "hello"
        /// etc.
        ctx.extendValues(with: values)
        do {
            try ctx.requestCompleted(withCollector: collector)
        } catch {
            let response = ctx.webResponse
            response.status = .internalServerError
            response.appendBody(string: "\(error)")
            response.completed()
        }
    }
}
```

To direct a web request to a Mustache template, call the `mustacheRequest` function. This function is defined as follows:

```
public func mustacheRequest(request req: HTTPRequest, response: HTTPResponse, handler: MustachePageHandler, templatePath: String)
```

Pass to this function the current request and response objects, your `MustachePageHandler`, and the path to the template file you wish to serve. `mustacheRequest` will perform the initial steps such as creating the Mustache template parser, locating the template file, and calling your Mustache handler to generate the values which will be used when completing the template content.

The following snippet illustrates how to use a Mustache template in your URL handler. In this example, the template named "test.html" would be located in your server's web root directory:

```
{
    request, response in
    let webRoot = request.documentRoot
    mustacheRequest(request: request, response: response, handler: TestHandler(), templatePath: webRoot + "/test.html")
}
```

Look at the [UploadEnumerator](#) example for a more concrete example.

## Standalone Usage

It is possible to use this Mustache processor in a non-web, standalone manner. You can accomplish this by either providing the path to a template file, or by supplying the template data as a string. In either case, the template content will be parsed, and any values that you supply will be filled in.

The first example uses raw template text as the source. The second example passes in a file path for the template:

```
let templateText = "TOP {\n{#name}}\n{name}}{}/{/name}}\n\nBOTTOM"
let d = ["name":"The name"] as [String:Any]
let context = MustacheEvaluationContext(templateContent: templateText, map: d)
let collector = MustacheEvaluationOutputCollector()
let responseString = try context.formulateResponse(withCollector: collector)
XCTAssertEqual(responseString, "TOP {\n\nThe name\n\nBOTTOM")
```

```
let templatePath = "path/to/template.mustache"
let d = ["name":"The name"] as [String:Any]
let context = MustacheEvaluationContext(templatePath: templatePath, map: d)
let collector = MustacheEvaluationOutputCollector()
let responseString = try context.formulateResponse(withCollector: collector)
```

## Tag Support

This Mustache template processor supports:

- `{{regularTags}}`
- `{{{unencodedTags}}}`
- `{{# sections}} ... {{/sections}}`
- `{{^ invertedSections}} ... {{/invertedSections}}`
- `{{! comments}}`
- `{{> partials}}`
- `lambdas`

### Partials

All files used for partials must be located in the same directory as the calling template. Additionally, all partial files *must* have the file extension of **.mustache**, but this extension must not be included in the partial tag itself. For example, to include the contents of the file *foo.mustache*, you would use the tag `{{> foo }}` .

### Encoding

By default, all encoded tags (i.e. regular tags) are HTML-encoded, and `<` & `>` entities will be escaped. In your handler you can manually set the

`MustacheEvaluationOutputCollector.defaultEncodingFunc` function to perform whatever encoding you need. For example, when outputting JSON data you would want to set this function to something like the following:

```
collector.defaultEncodingFunc = {
    string in
    return (try? string.jsonEncodedString()) ?? "bad string"
}
```

### Lambdas

Functions can be added to the values dictionary. These will be executed and the results will be added to the template output. Such functions should have the following signature:

```
(tag: String, context: MustacheEvaluationContext) -> String
```

The `tag` parameter will be the tag name. For example, the tag `{{name}}` would give you the value "name" for the tag parameter.

## Perfect-Markdown

This project provides a solution to convert markdown text into html presentations.

This package builds with Swift Package Manager and is part of the [Perfect](#) project but can also be used as an independent module.

## Acknowledgement

Perfect-Markdown is directly building on [GerHobbelt's "upskirt"](#) project.

## Swift Package Manager

---

Add dependencies to your Package.swift

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-Markdown.git", majorVersion: 1)
```

## Import Perfect Markdown Library

---

Add the following header to your swift source code:

```
import PerfectMarkdown
```

## Get HTML from Markdown Text

---

Once imported, a new String extension `markdownToHTML` would be available:

```
let markdown = "# some blah blah blah markdown text \n\n## with mojo 🇨🇦"

guard let html = markdown.markdownToHTML else {
    // conversion failed
}

print(html)
```

## HTTP Request Logging

To log HTTP requests to a file, use the `Perfect-RequestLogger` module.

## Relevant Examples

---

- [Perfect-HTTPRequestLogging](#)
- [Perfect-Session-Memory-Demo](#)

## Usage

---

Add the following dependency to the `Package.swift` file:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-RequestLogger.git", majorVersion: 1)
```

In each file you wish to implement logging, import the module:

```
import PerfectRequestLogger
```

## When using PerfectHTTP 2.1 or later

---

Add to `main.swift` after instantiating your `server` :

```
// Instantiate a logger
let httplogger = RequestLogger()

// Configure Server
var confData: [String:[String:Any]] = [
    "servers": [
        [
            "name":"localhost",
            "port":8181,
            "routes":[],
            "filters":[
                [
                    "type":"response",
                    "priority":"high",
                    "name":PerfectHTTPServer.HTTPFilter.contentCompression,
                ],
                [
                    "type":"request",
                    "priority":"high",
                    "name":RequestLogger.filterAPIRequest,
                ],
                [
                    "type":"response",
                    "priority":"low",
                    "name":RequestLogger.filterAPIResponse,
                ]
            ]
        ]
    ]
]
```

The important parts of the configuration spec to add for enabling the Request Logger are:

```
[
    "type":"request",
    "priority":"high",
    "name":RequestLogger.filterAPIRequest,
],
[
    "type":"response",
    "priority":"low",
    "name":RequestLogger.filterAPIResponse,
]
```

These request & response filters add the required hooks to mark the beginning and the completion of the HTTP request and response.

## When using PerfectHTTP 2.0

Add to `main.swift` after instantiating your `server` :

```
// Instantiate a logger
let httplogger = RequestLogger()

// Add the filters
// Request filter at high priority to be executed first
server.setRequestFilters([(httplogger, .high)])
// Response filter at low priority to be executed last
server.setResponseFilters([(httplogger, .low)])
```

These request & response filters add the required hooks to mark the beginning and the completion of the HTTP request and response.

## Setting a custom Logfile location

The default logfile location is `/var/log/perfectLog.log` . To set a custom logfile location, set the `RequestLogFile.location` property:

```
RequestLogFile.location = "/var/log/myLog.log"
```

## Example Log Output

```
[INFO] [62f940aa-f204-43ed-9934-166896eda21c] [servername/WuAyNIIU-1] 2016-10-07 21:49:04 +0000 "GET /one HTTP/1.1" from 127.0.0.1 - 200 64B in 0.000436007976531982s
[INFO] [ec6a9ca5-00b1-4656-9e4c-ddecae8dde02] [servername/WuAyNIIU-2] 2016-10-07 21:49:06 +0000 "GET /two HTTP/1.1" from 127.0.0.1 - 200 64B in 0.000207006931304932s
```

This module expands on earlier work by [David Fleming](#).

## WebSockets

WebSockets are designed as full-duplex communication channels over a single TCP connection. The WebSocket protocol facilitates the real-time data transfer from and to a server. This is made possible by providing a standardized way for the server to send content to the browser without being solicited by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way, a bi-directional (two-way) ongoing conversation can take place between browser and server. The communications are typically done over standard TCP ports, such as 80 or 443.

The WebSocket protocol is currently supported in most major browsers including Google Chrome, Microsoft Edge, Internet Explorer, Firefox, Safari and Opera. WebSockets also require support from web applications on the server.

## Relevant Examples

- [Perfect-Chat-Demo](#)
- [Perfect-WebSocketsServer](#)

## Getting started

Add the WebSocket dependency to your Package.swift file:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-WebSockets.git", majorVersion: 2)
```

Then import the WebSocket library into your Swift source code as needed:

```
import PerfectWebSockets
```

A typical scenario is communication inside a web page like a chat room where multiple users are interacting in near-real time.

This example sets up a WebSocket service handler interacting on the route `/echo` :

```
var routes = Routes()

routes.add(method: .get, uri: "/echo", handler: {
    request, response in
    // Provide your closure which will return the service handler.
    WebSocketHandler(handlerProducer: {
        (request: HTTPRequest, protocols: [String]) -> WebSocketSessionHandler? in

        // Check to make sure the client is requesting our "echo" service.
        guard protocols.contains("echo") else {
            return nil
        }

        // Return our service handler.
        return EchoHandler()
    }).handleRequest(request: request, response: response)
})

)
```

## Handling WebSocket Sessions

A WebSocket service handler must implement the `WebSocketSessionHandler` protocol.

This protocol requires the function `handleSession(request: HTTPRequest, socket: WebSocket)` . This function will be called once the WebSocket connection has been established, at which point it is safe to begin reading and writing messages.

The initial `HttpRequest` object which instigated the session is provided for reference.

Messages are transmitted through the provided `WebSocket` object.

- Call `WebSocket.sendStringMessage` or `WebSocket.sendBinaryMessage` to send data to the client.
- Call `WebSocket.readStringMessage` or `WebSocket.readBinaryMessage` to read data from the client.

By default, reading will block indefinitely until a message arrives or a network error occurs.

A read timeout can be set with `WebSocket.readTimeoutSeconds` .

Close the session using `WebSocket.close()` .

The example `EchoHandler` consists of the following:

```
class EchoHandler: WebSocketSessionHandler {

    // The name of the super-protocol we implement.
    // This is optional, but it should match whatever the client-side WebSocket is initialized with.
    let socketProtocol: String? = "echo"

    // This function is called by the WebSocketHandler once the connection has been established.
    func handleSession(request: HttpRequest, socket: WebSocket) {

        // Read a message from the client as a String.
        // Alternatively we could call `WebSocket.readBytesMessage` to get the data as an array of bytes.
        socket.readStringMessage {
            // This callback is provided:
            // the received data
            // the message's op-code
            // a boolean indicating if the message is complete
            // (as opposed to fragmented)
            string, op, fin in

            // The data parameter might be nil here if either a timeout
            // or a network error, such as the client disconnecting, occurred.
            // By default there is no timeout.
            guard let string = string else {
                // This block will be executed if, for example, the browser window is closed.
                socket.close()
                return
            }

            // Print some information to the console for informational purposes.
            print("Read msg: \(string) op: \(op) fin: \(fin)")

            // Echo the data received back to the client.
            // Pass true for final. This will usually be the case, but WebSockets has
            // the concept of fragmented messages.
            // For example, if one were streaming a large file such as a video,
            // one would pass false for final.
            // This indicates to the receiver that there is more data to come in
            // subsequent messages but that all the data is part of the same logical message.
            // In such a scenario one would pass true for final only on the last bit of the video.
            socket.sendStringMessage(string, final: true) {

                // This callback is called once the message has been sent.
                // Recurse to read and echo new message.
                self.handleSession(request, socket: socket)
            }
        }
    }
}
```

## FastCGI Caveat

WebSockets serving is only supported with the stand-alone Perfect HTTP server. At this time, the WebSocket server does not operate with the Perfect FastCGI connector.

## WebSocket Class

## enum OpcodeType

WebSocket messages can be various types: continuation, text, binary, close, ping or pong, or invalid types.

## var readTimeoutSeconds

When trying to read a message from the current socket, this property helps the socket to read a message before timeout. If this property has been set to `NetEvent.noTimeout (-1)`, it will wait infinitely.

## read message

There are two ways of reading messages: text or binary, which differ only in the data type returned, i.e. `String` and `[UInt8]` array respectively.

### read text message:

```
public func readStringMessage(continuation: @escaping (String?, _ opcode: OpcodeType, _ final: Bool) -> ())
```

### read binary message:

```
public func readBytesMessage(continuation: @escaping ([UInt8]?, _ opcode: OpcodeType, _ final: Bool) -> ())
```

There are three parameters when it calls back:

### String / [UInt8]

`readMessage` will deliver the text / binary data sent from the client to your closure by this parameter.

### opcode

Use opcode if you want more controls in the communication.

### final

This parameter indicates whether the message is completed or fragmented.

## send message

There are two ways of sending messages: text or binary, which differ only in the data type sent, i.e. `String` and `[UInt8]` array respectively.

### send text message:

```
public func sendStringMessage(string: String, final: Bool, completion: @escaping () -> ())
```

### send binary message:

```
public func sendBinaryMessage(bytes: [UInt8], final: Bool, completion: @escaping () -> ())
```

Parameter `final` indicates whether the message is completed or fragmented.

## ping & pong

Perfect WebSocket also provides a convenient way of testing the connection. The `ping` method starts the test and expects a `pong` back.

Check out these two methods:

```
/// Send a "pong" message to the client.
public func sendPong(completion: @escaping () -> ())

/// Send a "ping" message to the client.
/// Expect a "pong" message to follow.
public func sendPing(completion: @escaping () -> ())
```

## func close()

To close the WebSocket connection:

```
socket.close()
```

For a Perfect WebSockets server example, visit the [Perfect-WebSocketsServer](#) demo.

## Perfect Utilities

In addition to the core Web Service-related functionality, Perfect provides a range of fundamental building blocks with which to build server side and desktop applications.

Like the JSON library discussed in previous chapters, many of these functions can be achieved by Swift's provided functions; however, Perfect's APIs aim to increase the efficiency, readability and simplicity of the end user's code.

Consult the followings sections for detailed information on the utilities available in Perfect's libraries:

- [Bytes](#)
- [File](#)
- [Dir](#)
- [Threading](#)
- [UUID](#)
- [SysProcess](#)
- [Log](#)
- [CURL](#)
- [Zip](#)# Bytes

The bytes object provides simple streaming of common Swift values to and from a UInt8 array. It supports importing and exporting UInt8, UInt16, UInt32, and UInt64 values. When importing these values, they are appended to the end of the contained array. When exporting, a repositionable marker is kept indicating the current export location. The bytes object is included as part of PerfectLib. Make sure to `import PerfectLib` if you wish to use the Bytes object.

The primary purpose behind the bytes object is to enable binary network payloads to be easily assembled and decomposed. The resulting UInt8 array is available through the `data` property.

The following example illustrates importing values of various sizes, and then exporting and validating the values. It also shows how to reposition the marker and how to determine how many bytes remain for export.

```
let i8 = 254 as UInt8
let i16 = 54045 as UInt16
let i32 = 4160745471 as UInt32
let i64 = 17293541094125989887 as UInt64

let bytes = Bytes()

bytes.import64Bits(from: i64)
    .import32Bits(from: i32)
    .import16Bits(from: i16)
    .import8Bits(from: i8)

let bytes2 = Bytes()
bytes2.importBytes(from: bytes)

XCTAssert(i64 == bytes2.export64Bits())
XCTAssert(i32 == bytes2.export32Bits())
XCTAssert(i16 == bytes2.export16Bits())
bytes2.position -= sizeof(UInt16.self)
XCTAssert(i16 == bytes2.export16Bits())
XCTAssert(bytes2.availableExportBytes == 1)
XCTAssert(i8 == bytes2.export8Bits())
```

## File Operations

Perfect brings file system operations into your sever-side Swift environment to control how data is stored and retrieved in an accessible way.

First, ensure the `PerfectLib` is imported in your Swift file:

```
import PerfectLib
```

You are now able to use the `File` object to query and manipulate the file system.

## Setting Up a File Object Reference

Specify the absolute or relative path to the file:

```
let thisFile = File("/path/to/file/helloWorld.txt")
```

If you are not familiar with the file path, please read [Directory Operations](#) first.

## Opening a File for Read or Write Access

**Important:** Before writing to a file — even if it is a new file — it must be opened with the appropriate permissions.

To open a file:

```
try thisFile.open(<OpenMode>,permissions:<PermissionMode>)
```

For example, to write a file:

```
let thisFile = File("helloWorld.txt")
try thisFile.open(.readWrite)
try thisFile.write(string: "Hello, World!")
thisFile.close()
```

For full outlines of [OpenMode](#) and [PermissionMode](#) values, see their definitions later in this document.

## Checking If a File Exists

Use the `exists` method to return a Boolean value.

```
thisFile.exists
```

## Get the Modification Time for a File

Return the modification date for the file in the standard UNIX format of seconds since 1970/01/01 00:00:00 GMT, as an integer using:

```
thisFile.modificationTime
```

## File Paths

Regardless of how a file reference was defined, both the absolute (real) and internal path can be returned.

Return the "internal reference" file path:

```
thisFile.path
```

Return the "real" file path. If the file is a symbolic link, the link will be resolved:

```
thisFile.realpath
```

## Closing a File

Once a file has been opened for read or write, it is advisable to either close it at a specific place within the code, or by using `defer` :

```
let thisFile = File("/path/to/file/helloWorld.txt")
// Your processing here
thisFile.close()
```

## Deleting a File

To remove a file from the file system, use the `delete()` method.

```
thisFile.delete()
```

This also closes the file, so there is no need to invoke an additional `close()` method.

## Returning the Size of a File

`size` returns the size of the file in bytes, as an integer.

```
thisFile.size
```

## Determining if the File is a Symbolic Link

If the file is a symbolic link, the method will return Boolean `true`, otherwise `false`.

```
thisFile.isLink
```

## Determining if the File Object is a Directory

If the file object refers instead to a directory, `isDir` will return either a Boolean `true` or `false` value.

```
thisFile.isDir
```

## Returning the File's UNIX Permissions

`perms` returns the UNIX style permissions for the file as a `PermissionMode` object.

```
thisFile.perms
```

For example:

```
print(thisFile.perms)
>> PermissionMode(rawValue: 29092)
```

## Reading Contents of a File

### readSomeBytes

Reads up to the indicated number of bytes from the file:

```
let thisFile = File("/path/to/file/helloWorld.txt")
let contents = try thisFile.readSomeBytes(count: <Int>)
```

### Parameters

To read a specific byte range of a file's contents, enter the number of bytes you wish to read. For example, to read the first 10 bytes of a file:

```
let thisFile = File("/path/to/file/helloWorld.txt")
let contents = try thisFile.readSomeBytes(count: 10)
print(contents)
>> [35, 32, 80, 101, 114, 102, 101, 99, 116, 84]
```

### readString

`readString` reads the entire file as a string:

```
let thisFile = File("/path/to/file/helloWorld.txt")
let contents = try thisFile.readString()
```

## Writing, Copying, and Moving Files

**Important:** Before writing to a file — even if it is a new file — it must be opened with the appropriate permissions.

### Writing a String to a File

Use `write` to create or rewrite a string to the file using UTF-8 encoding. The method returns an integer which is the number of bytes written.

Note that this method uses the `@discardableResult` property, so it can be used without assignment if required.

```
let bytesWritten = try thisFile.write(string: <String>)
```

## Writing a Bytes Array to a File

An array of `bytes` can also be written directly to a file. The method returns an integer which is the number of bytes written.

Note that this method uses the `@discardableResult` property, so it can be used without assignment if required.

```
let bytesWritten = try thisFile.write(  
    bytes: <[UInt8]>,  
    dataPosition: <Int>,  
    length: <Int>  
)
```

### Parameters

- **bytes:** The array of UInt8 to write
- **dataPosition:** *Optional.* The offset within `bytes` at which to begin writing
- **length:** *Optional.* The number of bytes to write

## Moving a File

Once a file is defined, the `moveTo` method can be used to relocate the file to a new location in the file system. This can also be used to rename a file if desired. The operation returns a new file object representing the new location.

```
let newFile = thisFile.moveTo(path: <String>, overWrite: <Bool>)
```

### Parameters

- **path:** The path to move the file to
- **overWrite:** *Optional.* Indicates that any existing file at the destination path should first be deleted. Default is `false`

### Error Handling

The method throws `PerfectError.FileError` on error.

```
let thisFile = File("/path/to/file/helloWorld.txt")  
let newFile = try thisFile.moveTo(path: "/path/to/file/goodbyeWorld.txt")
```

## Copying a File

Similar to `moveTo`, `copyTo` copies the file to the new location, optionally overwriting any existing file. However, it does not delete the original file. A new file object is returned representing the new location.

Note that this method uses the `@discardableResult` property, so it can be used without assignment if required.

```
let newFile = thisFile.copyTo(path: <String>, overWrite: <Bool>)
```

### Parameters

- **path:** The path to copy the file to
- **overWrite:** *Optional.* Indicates that any existing file at the destination path should first be deleted. Default is `false`

### Error Handling

The method throws `PerfectError.FileError` on error.

```
let thisFile = File("/path/to/file/helloWorld.txt")  
let newFile = try thisFile.copyTo(path: "/path/to/file/goodbyeWorld.txt")
```

## File Locking Functions

The file locking functions allow *sections* of a file to be locked with *advisory-mode locks*.

All the locks for a file are removed when the file is closed or the process terminates.

**Note:** These are not file system locks, and do not prevent others from performing write operations on the affected files: The locks are "advisory-mode locks".

## Locking a File

Attempts to place an advisory lock starting from the current position marker up to the indicated byte count. This function will block the current thread until the lock can be performed.

```
let result = try thisFile.lock(byteCount: <Int>)
```

## Unlocking a File

Unlocks the number of bytes starting from the current position marker up to the indicated byte count.

```
let result = try thisFile.unlock(byteCount: <Int>)
```

## Attempt to Lock a File

Attempts to place an advisory lock starting from the current position marker up to the indicated byte count. This function will throw an exception if the file is already locked, but will not block the current thread.

```
let result = try thisFile.tryLock(byteCount: <Int>)
```

## Testing a Lock

Tests if the indicated bytes are locked. Returns a Boolean true or false.

```
let isLocked = try thisFile.testLock(byteCount: <Int>)
```

## File OpenMode

The OpenMode of a file is defined as an enum:

- `.read`: Opens the file for read-only access
- `.write`: Opens the file for write-only access, creating the file if it did not exist
- `.readWrite`: Opens the file for read-write access, creating the file if it did not exist
- `.append`: Opens the file for read-write access, creating the file if it did not exist and moving the file marker to the end
- `.truncate`: Opens the file for read-write access, creating the file if it did not exist and setting the file's size to zero

For example, to write a file:

```
let thisFile = File("helloWorld.txt")
try thisFile.open(.readWrite)
try thisFile.write(string: "Hello, World!")
thisFile.close()
```

## File PermissionMode

The PermissionMode for a directory or file is provided as a single option or as an array of options.

For example, to create a directory with specific permissions:

```
let thisDir = Dir("/path/to/dir/")
do {
    try thisDir.create(perms: [.rxUser, .rxGroup, .rxOther])
} catch {
    print("error")
}
//or
do {
    try thisDir.create(perms: .rxUser)
} catch {
    print("error")
}
```

## PermissionMode Options

- `.readUser` : Readable by user
- `.writeUser` : Writable by user
- `.executeUser` : Executable by user
- `.readGroup` : Readable by group
- `.writeGroup` : Writable by group
- `.executeGroup` : Executable by group
- `.readOther` : Readable by others
- `.writeOther` : Writable by others
- `.executeOther` : Executable by others
- `.rwxUser` : Read, write, execute by user
- `.rwUserGroup` : Read, write by user and group
- `.rxGroup` : Read, execute by group
- `.rxOther` : Read, execute by other

## Directory Operations

Perfect brings file system operations into your sever-side Swift environments to control how data is stored and retrieved in an accessible way.

## Relevant Examples

---

- [Perfect-Directory-Lister](#)
- [Perfect-FileHandling](#)

## Usage

---

First, ensure the `PerfectLib` is imported in your Swift file:

```
import PerfectLib
```

You are now able to use the `Dir` object to query and manipulate the file system.

### Setting Up a Directory Object Reference

Specify the absolute or relative path to the directory:

```
let thisDir = Dir("/path/to/directory/")
```

### Checking If a Directory Exists

Use the `exists` method to return a Boolean value.

```
let thisDir = Dir("/path/to/directory/")
thisDir.exists
```

### Returning the Current Directory Object's Name

`name` returns the name of the object's directory. Note that this is different from the "path".

```
thisDir.name
```

### Returning the Parent Directory

`parentDir` returns a `Dir` object representing the current directory object's parent. Returns nil if there is no parent.

```
let thisDir = Dir("/path/to/directory/")
let parent = thisDir.parentDir
```

### Revealing the Directory Path

`path` returns the path to the current directory.

```
let thisDir = Dir("/path/to/directory/")
let path = thisDir.path
```

## Returning the Directory's UNIX Permissions

`perms` returns the UNIX style permissions for the directory as a `PermissionMode` object.

```
thisDir.perms
```

For example:

```
print(thisDir.perms)
>> PermissionMode(rawValue: 29092)
```

## Creating a Directory

`create` creates the directory using the provided permissions. All directories along the path will be created if needed.

The following will create a new directory with the default permissions (Owner: read-write-execute, Group and Everyone: read-execute).

```
let newDir = Dir("/path/to/directory/newDirectory")
try newDir.create()
```

To create a directory with specific permissions, specify the `perms` parameter:

```
let newDir = Dir("/path/to/directory/newDirectory")
try newDir.create(perms: [.rwxUser, .rxGroup, .rxOther])
```

The method throws `PerfectError.FileError` if an error creating the directory was encountered.

## Deleting a Directory

Deleting a directory from the file system:

```
let newDir = Dir("/path/to/directory/newDirectory")
try newDir.delete()
```

The method throws `PerfectError.FileError` if an error deleting the directory was encountered.

## Working Directories

### Set the Working Directory to the Location of the Current Object

Use `setAsWorkingDir` to set the current working directory to the location of the object's path.

```
let thisDir = Dir("/path/to/directory/")
try thisDir.setAsWorkingDir()
```

### Return the Current Working Directory

Returns a new object containing the current working directory.

```
let workingDir = Dir.workingDir
```

## Reading the Directory Structure

`forEachEntry` enumerates the contents of the directory, passing the name of each contained element to the provided callback.

```
try thisDir.forEachEntry(closure: {
    n in
    print(n)
})
```

# Threading

Perfect provides a core threading library in the `PerfectThread` package. This package is designed to provide support for the rest of the systems in Perfect. `PerfectThread` is abstracted over the core operating system level threading package.

`PerfectThread` is imported by `PerfectNet` and so it is not generally required that one directly import it. However, if you need to do so you can `import PerfectThread`.

`PerfectThread` provides the following constructs:

- `Threading.Lock` - Mutually exclusive thread lock, a.k.a. a mutex or critical section
- `Threading.RWLock` - A many reader/single writer based thread lock
- `Threading.Event` - Wait/signal/broadcast type synchronization
- `Threading.sleep` - Block/pause; a single thread for a given period of time
- `Threading.queue` - Create either a serial or concurrent thread queue with a given name
- `Threading.dispatch` - Dispatch a closure on a named queue
- `Promise` - An API for executing one or more tasks on alternate threads and polling or waiting for return values or errors.

These systems provide internal concurrency for Perfect, and are heavily used in the `PerfectNet` package in particular.

## Locks

`PerfectThread` provides both mutex and rwlock synchronization objects.

### Mutex

Mutexes are provided through the `Threading.Lock` object. These are intended to protect shared resources from being accessed simultaneously by multiple threads at once. It provides the following functions:

```
/// A wrapper around a variety of threading related functions and classes.
public extension Threading {
    /// A mutex-type thread lock.
    /// The lock can be held by only one thread.
    /// Other threads attempting to secure the lock while it is held will block.
    /// The lock is initialized as being recursive.
    /// The locking thread may lock multiple times, but each lock should be accompanied by an unlock.
    public class Lock {
        /// Attempt to grab the lock.
        /// Returns true if the lock was successful.
        public func lock() -> Bool
        /// Attempt to grab the lock.
        /// Will only return true if the lock was not being held by any other thread.
        /// Returns false if the lock is currently being held by another thread.
        public func tryLock() -> Bool
        /// Unlock. Returns true if the lock was held by the current thread and was successfully unlocked, or the lock count was decremented.
        public func unlock() -> Bool
        /// Acquire the lock, execute the closure, release the lock.
        public func doWithLock(closure: () throws -> ()) rethrows
    }
}
```

The general usage pattern as as follows:

- Created a shared instance of `Threading.Lock`
- When a shared resource needs to be accessed, call the `lock()` function
  - If another thread already has the lock then the calling thread will block until it is unlocked
- Once the call to `lock()` returns the resource is safe to access
- When finished, call `unlock()`
  - Other threads are now free to acquire the lock

Alternatively, you can pass a closure to the `doWithLock` function. This will lock, call the closure, and then unlock.

The `tryLock` function will lock and return true if no other thread currently holds the lock. If another thread holds the lock, it will return false.

### Read/Write Lock

Read/Write Locks (`RWLock`) are provided through the `Threading.RWLock` object. `RWLocks` support many threads accessing a shared resource in a read-only capacity. For

example, it could permit many threads at once to be accessing values in a shared Dictionary. When a thread needs to perform a modification (a write) to a shared object it acquires a write lock. Only one thread can hold a write lock at a time, and all other threads attempting to read or write will block until the write lock is released. An attempt to lock for writing will block until any other read or write locks have been released. When attempting to acquire a write lock, no other read locks will be permitted, and threads attempting to read or write will be blocked until the write lock is held and then released.

RWLock is defined as follows:

```
/// A wrapper around a variety of threading related functions and classes.
public extension Threading {
  /// A read-write thread lock.
  /// Permits multiple readers to hold the while, while only allowing at most one writer to hold the lock.
  /// For a writer to acquire the lock all readers must have unlocked.
  /// For a reader to acquire the lock no writers must hold the lock.
  public final class RWLock {
    /// Attempt to acquire the lock for reading.
    /// Returns false if an error occurs.
    public func readLock() -> Bool
    /// Attempts to acquire the lock for reading.
    /// Returns false if the lock is held by a writer or an error occurs.
    public func tryReadLock() -> Bool
    /// Attempt to acquire the lock for writing.
    /// Returns false if an error occurs.
    public func writeLock() -> Bool
    /// Attempt to acquire the lock for writing.
    /// Returns false if the lock is held by readers or a writer or an error occurs.
    public func tryWriteLock() -> Bool
    /// Unlock a lock which is held for either reading or writing.
    /// Returns false if an error occurs.
    public func unlock() -> Bool
    /// Acquire the read lock, execute the closure, release the lock.
    public func doWithReadLock(closure: () throws -> ()) rethrows
    /// Acquire the write lock, execute the closure, release the lock.
    public func doWithWriteLock(closure: () throws -> ()) rethrows
  }
}
```

RWLock supports `tryReadLock` and `tryWriteLock`, both of which will return false if the lock cannot be immediately acquired. It also supports `doWithReadLock` and `doWithWriteLock` which will call the provided closure with the lock held and then release it when it has completed.

## Events

The `Threading.Event` object provides a way to safely signal or communicate among threads about particular events. For instance, to signal worker threads that a task has entered a queue. It's important to note that `Threading.Event` inherits from `Threading.Lock` as using the `lock` and `unlock` methods provided therein are vital to understanding thread event behaviour.

`Threading.Event` provides the following functions:

```
public extension Threading {
  /// A thread event object. Inherits from `Threading.Lock`.
  /// The event MUST be locked before `wait` or `signal` is called.
  /// While inside the `wait` call, the event is automatically placed in the unlocked state.
  /// After `wait` or `signal` return the event will be in the locked state and must be unlocked.
  public final class Event: Lock {
    /// Signal at most ONE thread which may be waiting on this event.
    /// Has no effect if there is no waiting thread.
    public func signal() -> Bool
    /// Signal ALL threads which may be waiting on this event.
    /// Has no effect if there is no waiting thread.
    public func broadcast() -> Bool
    /// Wait on this event for another thread to call signal.
    /// Blocks the calling thread until a signal is received or the timeout occurs.
    /// Returns true only if the signal was received.
    /// Returns false upon timeout or error.
    public func wait(seconds secs: Double = Threading.noTimeout) -> Bool
  }
}
```

The general usage pattern is illustrated by using a producer/consumer metaphor:

*Producer Thread*

- Producer thread wants to produce a resource and alert other threads about the occurrence
- Call the `lock` function
- Produce the resource
- Call the `signal` or `broadcast` function
- Call the `unlock` function

#### *Consumer Thread*

- Call the `lock` function
- If a resource is available for consumption:
  - Consume the resource and call the `unlock` function
- If a resource is not available for consumption:
  - Call the `wait` function
  - When `wait` returns true:
    - If a resource is available then consume the resource
  - Call the `unlock` function

These producer/consumer threads generally operate in a loop performing these steps repeatedly during the life of the program.

The `wait` function accepts an optional timeout parameter. If the timeout expires then `wait` will return false. By default, `wait` does not timeout.

The functions `signal` and `broadcast` differ in that `signal` will alert at most one waiting thread while `broadcast` will alert all currently waiting threads.

## Queues

The `PerfectThread` package provides an abstracted thread queue system. It is based loosely on Grand Central Dispatch (GCD), but is designed to mask the actual threading primitives, and as such, operate with a variety of underlying systems.

This queue system provides the following features:

- Named serial queues - one thread operating, removing and executing tasks
- Named concurrent queues - multiple threads operating, the count varying depending on the number of available CPUs, removing and executing tasks simultaneously
- Anonymous serial or concurrent queues which are not shared and can be explicitly destroyed.
- A default concurrent queue

This system provides the following functions:

```

/// A thread queue which can dispatch a closure according to the queue type.
public protocol ThreadQueue {
    /// The queue name.
    var name: String { get }
    /// The queue type.
    var type: Threading.QueueType { get }
    /// Execute the given closure within the queue's thread.
    func dispatch(_ closure: Threading.ThreadClosure)
}

public extension Threading {
    /// The function type which can be given to `Threading.dispatch`.
    public typealias ThreadClosure = () -> ()
    /// Queue type indicator.
    public enum QueueType {
        /// A queue which operates on only one thread.
        case serial
        /// A queue which operates on a number of threads, usually equal to the number of logical CPUs.
        case concurrent
    }
    /// Find or create a queue indicated by name and type.
    public static func getQueue(name nam: String, type: QueueType) -> ThreadQueue
    /// Returns an anonymous queue of the indicated type.
    /// This queue can not be utilized without the returned ThreadQueue object.
    /// The queue should be destroyed when no longer needed.
    public static func getQueue(type: QueueType) -> ThreadQueue
    /// Return the default queue
    public static func getDefaultQueue() -> ThreadQueue
    /// Terminate and remove a thread queue.
    public static func destroyQueue(_ queue: ThreadQueue)
    /// Call the given closure on the "default" concurrent queue
    /// Returns immediately.
    public static func dispatch(closure: Threading.ThreadClosure)
}

```

Calling `Threading.getQueue` will create the queue if it does not already exist. Once the queue object has been returned call, its `dispatch` function and pass it the closure which will be executed on that queue.

The system will automatically create a queue called "default". Calling the static `Threading.dispatch` function will always dispatch the closure on this queue.

## Promise

A Promise is an object which is shared between one or more threads. A promise will execute the closure/function given to it on a new thread. When the thread produces its return value a consumer thread will be able to obtain the value or handle the error if one occurred.

This object is generally used in one of two ways:

- By passing a closure/function which accepts zero parameters and returns some arbitrary type, followed by zero or more calls to `.then`

Example: Count to three on another thread

```

let v = try Promise { 1 }
    .then { try $0() + 1 }
    .then { try $0() + 1 }
    .wait()
XCTAssert(v == 3, "\(v)")

```

Note that the closure/function given to `.then` accepts a function which must be called to either throw the error produced by the previous call or return its value.

- By passing a closure/function which is executed on another thread and accepts the Promise as a parameter. The promise can at some later point be `.set` or `.fail`'ed, with a return value or error object, respectively. The Promise creator can periodically `.get` or `.wait` for the value or error. This provides the most flexible usage as the Promise can be `.set` at any point, for example after a series of asynchronous API calls.

Example: Pause then set a Bool value

```
let prom = Promise<Bool> {
    (p: Promise) in
        Threading.sleep(seconds: 2.0)
        p.set(true)
}
XCTAssert(try prom.get() == nil) // not fulfilled yet
XCTAssert(try prom.wait(seconds: 3.0) == true)
```

Regardless of which method is used, the Promise's closure will immediately begin executing on a new thread.

The full Promise API is as follows:

```
public class Promise<ReturnType> {
    /// Initialize a Promise with a closure. The closure is passed the promise object on which the
    /// return value or error can be later set.
    /// The closure will be executed on a new serial thread queue and will begin
    /// executing immediately.
    public init(closure: @escaping (Promise<ReturnType>) throws -> ()) throws -> ()
    /// Initialize a Promise with a closure. The closure will return a single value type which will
    /// fulfill the promise.
    /// The closure will be executed on a new serial thread queue and will begin
    /// executing immediately.
    public init(closure: @escaping () throws -> ReturnType) throws -> ()
    /// Chain a new Promise to an existing. The provided closure will receive the previous promise's
    /// value once it is available and should return a new value.
    public func then<NewType>(closure: @escaping (() throws -> ReturnType) throws -> NewType) -> Promise<NewType>
}

public extension Promise {
    /// Get the return value if it is available.
    /// Returns nil if the return value is not available.
    /// If a failure has occurred then the Error will be thrown.
    /// This is called by the consumer thread.
    public func get() throws -> ReturnType?
    /// Get the return value if it is available.
    /// Returns nil if the return value is not available.
    /// If a failure has occurred then the Error will be thrown.
    /// Will block and wait up to the indicated number of seconds for the return value to be produced.
    /// This is called by the consumer thread.
    public func wait(seconds: Double = Threading.noTimeout) throws -> ReturnType?
}

public extension Promise {
    /// Set the Promise's return value, enabling the consumer to retrieve it.
    /// This is called by the producer thread.
    public func set(_ value: ReturnType)
    /// Fail the Promise and set its error value.
    /// This is called by the producer thread.
    public func fail(_ error: Error)
}
```

## Networking

...

## OAuth2

The Perfect Authentication OAuth2 provides [OAuth2](#) libraries and OAuth2 provider drivers for Facebook, Google, and GitHub.

A demo application can be found at <https://github.com/PerfectExamples/Perfect-Authentication-Demo> that shows the usage of the libraries and providers.

## Adding to your project

Add this project as a dependency in your Package.swift file.

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-Authentication.git", majorVersion: 1)
```

To then use the OAuth2 module in your code:

```
import OAuth2
```

## Configuration

Each provider needs an "appid", also known as a "key", and a "secret". These are usually generated by the OAuth Host, such as Facebook, GitHub and Google developer consoles. These values, as well as an "endpointAfterAuth" and "redirectAfterAuth" value must be set for each provider you wish to use.

To configure Facebook as a provider:

```
FacebookConfig.appid = "yourAppID"
FacebookConfig.secret = "yourSecret"
FacebookConfig.endpointAfterAuth = "http://localhost:8181/auth/response/facebook"
FacebookConfig.redirectAfterAuth = "http://localhost:8181/"
```

To configure Google as a provider:

```
GoogleConfig.appid = "yourAppID"
GoogleConfig.secret = "yourSecret"
GoogleConfig.endpointAfterAuth = "http://localhost:8181/auth/response/google"
GoogleConfig.redirectAfterAuth = "http://localhost:8181/"
```

To configure GitHub as a provider:

```
GitHubConfig.appid = "yourAppID"
GitHubConfig.secret = "yourSecret"
GitHubConfig.endpointAfterAuth = "http://localhost:8181/auth/response/github"
GitHubConfig.redirectAfterAuth = "http://localhost:8181/"
```

## Adding Routes

The OAuth2 system relies on an authentication / exchange system, which requires a URL to be specially assembled that the user is redirected to, and a URL that the user is returned to after the user has committed the authorization action.

The first set of routes below are the action URLs that will redirect to the OAuth2 provider's system. They can be anything you wish them to be. The user will never see anything on them as they will be immediately redirected to the correct location.

The second set of routes below are where the OAuth2 provider should return the user to. Note that this is the same as the "endpointAfterAuth" configuration option. Once the "authResponse" function has been completed the user is automatically forwarded to the URL in the "redirectAfterAuth" option.

```
var routes: [[String: Any]] = [[String: Any]]()

routes.append(["method": "get", "uri": "/to/facebook", "handler": Facebook.sendToProvider])
routes.append(["method": "get", "uri": "/to/github", "handler": GitHub.sendToProvider])
routes.append(["method": "get", "uri": "/to/google", "handler": Google.sendToProvider])

routes.append(["method": "get", "uri": "/auth/response/facebook", "handler": Facebook.authResponse])
routes.append(["method": "get", "uri": "/auth/response/github", "handler": GitHub.authResponse])
routes.append(["method": "get", "uri": "/auth/response/google", "handler": Google.authResponse])
```

## Information returned and made available

After the user has been authenticated, certain information is gleaned from the OAuth2 provider.

Note that the session ID can be retrieved using:

```
request.session?.token
```

The user-specific information can be accessed as part of the session info:

```
// The UserID as defined by the provider
request.session?.userid

// designates the OAuth2 source - useful if you are allowing multiple OAuth providers
request.session?.data["loginType"]

// The access token obtained in the process
request.session?.data["accessToken"]

// The user's first name as supplied by the provider
request.session?.data["firstName"]

// The user's last name as supplied by the provider
request.session?.data["lastName"]

// The user's profile picture as supplied by the provider
request.session?.data["picture"]
```

With access to this information, you can now save to the database of your choice.

## UUID

Also known as a Globally Unique Identifier (GUID), a Universal Unique Identifier (UUID) is a 128-bit number used to uniquely identify some object or entity. The UUID relies upon a combination of components to ensure uniqueness.

### Create a New UUID object

A new UUID object can either be randomly generated, or assigned.

To randomly generate a v4 UUID:

```
let u = UUID()
```

To assign a v4 UUID from a string:

```
let u = UUID(<String>)
```

If the string is invalid, the object is assigned the following UUID instead: `00000000-0000-0000-0000-000000000000`

To return the string value of a UUID:

```
let u1 = UUID()
print(u1.string)
```

## SysProcess

Perfect provides the ability to execute local processes or shell commands through the `SysProcess` type. This type allows local processes to be launched with an array of parameters and shell variables. Some processes will execute and return a result immediately. Other processes can be left open for interactive read/write operations.

## Relevant Examples

- [Perfect-System](#)

## Setup

Add the "Perfect" project as a dependency in your Package.swift file:

```
.Package(
    url: "https://github.com/PerfectlySoft/Perfect.git",
    majorVersion: 2
)
```

In your file where you wish to use SysProcess, import the PerfectLib and add either SwiftGlibc or Darwin:

```
import PerfectLib

#if os(Linux)
    import SwiftGlibc
#else
    import Darwin
#endif
```

## Executing a SysProcess Command

The following function `runProc` accepts a command, an array of arguments, and optionally outputs the response from the command.

```
func runProc(cmd: String, args: [String], read: Bool = false) throws -> String? {
    let envs = [("PATH", "/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin")]
    let proc = try SysProcess(cmd, args: args, env: envs)
    var ret: String?
    if read {
        var ary = [UInt8]()
        while true {
            do {
                guard let s = try proc.stdout?.readSomeBytes(count: 1024) where s.count > 0 else {
                    break
                }
                ary.append(contentsOf: s)
            } catch PerfectLib.PerfectError.fileError(let code, _) {
                if code != EINTR {
                    break
                }
            }
        }
        ret = UTF8Encoding.encode(bytes: ary)
    }
    let res = try proc.wait(hang: true)
    if res != 0 {
        let s = try proc.stderr?.readString()
        throw PerfectError.systemError(Int32(res), s!)
    }
    return ret
}

let output = try runProc(cmd: "ls", args: ["-la"], read: true)
print(output)
```

Note that the `SysProcess` command is executed in this example with a hardcoded environment variable.

## SysProcess Members

### stdin

`stdin` is the standard input file stream.

### stdout

`stdout` is the standard output file stream.

### stderr

`stderr` is the standard error file stream.

### pid

`pid` is the process identifier.

## SysProcess Methods

### isOpen

Returns true if the process was opened and was running at some point.

Note that the process may not be currently running. Use `wait(false)` to check if the process is currently running.

```
myProcess.isOpen()
```

## close

`close` terminates the process and cleans up.

```
myProcess.close()
```

## detach

Detach from the process such that it will not be manually terminated when this object is uninitialized.

```
myProcess.detach()
```

## wait

Determine if the process has completed running and retrieve its result code.

```
myProcess.wait(hang: <Bool>)
```

## kill

Terminate the process and return its result code.

```
myProcess.kill(signal: <Int32 = SIGTERM>)
```

Response is an `Int32` result code.

# Log

Perfect has a built-in error logging system that allows messages to be logged at several different levels. Each log level can be routed to either the console or the system log.

The built-in log levels, in order of increasing severity:

- **debug:** Log lines are preceded by `[DBG]`
- **info:** Log lines are preceded by `[INFO]`
- **warning:** Log lines are preceded by `[WARN]`
- **error:** Log lines are preceded by `[ERR]`
- **critical:** Log lines are preceded by `[CRIT]`
- **terminal:** Log lines are preceded by `[TERM]`

## To Log Information to the Console:

```
Log.debug(message: "Line 123: value \$(myVar)")
Log.info(message: "At Line 123")
Log.warning(message: "Entered error handler")
Log.error(message: "Error condition: \$(errorMessage)")
Log.critical(message: "Exception Caught: \$(exceptionVar)")
Log.terminal(message: "Uncaught exception, terminating. \$(infoVar)")
```

## To Log Information to the System Log:

If you wish to pipe all log entries to the system log, set the `Log.logger` property to `SysLogger()` early in the application setup. Once this has been executed, all output will be logged to the System Log file, and echoed to the console.

```
Log.logger = SysLogger()
```

If you wish to change the logger process back to only the console at any point, set the property back to `ConsoleLogger()`

```
Log.logger = ConsoleLogger()
```

# File Logging

Using the `PerfectLogger` module, events can be logged to a specified file, in addition to the console.

## Usage

Add the dependency to your project's Package.swift file:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-Logger.git", majorVersion: 0),
```

Now add the `import` directive to the file you wish to use the logging in:

```
import PerfectLogger
```

To log events to the local console as well as a file:

```
LogFile.debug("debug message", logFile: "test.txt")
LogFile.info("info message", logFile: "test.txt")
LogFile.warning("warning message", logFile: "test.txt")
LogFile.error("error message", logFile: "test.txt")
LogFile.critical("critical message", logFile: "test.txt")
LogFile.terminal("terminal message", logFile: "test.txt")
```

To log to the default file, omit the file name parameter.

## Linking events with "eventid"

Each log event returns an event id string. If an eventid string is supplied to the directive then it will use the supplied eventid in the log file instead. This makes it easy to link together related events.

```
let eid = LogFile.warning("test 1")
LogFile.critical("test 2", eventid: eid)
```

returns:

```
[WARNING] [62f940aa-f204-43ed-9934-166896eda21c] [2016-11-16 15:18:02 GMT-05:00] test 1
[CRITICAL] [62f940aa-f204-43ed-9934-166896eda21c] [2016-11-16 15:18:02 GMT-05:00] test 2
```

The returned eventid is marked `@discardableResult` and therefore can be safely ignored if not required for re-use.

## Setting a custom Logfile location

The default logfile location is `./log.log`. To set a custom logfile location, set the `LogFile.location` variable:

```
LogFile.location = "/var/log/myLog.log"
```

Messages can now be logged directly to the file as set by using:

```
LogFile.debug("debug message")
LogFile.info("info message")
LogFile.warning("warning message")
LogFile.error("error message")
LogFile.critical("critical message")
LogFile.terminal("terminal message")
```

## Sample output

```
[DEBUG] [ec6a9ca5-00b1-4656-9e4c-ddecae8dde02] [2016-11-16 15:18:02 GMT-05:00] a debug message
[INFO] [ec6a9ca5-00b1-4656-9e4c-ddecae8dde02] [2016-11-16 15:18:02 GMT-05:00] an informational message
[WARNING] [ec6a9ca5-00b1-4656-9e4c-ddecae8dde02] [2016-11-16 15:18:02 GMT-05:00] a warning message
[ERROR] [62f940aa-f204-43ed-9934-166896eda21c] [2016-11-16 15:18:02 GMT-05:00] an error message
[CRITICAL] [62f940aa-f204-43ed-9934-166896eda21c] [2016-11-16 15:18:02 GMT-05:00] a critical message
[EMERG] [ec6a9ca5-00b1-4656-9e4c-ddecae8dde02] [2016-11-16 15:18:02 GMT-05:00] an emergency message
```

# Remote Logging

Using the `PerfectLogger` module, events can be logged to a specified remote Perfect Log Server, in addition to the console.

The [Perfect Log Server](#) is a stand-alone project that can be deployed on your own servers.

## Using in your project

To include the dependency in your project, add the following to your project's Package.swift file:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-Logger.git", majorVersion: 1),
```

Now add the import directive to the file you wish to use the logging in:

```
import PerfectLogger
```

## Configuration

Three configuration parameters are required:

```
// Your token
RemoteLogger.token = "<your token>"

// App ID (Optional)
RemoteLogger.appid = "<your appid>"

// URL to access the log server.
// Note, this is not the full API path, just the host and port.
RemoteLogger.logServer = "http://localhost:8181"
```

## Usage

To log events to the log server:

```
var obj = [String: Any]()
obj["one"] = "donkey"
RemoteLogger.critical(obj)
```

## Linking events with "eventid"

Each log event returns an event id string. If an eventid string is supplied to the directive then it will use the supplied eventid in the log directive instead. This makes it easy to link together related events.

```
let eid = RemoteLogger.critical(obj)
RemoteLogger.info(obj, eventid: eid)
```

The returned eventid is marked `@discardableResult` and therefore can be safely ignored if not required for re-use.

# Network Requests with Perfect-CURL

The Perfect-CURL package provides support for [curl](#) in Swift. This package builds with Swift Package Manager and is part of the [Perfect](#) project.

## Building

Ensure you have installed and activated the latest Swift 3.1+ tool chain.

Add this package as a dependency in your Package.swift file.

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-CURL.git", majorVersion: 2)
```

## Linux Build Notes

Ensure that you have installed libcurl.

```
sudo apt-get install libcurl4-openssl-dev
```

## Usage

```
import PerfectCURL
```

This package uses a simple request/response model to access URL contents. Start by creating a `CURLRequest` object and configure it according to your needs, then ask it to perform the request and return a response. Responses are represented by `CURLResponse` objects.

Requests can be executed either synchronously - blocking the calling thread until the request completes, asynchronously - delivering the response to a callback, or through a Promise object - performing the request on a background thread giving you a means to chain additional tasks, poll, or wait for completion.

## Creating Requests

`CURLRequest` objects can be created with a URL and a series of options, or they can be created blank and then fully configured. `CURLRequest` provides the following initializers:

```
open class CURLRequest {
    // Init with a url and options array.
    public convenience init(_ url: String, options: [Option] = [])
    /// Init with url and one or more options.
    public convenience init(_ url: String, _ option1: Option, _ options: Option...)
    /// Init with array of options.
    public init(options: [Option] = [])
}
```

Options can be provided using either `Array<Option>` or variadic parameters. Options can also be directly added to the `CURLRequest.options` property before the request is executed.

## Configuring Requests

`CURLRequest` options are represented by the `CURLRequest.Option` enum. Each enum case will have zero or more associated values which indicate the parameters for the particular option. For example, the URL for the request could be indicated with the option `.url("https://httpbin.org/post")`. Most of the options that curl makes available are represented in the `CURLRequest.Option` enum. The full list of available options is presented near the end of this document.

## POST Data

POST field data, including file uploads, are added to request the same way other options are. The `.postField(POSTField)`, `.postData([UInt8])`, and `.postString(String)` enum cases will set the request's POST content. The `.postField` case can be added to a request multiple times as each instance represents one set of name/value pair. The `.postData` and `.postString` cases should be considered mutually exclusive to other post cases as adding either will overwrite any previously set POST content. Adding POST content data of any sort will automatically set the HTTP method to POST.

The `CURLRequest.POSTField` struct is defined as follows:

```
open class CURLRequest {
    public struct POSTField {
        /// Init with a name, value and optional mime-type.
        public init(name: String, value: String, mimeType: String? = nil)
        /// Init with a name, file path and optional mime-type.
        public init(name: String, filePath: String, mimeType: String? = nil)
    }
}
```

The example below creates a POST request and adds several name/value pairs as well as a file. The executed request will automatically have a "multipart/form-data" content type.

```
let json = try CURLRequest(url, .failOnError,
    .postField(.init(name: "key1", value: "value1")),
    .postField(.init(name: "key2", value: "value2")),
    .postField(.init(name: "file1", filePath: testFile.path, mimeType: "text/plain")))
    .perform().bodyJSON
```

## Fetching Responses

To perform a request, call one of the `CURLRequest.perform` or `CURLRequest.promise` functions. If the request is successful then you will be provided a `CURLResponse` object which can be used to get response data. If the request fails then a `CURLResponse.Error` will be thrown. A request may fail if it is unable to connect, times out, receives a

malformed response, or receives a HTTP response with a status code equal to or greater than 400 when the `.failOnError` option is given. If the `.failOnError` option is not given then any valid HTTP response will be a success, regardless of the response status code.

The three functions for obtaining a response are as follows:

```
public extension CURLRequest {
    /// Execute the request synchronously.
    /// Returns the response or throws an Error.
    func perform() throws -> CURLResponse
    /// Execute the request asynchronously.
    /// The parameter passed to the completion callback must be called to obtain the response or throw an Error.
    func perform(_ completion: @escaping (CURLResponse.Confirmation) -> ())
    /// Execute the request asynchronously.
    /// Returns a Promise object which can be used to monitor the operation.
    func promise() -> Promise<CURLResponse>
}
```

The first `CURLRequest.perform` function executes the request synchronously on the calling thread. The function call will block until the request succeeds or fails. On failure, a `CURLResponse.Error` will be thrown.

The second `CURLRequest.perform` function executes the request asynchronously on background threads as necessary. The parameter passed to this function is a callback which will be given a `CURLResponse.Confirmation` once the request completes or fails. Calling the confirmation parameter from within your callback will either return the `CURLResponse` or throw a `CURLResponse.Error`.

The third function, `CURLRequest.promise`, will return a `Promise<CURLResponse>` object which can be used to chain further activities and poll or wait for the request to complete. As with the other response generating functions, a `CURLResponse.Error` will be thrown if an error occurs. Information on the Promise object in general can be found in the [Perfect-Thread](#) documentation.

The following three example shows how each of the functions are used. Each will perform a request and convert the resulting response body from JSON into a `[String:Any]` dictionary.

• Synchronously fetch an API endpoint and decode it from JSON:

```
let url = "https://httpbin.org/get?pl=v1&p2=v2"
let json: [String:Any] = try CURLRequest(url).perform().bodyJSON
```

• Asynchronously fetch an API endpoint and decode it from JSON:

```
let url = "https://httpbin.org/post"
CURLRequest(url).perform {
    confirmation in
    do {
        let response = try confirmation()
        let json: [String:Any] = response.bodyJSON

    } catch let error as CURLResponse.Error {
        print("Failed: response code \(error.response.responseCode)")
    } catch {
        print("Fatal error \(error)")
    }
}
```

• Asynchronously fetch an API endpoint using a Promise and decode it from JSON:

```
let url = "https://httpbin.org/get?pl=v1&p2=v2"
if let json = try CURLRequest(url).promise().then { return try $0().bodyJSON }.wait() {
    ...
}
```

The three available functions ranked according to efficiency would be ordered as:

1. Asynchronous `perform`
2. Asynchronous `promise`
3. Synchronous `perform`

When performing CURL requests on a high-traffic server it is advised that one of the asynchronous response functions be used.

## Reset Request

A `CURLRequest` object can be reused for subsequent connections by calling the `.reset` function. Resetting a request will clear out any previously set options, including the target

URL. The `.reset` function accepts as an optional parameter new options with which the request will be reconfigured.

The reset declaration follows:

```
public extension CURLRequest {
    /// Reset the request. Clears all options so that the object can be reused.
    /// New options can be provided.
    func reset(_ options: [Option] = [])
    /// Reset the request. Clears all options so that the object can be reused.
    /// New options can be provided.
    func reset(_ option: Option, _ options: Option...)
}
```

Resetting a request will invalidate any previously executed `CURLResponse` objects. The reconfigured request should be reexecuted to obtain an updated `CURLResponse`.

## Response Data

A `CURLResponse` object provides access to the response's content body as either raw bytes, a String or as a JSON decoded `[String:Any]` dictionary. In addition, meta-information such as the response HTTP headers and status code can be retrieved.

Response body data is made available through a series of get-only `CURLResponse` properties:

```
public extension CURLResponse {
    /// The response's raw content body bytes.
    public var bodyBytes: [UInt8]
    /// Get the response body converted from UTF-8.
    public var bodyString: String
    /// Get the response body decoded from JSON into a [String:Any] dictionary.
    /// Invalid/non-JSON body data will result in an empty dictionary being returned.
    public var bodyJSON: [String:Any]
}
```

The remaining response data can be retrieved by calling one of the `CURLResponse.get` functions and passing in an enum value corresponding to the desired data. The enums indicating these values are separated into three groups, each according to the type of data that would be returned; one of String, Int or Double. The enum types are `CURLResponse.Info.StringValue`, `CURLResponse.Info.IntValue`, and `CURLResponse.Info.DoubleValue`. In addition, `get` functions are provided for directly pulling header values from the response.

```
public extension CURLResponse {
    /// Get an response info String value.
    func get(_ stringValue: Info.StringValue) -> String?
    /// Get an response info Int value.
    func get(_ intValue: Info.IntValue) -> Int?
    /// Get an response info Double value.
    func get(_ doubleValue: Info.DoubleValue) -> Double?
    /// Get a response header value. Returns the first found instance or nil.
    func get(_ header: Header.Name) -> String?
    /// Get a response header's values. Returns all found instances.
    func get(all header: Header.Name) -> [String]
}
```

For convenience properties have been added for pulling commonly requested data from a response such as `url` and `responseCode`.

The following examples show how to pull header and other meta-data from the response:

```
// get the response code
let code = response.get(.responseCode)
```

```
// get the response code using the accessor
let code = response.responseCode
```

```
// get the "Last-Modified" header from the response
if let lastMod = response.get(.lastModified) {
    ...
}
```

## Failures

When a failure occurs a `CURLResponse.Error` object will be thrown. This object provides the CURL error code which was generated (not that this is different from any HTTP

response code and is CURL specific). It also provides access to an error message string, and a CURLResponse object which can be used to further inquire about the resulting error.

Note that, depending on the options which were set on the request, the response object obtained after an error may not have any associated content body data.

`CURLResponse.Error` is defined as follows:

```
open class CURLResponse {
    /// An error thrown while retrieving a response.
    public struct Error: Swift.Error {
        /// The curl specific request response code.
        public let code: Int
        /// The string message for the curl response code.
        public let description: String
        /// The response object for this error.
        public let response: CURLResponse
    }
}
```

CURLRequest.Option List

The following is a list of the numerous `CURLRequest` options which can be set. Each enum case indicates the parameter types for the option. These enum values can be used when creating a new `CURLRequest` object or by adding them to an existing object's `.options` array property.

| CURLRequest.Option enum case    | Description                                                                                                                                                                      |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .url(String)                    | The URL for the request.                                                                                                                                                         |
| .port(Int)                      | Override the port for the request.                                                                                                                                               |
| .failOnError                    | Fail on http error codes >= 400.                                                                                                                                                 |
| .userPwd(String)                | Colon separated username/password string.                                                                                                                                        |
| .proxy(String)                  | Proxy server address.                                                                                                                                                            |
| .proxyUserPwd(String)           | Proxy server username/password combination.                                                                                                                                      |
| .proxyPort(Int)                 | Port override for the proxy server.                                                                                                                                              |
| .timeout(Int)                   | Maximum time in seconds for the request to complete. The default timeout is never.                                                                                               |
| .connectTimeout(Int)            | Maximum time in seconds for the request connection phase. The default timeout is 300 seconds.                                                                                    |
| .lowSpeedLimit(Int)             | The average transfer speed in bytes per second that the transfer should be below during <code>.lowSpeedLimit</code> seconds for the request to be too slow and abort.            |
| .lowSpeedTime(Int)              | The time in seconds that the transfer speed should be below the <code>.lowSpeedLimit</code> for the request to be considered too slow and aborted.                               |
| .range(String)                  | Range request value as a string in the format "X-Y", where either X or Y may be left out and X and Y are byte indexes                                                            |
| .resumeFrom(Int)                | The offset in bytes at which the request should start from.                                                                                                                      |
| .cookie(String)                 | Set one or more cookies for the request. Should be in the format "name=value". Separate multiple cookies with a semi-colon: "name1=value1; name2=value2".                        |
| .cookieFile(String)             | The name of the file holding cookie data for the request.                                                                                                                        |
| .cookieJar(String)              | The name of the file to which received cookies will be written.                                                                                                                  |
| .followLocation(Bool)           | Indicated that the request should follow redirects. Default is false.                                                                                                            |
| .maxRedirects(Int)              | Maximum number of redirects the request should follow. Default is unlimited.                                                                                                     |
| .maxConnects(Int)               | Maximum number of simultaneously open persistent connections that may cached for the request.                                                                                    |
| .autoReferer(Bool)              | When enabled, the request will automatically set the Referer: header field in HTTP requests when it follows a Location: redirect                                                 |
| .krbLevel(KBRLevel)             | Sets the kerberos security level for FTP. Value should be one of the following: <code>.clear</code> , <code>.safe</code> , <code>.confidential</code> or <code>.private</code> . |
| .addHeader(Header.Name, String) | Add a header to the request.                                                                                                                                                     |
| .addHeaders([(Header.Name,      |                                                                                                                                                                                  |

|                                     |                                                                                                                                                                                                                                                                                                                                     |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| String)))                           | Add a series of headers to the request.                                                                                                                                                                                                                                                                                             |
| .replaceHeader(Header.Name, String) | Add or replace a header.                                                                                                                                                                                                                                                                                                            |
| .removeHeader(Header.Name)          | Remove a default internally added header.                                                                                                                                                                                                                                                                                           |
| .sslCert(String)                    | Path to the client SSL certificate.                                                                                                                                                                                                                                                                                                 |
| .sslCertType(SSLFileType)           | Specifies the type for the client SSL certificate. Defaults to <code>.pem</code> .                                                                                                                                                                                                                                                  |
| .sslKey(String)                     | Path to client private key file.                                                                                                                                                                                                                                                                                                    |
| .sslKeyPwd(String)                  | Password to be used if the SSL key file is password protected.                                                                                                                                                                                                                                                                      |
| .sslKeyType(SSLFileType)            | Specifies the type for the SSL private key file.                                                                                                                                                                                                                                                                                    |
| .sslVersion(TLSMethod)              | Force the request to use a specific version of TLS or SSL.                                                                                                                                                                                                                                                                          |
| .sslVerifyPeer(Bool)                | Indicates whether the request should verify the authenticity of the peer's certificate.                                                                                                                                                                                                                                             |
| .sslVerifyHost(Bool)                | Indicates whether the request should verify that the server cert is for the server it is known as.                                                                                                                                                                                                                                  |
| .sslCAFilePath(String)              | Path to file holding one or more certificates which will be used to verify the peer.                                                                                                                                                                                                                                                |
| .sslCADirPath(String)               | Path to directory holding one or more certificates which will be used to verify the peer.                                                                                                                                                                                                                                           |
| .sslCiphers([String])               | Override the list of ciphers to use for the SSL connection.                                                                                                                                                                                                                                                                         |
| .sslPinnedPublicKey(String)         | File path to the pinned public key. When negotiating a TLS or SSL connection, the server sends a certificate indicating its identity. A public key is extracted from this certificate and if it does not exactly match the public key provided to this option, curl will abort the connection before sending or receiving any data. |
| .ftpPreCommands([String])           | List of (S)FTP commands to be run before the file transfer.                                                                                                                                                                                                                                                                         |
| .ftpPostCommands([String])          | List of (S)FTP commands to be run after the file transfer.                                                                                                                                                                                                                                                                          |
| .ftpPort(String)                    | Specifies the local connection port for active FTP transfers.                                                                                                                                                                                                                                                                       |
| .ftpResponseTimeout(Int)            | The time in seconds that the request will wait for FTP server responses.                                                                                                                                                                                                                                                            |
| .sshPublicKey(String)               | Path to the public key file used for SSH connections.                                                                                                                                                                                                                                                                               |
| .sshPrivateKey(String)              | Path to the private key file used for SSH connections.                                                                                                                                                                                                                                                                              |
| .httpMethod(HTTPMethod)             | HTTP method to be used for the request.                                                                                                                                                                                                                                                                                             |
| .postField(POSTField)               | Adds a single POST field to the request. Generally, multiple POST fields are added for a request.                                                                                                                                                                                                                                   |
| .postData([UInt8])                  | Raw bytes to be used for a POST request.                                                                                                                                                                                                                                                                                            |
| .postString(String)                 | Raw string data to be used for a POST request.                                                                                                                                                                                                                                                                                      |
| .mailFrom(String)                   | Specifies the sender's address when performing an SMTP request.                                                                                                                                                                                                                                                                     |
| .mailRcpt(String)                   | Specifies the recipient when performing an SMTP request. Multiple recipients may be specified by using this option multiple times.                                                                                                                                                                                                  |

## CURLResponse.Info List

The lists which follow describe the `CURLResponse.Info` cases which are used with the `CURLResponse.get` function to retrieve response information. The lists are grouped according to the type of data which would be returned; `StringValue`, `IntValue`, and `DoubleValue`, respectively.

| <b>CURLResponse.Info.StringValue enum case</b> | <b>Description</b>                                                                                                                                                               |
|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .url                                           | The effective URL for the request/response. This is ultimately the URL from which the response data came from. This may differ from the request's URL in the case of a redirect. |
| .ftpEntryPath                                  | The initial path that the request ended up at after logging in to the FTP server.                                                                                                |
| .redirectURL                                   | The URL that the request <i>would have</i> been redirected to.                                                                                                                   |
| .localIP                                       | The local IP address that the request used most recently.                                                                                                                        |
| .primaryIP                                     | The remote IP address that the request most recently connected to.                                                                                                               |
| .contentType                                   | The content type for the request. This is read from the "Content-Type" header.                                                                                                   |

| <b>CURLResponse.Info.IntValue enum case</b> | <b>Description</b>                                                                                                                     |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| .responseCode                               | The last received HTTP, FTP or SMTP response code.                                                                                     |
| .headerSize                                 | The total size in bytes of all received headers.                                                                                       |
| .requestSize                                | The total size of the issued request in bytes. This will indicate the cumulative total of all requests sent in the case of a redirect. |
| .sslVerifyResult                            | The result of the SSL certificate verification.                                                                                        |
| .redirectCount                              | The total number of redirections that were followed.                                                                                   |
| .httpConnectCode                            | The last received HTTP proxy response code to a CONNECT request.                                                                       |
| .osErrno                                    | The OS level errno which may have triggered a failure.                                                                                 |
| .numConnects                                | The number of connections that the request had to make in order to produce a response.                                                 |
| .primaryPort                                | The remote port that the request most recently connected to                                                                            |
| .localPort                                  | The local port that the request used most recently                                                                                     |

| <b>CURLResponse.Info.DoubleValue enum case</b> | <b>Description</b>                                                                                                                                          |
|------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .totalTime                                     | The total time in seconds for the previous request.                                                                                                         |
| .nameLookupTime                                | The total time in seconds from the start until the name resolving was completed.                                                                            |
| .connectTime                                   | The total time in seconds from the start until the connection to the remote host or proxy was completed.                                                    |
| .preTransferTime                               | The time, in seconds, it took from the start until the file transfer is just about to begin.                                                                |
| .sizeUpload                                    | The total number of bytes uploaded.                                                                                                                         |
| .sizeDownload                                  | The total number of bytes downloaded.                                                                                                                       |
| .speedDownload                                 | The average download speed measured in bytes/second.                                                                                                        |
| .speedUpload                                   | The average upload speed measured in bytes/second.                                                                                                          |
| .contentLengthDownload                         | The content-length of the download. This value is obtained from the Content-Length header field.                                                            |
| .contentLengthUpload                           | The specified size of the upload.                                                                                                                           |
| .startTransferTime                             | The time, in seconds, it took from the start of the request until the first byte was received.                                                              |
| .redirectTime                                  | The total time, in seconds, it took for all redirection steps include name lookup, connect, pre-transfer and transfer before final transaction was started. |
| .appConnectTime                                | The time, in seconds, it took from the start until the SSL/SSH connect/handshake to the remote host was completed.                                          |

Perfect provides XML & HTML parsing support via the Perfect-XML module.

## Getting Started

In addition to the PerfectLib, you will need the Perfect-XML dependency in the Package.swift file:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-XML.git", majorVersion: 2)
```

In each Swift source file that references the XML classes, include the import directive:

```
import PerfectXML
```

## macOS Build Notes

If you receive a compile error that says the following, you need to install and link libxml2.

```
note: you may be able to install libxml-2.0 using your system-packager:

    brew install libxml2

Compile Swift Module 'PerfectXML' (2 sources)
<module-includes>:1:9: note: in file included from <module-includes>:1:
#import "libxml2.h"
```

To install and link libxml2 with Homebrew, use the following two commands:

```
brew install libxml2
brew link --force libxml2
```

## Linux Setup note

Ensure that you have installed libxml2-dev and pkg-config.

```
sudo apt-get install libxml2-dev pkg-config
```

## Parsing an XML string

Instantiate an XDocument object with your XML string:

```
let xDoc = XDocument(fromSource: <XMLSource>)
```

Now you can get the root node of the XML structure by using the documentElement property.

## Convert the node tree to String

Converting the XML node tree to a string, with optional pretty-print formatting:

```
let xDoc = XDocument(fromSource: rssXML)
let prettyString = xDoc?.string(pretty: true)
let plainString = xDoc?.string(pretty: false)
```

## Working with Nodes

XML is a structured document standard consisting of nodes in the format `<A>B</A>`. Each node has a tag name and either a value, or sub nodes we call "children". Each node has several important properties:

- nodeValue
- nodeName
- parentNode
- childNodes

Each node also has a `getElementsByTagName` method that recursively searches through it and its children to return an array of all nodes that have that name. This method makes it easy to find a single value in the XML file.

To demonstrate, this recursive function iterates through all elements in the node:

```

func printChildrenName(_ xNode: XNode) {

    // try treating the current node as a text node
    guard let text = xNode as? XText else {

        // print out the node info
        print("Name:\t\(xNode.nodeName)\tType:\t\(xNode.nodeType)\t(children)\n")

        // find out children of the node
        for n in xNode.childNodes {

            // call the function recursively and take the same produces
            printChildrenName(n)
        }
        return
    }

    // it is a text node and print it out
    print("Name:\t\(xNode.nodeName)\tType:\t\(xNode.nodeType)\tValue:\t\text.nodeValue!\)\n")
}

printChildrenName( xDoc!)

```

## Access Node by Tag Name

The snippet below shows the method of `getElementsByTagName` in a general manner:

```

func testTag(tagName: String) {

    // use .getElementsByTagName to get this node.
    // check if there is such a tag in the xml document
    guard let node = xDoc?.documentElement?.getElementsByTagName(tagName) else {
        print("There is no such a tag: `\(tagName)`\n")
        return
    }

    // if is, get the first matched
    guard let value = node.first?.nodeValue else {
        print("Tag `\(tagName)` has no value\n")
        return
    }

    // show the node value
    print("Tag '\(tagName)' has a value of '\(value)'\n")
}

testTag(tagName: "link")
testTag(tagName: "description")

```

## Access Node by ID

Alternatively querying a node by its id using `.getElementById()` :

```

func testID(id: String) {

    // Access node by its id, if available
    guard let node = xDoc?.getElementById(id) else {
        print("There is no such a id: `\(id)`\n")
        return
    }

    guard let value = node.nodeValue else {
        print("id `\(id)` has no value\n")
        return
    }

    print("id `\(id)` has a value of `\(value)`\n")
}

testID(id: "rssID")
testID(id: "xmlID")

```

## XElement

The method `.getElementsByTagName()` returns an array of nodes of type XElement.

The following code demonstrates how to iterate all element in this array:

```

func showItems() {
    // get all items with tag name of "item"
    let feedItems = xDoc?.documentElement?.getElementsByTagName("item")

    // checkout how many items indeed.
    let itemCount = feedItems?.count
    print("There are totally \(itemCount!) Items Found\n")

    // iterate all items in the result set
    for item in feedItems! {
        let title = item.getElementsByTagName("title").first?.nodeValue
        let link = item.getElementsByTagName("link").first?.nodeValue
        let description = item.getElementsByTagName("description").first?.nodeValue
        print("Title: \(title!)\tLink: \(link!)\tDescription: \(description!)\n")
    }
}

showItems()

```

## Relationships of a Node Family

PerfectXML provides a convenient way to access all relationships of a specific XML node: Parent, Siblings & Children.

### Parent of a Node

Parent node can be accessed using `parentNode` :

```

func showParent(tag: String) {

    guard let node = xDoc?.documentElement?.getElementsByTagName(tag).first else {
        print("There is no such a tag `\(tag)`.\n")
        return
    }

    // Accessing the parent node
    guard let parent = node.parentNode else {
        print("Tag `\(tag)` is the root node.\n")
        return
    }
    let name = parent.nodeName
    print("Parent of `\(tag)` is `\(name)`\n")
}

showParent(tag: "link")

```

## Node Siblings

Each XML node can have two siblings: `previousSibling` and `nextSibling` .

```
func showSiblings (tag: String) {

    let node = xDoc?.documentElement?.getElementsByTagName(tag).first

    // Check the previous sibling of current node
    let previousNode = node?.previousSibling
    var name = previousNode?.nodeName
    var value = previousNode?.nodeValue
    print("Previous Sibling of \(tag) is \(name!)\t\(value!)\n")

    // Check the next sibling of current node
    let nextNode = node?.nextSibling
    name = nextNode?.nodeName
    value = nextNode?.nodeValue
    print("Next Sibling of \(tag) is \(name!)\t\(value!)\n")

}

showSiblings(tag: "link")
showSiblings(tag: "description")
```

## First & Last Child

If an XML node has a child, `.firstChild` and `.lastChild` can be used:

```
func firstLast() {
    let node = xDoc?.documentElement?.getElementsByTagName("channel").first

    /// retrieve the first child
    let firstChild = node?.firstChild
    var name = firstChild?.nodeName
    var value = firstChild?.nodeValue
    print("First Child of Channel is \(name!)\t\(value!)\n")

    /// retrieve the last child
    let lastChild = node?.lastChild
    name = lastChild?.nodeName
    value = lastChild?.nodeValue
    print("Last Child of Channel is \(name!)\t\(value!)\n")

}

firstLast()
```

## Node Attributes

Any XML node or element can have attributes:

```
<node attribute1="value of attribute1" attribute2="value of attribute2">
</node>
```

Node method `.getAttribute(name: String)` provides the functionality of accessing attributes:

```
func showAttributes() {
    let node = xDoc?.documentElement?.getElementsByTagName("title").first

    /// get some attributes of a node
    let att1 = node?.getAttribute(name: "attribute1")
    print("attribute1 of title is \(att1)\n")

    let att2 = node?.getAttributeNode(name: "attribute2")
    print("attribute2 of title is \(att2?.value)\n")

}

showAttributes()
```

# Namespaces

XML namespaces are used for providing uniquely named elements and attributes in an XML document. An XML instance may contain element or attribute names from more than one XML vocabulary. If each vocabulary is given a namespace, the ambiguity between identically named elements or attributes can be resolved.

Both `.getElementsByTagName()` and `.getAttributeNode()` have namespace versions: `.getElementsByTagNameNS()` and `.getAttributeNodeNS()`. In these cases `namespaceURI` and `localName` are required.

The following code demonstrates the usage of `.getElementsByTagNameNS()` and `.getNamedItemNS()`:

```
func showNamespaces(){
    let deeper = xDoc?.documentElement?.getElementsByTagName("deeper").first
    let atts = deeper?.firstChild?.attributes;
    let item = atts?.getNamedItemNS(namespaceURI: "foo:bar", localName: "atr2")
    print("Namespace of deeper has an attribute of \(item?.nodeValue)\n")

    let foos = xDoc?.documentElement?.getElementsByTagNameNS(namespaceURI: "foo:bar", localName: "fool")
    var count = foos?.count
    let node = foos?.first
    let name = node?.nodeName
    let localName = node?.localName
    let prefix = node?.prefix
    let nuri = node?.namespaceURI

    print("Namespace of 'foo:bar' has \(count!) element:\n")
    print("Node Name: \(name!)\n")
    print("Local Name: \(localName!)\n")
    print("Prefix: \(prefix!)\n")
    print("Namespace URI: \(nuri!)\n")

    let children = node?.childNodes

    count = children?.count

    let a = node?.firstChild
    let b = node?.lastChild

    let na = a?.nodeName
    let nb = b?.nodeName
    let va = a?.nodeValue
    let vb = b?.nodeValue

    print("This node also has \(count!) children.\n")
    print("The first one is called '\(na!)' with value of '\(va!)\n")
    print("And the last one is called '\(nb!)' with value of '\(vb!)\n")
}

showNamespaces()
```

# XPath

XPath (XML Path Language) is a query language for selecting nodes from an XML document. In addition, XPath may be used to compute values (e.g. strings, numbers, or Boolean values) from the content of an XML document.

The following code demonstrates extracting a specific path:

```
func showXPath(xpath: String) {
    /// Use .extract() method to deal with the XPath request.
    let pathResource = xDoc?.extract(path: xpath)
    print("XPath '\(xpath)':\n\(pathResource!)\n")
}

showXPath(xpath: "/rss/channel/item")
showXPath(xpath: "/rss/channel/title/@attribute1")
showXPath(xpath: "/rss/channel/link/text()")
showXPath(xpath: "/rss/channel/item[2]/deeper/foo:bar")
```

# Perfect INI File Parser

This project provides an express parser for [INI](#) files.

This package builds with Swift Package Manager of Swift 3.1 Tool Chain and is part of the [Perfect](#) project but can be used as an independent module.

## Quick Start

---

Configure Package.swift:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-INIParser.git", majorVersion: 1)
```

Import library into your code:

```
import INIParser
```

Load the objective INI file by initializing a `INIParser` object:

```
let ini = try INIParser("/path/to/somefile.ini")
```

Then it should be possible to access variables inside the file.

### Variables with Specific Section

For most regular lines under a certain section, use `sections` attribute of `INIParser`. Take example:

```
[GroupA]  
myVariable = myValue
```

Then `let v = ini.sections["[GroupA]"]?["myVariable"]` will get the value as `"myValue"`.

### Variables without Section

However, some ini files may not have any available sections but directly put all variables together:

```
freeVar1 = 1
```

In this case, call `anonymousSection` to load the corresponding value:

```
let v = ini.anonymousSection["freeVar1"]
```

## Perfect's Zip Toolkit

Perfect provides a wrapper around the minizip C library and implements a set of convenience functions for use when the PerfectZip package is imported.

## Relevant Examples

---

- [Perfect-Zip-Example](#)

## Getting Started

---

On MacOS, install minizip using Homebrew:

```
brew install minizip
```

On Ubuntu, install minizip:

```
apt-get install libminizip-dev
```

In addition to the PerfectLib, you will need the Perfect-Zip dependency in the Package.swift file:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-Zip.git", majorVersion: 2)
```

# Using Perfect Zip

---

The two main functions of the Perfect Zip module are to zip files, or unzip a zip file.

## Declaring an instance of the Zip class

Before initiating compression or decompression, an instance of the class must be created:

```
let myVar = zip()
```

## Zip

To compress a file, use the `.zipFiles(...)` method.

```
zipFiles(  
  paths: [String],  
  zipFilePath: String,  
  overwrite: Bool,  
  password: String?  
) -> ZipStatus
```

This method returns the success/fail status of the operation as a `ZipStatus` enum.

### Parameters:

- **paths:** The array of file paths to add to the zip file
- **zipFilePath:** The path and filename of the destination zip file
- **overwrite:** A boolean declaring the behaviour to attempt when the destination zip file exists
- **password:** The password string to use for password-protecting the zip file. Optional. Leave empty or omit to create an unprotected zip file.

## UnZip

To decompress a file, use the `.unzipFile(...)` method.

```
unzipFile(  
  source: String,  
  destination: String,  
  overwrite: Bool,  
  password: String = ""  
) -> ZipStatus
```

This method returns the success/fail status of the operation as a `ZipStatus` enum.

### Parameters:

- **source:** The file path of zipped file
- **destination:** The path to the directory into which the contents of the zip file are to be placed
- **overwrite:** A boolean declaring the behaviour to attempt when the destination directory or file exists
- **password:** The password string to use for decrypting a password-protected zip file. Optional.

## ZipStatus

The `ZipStatus` values are as follows:

- `.FileNotFound`
- `.UnzipFail`
- `.ZipFail`
- `.ZipCannotOverwrite`
- `.ZipSuccess`

The enum has a `.description` variable which returns human-readable descriptors of each enum value.

- `.FileNotFound` - "File not found."
- `.UnzipFail` - "Failed to unzip file."
- `.ZipFail` - "Failed to zip file."

- `.ZipCannotOverwrite` - **"Cannot overwrite destination file."**
- `.ZipSuccess` - **"Success."**

## Usage

---

The following will zip the specified directory:

```
import PerfectZip

let myZip = Zip()

let thisZipFile = "/path/to/ZipFile.zip"
let sourceDir = "/path/to/files/"

let ZipResult = myZip.zipFiles(
    paths: [sourceDir],
    zipFilePath: thisZipFile,
    overwrite: true, password: ""
)

print("ZipResult Result: \(ZipResult.description)")
```

To unzip a file:

```
import PerfectZip

let myZip = Zip()

let sourceDir = "/path/to/files/"
let thisZipFile = "/path/to/ZipFile.zip"

let UnZipResult = myZip.unzipFile(
    source: thisZipFile,
    destination: sourceDir,
    overwrite: true
)

print("Unzip Result: \(UnZipResult.description)")
```

## Crypto

The Perfect-Crypto package is a general purpose cryptography library built on OpenSSL. It provides high level objects for dealing with the following cryptographic tasks:

- Message digests and hashes, sign/verify
- Cipher based encryption/decryption
- Random data generation of arbitrary byte lengths
- HMAC key generation
- PEM format public/private key reading
- JWT (JSON Web Token) creation and validation

It also provides some encoding related functions which are generally useful but are commonly used along with cryptography, particularly when converting binary data to and from a character printable state.

To use the functionality of this package ensure you `import PerfectCrypto`.

## Initialization

---

The underlying cryptography library requires a one-time initialization to be performed before any related functions are used. This is generally done once when your program starts before performing any actual tasks. Calling `PerfectCrypto.isInitialized` will initialize the library and return true. Calling this more than once is fine, however one would generally only call this once when boot-strapping your application.

## Extensions

---

Much of the functionality provided by this package is through extensions on several of the Swift builtin types, namely: `String`, `Array<UInt8>`, `UnsafeRawBufferPointer`, and `UnsafeMutableRawBufferPointer`. The extensions mainly consist of adding `encode/decode`, `encrypt/decrypt`, `sign/verify` and `digest` functions for these types. Others provide convenience functions for dealing with raw binary data, generating random data, etc.

The extensions are listed below in order of "convenience". The higher level functions are listed first and the more efficient "unsafes" are listed at the end.

## String

The String extensions are divided into two groups. The first group provides the encode, decode and digest functions for Strings.

```
public extension String {
    /// Decode the String into an array of bytes using the indicated encoding.
    /// The string's UTF8 characters are decoded.
    func decode(_ encoding: Encoding) -> [UInt8]?
    /// Encode the String into an array of bytes using the indicated encoding.
    /// The string's UTF8 characters are encoded.
    func encode(_ encoding: Encoding) -> [UInt8]?
    /// Perform the digest algorithm on the String's UTF8 bytes.
    func digest(_ digest: Digest) -> [UInt8]?
    /// Sign the String data into an array of bytes using the indicated algorithm and key.
    func sign(_ digest: Digest, key: Key) -> [UInt8]?
    /// Verify the signature against the String data.
    /// Returns true if the signature is verified. Returns false otherwise.
    func verify(_ digest: Digest, signature: [UInt8], key: Key) -> Bool
    /// Encrypt this buffer using the indicated cipher, password, and salt.
    /// The string's UTF8 characters are encoded.
    /// Resulting data is in PEM encoded CMS format.
    func encrypt(_ cipher: Cipher,
                 password: String,
                 salt: String,
                 keyIterations: Int = 2048,
                 keyDigest: Digest = .md5) -> String?
    /// Decrypt this PEM encoded CMS buffer using the indicated password and salt.
    /// Resulting decrypted data must be valid UTF-8 characters or the operation will fail.
    func decrypt(_ cipher: Cipher,
                 password: String,
                 salt: String,
                 keyIterations: Int = 2048,
                 keyDigest: Digest = .md5) -> String?
}
```

The `encode` and `decode` functions are given an `Encoding` type and will return the result as a `[UInt8]` or nil if the input data was invalid for that particular encoding. The encoding type must be one of the valid `Encoding` enums. For example `.hex`, or `.base64url`.

This example shows how one would encode a String as base64 and decode it back into the original.

```
let testStr = "Räksmörgåsen"
if let encoded = testStr.encode(.base64),
    let decoded = encoded.decode(.base64),
    let decodedStr = String(validatingUTF8: decoded) {
    print(decodedStr)
    // Räksmörgåsen
}
```

The `digest` function allows one to perform a message digest operation on the String's characters. The digest data is returned as a `[UInt8]`. nil is returned if the indicated encoding is not supported by the underlying system.

This example shows a sha256 digest calculated for a String. The resulting value is then converted into a printable hexadecimal String.

```
let testStr = "Hello, world!"
if let digestBytes = testStr.digest(.sha256),
    let hexBytes = digestBytes.encode(.hex),
    let hexBytesStr = String(validatingUTF8: hexBytes) {
    print(hexBytesStr)
    // 315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
}
```

The `sign` and `verify` functions permit a block of data to be cryptographically signed using a specified key and then later verified to ensure that the data has not been modified in any way. Signing can be done with either HMAC, RSA or EC type keys.

The `encrypt` and `decrypt` functions accept a Cipher, password and salt value and encrypt/decrypt the data. These encryption functions produce and consume PEM encoded Cryptographic Message Syntax (CMS) data.

```

let cipher = Cipher.aes_256_cbc
let password = "this is a good pw"
let salt = "this is a salty salt"
let data = "ABCDEFGHijklMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
guard let result = data.encrypt(cipher, password: password, salt: salt) else {
    return // fatal error
}
guard let decryptedData = result.decrypt(cipher, password: password, salt: salt) else {
    return // decryption error
}
// decryptedData == data

```

The second group of String extensions add convenience functions for creating a String from *non null terminated* UTF-8 characters. These characters can be given through either a `[UInt8]` or `UnsafeRawBufferPointer`.

```

public extension String {
    /// Construct a string from a UTF8 character array.
    /// The array's count indicates how many characters are to be converted.
    /// Returns nil if the data is invalid.
    init?(validatingUTF8 a: [UInt8])
    /// Construct a string from a UTF8 character pointer.
    /// Character data does not need to be null terminated.
    /// The buffer's count indicates how many characters are to be converted.
    /// Returns nil if the data is invalid.
    init?(validatingUTF8 ptr: UnsafeRawBufferPointer?)
    /// Obtain a buffer pointer for the String's UTF8 characters.
    func withBufferPointer<Result>(_ body: (UnsafeRawBufferPointer) throws -> Result) rethrows -> Result
}

```

The final function, `withBufferPointer`, is useful for obtaining an `UnsafeRawBufferPointer` containing the String's UTF8 characters. The buffer is valid only within the provided closure/function's body.

## Array<UInt8>

The extensions on Array are only provided for those containing `UInt8` values. These functions provide encode/decode, encrypt/decrypt and digest operations as well as an initializer which allows creating an array of randomly generated data.

```

public extension Array where Element: Octal {
    /// Encode the Array into An array of bytes using the indicated encoding.
    func encode(_ encoding: Encoding) -> [UInt8]?
    /// Decode the Array into an array of bytes using the indicated encoding.
    func decode(_ encoding: Encoding) -> [UInt8]?
    /// Digest the Array data into an array of bytes using the indicated algorithm.
    func digest(_ digest: Digest) -> [UInt8]?
    /// Sign the Array data into an array of bytes using the indicated algorithm and key.
    func sign(_ digest: Digest, key: Key) -> [UInt8]?
    /// Verify the array against the signature.
    /// Returns true if the signature is verified. Returns false otherwise.
    func verify(_ digest: Digest, signature: [UInt8], key: Key) -> Bool
    /// Decrypt this buffer using the indicated cipher, key an iv (initialization vector).
    func encrypt(_ cipher: Cipher, key: [UInt8], iv: [UInt8]) -> [UInt8]?
    /// Encrypt this buffer using the indicated cipher, key an iv (initialization vector).
    func decrypt(_ cipher: Cipher, key: [UInt8], iv: [UInt8]) -> [UInt8]?
    /// Encrypt this buffer using the indicated cipher, password, and salt.
    /// Resulting data is PEM encoded CMS format.
    func encrypt(_ cipher: Cipher,
                 password: [UInt8],
                 salt: [UInt8],
                 keyIterations: Int = 2048,
                 keyDigest: Digest = .md5) -> [UInt8]?
    /// Decrypt this PEM encoded CMS buffer using the indicated password and salt.
    func decrypt(_ cipher: Cipher,
                 password: [UInt8],
                 salt: [UInt8],
                 keyIterations: Int = 2048,
                 keyDigest: Digest = .md5) -> [UInt8]?
}

```

The encode, decode and digest functions work identically to the String versions, except that the input data is a `[UInt8]`. The encode and decode functions are given an `Encoding` type and will return the result as a `[UInt8]` or nil if the input data was invalid for that particular encoding. The encoding type must be one of the valid `Encoding` enums. For

example `.hex` , or `.base64url` .

The `digest` function allows one to perform a message digest operation on the array values. The digest data is returned as a `[UInt8]` . nil is returned if the indicated digest is not supported by the underlying system.

The first set of `encrypt` and `decrypt` functions will encrypt/decrypt the array data based on the indicated `Cipher` enum value. These operations also require input for the key and initialization vector (iv) parameters.

The second set of `encrypt` and `decrypt` functions accept a Cipher, password and salt value and encrypt/decrypt the data. These encryption functions produce and consume PEM encoded Cryptographic Message Syntax (CMS) data.

The sizes of the key and iv arrays will differ based the cipher in use. `Cipher` enum values provide properties for the individual cipher's `blockSize` , `keyLength` and `ivLength` . All of these values are indicated in bytes.

The snippet below will use the `.aes_256_cbc` cipher to encrypt an array of random bytes. The key and the iv for this example are also generated at random based on the sizes required for the cipher.

```
let cipher = Cipher.aes_256_cbc
// the data which will be encrypted
let random = [UInt8](randomCount: 2048)
// The key value for the encrypt/decrypt
let key = [UInt8](randomCount: cipher.keyLength)
// Initialization vector
let iv = [UInt8](randomCount: cipher.ivLength)

if let encrypted = random.encrypt(cipher, key: key, iv: iv),
    let decrypted = encrypted.decrypt(cipher, key: key, iv: iv) else {

    for (a, b) in zip(decrypted, random) {
        (a == b)
    }
}
```

Arrays containing randomly generated bytes can be produced with the following extension.

```
public extension Array where Element: Octal {
    /// Creates a new array containing the specified number of a single random values.
    init(randomCount count: Int)
}
```

Each byte in the resulting array will be generated randomly. The example below generates 16 bytes of random data and converts it to a printable base64 string.

```
// generate 16 random bytes of data
let random = [UInt8](randomCount: 16)
if let base64 = random.encode(.base64),
    let base64Str = String(validatingUTF8: base64) {
    print(base64Str)
}
```

## UnsafeXX Extensions

The extensions provided on `UnsafeMutableRawBufferPointer` and `UnsafeRawBufferPointer` give lower level access to the buffers used for the crypto operations. They provide more efficient behaviour but all the same operations as the Array extension counterparts.

### UnsafeMutableRawBufferPointer

The extensions on `UnsafeMutableRawBufferPointer` permit one to generate or fill a buffer of randomly generated data.

```
public extension UnsafeMutableRawBufferPointer {
    /// Allocate memory for `size` bytes with word alignment from the encryption library's
    /// random number generator.
    static func allocateRandom(count size: Int) -> UnsafeMutableRawBufferPointer?
    /// Initialize the buffer with random bytes.
    func initializeRandom()
}
```

The static `allocateRandom` function will return a newly allocated buffer containing the randomized data. You must `deallocate` this buffer just as if you had called the standard `allocate` function.

The `initializeRandom` function will fill the buffer (which is assumed to have been allocated through some other means) with randomized data, up to the buffer's `.count` value.

## UnsafeRawBufferPointer

These extensions provide the same `allocateRandom` static function as above, as well as all of the encode/decode, encrypt/decrypt and digest functions.

```
public extension UnsafeRawBufferPointer {
    /// Allocate memory for `size` bytes with word alignment from the encryption library's
    /// random number generator.
    static func allocateRandom(count size: Int) -> UnsafeRawBufferPointer?
    /// Encode the buffer using the indicated encoding.
    /// The return value must be deallocated by the caller.
    func encode(_ encoding: Encoding) -> UnsafeMutableRawBufferPointer?
    /// Decode the buffer using the indicated encoding.
    /// The return value must be deallocated by the caller.
    func decode(_ encoding: Encoding) -> UnsafeMutableRawBufferPointer?
    /// Digest the buffer using the indicated algorithm.
    /// The return value must be deallocated by the caller.
    func digest(_ digest: Digest) -> UnsafeMutableRawBufferPointer?
    /// Sign the buffer using the indicated algorithm and key.
    /// The return value must be deallocated by the caller.
    func sign(_ digest: Digest, key: Key) -> UnsafeMutableRawBufferPointer?
    /// Verify the signature against the buffer.
    /// Returns true if the signature is verified. Returns false otherwise.
    func verify(_ digest: Digest, signature: UnsafeRawBufferPointer, key: Key) -> Bool
    /// Encrypt this buffer using the indicated cipher, key and iv (initialization vector).
    /// Returns a newly allocated buffer which must be freed by the caller.
    func encrypt(_ cipher: Cipher, key: UnsafeRawBufferPointer, iv: UnsafeRawBufferPointer) -> UnsafeMutableRawBufferPointer?
    /// Decrypt this buffer using the indicated cipher, key and iv (initialization vector).
    /// Returns a newly allocated buffer which must be freed by the caller.
    func decrypt(_ cipher: Cipher, key: UnsafeRawBufferPointer, iv: UnsafeRawBufferPointer) -> UnsafeMutableRawBufferPointer?
    /// Encrypt this buffer to PEM encoded CMS format using the indicated cipher, password, and salt.
    /// Returns a newly allocated buffer which must be freed by the caller.
    func encrypt(_ cipher: Cipher,
                 password: UnsafeRawBufferPointer,
                 salt: UnsafeRawBufferPointer,
                 keyIterations: Int = 2048,
                 keyDigest: Digest = .md5) -> UnsafeMutableRawBufferPointer?
    /// Decrypt this PEM encoded CMS buffer using the indicated password and salt.
    /// Returns a newly allocated buffer which must be freed by the caller.
    func decrypt(_ cipher: Cipher,
                 password: UnsafeRawBufferPointer,
                 salt: UnsafeRawBufferPointer,
                 keyIterations: Int = 2048,
                 keyDigest: Digest = .md5) -> UnsafeMutableRawBufferPointer?
}
```

It's very important to adhere to the proper memory ownership guidelines when using these functions. Any `UnsafeMutableRawBufferPointer` returned by one of these functions must at some point be deallocated by the caller. All such return buffers will properly have their `.count` properties set to indicate their size. No `UnsafeRawBufferPointer` passed into a function will be deallocated or otherwise modified.

## JSON Web Tokens (JWT)

This crypto package provides an means for creating new JWT tokens and validating existing tokens.

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA. Source: [JWT](#).

New JWT tokens are created through the `JWTCreator` object.

```

/// Creates and signs new JWT tokens.
public struct JWTCreator {
    /// Creates a new JWT token given a payload.
    /// The payload can then be signed to generate a JWT token string.
    public init?(payload: [String:Any])
    /// Sign and return a new JWT token string using an HMAC key.
    /// Additional headers can be optionally provided.
    /// Throws a JWT.Error.signingError if there is a problem generating the token string.
    public func sign(alg: JWT.Alg, key: String, headers: [String:Any] = [:]) throws -> String
    /// Sign and return a new JWT token string using the given key.
    /// Additional headers can be optionally provided.
    /// The key type must be compatible with the indicated `algo`.
    /// Throws a JWT.Error.signingError if there is a problem generating the token string.
    public func sign(alg: JWT.Alg, key: Key, headers: [String:Any] = [:]) throws -> String
}

```

Existing JWT tokens can be validated through the `JWTVerifier` object.

```

/// Accepts a JWT token string and verifies its structural validity and signature.
public struct JWTVerifier {
    /// The headers obtained from the token.
    public var header: [String:Any]
    /// The payload carried by the token.
    public var payload: [String:Any]
    /// Create a JWTVerifier given a source string in the "aaaa.bbbb.cccc" format.
    /// Returns nil if the given string is not a valid JWT.
    /// *Does not perform verification in this step.* Call `verify` with your key to validate.
    /// If verification succeeds then the `.headers` and `.payload` properties can be safely accessed.
    public init?(_ jwt: String)
    /// Verify the token based on the indicated algorithm and HMAC key.
    /// Throws a JWT.Error.verificationError if any aspect of the token is incongruent.
    /// Returns without any error if the token was able to be verified.
    /// The parameter `algo` must match the token's "alg" header.
    public func verify(algo: JWT.Alg, key: String) throws
    /// Verify the token based on the indicated algorithm and key.
    /// Throws a JWT.Error.verificationError if any aspect of the token is incongruent.
    /// Returns without any error if the token was able to be verified.
    /// The parameter `algo` must match the token's "alg" header.
    /// The key type must be compatible with the indicated `algo`.
    public func verify(algo: JWT.Alg, key: Key) throws
}

```

The following example will create and then verify a token using the "HS256" alg scheme.

```

let name = "John Doe"
let tstPayload = ["sub": "1234567890", "name": name, "admin": true] as [String : Any]
let secret = "secret"
guard let jwt1 = JWTCreator(payload: tstPayload) else {
    return // fatal error
}
let token = try jwt1.sign(alg: .hs256, key: secret)
guard let jwt = JWTVerifier(token) else {
    return // fatal error
}
try jwt.verify(algo: .hs256, key: HMACKey(secret))
let fndName = jwt.payload["name"] as? String
// name == fndName!

```

## Algorithms

The available encoding, digest and cipher algorithms are enumerated in the `Encoding`, `Digest` and `Cipher` enums, respectively.

- Encoding: base64, base64url, hex
- Digest: md4, md5, sha, sha1, dss, dss1, ecdsa, sha224, sha256, sha384, sha512, ripemd160, whirlpool, custom(String)
- Cipher: des\_ecb, des\_edc, des\_edc3, des\_edc\_ecb, des\_edc3\_ecb, des\_cfb64, des\_cfb1, des\_cfb8, des\_edc\_cfb64, des\_edc3\_cfb1, des\_edc3\_cfb8, des\_ofb, des\_edc\_ofb, des\_edc3\_ofb, des\_cbc, des\_edc\_cbc, des\_edc3\_cbc, desx\_cbc, des\_edc3\_wrap, rc4, rc4\_40, rc4\_hmac\_md5, rc2\_ecb, rc2\_cbc, rc2\_40\_cbc, rc2\_64\_cbc, rc2\_cfb64, rc2\_ofb, bf\_ecb, bf\_cbc, bf\_cfb64, bf\_ofb, cast5\_ecb, cast5\_cbc, cast5\_cfb64, cast5\_ofb, aes\_128\_ecb, aes\_128\_cbc, aes\_128\_cfb1, aes\_128\_cfb8, aes\_128\_cfb128, aes\_128\_ofb, aes\_128\_ctr, aes\_128\_ccm, aes\_128\_gcm, aes\_128\_xts, aes\_128\_wrap, aes\_192\_ecb, aes\_192\_cbc, aes\_192\_cfb1, aes\_192\_cfb8, aes\_192\_cfb128, aes\_192\_ofb, aes\_192\_ctr, aes\_192\_ccm, aes\_192\_gcm, aes\_192\_wrap, aes\_256\_ecb, aes\_256\_cbc, aes\_256\_cfb1, aes\_256\_cfb8, aes\_256\_cfb128, aes\_256\_ofb, aes\_256\_ctr, aes\_256\_ccm,

aes\_256\_gcm, aes\_256\_xts, aes\_256\_wrap, aes\_128\_cbc\_hmac\_sha1, aes\_256\_cbc\_hmac\_sha1, aes\_128\_cbc\_hmac\_sha256, aes\_256\_cbc\_hmac\_sha256, camellia\_128\_ecb, camellia\_128\_cbc, camellia\_128\_cfb1, camellia\_128\_cfb8, camellia\_128\_cfb128, camellia\_128\_ofb, camellia\_192\_ecb, camellia\_192\_cbc, camellia\_192\_cfb1, camellia\_192\_cfb8, camellia\_192\_cfb128, camellia\_192\_ofb, camellia\_256\_ecb, camellia\_256\_cbc, camellia\_256\_cfb1, camellia\_256\_cfb8, camellia\_256\_cfb128, camellia\_256\_ofb, seed\_ecb, seed\_cbc, seed\_cfb128, seed\_ofb, custom(String)

The Digest and Cipher enums provide a `custom` case which can be used to indicate the digest or encoding by name. This can be useful if your system includes additional algorithms which are not made explicitly available by this package. All of the digest and cipher algorithms are implemented by the underlying crypto library (OpenSSL). Some of the encodings are implemented by the underlying library but others may be implemented directly by this package (hex, for example).

## Perfect – SMTP

This project provides an SMTP library.

This package builds with Swift Package Manager and is part of the [Perfect](#) project. Ensure you have installed and activated the latest Swift 3.0 tool chain.

## Linux Build Note

Make sure libssl-dev was installed on Ubuntu 16.04:

```
sudo apt-get install libssl-dev
```

## Relevant Examples

- [Perfect-SMTP-Demo](#)

## Quick Start

To use SMTP class, modify the Package.swift file and add following dependency:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-SMTP.git", majorVersion: 1)
```

Then import SMTP library into the Swift source code:

```
import PerfectSMTP
```

## Data Structures

Perfect SMTP contains three different data structures: SMTPClient, Recipient and EMail.

### SMTPClient

SMTPClient object is a data structure to store mail server login information:

```
let client = SMTPClient(url: "smtp://mailserver.address", username: "someone@some.where", password:"secret")
```

### Recipient

Recipient object is a data structure which store one's name and email address:

```
let recipient = Recipient(name: "Someone's Full Name", address: "someone@some.where")
```

### EMail

Use email object to compose and send an email. Check the following example code:

```
// initialize an email draft with mail connection / login info
var email = EMail(client: client)

// set the title of email
email.subject = "Mail Title"

// set the sender info
email.from = Recipient(name: "My Full Name", address: "mynickname@my.home")

// fill in the main content of email, plain text or html
email.html = "<h1>Hello, world!</h1><hr><img src='http://www.perfect.org/images/perfect-logo-2-0.svg'>"

// set the mail recipients, to / cc / bcc are all arrays
email.to.append(Recipient(name: "First Receiver", address: "someone@some.where"))
email.cc.append(Recipient(name: "Second Receiver", address: "someOtherOne@some.where"))
email.bcc.append(Recipient(name: "An invisible receiver", address: "someoneElse@some.where"))

// add attachments
email.attachments.append("/path/to/file.txt")
email.attachments.append("/path/to/img.jpg")

// send the email and call back if done.
do {
    try email.send { code, header, body in
        /// response info from mail server
        print(code)
        print(header)
        print(body)
    } //end send
} catch (let err) {
    /// something wrong
}
}
```

## Members of EMail Object

- client: SMTPClient, login info for mail server connection
- to: [Recipient], array of mail recipients
- cc: [Recipient], array of mail recipients, "copy / forward"
- bcc: [Recipient], array of mail recipients, will not appear in the to / cc mail.
- from: Recipient, email address of the current sender
- subject: String, title of the email
- attachments: [String], full path of attachments, i.e., ["/path/to/file1.txt", "/path/to/file2.gif" ...]
- content: String, mail body in text, plain text or html
- html: String, alias of `content` (shares the same variable as `content` )
- text: String, set the content to plain text
- send(completion: @escaping ((Int, String, String)->Void)), function of sending email with callback. The completion callback has three parameters; check Perfect-CURL `performFully()` for more information:
  - code: Int, mail server response code. Zero for OK.
  - header: String, mail server response header string.
  - body: String, mail server response body string.

## Summary

For better understanding, here is the brief structure of sending emails:

```
import PerfectSMTP

let client = SMTPClient(url: "smtp://smtp.gmx.com", username: "yourname@youraddress.com", password:"yourpassword")

var email = EMail(client: client)

email.subject = "a topic"
email.content = "a message"

email.cc.append(Recipient(address: "who@where.com"))

do {
    try email.send { code, header, body in
        /// response info from mail server
        print(code)
    } //end send
} catch(let err) {
    /// something wrong
}
```

## Example

---

A demo can be found here: [Perfect SMTP Demo](#)

## Tips for SMTPS

---

We've received a lot of requests about google smtp examples, Thanks for @ucotta @james and of course the official Perfect support from @iamjono, this note might be helpful for building gmail applications: ⚠️ *the SMTPClient url needs to be `smtps://smtp.gmail.com:465`, and you may need to "turn on access for less secure apps" in the google settings.* ⚠️

Please check the SMTPS code below, note the only difference is the URL pattern:

```
import PerfectSMTP

let client = SMTPClient(url: "smtps://smtp.gmail.com:465", username: "yourname@gmail.com", password:"yourpassword")

var email = EMail(client: client)

email.subject = "a topic"
email.content = "a message"

email.cc.append(Recipient(address: "who@where.com"))

do {
    try email.send { code, header, body in
        /// response info from mail server
        print(code)
    } //end send
} catch(let err) {
    /// something wrong
}
```

## Google Analytics Measurement Protocol

The Google Analytics Measurement Protocol is the server-side equivalent of embedding Google Analytics into a web page.

It means that you can log any sort of activity - Raw TCP or UDP events, or specific interactions that are triggered by events like AJAX.

## API Documentation

---

For full API documentation, visit <https://www.perfect.org/docs/api-Perfect-GoogleAnalytics-MeasurementProtocol.html>.

The API documentation explains every property that can be set within the system.

## Configuration

---

The `PerfectGAMeasurementProtocol` struct enables the setting of application-wide defaults for Property ID and Hit Type.

```
PerfectGAMeasurementProtocol.propertyid = "UA-XXXXXXX-X"  
PerfectGAMeasurementProtocol.hitType = "pageview"
```

## Building

Add this project as a dependency in your Package.swift file.

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-GoogleAnalytics-MeasurementProtocol.git", majorVersion: 0)
```

## Example Usage

To set up and execute the logging of an event:

```
PerfectGAMeasurementProtocol.propertyid = "UA-XXXXXXX-X"  
let gaex = PerfectGAEvent()  
gaex.user.uid = "donkey"  
gaex.user.cid = "kong"  
gaex.session.ua = "aua"  
gaex.traffic.ci = "ci"  
gaex.system.fl = "x"  
gaex.hit.ni = 2  
  
do {  
    let str = try gaex.generate()  
    print(str)  
    let resp = gaex.makeRequest(useragent: "TestingAPI1.0", body: str)  
    print(resp)  
} catch {  
    print("\(error)")  
}
```

A series of common hit types and configurations can be found in Google's documentation,  
<https://developers.google.com/analytics/devguides/collection/protocol/v1/devguide#commonhits>

## Perfect Repeater

This package provides a method to schedule repeating/recurring events.

## Relevant Examples

- [Perfect-Repeater-Example](#)

## Getting Started

In addition to the PerfectLib, you will need the Perfect-Repeater dependency in the Package.swift file:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-Repeater.git", majorVersion: 1)
```

## Using Perfect Repeater

Import the Perfect Repeater into each file that you wish to use the functions in:

```
import PerfectRepeater
```

The base form of executing this is:

```
Repeater.exec(timer: <Double>, callback: <Closure>)
```

The `timer` value is the time in seconds to repeat the event.

The `callback` contains a closure containing code to execute. This must contain a boolean return value. Returning `true` will re-queue the event, and `false` will remove the event from the queue.

The following code demonstrates the process of repeating a closure containing your code and optionally re-queuing:

```
var opt = 1

let c = {
  () -> Bool in
    print("XXXXXX")
    return true
}

let cc = {
  () -> Bool in
    print("Hello, world! \(opt)")
    if opt < 10 {
      opt += 1
      return true
    } else {
      print("cc exiting.")
      return false
    }
}

Repeater.exec(timer: 3.0, callback: c)
Repeater.exec(timer: 2.0, callback: cc)
```# Database Connectors
```

At the heart of any server side application is database access. Perfect supplies a number of connectors to popular datasources such as:

```
* [SQLite](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/SQLite.md)
* [MySQL](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/MySQL.md)
* [PostgreSQL](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/PostgreSQL.md)
* [MongoDB](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/MongoDB.md)
* [FileMaker](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/filemaker.md)
```

# SQLite

The SQLite connector provides a wrapper around SQLite3, allowing interaction between your Perfect applications and SQLite databases.

## Relevant Examples

```
* [PerfectArcade](https://github.com/PerfectExamples/PerfectArcade)
* [Perfect-Polling](https://github.com/PerfectExamples/Perfect-Polling)
* [Perfect-Turnstile-SQLite-Demo](https://github.com/PerfectExamples/Perfect-Turnstile-SQLite-Demo)
* [Perfect-Session-SQLite-Demo](https://github.com/PerfectExamples/Perfect-Session-SQLite-Demo)
```

## System Requirements

### macOS

No additional configuration should be required, as SQLite3 is already built in.

### Linux

Make sure that you have SQLite3 installed: ``sudo apt-get install sqlite3``

## Setup

Add the "Perfect-SQLite" project as a dependency in your Package.swift file:

```
``` swift
.Package(
  url: "https://github.com/PerfectlySoft/Perfect-SQLite.git",
  majorVersion: 2
)
```

## Import

First and foremost, in any of the source files you intend to use with SQLite, import the module with:

```
import SQLite
```

## Quick Start

### Access the Database

The database is accessed via its local file path, so the first step is to store the file path to your SQLite data:

```
let dbPath = "./db/database"
```

Once you've got that, you can open a connection to the database using a do-try-catch to make sure that errors are handled:

```
let dbPath = "./db/database"

do {
    let sqlite = try SQLite(dbPath)
    defer {
        sqlite.close() // This makes sure we close our connection.
    }
} catch {
    //Handle Errors
}
```

### Create Tables

Expanding on our connection above, we're able to run queries to create database tables by trying the [execute](#) method on our connection like so:

```
let dbPath = "./db/database"

do {
    let sqlite = try SQLite(dbPath)
    defer {
        sqlite.close() // This makes sure we close our connection.
    }

    try sqlite.execute(statement: "CREATE TABLE IF NOT EXISTS demo (id INTEGER PRIMARY KEY NOT NULL, option TEXT NOT NULL, value TEXT)")
} catch {
    //Handle Errors
}
```

### Run Queries

- A quick note about string interpolation: Variables in queries do not work as interpolated strings. In order to use variables, you need to use the binding system, described in the [next section](#).

Once you have a database and tables, the next step is to query and return data. In this example, we will store our statement in a string, and pass it into the [forEachRow](#) method, which will iterate through each returned row, where you can (most often) append to a dictionary.

```

let dbPath = "./db/database"
var contentDict = [String: Any]()

do {
    let sqlite = try SQLite(dbPath)
    defer {
        sqlite.close() // This makes sure we close our connection.
    }

    let demoStatement = "SELECT * FROM demo"

    try sqlite.forEachRow(statement: demoStatement) {(statement: SQLiteStmt, i:Int) -> () in

        self.contentDict.append([
            "id": statement.columnText(position: 0),
            "second_field": statement.columnText(position: 1),
            "third_field": statement.columnText(position: 2)
        ])
    }
} catch {
    //Handle Errors
}

```

## Binding Variables to Queries

One thing you'll definitely want to do is add variables to your queries. As noted above, you cannot do this with string interpolation; instead you can use the binding system. For example:

```

let dbPath = "./db/database"
var contentDict = [String: Any]()

do {
    let sqlite = try SQLite(dbPath)
    defer {
        sqlite.close() // This makes sure we close our connection.
    }

    let demoStatement = "SELECT post_title, post_content FROM posts ORDER BY id DESC LIMIT :1"

    try sqlite.forEachRow(statement: demoStatement, doBindings {

        (statement: SQLiteStmt) -> () in

        let bindValue = 5
        try statement.bind(position: 1, bindValue)

    }) {(statement: SQLiteStmt, i:Int) -> () in

        self.contentDict.append([
            "id": statement.columnText(position: 0),
            "second_field": statement.columnText(position: 1),
            "third_field": statement.columnText(position: 2)
        ])
    }
} catch {
    //Handle Errors
}

```

If that looks a little tricky, that's okay. Our "doBindings:" argument takes a closure to handle adding variables to the positions you've defined, and our last argument (technically "handleRow:") is omitted and passed the closure it takes afterward, as it's the standard way to use Swift. After a few practice runs, it gets much easier to read.

## Full API Reference

The full API consists of:

```

init(_:readOnly:)
close()
prepare(_:)
lastInsertRowID()
totalChanges()
changes()
errCode()
errMsg()
execute(_:)
execute(_:doBindings:)
execute(_:count:doBindings:)
doWithTransaction(_:)
forEachRow(_:handleRow:)
forEachRow(_:doBindings:handleRow:)

```

Each of these is detailed below:

## init

```
public init(_ path: String, readOnly: Bool = false) throws
```

Is the basic initializer for creating a new instance of the class. It's used by passing a file path to an SQLite database and has a secondary parameter used to put the database in read only mode, as necessary. Since `readOnly`: defaults to `false`, it's not necessary to include it if you want to both read and write to the file. This function also throws, which means that you must use it within `do-try-catch`.

## close

```
public func close()
```

This does exactly what you think it does: it closes the database connection. Its default usage is a `defer` method inside the `do-try-catch` that you initialize the database with. This way you are guaranteed to close the connection to the database whether or not your try succeeds or terminates in an error.

## prepare

```
public func prepare(stat: String) throws -> SQLiteStmt
```

This function returns a compiled SQLite statement object that represents the compiled statement. This function is a dependency of other class functions (like `execute` and `forEachRow`) which take strings as arguments, pass them to this function, and use the returned object to communicate with the database. There is a very low chance you will ever need to use this function directly, but it is available if you come up with a use case.

## lastInsertRowID

```
public func lastInsertRowID() -> Int
```

This function returns the value of the last row that was inserted into the database. It must be used within your `do-try-catch` *before* the connection to the database is severed, otherwise it will always return "0". A return value of "0" either means that there is no open connection, or there are no rows. If it is called while another insert is being performed on the same table by a different thread, the result can be slightly unpredictable, so be careful with it. You can visit the [SQLite3 Documentation](#) for further information about how this function works.

## totalChanges

```
public func totalChanges() -> Int
```

Returns the value of `sqlite3_total_changes` . From [SQLite3's documentation](#):

*"This function returns the total number of rows inserted, modified or deleted by all INSERT, UPDATE or DELETE statements completed since the database connection was opened, including those executed as part of trigger programs. Executing any other type of SQL statement does not affect the value returned by `sqlite3_total_changes()` ."*

Once the connection closes, you are not going to get much value from this, so make sure you include it in the `do-try-catch` where the related statements are executing.

## changes

```
public func changes() -> Int
```

This function will return the value of `sqlite3_changes` . The major difference between this and `totalChanges` is that the resulting number of changes will not include anything that the

statement triggers; only those rows affected directly by the statement itself.

## errCode

```
public func errCode() -> Int
```

Returns the value of `sqlite3_errcode`. You can learn more about what those mean [here](#).

## errMsg

```
public func errMsg() -> String
```

Returns the value of `sqlite3_errmsg`. Learn more about error codes and messages [here](#).

## execute

```
public func execute(statement: String) throws
```

Runs a statement that expects no return (such as an *INSERT* or *CREATE TABLE*). Since this function throws, make sure to use it within your do-try-catch.

## execute with doBindings:

```
public func execute(statement: String, doBindings: (SQLiteStmt) throws -> ()) throws
```

A variant of execute that also allows for binding variables to the statement before it runs. Also must be in the do-try-catch, as it throws. Learn more about binding variables [here](#).

## execute with count: and doBindings:

```
public func execute(statement: String, count: Int, doBindings: (SQLiteStmt, Int) throws -> ()) throws
```

This last variant of the execute function allows you to repeat the statement for count: number of times. This is especially useful in situations where you need to loop an insert. Also allows for the [binding of variables](#). Must be in the do-try-catch, as it throws.

## doWithTransaction

```
public func doWithTransaction(closure: () throws -> ()) throws
```

"Executes a BEGIN, calls the provided closure and executes a ROLLBACK if an exception occurs or a COMMIT if no exception occurs."

That means that you can run a closure with a result, and if that result fails, no changes will actually be made to the database unless that result succeeds. You can read more about SQLite3 transactions [here](#).

## forEachRow with handleRow:

```
public func forEachRow(statement: String, handleRow: (SQLiteStmt, Int) -> ()) throws
```

Executes the given statement, then calls the closure given at handleRow for every row returned. This is especially useful for appending items from the rows returned by the statement into things like dictionaries and arrays. As it throws, it must be placed in your do-try-catch.

## forEachRow with doBindings: and handleRow:

```
public func forEachRow(statement: String, doBindings: (SQLiteStmt) throws -> (), handleRow: (SQLiteStmt, Int) -> ()) throws
```

As with the previous, this also allows for calling a closure on each row given, but includes the ability to bind variables to the query before it is run. Again, throws means that you need to include this in the do-try-catch.

## Example

The following example is part of a blog system. It's a function that encapsulates loading page content for a blog post kept in an SQLite database and appending it to a dictionary declared in the class as `var content = [String: Any]()`, which in this particular case would be further used as part of a mustache template that displays that content. It will load content for a page of five posts, given the page number as an argument.

```

func loadPageContent(forPage: Int) {
    do {
        let sqlite = try SQLite(DB_PATH)
        defer {
            sqlite.close() // defer ensures we close our db connection at the end of this request
        }
        let sqlStatement = "SELECT post_content, post_title FROM posts ORDER BY id DESC LIMIT 5 OFFSET :1"

        try sqlite.forEachRow(statement: sqlStatement, doBindings: {
            (statement: SQLiteStmt) -> () in

            let bindPage: Int

            if self.page == 0 || self.page == 1 {
                bindPage = 0
            } else {
                bindPage = forPage * 5 - 5
            }

            try statement.bind(position: 1, bindPage)
        }) {
            (statement: SQLiteStmt, i: Int) -> () in

            self.content.append([
                "postContent": statement.columnText(position: 0),
                "postTitle": statement.columnText(position: 1)
            ])
        }
    } catch {
    }
}

```

## MySQL

The MySQL connector provides a wrapper around MySQL, allowing interaction between your Perfect Applications and MySQL databases.

### System Requirements

#### macOS

Requires the use of Homebrew's MySQL.

```
brew install mysql
```

If you need Homebrew, you can install it with:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Unfortunately, at this point in time you will need to edit the mysqlclient.pc file located here:

```
/usr/local/lib/pkgconfig/mysqlclient.pc
```

Remove the occurrence of "-fno-omit-frame-pointer". This file is read-only by default so you will need to change its permissions first.

#### Linux

Ensure that you have installed libmysqlclient-dev for MySQL version 5.6 or greater:

```
sudo apt-get install libmysqlclient-dev
```

Please note that Ubuntu 14 defaults to including a version of MySQL client which will not compile with this package. Install MySQL client version 5.6 or greater manually.

### Setup

Add the "Perfect-MySQL" project as a dependency in your Package.swift file:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-MySQL.git", majorVersion: 2)
```

## Import

First and foremost, in any of the source files you intend to use with MySQL, import the required module with:

```
import MySQL
```

## Quick Start

### Access the Database

In order to access the database, set up your credentials:

```
let testHost = "127.0.0.1"
let testUser = "test"
let testPassword = "password"
let testDB = "schema"

//Obviously change these details to a database and user you have already defined
```

There are two common ways to connect to MySQL. First, you can omit the schema, so that you can use a separate selector. This is handy if you have multiple schemas that your program can choose:

```
func fetchData() {

    let mysql = MySQL() // Create an instance of MySQL to work with

    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword)

    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }

    defer {
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result
    }

    //Choose the database to work with
    guard mysql.selectDatabase(named: testDB) else {
        Log.info(message: "Failure: \(mysql.errorCode()) \(dataMysql.errorMessage())")
        return
    }
}
```

Alternatively, you can pass the database you would like to access into the connection and skip selection:

```
func fetchData() {

    let mysql = MySQL() // Create an instance of MySQL to work with

    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword, db: testDB)

    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }

    defer {
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result
    }
}
```

## Create Tables

Choosing the database is great, but it is much more helpful to run queries, such as adding tables programmatically. Expanding on our connection example above, it is relatively simple to add a query:

```
func setupMySQLDB() {  
  
    let mysql = MySQL() // Create an instance of MySQL to work with  
  
    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword, db: testDB)  
  
    guard connected else {  
        // verify we connected successfully  
        print(mysql.errorMessage())  
        return  
    }  
  
    defer {  
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result  
    }  
  
    //Run Query to Add Tables  
  
}
```

## Run Queries

Getting data from your schema is essential, and relatively easy to do. After running a query, save your data and then act on it. In the example below, we're assuming we have a table called options with a row id, an option name (text) and an option value (text):

```

func fetchData() {

    let mysql = MySQL() // Create an instance of MySQL to work with

    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword, db: testDB)

    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }

    defer {
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result
    }

    // Run the Query (for example all rows in an options table)
    let querySuccess = mysql.query(statement: "SELECT option_name, option_value FROM options")
    // make sure the query worked
    guard querySuccess else {
        return
    }

    // Save the results to use during this session
    let results = mysql.storeResults()! //We can implicitly unwrap because of the guard on the querySuccess. You're welcome to use an if-let
    here if you like.

    var ary = [[String:Any]]() //Create an array of dictionaries to store our results in for use

    results.forEachRow { row in
        let optionName = getRowString(forRow: row[0]) //Store our Option Name, which would be the first item in the row, and therefore row[0]
        let optionValue = getRowString(forRow: row[1]) //Store our Option Value

        ary.append("\(optionName)":optionValue) //store our options
    }
}

```

## MySQL Server API

The MySQL server API provides you with a set of tools to connect to and work with MySQL server instances. This includes basic connections, disconnections, querying the instance for databases/tables, and running queries (which is actually a light wrapper for the full [Statements API](#)). Results returned, however, are handled and manipulated with the [Results API](#). Statements also have a [Statements API](#) that lets you work with statements in much more detail than simply running queries though the main MySQL class.

### init

```
public init()
```

Creates an instance of the MySQL class that allows you to interact with MySQL databases.

### NOTE for ⚠️Character Encoding⚠️

If your dataset contains non-ascii characters, please set this option for proper encoding:

```
setOption(.MYSQL_SET_CHARSET_NAME, "utf8")
```

### close

```
public func close()
```

Closes a connection to MySQL. Most commonly used as a defer after guarding a connection, making sure that your session will close no matter what the outcome.

### clientInfo

```
public static func clientInfo() -> String
```

This will give you a string of the MySQL client library version, e.g. "5.7.x" or similar depending on your MySQL installation.

## errorCode & errorMessage

```
public func errorCode() -> UInt32
```

```
public func errorMessage() -> String
```

Error codes and messages are useful when debugging. These functions retrieve, display, and make use of those in Swift. You can learn more about what those mean [here](#). This is especially useful after connecting or running queries. Example:

```
let mysql = MySQL()
let connected = mysql.connect(host: dbHost, user: dbUser, password: dbPassword, db: dbName)
    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }
```

In this case, the console output would print any error messages that came up during a connection failure.

## serverVersion

```
public func serverVersion() -> Int
```

Returns an integer representation of the MySQL server's version.

## connect

```
public func connect(host hst: String? = nil, user: String? = nil, password: String? = nil, db: String? = nil, port: UInt32 = 0, socket: String? = nil, flag: UInt = 0) -> Bool
```

Opens a connection to the MySQL database when supplied with the bare minimum credentials for your server (usually a host, user, and password). As an option, you can specify the port, database, or socket. Specifying the schema is not required, as you can use the [selectDatabase\(\)](#) method after the connection has been made.

## selectDatabase

```
public func selectDatabase(named namd: String) -> Bool
```

Selects a database from the active MySQL connection.

## listTables

```
public func listTables(wildcard wild: String? = nil) -> [String]
```

Returns an array of strings representing the different tables available on the selected database.

## listDatabases

```
public func listDatabases(wildcard wild: String? = nil) -> [String]
```

Returns an array of strings representing the databases available on the MySQL server currently connected.

## commit

```
public func commit() -> Bool
```

Commits the transaction.

## rollback

```
public func rollback() -> Bool
```

Rolls back the transaction.

## moreResults

```
public func moreResults() -> Bool
```

Checks `mysql_more_results` to see if any more results exist.

## nextResult

```
public func nextResult() -> Int
```

Returns the next result in a multi-result execution. Most commonly used in a while loop to produce an effect similar to running `forEachRow()`. For example:

```
var results = [[String?]]()

while let row = results?.next() {
    results.append(row)
}
```

## query

```
public func query(statement stmt: String) -> Bool
```

Runs an SQL Query given as a string.

## storeResults

```
public func storeResults() -> MySQL.Results?
```

This retrieves a complete result set from the server and stores it on the client. This should be run after your query and before a function like `forEachRow()` or `next()` so that you can ensure that you iterate through all results.

## setOption

```
public func setOption(_ option: MySQLOpt) -> Bool
public func setOption(_ option: MySQLOpt, _ b: Bool) -> Bool
public func setOption(_ option: MySQLOpt, _ i: Int) -> Bool
public func setOption(_ option: MySQLOpt, _ s: String) -> Bool
```

Sets the options for connecting and returns a Boolean for success or failure. Requires a `MySQLOpt` and has several versions to support setting options that require Booleans, integers, or strings as values.

`MySQLOpt` values that are available to use are defined by the following enumeration:

```
public enum MySQLOpt {
    case MySQL_OPT_CONNECT_TIMEOUT, MySQL_OPT_COMPRESS, MySQL_OPT_NAMED_PIPE,
        MySQL_INIT_COMMAND, MySQL_READ_DEFAULT_FILE, MySQL_READ_DEFAULT_GROUP,
        MySQL_SET_CHARSET_DIR, MySQL_SET_CHARSET_NAME, MySQL_OPT_LOCAL_INFILE,
        MySQL_OPT_PROTOCOL, MySQL_SHARED_MEMORY_BASE_NAME, MySQL_OPT_READ_TIMEOUT,
        MySQL_OPT_WRITE_TIMEOUT, MySQL_OPT_USE_RESULT,
        MySQL_OPT_USE_REMOTE_CONNECTION, MySQL_OPT_USE_EMBEDDED_CONNECTION,
        MySQL_OPT_GUESS_CONNECTION, MySQL_SET_CLIENT_IP, MySQL_SECURE_AUTH,
        MySQL_REPORT_DATA_TRUNCATION, MySQL_OPT_RECONNECT,
        MySQL_OPT_SSL_VERIFY_SERVER_CERT, MySQL_PLUGIN_DIR, MySQL_DEFAULT_AUTH,
        MySQL_OPT_BIND,
        MySQL_OPT_SSL_KEY, MySQL_OPT_SSL_CERT,
        MySQL_OPT_SSL_CA, MySQL_OPT_SSL_CAPATH, MySQL_OPT_SSL_CIPHER,
        MySQL_OPT_SSL_CRL, MySQL_OPT_SSL_CRLPATH,
        MySQL_OPT_CONNECT_ATTR_RESET, MySQL_OPT_CONNECT_ATTR_ADD,
        MySQL_OPT_CONNECT_ATTR_DELETE,
        MySQL_SERVER_PUBLIC_KEY,
        MySQL_ENABLE_CLEARTEXT_PLUGIN,
        MySQL_OPT_CAN_HANDLE_EXPIRED_PASSWORDS
}
```

## MySQL Results API

The results API set provides a set of tools for working with result sets that are obtained from running queries.

### close

```
public func close()
```

Closes the result set by releasing the results. Make sure you have a close function that is always executed after each session with a connection, otherwise you are going to have some memory problems. This is most commonly found in the defer block, just like in the examples at the top of this page.

### dataSeek

```
public func dataSeek(_ offset: UInt)
```

Moves to an arbitrary row number in the results given an unsigned integer as an offset.

### numRows

```
public func numRows() -> Int
```

Lets you know how many rows are in a result set.

### numFields

```
public func numFields() -> Int
```

Similar to numRows, but returns the number of columns in a result set instead. Very useful for `Select *` type queries where you may need to know how many columns are in the results.

### next

```
public func next() -> Element?
```

Returns the next row in the result set as long as one exists.

### forEachRow

```
public func forEachRow(callback: (Element) -> ())
```

Iterates through all rows in query results. Most useful for appending elements to an array or dictionary, just like we did in the [quick start guide](#).

## MySQL Statements API

## init

```
public init(_ mysql: MySQL)
```

Initializes the MySQL statement structure. This is very commonly used by other API functions to create a statement structure after you've passed in a string.

## close

```
public func close()
```

This frees the MySQL statement structure pointer. Use it or lose valuable memory to the underlying MySQL C API. Most commonly in a defer block, just like we used in [quick start](#).

## reset

```
public func reset()
```

Resets the statement buffers that are in the server. This doesn't affect bindings or stored result sets. Learn more about this feature [here](#).

## clearBinds

```
func clearBinds()
```

Clears the current bindings.

## freeResult

```
public func freeResult()
```

Releases memory tied up in with the result set produced by execution of a prepared statement. Also closes a cursor if one is open for the statement.

## errorCode & errorMessage

```
public func errorCode() -> UInt32
```

```
public func errorMessage() -> String
```

Error codes and messages are useful when debugging. These functions retrieve, display, and make use of both in Swift. You can learn more about what those mean [here](#). This is especially useful after connecting or running queries. Example:

```
let mysql = MySQL()
let connected = mysql.connect(host: dbHost, user: dbUser, password: dbPassword, db: dbName)
    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }
```

In this case, the console output would print any error messages that came up during a connection failure.

## prepare

```
public func prepare(statement query: String) -> Bool
```

Prepares a SQL statement for execution. More commonly called by other functions in the API, but public if you need it.

## execute

```
public func execute() -> Bool
```

Executes a prepared statement.

## results

```
public func results() -> MySQLStmt.Results
```

Returns current results from the server.

## fetch

```
public func fetch() -> FetchResult
```

Fetches the next row of data from the result set.

## numRows

```
public func numRows() -> UInt
```

Returns the row count in a buffered statement result set.

## affectedRows

```
public func affectedRows() -> UInt
```

Returns the number of rows that were changed, deleted, or inserted by a prepared statement or an UPDATE, DELETE, or INSERT statement that was executed.

## insertId

```
public func insertId() -> UInt
```

Returns the row id number for the last row inserted by a prepared statement, as long as id was an auto-increment enabled column.

## fieldCount

```
public func fieldCount() -> UInt
```

Returns the number of columns in the results for the most recently executed statement.

## nextResult

```
public func nextResult() -> Int
```

Returns the next result in a multi-result execution.

## dataSeek

```
public func dataSeek(offset: Int)
```

Given an offset, it will seek to an arbitrary row in a statement result set.

## paramCount

```
public func paramCount() -> Int
```

Returns the number of parameters in a prepared statement.

## bindParam

```
public func bindParam()  
func bindParam(_ s: String, type: enum_field_types)  
public func bindParam(_ d: Double)  
public func bindParam(_ i: Int)  
public func bindParam(_ i: UInt64)  
public func bindParam(_ s: String)  
public func bindParam(_ b: UnsafePointer<Int8>, length: Int)  
public func bindParam(_ b: [UInt8])
```

Variations above on the `bindParam()` allow binding to statement parameters with different types. If no arguments are passed, it creates a null binding.

# MariaDB

The MariaDB connector provides a wrapper around MariaDB, allowing interaction between your Perfect Applications and MariaDB databases.

**Note:** Because of the history of MySQL & MariaDB, MariaDB keeps the same names as MySQL API functions; however, it is not possible to compile MariaDB in a MySQL configuration "as-is" context. This is why Perfect 2.0 includes both MySQL & MariaDB connectors.

## System Requirements

### macOS

Requires the use of Homebrew's MariaDB connector.

```
brew install mariadb-connector-c
```

If you need Homebrew, you can install it with:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You will need to edit the `mariadb.pc` file:

```
/usr/local/lib/pkgconfig/mariadb.pc
```

Remove the occurrence of `"-fno-omit-frame-pointer"`. This file is read-only by default so you will need to change its permissions first.

A typical configuration of `mariadb.pc` may look like:

```
prefix=/usr/local
exec_prefix=${prefix}/bin
libdir=${prefix}/lib/mariadb
includedir=${prefix}/include/mariadb

Name: mariadb
Description: MariaDB Connector/C
Version: 5.5.1
Requires:
Libs: -L${libdir} -lmariadb -ldl -lm -lpthread
Cflags: -I${includedir}
Libs_r: -L${libdir} -lmariadb -ldl -lm -lpthread
```

Edit your `~/bash_profile` with the following line:

```
export PKG_CONFIG_PATH="/usr/local/lib/pkgconfig:/usr/lib/pkgconfig"
```

### Linux

Ensure that you have installed `libmariadb2` `libmariadb-client-lgpl-dev` for MariaDB:

```
sudo apt-get install clang pkg-config libmariadb2 libmariadb-client-lgpl-dev libcurl4-openssl-dev
```

Ensure the `pkg-config` file `/usr/lib/pkgconfig/mariadb.pc` specified for MariaDB should be MANUALLY added and corrected before building. The file will look similar to the following:

```
prefix=/usr
exec_prefix=${prefix}/bin
libdir=${prefix}/lib/mariadb
includedir=${prefix}/include/mariadb

Name: mariadb
Description: MariaDB Connector/C
Version: 5.5.0
Requires:
Libs: -L${libdir} -lmariadb -ldl -lm -lpthread
Cflags: -I${includedir}
Libs_r: -L${libdir} -lmariadb -ldl -lm -lpthread
```

## Setup

Add the "Perfect-MariaDB" project as a dependency in your Package.swift file:

```
.Package(url:"https://github.com/PerfectlySoft/Perfect-MariaDB.git", majorVersion: 2)
```

## Import

To use MariaDB within a file, import the required module:

```
import MariaDB
```

## Quick Start

### Access the Database

In order to access the database, set up your credentials:

```
let testHost = "127.0.0.1"
let testUser = "test"
let testPassword = "password"
let testDB = "schema"
```

These are example credentials only. Replace with your own.

There are two common ways to connect to MariaDB. First, you can omit the schema, so that you can use a separate selector. This is handy if you have multiple schemas that your program can choose:

```
func fetchData() {

    let dataMysql = MySQL() // Create an instance of MySQL to work with

    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword)

    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }

    defer {
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result
    }

    //Choose the database to work with
    guard dataMysql.selectDatabase(named: testDB) else {
        Log.info(message: "Failure: \(dataMysql.errorCode()) \(dataMysql.errorMessage())")
        return
    }
}
```

Alternatively, you can pass the database you would like to access into the connection and skip selection:

```

func fetchData() {

    let dataMysql = MySQL() // Create an instance of MariaDB MySQL Object to work with

    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword, db: testDB)

    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }

    defer {
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result
    }
}

```

## Create Tables

Choosing the database is great, but it is much more helpful to run queries, such as adding tables programmatically. Expanding on our connection example above, it is relatively simple to add a query:

```

func setupMySQLDB() {

    let dataMysql = MySQL() // Create an instance of MySQL to work with

    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword, db: testDB)

    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }

    defer {
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result
    }

    //Run Query to Add Tables

}

```

## Run Queries

Getting data from your schema is essential, and relatively easy to do. After running a query, save your data and then act on it. In the example below, we're assuming we have a table called options with a row id, an option name (text) and an option value (text):

```

func fetchData() {

    let dataMysql = MySQL() // Create an instance of MySQL to work with

    let connected = mysql.connect(host: testHost, user: testUser, password: testPassword, db: testDB)

    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }

    defer {
        mysql.close() //This defer block makes sure we terminate the connection once finished, regardless of the result
    }

    // Run the Query (for example all rows in an options table)
    let querySuccess = mysql.query(statement: "SELECT option_name, option_value FROM options")
    // make sure the query worked
    guard querySuccess else {
        return
    }

    // Save the results to use during this session
    let results = mysql.storeResults()! //We can implicitly unwrap because of the guard on the querySuccess. You're welcome to use an if-let
    here if you like.

    var ary = [[String:Any]]() //Create an array of dictionaries to store our results in for use

    results.forEachRow { row in
        let optionName = getRowString(forRow: row[0]) //Store our Option Name, which would be the first item in the row, and therefore row[0]
        let optionValue = getRowString(forRow: row[1]) //Store our Option Value

        ary.append("\(optionName)":optionValue) //store our options
    }
}

```

## MariaDB Server API

The MariaDB server API provides you with a set of tools to connect to and work with MariaDB server instances. This includes basic connections, disconnections, querying the instance for databases/tables, and running queries (which is actually a light wrapper for the full [Statements API](#). Results returned, however, are handled and manipulated with the [Results API](#). Statements also have a [Statements API](#) that lets you work with statements in much more detail than simply running queries though the main MySQL class.

### init

```
public init()
```

Creates an instance of the MySQL class that allows you to interact with MariaDB databases.

### NOTE for ⚠️Character Encoding⚠️

If your dataset contains non-ascii characters, please set this option for proper encoding:

```
setOption(MYSQL_SET_CHARSET_NAME, "utf8")
```

### close

```
public func close()
```

Closes a connection to MariaDB. Most commonly used as a defer after guarding a connection, making sure that your session will close no matter what the outcome.

### clientInfo

```
public static func clientInfo() -> String
```

This will give you a string of the MariaDB client library version, e.g. "5.5.x" or similar depending on your MariaDB installation.

## errorCode & errorMessage

```
public func errorCode() -> UInt32
```

```
public func errorMessage() -> String
```

Error codes and messages are useful when debugging. These functions retrieve, display, and make use of those in Swift. You can learn more about what those mean [here](#). This is especially useful after connecting or running queries. Example:

```
let mysql = MySQL()
let connected = mysql.connect(host: dbHost, user: dbUser, password: dbPassword, db: dbName)
    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }
```

In this case, the console output would print any error messages that came up during a connection failure.

## serverVersion

```
public func serverVersion() -> Int
```

Returns an integer representation of the MariaDB server's version.

## connect

```
public func connect(host hst: String? = nil, user: String? = nil, password: String? = nil, db: String? = nil, port: UInt32 = 0, socket: String? = nil, flag: UInt = 0) -> Bool
```

Opens a connection to the MariaDB database when supplied with the bare minimum credentials for your server (usually a host, user, and password). As an option, you can specify the port, database, or socket. Specifying the schema is not required, as you can use the [selectDatabase\(\)](#) method after the connection has been made.

## selectDatabase

```
public func selectDatabase(named namd: String) -> Bool
```

Selects a database from the active MariaDB connection.

## listTables

```
public func listTables(wildcard wild: String? = nil) -> [String]
```

Returns an array of strings representing the different tables available on the selected database.

## listDatabases

```
public func listDatabases(wildcard wild: String? = nil) -> [String]
```

Returns an array of strings representing the databases available on the MariaDB server currently connected.

## commit

```
public func commit() -> Bool
```

Commits the transaction.

## rollback

```
public func rollback() -> Bool
```

Rolls back the transaction.

## moreResults

```
public func moreResults() -> Bool
```

Checks `mysql_more_results` to see if any more results exist.

## nextResult

```
public func nextResult() -> Int
```

Returns the next result in a multi-result execution. Most commonly used in a while loop to produce an effect similar to running `forEachRow()`. For example:

```
var results = [[String?]]()

while let row = results?.next() {
    results.append(row)
}
```

## query

```
public func query(statement stmt: String) -> Bool
```

Runs an SQL Query given as a string.

## storeResults

```
public func storeResults() -> MySQL.Results?
```

This retrieves a complete result set from the server and stores it on the client. This should be run after your query and before a function like `forEachRow()` or `next()` so that you can ensure that you iterate through all results.

## setOption

```
public func setOption(_ option: MySQLOpt) -> Bool
public func setOption(_ option: MySQLOpt, _ b: Bool) -> Bool
public func setOption(_ option: MySQLOpt, _ i: Int) -> Bool
public func setOption(_ option: MySQLOpt, _ s: String) -> Bool
```

Sets the options for connecting and returns a Boolean for success or failure. Requires a `MySQLOpt` and has several versions to support setting options that require Booleans, integers, or strings as values.

`MySQLOpt` values that are available to use are defined by the following enumeration:

```
public enum MySQLOpt {
    case MYSQL_OPT_CONNECT_TIMEOUT, MYSQL_OPT_COMPRESS, MYSQL_OPT_NAMED_PIPE,
    MYSQL_INIT_COMMAND, MYSQL_READ_DEFAULT_FILE, MYSQL_READ_DEFAULT_GROUP,
    MYSQL_SET_CHARSET_DIR, MYSQL_SET_CHARSET_NAME, MYSQL_OPT_LOCAL_INFILE,
    MYSQL_OPT_PROTOCOL, MYSQL_SHARED_MEMORY_BASE_NAME, MYSQL_OPT_READ_TIMEOUT,
    MYSQL_OPT_WRITE_TIMEOUT, MYSQL_OPT_USE_RESULT,
    MYSQL_OPT_USE_REMOTE_CONNECTION, MYSQL_OPT_USE_EMBEDDED_CONNECTION,
    MYSQL_OPT_GUESS_CONNECTION, MYSQL_SET_CLIENT_IP, MYSQL_SECURE_AUTH,
    MYSQL_REPORT_DATA_TRUNCATION, MYSQL_OPT_RECONNECT,
    MYSQL_OPT_SSL_VERIFY_SERVER_CERT, MYSQL_PLUGIN_DIR, MYSQL_DEFAULT_AUTH,
    MYSQL_OPT_BIND,
    MYSQL_OPT_SSL_KEY, MYSQL_OPT_SSL_CERT,
    MYSQL_OPT_SSL_CA, MYSQL_OPT_SSL_CAPATH, MYSQL_OPT_SSL_CIPHER,
    MYSQL_OPT_SSL_CRL, MYSQL_OPT_SSL_CRLPATH,
    MYSQL_OPT_CONNECT_ATTR_RESET, MYSQL_OPT_CONNECT_ATTR_ADD,
    MYSQL_OPT_CONNECT_ATTR_DELETE
}
```

## MariaDB Results API

---

The results API set provides a set of tools for working with result sets that are obtained from running queries.

### close

```
public func close()
```

Closes the result set by releasing the results. Make sure you have a close function that is always executed after each session with a connection, otherwise you are going to have some memory problems. This is most commonly found in the defer block, just like in the examples at the top of this page.

### dataSeek

```
public func dataSeek(_ offset: UInt)
```

Moves to an arbitrary row number in the results given an unsigned integer as an offset.

### numRows

```
public func numRows() -> Int
```

Lets you know how many rows are in a result set.

### numFields

```
public func numFields() -> Int
```

Similar to numRows, but returns the number of columns in a result set instead. Very useful for `Select *` type queries where you may need to know how many columns are in the results.

### next

```
public func next() -> Element?
```

Returns the next row in the result set as long as one exists.

### forEachRow

```
public func forEachRow(callback: (Element) -> ())
```

Iterates through all rows in query results. Most useful for appending elements to an array or dictionary, just like we did in the [quick start guide](#).

## MariaDB Statements API

### init

```
public init(_ mysql: MySQL)
```

Initializes the MySQL statement structure. This is very commonly used by other API functions to create a statement structure after you've passed in a string.

### close

```
public func close()
```

This frees the MySQL statement structure pointer. Use it or lose valuable memory to the underlying MariaDB C API. Most commonly in a defer block, just like we used in [quick start](#).

### reset

```
public func reset()
```

Resets the statement buffers that are in the server. This doesn't affect bindings or stored result sets. Learn more about this feature [here](#).

## clearBinds

```
func clearBinds()
```

Clears the current bindings.

## freeResult

```
public func freeResult()
```

Releases memory tied up in with the result set produced by execution of a prepared statement. Also closes a cursor if one is open for the statement.

## errorCode & errorMessage

```
public func errorCode() -> UInt32
```

```
public func errorMessage() -> String
```

Error codes and messages are useful when debugging. These functions retrieve, display, and make use of both in Swift. You can learn more about what those mean [here](#). This is especially useful after connecting or running queries. Example:

```
let mysql = MySQL()
let connected = mysql.connect(host: dbHost, user: dbUser, password: dbPassword, db: dbName)
    guard connected else {
        // verify we connected successfully
        print(mysql.errorMessage())
        return
    }
```

In this case, the console output would print any error messages that came up during a connection failure.

## prepare

```
public func prepare(statement query: String) -> Bool
```

Prepares a SQL statement for execution. More commonly called by other functions in the API, but public if you need it.

## execute

```
public func execute() -> Bool
```

Executes a prepared statement.

## results

```
public func results() -> MySQLStmt.Results
```

Returns current results from the server.

## fetch

```
public func fetch() -> FetchResult
```

Fetches the next row of data from the result set.

## numRows

```
public func numRows() -> UInt
```

Returns the row count in a buffered statement result set.

## affectedRows

```
public func affectedRows() -> UInt
```

Returns the number of rows that were changed, deleted, or inserted by a prepared statement or an UPDATE, DELETE, or INSERT statement that was executed.

## insertId

```
public func insertId() -> UInt
```

Returns the row id number for the last row inserted by a prepared statement, as long as id was an auto-increment enabled column.

## fieldCount

```
public func fieldCount() -> UInt
```

Returns the number of columns in the results for the most recently executed statement.

## nextResult

```
public func nextResult() -> Int
```

Returns the next result in a multi-result execution.

## dataSeek

```
public func dataSeek(offset: Int)
```

Given an offset, it will seek to an arbitrary row in a statement result set.

## paramCount

```
public func paramCount() -> Int
```

Returns the number of parameters in a prepared statement.

## bindParam

```
public func bindParam()  
func bindParam(_ s: String, type: enum_field_types)  
public func bindParam(_ d: Double)  
public func bindParam(_ i: Int)  
public func bindParam(_ i: UInt64)  
public func bindParam(_ s: String)  
public func bindParam(_ b: UnsafePointer<Int8>, length: Int)  
public func bindParam(_ b: [UInt8])
```

Variations above on the bindParam() allow binding to statement parameters with different types. If no arguments are passed, it creates a null binding.

# PostgreSQL

This project provides a Swift wrapper around the libpq client library, enabling access to PostgreSQL servers.

## System Requirements

### macOS

This package requires the [Homebrew](#) build of PostgreSQL.

To install Homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

To install Postgres:

```
brew install postgres
```

## Linux

Ensure that you have installed libpq-dev.

```
sudo apt-get install libpq-dev
```

## Setup

Add the "Perfect-PostgreSQL" project as a dependency in your Package.swift file:

```
.Package(  
    url: "https://github.com/PerfectlySoft/Perfect-PostgreSQL.git",  
    majorVersion: 2  
)
```

Remember to rebuild your Xcode project file after making any changes to your Package.swift file.

```
swift package generate-xcodeproj
```

## Import

To use the PostgreSQL connector in a source file, import the module:

```
import PostgreSQL
```

## Quick Start

### Connect to the database

The connection string defines how you "get" to your database. For more on connection strings and URLs see the `PGConnection` section below.

```
do {  
    let p = PGConnection()  
    let status = p.connectdb("host=localhost dbname=postgres")  
    defer {  
        p.close() // close the connection  
    }  
} catch {  
    // handle errors  
}
```

The status returned from the `p.connectdb()` method is either `.ok` or `.bad`. This allows you to quickly catch and abort your connection processing if the database server is unavailable.

### Running Queries

Once you have your connection established, you normally want to run queries on the database to store or retrieve data.

```
let p = PGConnection()  
let status = p.connectdb("host=localhost dbname=postgres")  
  
let result = p.exec(  
    statement: "  
        select datname,datdba,encoding,datistemplate  
        from pg_database  
    ")
```

The result of the query ( `result` ) is returned as type `PGResult`. For more on how to work with these result sets, see the `PGResult` documentation below.

### Parameter binding

"Parameter binding" is an alternative method of passing data to the database. Instead of putting the values directly into the SQL statement, you just use a placeholder and provide the actual values in a `params` array.

```
let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

let result = p.exec(
  statement: "
    insert into test_db (col1, col2, col3)
    values($1, $2, $3)
  ",params: [6, "hello world", "another value"])
```

Saving data can be done without Parameter Binding, however, this shifts the burden of sanitizing your data completely to your data validation prior to addressing the database.

## Accessing results

```
let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

let res = p.exec(statement: "
  select datname,datdba,encoding,datistemplate
  from pg_database
  where encoding = $1",
  params: ["6"])

let num = res.numTuples()
for x in 0..

```

What's happening:

- `res` is assigned the results of the query, its type is `PGResult`
- `res.numTuples()` provides the number of rows returned in the result
- `PGResult` is iterable, so `for x in 0.. provides access to each row in turn`
- `getFieldString`, `getFieldInt`, `getFieldBool` methods are processing the contents of the row and field number requested

## Managing the Connection: PGConnection

### Opening a Connection

There are two accepted formats for the connection string: plain keyword = value strings, and RFC 3986 URIs.

```
let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

//or
let p = PGConnection()
let status = p.connectdb("postgresql://user:password@localhost:5432/dbname")
```

When using keyword/value connection strings, each parameter is in the form keyword = value. To write an empty value, or a value containing spaces, surround it with single quotes. For instance, keyword = 'a value'.

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

For a full list of parameter keywords see the relevant PostgreSQL documentation: <https://www.postgresql.org/docs/current/static/libpq-connect.html#LIBPQ-PARAMKEYWORDS>

When specifying the connection string using a connection URI, the general form is:

```
postgresql://[user[:password]@][netloc][:port]/[dbname][?param1=value1&...]
```

The URI scheme designator can be either `postgresql://` or `postgres://`. Each of the URI parts is optional. The following examples illustrate valid URI syntax uses:

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
```

## Closing a Connection

Once a connection has been opened, it is important to specify a close of the connection.

This can be done with `.finish()`:

```
let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

defer {
    p.finish()
}
```

Note that `.close()` is also valid and is functionally equivalent to `.finish()`. It is included for syntax consistency with other connectors.

## Getting the Status of a Connection

After opening a connection you can see its success status:

```
let p = PGConnection()
let connection = p.connectdb("host=localhost dbname=postgres")

print("The connection status is: \(p.status)")

defer {
    p.finish()
}
```

## The Most Recent Error Status

`.errorMessage()` returns the error message most recently generated by an operation on the connection.

```
p.errorMessage()
```

## Executing SQL Statements

Using the `.exec` method, raw SQL statements are passed to the PostgreSQL server. The statements can be provided either as complete strings or parameterized.

```
let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

// name, oid, integer, boolean
let result = p.exec(
    statement: "
        select datname,datdba,encoding,datistemplate
        from pg_database
    ")
```

The SQL statements can be anything from SELECT, INSERT, UPDATE, DELETE as well as table creation statements or complete database setup scripts.

"Parameter binding" is an alternative method of passing data to the database. It shifts the burden of sanitizing your data completely to your data validation prior to addressing the database. Instead of putting the values directly into the SQL statement, you just use a placeholder and provide the actual values in a `params` array.

```

let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

// name, oid, integer, boolean
let result = p.exec(
  statement: "
    insert into test_db (col1, col2, col3)
    values($1, $2, $3)
    ",params: [6, "hello world", "another value"])

```

## Working with Results: PGResult

PGResult is the container type for all `.exec` method responses.

In the `PGResult` response of a SELECT statement, the number of rows, and the value at a given row and field index can be accessed as below:

```

let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

// name, oid, integer, boolean
let res = p.exec(statement: "
  select datname,datdba,encoding,datistemplate
  from pg_database
  where encoding = $1",
  params: ["6"])

let num = res.numTuples()
for x in 0..

```

What's happening:

- `res` is assigned the results of the query, its type is `PGResult`
- `res.numTuples()` provides the number of rows returned in the result
- `PGResult` is iterable, so `for x in 0.. provides access to each row in turn`
- `getFieldString`, `getFieldInt`, `getFieldBool` methods are processing the contents of the row and field number requested

## Returning the Status of the Executed Statement

```

let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

// name, oid, integer, boolean
let result = p.exec(
  statement: "
    insert into test_db (col1, col2, col3)
    values($1, $2, $3)
    ",params: [6, "hello world", "another value"])

print("Insert statement result status: \(result.status())")

```

The possible values for `.status()` are:

- `.emptyQuery` - The string sent to the server was empty
- `.commandOK` - Successful completion of a command returning no data
- `.tuplesOK` - Successful completion of a command returning data (such as a SELECT or SHOW)
- `.badResponse` - The server's response was not understood
- `.nonFatalError` - A nonfatal error (a notice or warning) occurred
- `.fatalError` - A fatal error occurred
- `.singleTuple` - The result contains a single result tuple from the current command. This status occurs only when single-row mode has been selected for the query
- `.unknown` - An unknown result status was returned. Look in the console log for more detail

## Getting the Number of Rows Returned

In an earlier example above ("Working with results: PGResult"), the select statement returns the `PGResult` into `res`. The number of rows contained in `res` can be accessed using `.numTuples()`:

```
let num = res.numTuples()
```

## Result Field Count

`.numFields()` returns the number of fields returned. Note that this is not the number of rows, but the fields as in a SELECT statement.

## Determining Field Name and Type

Because fields are not addressed using name but by index, if the precise order of the response is not certain, it may be important to determine the name and type. In this case use

`.fieldName(index)` and `.fieldType(index)`:

```
print("The first field name is: \" + res.fieldName(0) + "\")
print("The first field type is: \" + res.fieldType(0) + "\")
```

## Getting Row Data

Once the result has been returned, we can iterate through the results using a for loop. The `tupleIndex` parameter below is a zero-based index.

Use the `.getField*` methods to access data as appropriate.

```
let num = res.numTuples()
for x in 0..<num {
    let c1 = res.getFieldString(tupleIndex: x, fieldIndex: 0)
    let c2 = res.getFieldInt(tupleIndex: x, fieldIndex: 1)
    let c3 = res.getFieldInt(tupleIndex: x, fieldIndex: 2)
    let c4 = res.getFieldBool(tupleIndex: x, fieldIndex: 3)
    print("c1=\" + c1 + "\" c2=\" + c2 + "\" c3=\" + c3 + "\" c4=\" + c4 + "\")
}
```

The `.getField*` methods are:

- `.getFieldString(tupleIndex: Int, fieldIndex: Int)` returns a String type
- `.getFieldInt(tupleIndex: Int, fieldIndex: Int)` returns an Int type
- `.getFieldBool(tupleIndex: Int, fieldIndex: Int)` returns a Boolean type
- `.getFieldInt8(tupleIndex: Int, fieldIndex: Int)` returns an Int8 type
- `.getFieldInt16(tupleIndex: Int, fieldIndex: Int)` returns an Int16 type
- `.getFieldInt32(tupleIndex: Int, fieldIndex: Int)` returns an Int32 type
- `.getFieldInt64(tupleIndex: Int, fieldIndex: Int)` returns an Int64 type
- `.getFieldDouble(tupleIndex: Int, fieldIndex: Int)` returns a Double type
- `.getFieldFloat(tupleIndex: Int, fieldIndex: Int)` returns a Float type
- `.getFieldBlob(tupleIndex: Int, fieldIndex: Int)` returns an [Int8] array

## Testing If a Field Has a Null Value

Similar to the `.getField*` methods, `.fieldIsNull` requires a row (tuple) and field index. The returned value is a Boolean true or false.

```
let nullTest = res.fieldIsNull(tupleIndex: <Int>, fieldIndex: <Int>)
```

## ErrorMessage

To access the error message of the query, use `.errorMessage()`:

```
let result = p.exec(statement: "SELECT * FROM x")

print("Error Message: \" + result.errorMessage() + "\")
```

## Clearing the Result Cursor

Once a result has been processed, leaving the cursor with all the results in memory can be sub-optimal. Clearing the cursor will release the allocated memory.

Clear the result using `.clear()` :

```
let p = PGConnection()
let status = p.connectdb("host=localhost dbname=postgres")

// name, oid, integer, boolean
let result = p.exec(statement: "...")
// process result here

// clearing the cursor:
result.clear()
p.finish()
```

## MongoDB

The MongoDB Connector provides a Swift wrapper around the mongo-c client library, enabling access to MongoDB servers.

This package builds with the Swift Package Manager and is part of the [Perfect](#) project. It was written to be standalone, and does not require PerfectLib or any other components.

Ensure you have installed and activated the latest Swift 3.0 toolchain.

## Relevant Examples

- [MongoDBStORM-Demo](#)
- [Perfect-Session-MongoDB-Demo](#)
- [Perfect-Turnstile-MongoDB-Demo](#)

## Platform-Specific Preparation

### OS X

This package requires the [Homebrew](#) build of mongo-c.

To install Homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

To install mongo-c:

```
brew install mongo-c-driver
```

### Linux

Ensure that you have installed libmongoc.

```
sudo apt-get install libmongoc
```

## Including the MongoDB Driver in Your Project

Add this project as a dependency in your Package.swift file.

```
.Package(
    url: "https://github.com/PerfectlySoft/Perfect-MongoDB.git",
    majorVersion: 2
)
```

For more information about using Perfect libraries with your project, see the chapter on "[Building with Swift Package Manager](#)".

### Quick Start

The following will clone an empty starter project:

```
git clone https://github.com/PerfectlySoft/PerfectTemplate.git
cd PerfectTemplate
```

Add this to the Package.swift file:

```
let package = Package(
    name: "PerfectTemplate",
    targets: [],
    dependencies: [
        .Package(url:"https://github.com/PerfectlySoft/Perfect.git", versions: Version(0,0,0)..
```

Create the Xcode project:

```
swift package generate-xcodeproj
```

Open the generated `PerfectTemplate.xcodeproj` file in Xcode.

The project will now build in Xcode and start a server on localhost port 8181.

**Important:** When a dependency has been added to the project, the Swift Package Manager must be invoked to generate a new Xcode project file. Be aware that any customizations that have been made to this file will be lost.

## Importing MongoDB for Use in Your Project

At the head of your Swift file, import the MongoDB package:

```
import MongoDB
```

## Creating a MongoDB Connection

When you are opening a new connection to a MongoDB server, firstly obtain the connection URL. This will be the fully qualified domain name or IP address of the MongoDB server, with an optional port.

Once you know the connection URL, open a connection as follows:

```
let client = try! MongoClient(uri: "mongodb://localhost")
```

Where "localhost" is replaced by the actual server address.

## Defining a Database

Once the connection has been opened, a database can be assigned:

```
let db = client.getDatabase(name: "test")
```

## Defining a MongoDB Collection

In order to work with a MongoDB Collection, it must be defined:

```
let collection = db.getCollection(name: "testcollection")
```

## Closing Open Connections

Once a connection and associated connections are defined, it is wise to set them up to be closed using a `defer` statement.

```
defer {
    collection.close()
    db.close()
    client.close()
}
```

## Performing a Find

Use the `find` method to find all documents in the collection:

```
let fnd = collection.find(query: BSON())

// Initialize empty array to receive formatted results
var arr = [String]()

// The "fnd" cursor is typed as MongoCursor, which is iterable
for x in fnd! {
    arr.append(x.asString)
}
```

For more detailed documentation on the MongoDB Collections class, see the chapter on [MongoDB Collections](#).

## MongoDB Database

Use the MongoDB Database class to create a reference to a named database using a provided MongoClient instance.

Create a new Mongo Database connection:

```
let database = try! MongoDBDatabase(
    client: <MongoClient>,
    dbName: <String>
)
```

### Closing the Connection

Once the connection is established and the database and collections have been defined, set the connection to close once completed using `defer`. This is done in reverse order: close collections, then databases, then finally the client connection.

```
defer {
    collection.close()
    db.close()
    client.close()
}
```

### Drop the Current Database

Drops the current database, deleting the associated data files.

```
database.drop()
```

### Current Database Name

`name()` returns the name of the current database.

```
let name = database.name()
```

### Create a New Collection

```
database.createCollection(name: <String>, options: <BSON>)
```

#### Parameters

- **name:** String, name of collection to be created
- **options:** BSON document listing options for new collection

### Create Reference to MongoDB Collection Referenced by Name

Use `getCollection` to create a reference to a MongoCollection:

```
let collection = database.getCollection(name: <String>)
```

### Create String Array of Current Database Collections' Names

Use `collectionNames` to create an array of the databases' collection names:

```
let collection = database.collectionNames()
```

## MongoDB Collections

Once your connection and database are defined, you will be working with a collection to create, update, delete, and query documents within that collection.

A collection can be defined either by "cascading" through defining the connection client, the database then the collection:

```
let client = try! MongoClient(uri: "mongodb://localhost")
let db = client.getDatabase(name: "test")
let collection = db.getCollection(name: "testcollection")
```

Or by assigning directly after opening the connection:

```
let client = try! MongoClient(uri: "mongodb://localhost")
let collection = MongoCollection(
    client: client,
    dbName: "test",
    collectionName: "testcollection"
)
```

### Closing Connections

Remember to set up your connections to close:

```
defer {
    collection.close()
    db.close()           // if created by "cascade"
    client.close()
}
```

### Collection Name

To return the collection name as a string:

```
collection.name()
```

### Rename Collection

To rename the collection using `newDbName` and `newCollectionName`, with an option to drop any existing collection immediately instead of after the move:

```
collection.rename(
    newDbName: <String>,
    newCollectionName: <String>,
    dropExisting: <Bool>
)
```

#### Parameters

- **newDbName:** String name for db after move
- **newCollectionName:** String name for collection after move
- **dropExisting:** Bool option to drop any existing collection immediately instead of after move

The returned value is the status of the renaming action.

### Drop Collection

`.drop` removes a collection from the database. The method also removes any indexes associated with the dropped collection.

```
collection.drop()
```

The returned value is the status of the drop action.

### Inserting a Document into a Collection

Insert document into the current collection returning a result status:

```
collection.insert(_  
  document: <BSON>,  
  flag: <MongoInsertFlag>  
)
```

### Parameters

- **document:** BSON document to be inserted
- **flag:** Optional MongoInsertFlag defaults to .None

The returned value is the status of the insert action.

### MongoInsertFlag Options

The MongoInsertFlag enum has the following options:

- **None:** No additional action is to be taken. This is the default option
- **ContinueOnError:** Instruct MongoDB to ignore errors
- **NoValidate:** Ignore validation process

## Updating a Document

To update a document in the collection, assemble the BSON object to replace the existing document, and the selector.

```
collection.update(  
  update: <BSON>,  
  selector: <BSON>,  
  flag: <MongoUpdateFlag>  
)
```

### Parameters

- **update:** BSON document to replace the existing document
- **selector:** BSON document with selection criteria
- **flag:** Optional MongoUpdateFlag defaults to .None

The returned value is the status of the update action.

### MongoUpdateFlag options

The MongoUpdateFlag enum has the following options:

- **None:** No additional action is to be taken. This is the default option
- **Upsert:** Insert, or if selector matches a record, update
- **MultiUpdate:** Update more than one document id matched by the selector
- **NoValidate:** Ignore validation process

## Save

Updates an existing document or inserts a new document, depending on its document parameter.

```
collection.save(document: <BSON>)
```

### Parameters

- **document:** BSON document to save

The returned value is the status of the save action.

- If the document does not contain an `_id` field, a new document will be created
- If an `_id` is specified, `save` will perform an "upsert": If a matching `_id` is found in the collection an update will occur, otherwise an insert will be performed

```
let bson = BSON()
defer {
    bson.close()
}

bson.append(key: "stringKey", string: "String Value")
bson.append(key: "intKey", int: 42)

let result2 = collection.save(document: bson)
```

## Find

Selects documents in a collection and returns a cursor to the selected documents.

```
collection.find(
    query: <BSON>,
    fields: <BSON>,
    flags: <MongoQueryFlag>,
    skip: <Int>,
    limit: <Int>,
    batchSize: <Int>
)
```

### Parameters

- **query:** *(Optional)* Specifies selection filter using query operators. To return all documents in a collection, omit this parameter or pass an empty document (`{}`)
- **fields:** *(Optional)* Specifies the fields to return in the documents that match the query filter. To return all fields in the matching documents, omit this parameter
- **flags:** *(Optional)* Set queryFlags for the current search
- **skip:** *(Optional)* Skip the supplied number of records
- **limit:** *(Optional)* Return no more than the supplied number of records
- **batchSize:** *(Optional)* Change number of automatically iterated documents

### Return Value

A cursor to the documents that match the query criteria. When the `find()` method "returns documents" the method is actually returning a cursor to the documents.

## Find and Modify

Modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the **new** option.

```
collection.findAndModify(
    query: <BSON>,
    sort: <BSON>,
    update: <BSON>,
    fields: <BSON>,
    remove: <Bool>,
    upsert: <Bool>,
    new: <Bool>
)
```

### Parameters

- **query:** *Optional.* The selection criteria for the modification. The query field employs the same query selectors as used in the `db.collection.find()` method. Although the query may match multiple documents, `findAndModify()` will only select one document to modify
- **sort:** *Optional.* Determines which document the operation modifies if the query selects multiple documents. `findAndModify()` modifies the first document in the sort order specified by this argument
- **update:** Must specify either the remove or the update field. Performs an update of the selected document. The update field employs the same update operators or field: value specifications to modify the selected document
- **fields:** *Optional.* A subset of fields to return. The fields document specifies an inclusion of a field with 1, as in: `fields: { : 1, : 1, ... }`
- **remove:** Must specify either the remove or the update field. Removes the document specified in the query field. Set this to true to remove the selected document. The default is false
- **upsert:** *Optional.* Used in conjunction with the update field. When true, `findAndModify()` creates a new document if no document matches the query, or if documents match the query, `findAndModify()` performs an update. To avoid multiple upserts, ensure that the query fields are uniquely indexed. The default is false
- **new:** *Optional.* When true, returns the modified document rather than the original. The `findAndModify()` method ignores the new option for remove operations. The default is false

## Count

Returns the count of documents that would match a `find()` query. The `count()` method does not perform the `find()` operation but instead counts and returns the number of results that match a query.

```
collection.count(  
  query: <BSON>,  
  fields: <BSON>,  
  flags: <MongoQueryFlag>,  
  skip: <Int>,  
  limit: <Int>,  
  batchSize: <Int>  
)
```

### Parameters

- **query:** The query selection criteria
- **fields:** Optional. Specifies the fields to return in the documents that match the query filter. To return all fields in the matching documents, omit this parameter
- **flags:** Optional. Set queryFlags for the current search
- **skip:** Optional. Skip the supplied number of records
- **limit:** Optional. Returns no more than the supplied number of records
- **batchSize:** Optional. Change number of automatically iterated documents

### Return Value

The count of documents that would match a `find()` query. The `count()` method does not perform the `find()` operation but instead counts and returns the number of results that match a query.

## Deleting

Remove the document found using selector returning a result status:

```
collection.remove(  
  selector: <BSON>,  
  flag: <MongoRemoveFlag>  
)
```

### Parameters

- **selector:** BSON document with selection criteria
- **flag:** *Optional* MongoRemoveFlag defaults to `.None`

## Creating an Index

Creates indexes on collections.

```
collection.createIndex(  
  keys: <BSON>,  
  options: <MongoIndexOptions>  
)
```

### Parameters

- **keys:** A document that contains the field and value pairs where the field is the index key and the value describes the type of index for that field. For an ascending index on a field, specify a value of 1; for descending index, specify a value of -1
- **options:** *Optional*. A document that contains a set of options that controls the creation of the index. See MongoIndexOptions for details

## Dropping an Index

Drops or removes the specified index from a collection.

```
collection.dropIndex(name: <String>)
```

## Stats

Returns statistics about the collection formatted according to the options document.

```
collection.stats(options: <BSON>)
```

The options document can contain the following fields and values:

- **scale:** *number, Optional.* The scale used in the output to display the sizes of items. By default, output displays sizes in bytes. To display kilobytes rather than bytes, specify a scale value of 1024
- **indexDetails:** *boolean, Optional.* If true, stats() returns index details in addition to the collection stats. Only works for WiredTiger storage engine. Defaults to false
- **indexDetailsKey:** *document, Optional.* If indexDetails is true, you can use indexDetailsKey to filter index details by specifying the index key specification. Only the index that exactly matches indexDetailsKey will be returned. If no match is found, indexDetails will display statistics for all indexes
- **indexDetailsName:** *string, Optional.* If indexDetails is true, you can use indexDetailsName to filter index details by specifying the index name. Only the index name that exactly matches indexDetailsName will be returned. If no match is found, indexDetails will display statistics for all indexes

## Validate

Validates a collection. The method scans a collection's data structures for correctness and returns a single document that describes the relationship between the logical collection and the physical representation of the data.

```
collection.validate(full: <bool>)
```

The full parameter is optional. Specify true to enable a full validation and to return full statistics. MongoDB disables full validation by default because it is a potentially resource-intensive operation.

## GetLastError

`getLastError()` returns a BSON document with description of last transaction status.

```
collection.getLastError()
```

# MongoDB Client

The MongoClient class is where the initial connection to the MongoDB server is defined.

Create a new Mongo client connection:

```
let client = try! MongoClient(uri: "mongodb://localhost")
```

## Closing the Connection

Once the connection is established and the database and collections have been defined, set the connection to close once completed using `defer`. This is done in reverse order: close collections, then databases, and then finally the client connection.

```
defer {  
    collection.close()  
    db.close()  
    client.close()  
}
```

## Create Database Reference

`getDatabase` returns the specified MongoDB using the current connection.

```
let db = client.getDatabase(  
    dbName: <String>  
)
```

## Parameters

- **dbName:** String name of database to be used

## Create Collection Reference

`getCollection` returns the specified `MongoCollection` from the specified database using the current connection.

```
let collection = client.getCollection(  
    dbName: <String>,  
    collectionName: <String>  
)
```

### Parameters

- **dbName:** String name of database to be used
- **collectionName:** String name of collection to be retrieved

## Get Current Mongo Server Status

`serverStatus` returns: a `Result` object representing the server status.

```
let status = client.serverStatus()
```

## Return String Array of Current Database Names

Use `databaseNames` to build a string array of current database names:

```
let dbnames = client.databaseNames()
```

# Working with BSON

BSON, short for Binary JSON, is a binary-encoded serialization of JSON-like documents. Like JSON, BSON supports the embedding of documents and arrays within other documents and arrays. BSON also contains extensions that allow representation of data types that are not part of the JSON spec. For example, BSON has a `Date` type and a `BinData` type.

## Creating a BSON Object

There are several ways to create a new BSON object:

As an empty BSON object

```
let bsonEmpty = BSON()
```

Specifying a bytes parameter creates a new BSON doc structure using the data provided. The bytes array contains a serialized BSON document.

```
let bsonBytes = BSON(bytes: [UInt8])
```

With a JSON parameter, a new BSON document will be created from the JSON structure provided.

```
let bsonJSON = BSON(json: String)
```

When specifying a document parameter containing BSON data, a new document will be created with the specified BSON data.

```
let bsonBSON = BSON(document: BSON)
```

## Closing the BSON Object

To close, destroy and release the current BSON document:

```
let bson = BSON()  
defer {  
    bson.close()  
}
```

## Converting a BSON Object to a JSON String

Use the `asString` method to convert the BSON document to an extended JSON string.

See <http://docs.mongodb.org/manual/reference/mongodb-extended-json/> for more information on extended JSON.

```
let bson.close = BSON(document: <BSON>)
defer {
    bson.close()
}
print(bson.asString())
```

Use `asArrayString` for outermost arrays:

```
print(bson.asArrayString())
```

## Converting a BSON Object to a Byte Array

Use the `asBytes` method to convert the BSON document to a `[UInt8]` byte array.

```
let byteArray = bson.asBytes()
```

## Appending Data to the BSON Document

Using the `append` method, data can be added to the BSON document.

The `append` method uses the `@discardableResult` property. The result of the operation can be returned or ignored.

If the result of the `append` operation is returned:

- **true**: successful append
- **false**: append would overflow max size

### Variations

Appends a new field to the BSON document of type `BSON_TYPE_OID` using the contents of oid.

```
bson.append(key: <String>, oid: <bson_oid_t>)
```

Appends a new field to the BSON document of type `BSON_TYPE_DATE_TIME` using the contents of dateTime.

```
bson.append(key: <String>, dateTime: <Int64>)
```

Appends a new field to the BSON document of type `BSON_TYPE_DATE_TIME` using the contents of time, a `time_t` @value for the number of seconds since UNIX epoch in UTC.

```
bson.append(key: <String>, time: <time_t>)
```

Appends a new field to the BSON document of type `BSON_TYPE_DOUBLE` using the contents of double.

```
bson.append(key: <String>, double: <Double>)
```

Appends a new field to the BSON document of type `BSON_TYPE_BOOL` using the contents of bool.

```
bson.append(key: <String>, bool: <Bool>)
```

Appends a new field to the BSON document of type UTF-8 encoded string using the contents of string.

```
bson.append(key: <String>, string: <String>)
```

Appends a bytes buffer to the BSON document

```
bson.append(key: <String>, bytes: <[UInt8]>)
```

Appends a new field to self.doc of type `BSON_TYPE_REGEX`. @regex should be the regex string. @options should contain the options for the regex.

Valid options for @options are:

- 'i' for case-insensitive
- 'm' for multiple matching
- 'x' for verbose mode

- 'l' to make \w and \W locale dependent
- 's' for dotall mode ('.' matches everything)
- 'u' to make \w and \W match unicode

For more information on what comprises a BSON regex, see [www.bsonspec.org](http://www.bsonspec.org).

- **key**: The key of the field
- **regex**: The regex to append to the BSON
- **options**: Options for @regex

```
bson.append(key: <String>, regex: <String>, options: <String>)
```

## Appending Arrays to a BSON Document

The `appendArray` method will append a complete array to the BSON document. BSON arrays are like documents where the key is the string version of the index. For example, the first item of the array would have the key "0". The second item would have the index "1".

```
bson.appendArray(key: <String>, array: <BSON>)
```

It is also possible to begin an array append operation, then finish it after further processing, using `appendArrayBegin` and `appendArrayEnd`.

`appendArrayBegin` appends a new field named key to the BSON document; the field is, however, incomplete. @child will be initialized so that you may add fields to the child array. Child will use a memory buffer owned by BSON document, and therefore, grow the parent buffer as additional space is used. This allows a single malloc'd buffer to be used when building arrays which can help reduce memory fragmentation.

The type of @child will be `BSON_TYPE_ARRAY`, and therefore, the keys inside of it MUST be "0", "1", etc.

```
bson.appendArrayBegin(key: <String>, child: <BSON>)
```

`appendArrayEnd` finishes the appending of an array to the BSON document. Child is considered disposed after this call and should not be used any further.

```
bson.appendArrayEnd(key: <String>, child: <BSON>)
```

## Concatenate BSON Documents

Use `concat` to concatenate the `src` parameter contents with the BSON document.

```
bson.concat(src: <BSON>)
```

## Count Elements in BSON Document

Returns the number of elements found in the BSON document.

```
bson.countKeys()
```

## Checks to See if BSON Document Contains a Field Named @key

This function is case-sensitive. Returns `true` if the key was found, otherwise `false`.

```
let hasKey = bson.hasField(key: <String>)
```

## Compare Two BSON Documents for Equality

```
guard let bson == bson2 else {
    return false
}
```

## Compare Two BSON Documents for Sort Priority

Returns `true` if lhs sorts above rhs, `false` otherwise.

```
guard let bson < bson2 else {
    return false
}
```

# MongoDB GridFS

GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16 MB by dividing a single document into parts. Perfect-GridFS provides two different classes to access MongoDB GridFS:

- GridFS: list(), search(), delete(), upload(), download() and drop()
- GridFile: provides file info, download() and dedicated large file operations such as tell() / seek(), partially read()/write(), and download()

## macOS Building Notes

Please *NOTE* that a special BUG must be fixed on mongoc-library before compiling this module. Please use mongo-c-driver 1.7.1+ versions. [More Information about mongoc BSON-API file property setting issue](#)

## GridFS Class

Accessing a GridFS object in a Perfect MongoDB context is simple. First, make sure that a mongo client is already available, then call `gridfs()` method with database name and a prefix string as an option. Check the example code below:

```
// suppose let client = try MongoClient(uri: "mongodb://...")
let gridfs = try client.gridfs(database: "test")
```

or get the gripfs handle with an optional prefix:

```
let gridfs = try client.gridfs(database: "test", prefix: "clip")
```

## GridFS Methods

GridFS has methods listed below:

- list(optional filter): show all / parts of files on the server
- drop(): drop all files and the gridfs system from the database
- search(filename): search a file on the server
- delete(filename): delete a file from the server
- upload(from, to, contentType, md5, metaData, aliases): upload a local file to server
- download(from, to): download a remote to local path
- close(): close gridfs handle

### List files

`list()` returns an array of GridFile objects with an optional BSON type parameter called `filter`, which can narrow down the list result by setting up a proper query.

```
func list(filter: BSON? = nil) throws -> [GridFile]
```

By default, `list()` doesn't require any parameters but actually will perform a listing in alphabetic order by setting a bson query with a key `$order` internally:

```
let list = try gridfs.list()
print("\(list.count) file(s) found")
```

### Drop the Current GridFS

Method `drop()` Requests that an entire GridFS be dropped, including all files associated with it.

```
try gridfs.drop()
```

### Search for a file

To search for a file on server and get information or content back, use `search()` method. It will return a `GridFile` object which will be discussed later in this article. If failed, it will throw a MongoClientError:

```
let f = try gridfs.search(name: "file.name.on.server")
print(f.id)
print(f.length)
```

## Delete a file

Call method `delete()` to remove a file on server. If failed, it will throw a `MongoClientError`:

```
try gridfs.delete(name: "file.name.on.server")
```

## Upload a file

Specify a local file path and its expected remote file name to upload with properties:

```
let gridfile = try gridfs.upload(from: "/path/to/local.file", to: "name.on.server.type", contentType: "primaryType/subType", md5: "MD5 string",
metadata: BSON(json: "meta data in a json string"), aliases: BSON(json: "aliases in a json string"))
print(gridfile.length)
```

Parameters:

- from: a required string stands for the local path of file to upload
- to: a required string stands for the destined file name on the remote server
- contentType: an optional string stands for the file type. Default value is `text/plain`
- md5: an optional string stands for the MD5 hash of the file to upload. *NOTE* [Perfect COpenSSL](#) and [Perfect COpenSSL Linux Edition](#) provide MD5 caculation API
- metaData: an optional BSON stands for the meta data of the file to upload.
- aliases: an optional BSON stands for the aliases of the file to upload.

## Download a file

Specify a local path to download a file existing on server or throws a `MongoClientError` if failed:

```
do {
    let bytes = try gridfs.download(from: "name.on.server", to: "/local/path/downloaded.name")
    print("\(bytes) downloaded")
}catch(let err) {
    print("Unexpected \(err) in downloading")
}
```

## Closing the Connection

Practically, Swift's garbage collection mechanism will automatically release the `GridFS` / `GridFile` objects and related resources, however, users can also call defer closing manually if preferred:

```
defer {
    gridfs.close()
} //end defer
```

## GridFile class

Method `search()` and `list()` return `GridFile` and `[GridFile]` array correspondingly. `GridFile` is for dealing with file information and more complicated file operations such as file cursor and partially read / write, as described below:

### GridFile properties (READONLY):

- id: oid string property of `GridFile`.
- md5: md5 string property of `GridFile`.
- aliases: a BSON property which represents the aliases of the `GridFile`.
- contentType: string property of `GridFile`.
- length: length of the `GridFile`, an `Int64` number.
- uploadDate: `Int64` unix epoch time stamp of the `GridFile`.
- fileName: file name stored in the remote server.
- metaData: a BSON type property of `GridFile` to hold the meta data.

### GridFile methods:

Besides above properties, `GridFile` also has a few available methods as below:

- download(): directly download the whole file
- tell(): a `UInt64` position of current gridfile instance's cursor is pointing at

- `seek(cursor, optional whence)`: move the file cursor to objective position referring to whence it goes
- `paritallyRead(amount, optional timeout)`: read a part of the file from the current file cursor position and return it in a binary buffer [UInt8]
- `partiallyWrite(bytes, optional timeout)`: write a buffer to the current file cursor position
- `close()`: close the GridFile instance handle

## download()

Method `download()` of GridFile class is the lower base of `GridFS.download()`, which actually calls `GridFile.download()` internally. If there is an available GridFile handler, simply apply the local path for downloading:

```
do {
    let bytes = try gridfile.download(to: "/local/path/downloaded.name")    print("Totally \(bytes) downloaded.")
}catch(let err) {
    print("Unexpected \(err) in downloading")
}
```

## tell()

Method `tell()` will return a `UInt64` integer to represent the current cursor position of `GridFile` instance:

```
let position = gridfile.tell()
print("current position is \(position)")
```

## seek()

Use `seek()` to move the current GridFile cursor to a new position, with an optional supporting parameter to indication whence the cursor will offset to, i.e., `.begin` as default for calculate offset from beginning of the file, `.current` for an offset from current position, and `.end` represents an offset from the every end of `GridFile`:

```
enum Whence {
    // offset from starting point of file
    case begin
    // offset from current file cursor
    case current
    // offset from the last byte of the file
    case end
} //end whence

func seek(cursor: Int64, whence: Whence = .begin) throws
```

For example, the following code demonstrates how to move the file cursor to 1MB from the starting point:

```
try gridfile.seek(1048576)
```

## paritallyRead()

To deal with large files, GridFile provides partially read() / write() methods:

```
func paritallyRead(amount: UInt32, timeout: UInt32 = 0) throws -> [UInt8]
```

Parameter `amount` is the bytes count to read, and the `timeout` value represents the milliseconds to wait for reading. If failed to read, it would throw a `MongoClientError`. Please **⚠️NOTE⚠️** the reading starting point is controlled by `seek()` and is subjected to changed after every `paritallyRead()` or `partiallyWrite()`. The following example demonstrates how it works:

```
let file = try gridfs.search("a.large.file")
let cursor = file?.tell()
print("if no operations before, the cursor should be zero, now it is \(cursor)")
let oneMegaByte = 1048576
// move the cursor to 100MB position from the file starting point
try file?.seek(cursor: Int64(oneMegaByte * 100))
// then read 1MB bytes into a [UInt8] buffer
let bytes = try file?.paritallyRead(amount: UInt32(oneMegaByte))
```

## partiallyWrite()

Method `partiallyWrite()` performs an opposite operation against to `paritallyRead()`. If failed to write, it would throw a `MongoClientError`.

```
public func partiallyWrite(bytes:[UInt8], timeout:UInt32 = 0) throws -> Int
```

The first parameter `bytes` is the buffer to write, with a second optional parameter of timeout in milliseconds to wait for writing. If ignored, the writing operation would immediately return. ⚠️ *Again* ⚠️, the writing starting point is controlled by `seek()` and is subjected to changed after every `paritallyRead()` or `partiallyWrite()` :

```
// move the file cursor to an offset of 1k from end of file
try file?.seek(cursor: 1024, whence: .end)
let buffer:[UInt8] = [1,2,3,4,5]
// write the array to the current position
let totalWritten = try file?.partially(bytes: buffer)
print(totalWritten)
```

If succeed, the above code would write five different bytes at the 1k point from end of the file.

### **close()**

Practically, Swift's garbage collection mechanism will automatically release the `GridFS` / `GridFile` objects and related resources, however, users can also call defer closing manually if preferred:

```
defer {
    gridfile.close()
} //end defer
```

## Performance Suggestion

Since large file uploads / downloads are usually time consuming, try using `Threading.dispatch {}` to avoid blocking the main thread, or use partially read()/write() incrementally. For more information, see [Perfect Threading](#)

# Apache CouchDB

The CouchDB connector provides access to Apache CouchDB servers. It allows you to search for, add, update and delete documents in a CouchDB server.

## System Requirements

This module uses libcurl along with the Perfect-CURL package.

### macOS

macOS currently includes suitable versions of the dependent libraries and no manual installation is necessary.

### Linux

Install the libcurl4-openssl-dev package through apt.

```
sudo apt-get install libcurl4-openssl-dev
```

## Setup

Add the "Perfect-CouchDB" project as a dependency in your Package.swift file:

```
.Package(
    url:"https://github.com/PerfectlySoft/Perfect-CouchDB.git",majorVersion: 0)
```

### Import

In any of the source files where you intend to use with this module, add the following import:

```
import PerfectCouchDB
```

## Overview

The main objects that you will be working with when accessing CouchDB databases are described here.

## CouchDB()

This is the main interface to a CouchDB server. The class has 4 publicly accessible properties:

- `debug`: Defines the level of console & log output. Default: `false`;
- `database`: The instance's current database;
- `connector`: The instance's connector information;
- `authentication`: A container for the authentication settings.

## Server API Functions

- `serverInfo()` - returns the current server information
- `serverActiveTasks()` - returns the active tasks on the server
- `listDatabases()` - lists all accessible databases on the server

## Database operations

To check if a database exists, use the `databaseExists(String)` method, which returns a true or false response.

```
db.databaseExists("mydb")
```

Deleting a database is done via the `databaseDelete(String)` method. A `CouchDBResponse` object is returned containing the result of the operation.

```
db.databaseDelete("mydb")
```

Getting database information is accomplished using `databaseInfo(String)`. The response is a tuple of `CouchDBResponse, [String:Any]` where `CouchDBResponse` is the status of the request (200 OK; 404 Not Found), and the `[String:Any]` are the name/value pairs of the response.

```
db.databaseInfo("mydb")
```

Possible information responses include:

- `committed_update_seq (Int)` – The number of committed update.
- `compact_running (Bool)` – Set to true if the database compaction routine is operating on this database.
- `db_name (String)` – The name of the database.
- `disk_format_version (Int)` – The version of the physical format used for the data when it is stored on disk.
- `data_size (Int)` – The number of bytes of live data inside the database file.
- `disk_size (Int)` – The length of the database file on disk. Views indexes are not included in the calculation.
- `doc_count (Int)` – A count of the documents in the specified database.
- `doc_del_count (Int)` – Number of deleted documents
- `instance_start_time (String)` – Timestamp of when the database was opened, expressed in microseconds since the epoch.
- `purge_seq (Int)` – The number of purge operations on the database.
- `update_seq (Int)` – The current number of updates to the database.

## Adding a document

To add a new document to the database, using the supplied JSON document structure, use the `addDoc` method.

If the JSON structure includes the `_id` field, then the document will be created with the specified document ID. If the `_id` field is not specified, a new unique ID will be generated, following whatever UUID algorithm is configured for that server.

Parameters:

- `db`: The database name
- `doc`: The document to be stored.

The stored document can be supplied as a JSON encoded string, or as a `[String: Any]` type.

This returns a tuple of `CouchDBResponse` and the returned JSON as `[String:Any]`.

```
let data = [ "one": "ONE", "two": "TWO" ]
db.addDoc("mydb", doc: data)
```

Alternatively, you can use the `create()` method. This creates a new named document, or creates a new revision of the existing document. Unlike the `addDoc` method, you must

specify the document ID in the request URL.

Parameters:

- db – Database name
- docid – Document ID
- doc - [String: Any] object to be stored as JSON

Response JSON Object:

- id (string) – Document ID
- ok (boolean) – Operation status
- rev (string) – Revision MVCC token

Status Codes:

- 201 Created – Document created and stored on disk
- 202 Accepted – Document data accepted, but not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Write privileges required
- 404 Not Found – Specified database or document ID doesn't exist
- 409 Conflict – Document with the specified ID already exists or specified revision is not latest for target document

```
db.create("mydb", docid: String, doc: [String: Any])
```

## Duplicating a specific document

The `copy` method copies an existing document to a new or existing document. The source document docid is specified, with the destination parameter specifying the target document.

Parameters:

- db – Database name
- docid – Document ID
- rev (string) – Revision to copy from. Optional
- destination – Destination document

Response JSON Object:

- id (string) – Document document ID
- ok (boolean) – Operation status
- rev (string) – Revision MVCC token

Status Codes:

- 201 Created – Document successfully created
- 202 Accepted – Request was accepted, but changes are not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Read or write privileges required
- 404 Not Found – Specified database, document ID or revision doesn't exist
- 409 Conflict – Document with the specified ID already exists or specified revision is not latest for target document

```
db.copy("mydb", docid: String, doc: [String:Any], rev: String, destination: String)
```

## Searching for documents

The `filter` method returns a JSON structure of all of the documents in a given database. The information is returned as a JSON structure containing meta information about the return structure, including a list of all documents and basic contents, consisting the ID, revision and key. The key is the from the document's `_id`.

Parameters:

- db – Database name
- queryParams - CouchDBQuery object.

Response JSON Object:

- offset (Int) – Offset where the document list started
- rows (Array) – Array of view row objects. By default the information returned contains only the document ID and revision.
- total\_rows (Int) – Number of documents in the database/view. Note that this is not the number of rows returned in the actual query.

- `update_seq` (Int) – Current update sequence for the database

```
let query = CouchDBQuery()
// See the CouchDBQuery section for configuration options
db.filter("mydb", query)
```

Alternatively, you can use the `find` method. This differs from `filter` in the options supplied to the API. `find` uses a declarative JSON querying syntax.

Parameters supported by the CouchDBQuery in this context:

- `selector` ([String:Any]) – JSON object describing criteria used to select documents. More information provided in the section on selector syntax.
- `limit` (Int) – Maximum number of results returned. Default is 25. Optional
- `skip` (Int) – Skip the first ‘n’ results, where ‘n’ is the value specified. Optional
- `sort` ([String:Any]) – JSON array following sort syntax. Optional
- `fields` ([String:Any]) – JSON array specifying which fields of each object should be returned. If it is omitted, the entire object is returned. More information provided in the section on filtering fields. Optional
- `use_index` ([String:Any]) – Instruct a query to use a specific index.

Response JSON Object:

- `docs` (object) – Documents matching the selector

Status Codes:

- 200 OK – Request completed successfully
- 400 Bad Request – Invalid request
- 401 Unauthorized – Read permission required
- 500 Internal Server Error – Query execution error

```
let findObject = CouchDBQuery()
findObject.selector = data
findObject.limit = 10
findObject.skip = 0

do {
    let (code, response) = try db.find("mydb", queryParams: findObject)
} catch {
    print("\(error)")
}
```

## Retrieving a specific document

To retrieve a specific document from a database, use the `get()` method.

This method returns document by the specified docid from the specified db. Unless you request a specific revision, the latest revision of the document will always be returned.

Parameters:

- `db` – Database name
- `docid` – Document ID

Query Parameters:

- `attachments` (Bool) – Includes attachments bodies in response. Default is false
- `att_encoding_info` (Bool) – Includes encoding information in attachment stubs if the particular attachment is compressed. Default is false.
- `atts_since` (array) – Includes attachments only since specified revisions. Doesn’t include attachments for specified revisions. Optional
- `conflicts` (Bool) – Includes information about conflicts in document. Default is false
- `deleted_conflicts` (Bool) – Includes information about deleted conflicted revisions. Default is false
- `latest` (Bool) – Forces retrieving latest “leaf” revision, no matter what rev was requested. Default is false
- `local_seq` (Bool) – Includes last update sequence for the document. Default is false
- `meta` (Bool) – Acts same as specifying all conflicts, deleted\_conflicts and open\_revs query parameters. Default is false
- `open_revs` (array) – Retrieves documents of specified leaf revisions. Additionally, it accepts value as all to return all leaf revisions. Optional
- `rev` (String) – Retrieves document of specified revision. Optional
- `revs` (Bool) – Includes list of all known document revisions. Default is false
- `revs_info` (Bool) – Includes detailed information for all known document revisions. Default is false

Note that all these query params are optional apart from the document id (docid).

Response JSON Object:

- `_id` (string) – Document ID
- `_rev` (string) – Revision MVCC token
- `_deleted` (boolean) – Deletion flag. Available if document was removed
- `_attachments` (object) – Attachment's stubs. Available if document has any attachments
- `_conflicts` (array) – List of conflicted revisions. Available if requested with `conflicts=true` query parameter
- `_deleted_conflicts` (array) – List of deleted conflicted revisions. Available if requested with `deleted_conflicts=true` query parameter
- `_local_seq` (string) – Document's update sequence in current database. Available if requested with `local_seq=true` query parameter
- `_revs_info` (array) – List of objects with information about local revisions and their status. Available if requested with `open_revs` query parameter
- `_revisions` (object) – List of local revision tokens without. Available if requested with `revs=true` query parameter

Status Codes:

- 200 OK – Request completed successfully
- 304 Not Modified – Document wasn't modified since specified revision
- 400 Bad Request – The format of the request or revision was invalid
- 401 Unauthorized – Read privilege required
- 404 Not Found – Document not found

```
db.get(
  "mydb",
  docid: "docidString",
  attachments: false,
  att_encoding_info: false,
  atts_since: [String](),
  conflicts: false,
  deleted_conflicts: false,
  latest: false,
  local_seq: false,
  meta: false,
  open_revs: [String](),
  rev: "specificRevision",
  revs: false,
  revs_info: false
)
```

## Update a document

To update an existing document you must specify the current revision number within the `rev` parameter.

Parameters:

- `db` – Database name
- `docid` – Document ID
- `doc` – Document body as `[String: Any]`
- `rev` – Document Revision

Response JSON Object:

- `id` (string) – Document document ID
- `ok` (boolean) – Operation status
- `rev` (string) – Revision MVCC token

```
db.update("mydb", docid: String, doc: [String:Any], rev: String)
```

## Delete a document

The `delete` method marks the specified document as deleted by adding a field `_deleted` with the value `true`. Documents with this field will not be returned within requests anymore, but stay in the database. You must supply the current (latest) revision by using the `rev` parameter.

Note: CouchDB doesn't completely delete the specified document. Instead, it leaves a tombstone with very basic information about the document. The tombstone is required so that the delete action can be replicated across databases.

Parameters:

- `db` – Database name
- `docid` – Document ID

- **rev** - Actual document's revision

Response JSON Object:

- **id** (string) – Document ID
- **ok** (boolean) – Operation status
- **rev** (string) – Revision MVCC token

Status Codes:

- 200 OK – Document successfully removed
- 202 Accepted – Request was accepted, but changes are not yet stored on disk
- 400 Bad Request – Invalid request body or parameters
- 401 Unauthorized – Write privileges required
- 404 Not Found – Specified database or document ID doesn't exist
- 409 Conflict – Specified revision is not the latest for target document

```
db.delete("mydb", docid: String, rev: String)
```

## CouchDBAuthentication()

This is the container struct for holding the CouchDB Authentication information.

- **authType**: The authentication mode. `.none`, `.basic`, or `.session`
- **username**: Username for the authentication
- **password**: Password for the authentication
- **token**: The token obtained if auth type is `.session`. Irrelevant if auth type is not `.session`

An instance of the struct can be set up in the following ways:

```
let auth = CouchDBAuthentication("username","pwd")
let auth = CouchDBAuthentication("username","pwd", auth: .basic)
```

Calculating the base64 token:

```
let auth = CouchDBAuthentication("username","pwd")
let token = auth.basic()
```

## CouchDBConnector()

This struct holds the basic CouchDB connector information.

- **ssl**: Bool - SSL mode enabled/disabled. Default value: disabled (false)
- **host**: String - CouchDB Server hostname or IP address. Default: localhost
- **port**: Int - CouchDB Server port. Default: 5984

```
var thisConnector = CouchDBConnector()
thisConnector.host = "192.168.0.12"
thisConnector.ssl = true
thisConnector.port = 5984
```

## CouchDBQuery()

CouchDBQuery is container class for query params used in the filter and find functions.

It has the following properties:

- **conflicts** : Bool – Includes conflicts information in response. Ignored if `include_docs` isn't true. Default is false. Used for a filter query only.
- **descending** : Bool – Return the documents in descending by key order. Default is false. Used for a filter query only.
- **endkey** : String – Stop returning records when the specified key is reached. Optional. Used for a filter query only.
- **endkey\_docid** : String – Stop returning records when the specified document ID is reached. Optional. Used for a filter query only.
- **include\_docs** : Bool – Include the full content of the documents in the return. Default is false. Used for a filter query only.
- **inclusive\_end** : Bool – Specifies whether the specified end key should be included in the result. Default is true. Used for a filter query only.
- **key** : String – Return only documents that match the specified key. Optional. Used for a filter query only.
- **keys** : [String] – Return only documents that match the specified keys. Optional. Used for a filter query only.
- **limit** : Int – Limit the number of the returned documents to the specified number. Optional.

- **skip** : Int – Skip this number of records before starting to return the results. Default is 0.
- **stale** : CouchDBQueryStale – Allow the results from a stale view to be used, without triggering a rebuild of all views within the encompassing design doc. Supported values: ok and update\_after. Optional, default is .ignore. Used for a filter query only.
- **startkey** : String – Return records starting with the specified key. Optional. Used for a filter query only.
- **startkey\_docid** : String – Return records starting with the specified document ID. Optional. Used for a filter query only.
- **update\_seq** : Bool – Response includes an update\_seq value indicating which sequence id of the underlying database the view reflects. Default is false. Used for a filter query only.
- **selector** : [String:Any] – JSON object describing criteria used to select documents. More information provided in the section on selector syntax. Used for a find query only.
- **sort** : [String:Any] – Name/value array following sort syntax. Optional. Used for a find query only.
- **fields** : [String:Any] – Name/value array specifying which fields of each object should be returned. If it is omitted, the entire object is returned. More information provided in the section on filtering fields. Optional. Used for a find query only.
- **use\_index** : [String:Any] – Instruct a query to use a specific index. Specified either as "<design\_document>" or ["<design\_document>", "<index\_name>"]. Optional. Used for a find query only.

## Error Responses

### enum CouchDBResponse

The CouchDBResponse enum details the available conditions encountered when executing a request to the CouchDB server.

- **.undefined** : Undefined.
- **.ok** : Request completed successfully.
- **.created** : Document created successfully.
- **.accepted** : Request has been accepted, but the corresponding operation may not have completed. This is used for background operations, such as database compaction.
- **.notModified** : The additional content requested has not been modified. This is used with the ETag system to identify the version of information returned.
- **.badRequest** : Bad request structure. The error can indicate an error with the request URL, path or headers. Differences in the supplied MD5 hash and content also trigger this error, as this may indicate message corruption.
- **.unauthorized** : The item requested was not available using the supplied authorization, or authorization was not supplied.
- **.forbidden** : The requested item or operation is forbidden.
- **.notFound** : The requested content could not be found. The content will include further information, as a JSON object, if available. The structure will contain two keys, error and reason. For example `{ "error": "not_found", "reason": "no_db_file" }`
- **.resourceNotAllowed** : A request was made using an invalid HTTP request type for the URL requested. For example, you have requested a PUT when a POST is required. Errors of this type can also triggered by invalid URL strings.
- **.notAcceptable** : The requested content type is not supported by the server.
- **.conflict** : Request resulted in an update conflict.
- **.preconditionFailed** : The request headers from the client and the capabilities of the server do not match.
- **.badContentType** : The content types supported, and the content type of the information being requested or submitted indicate that the content type is not supported.
- **.requestedRangeNotSatisfiable** : The range specified in the request header cannot be satisfied by the server.
- **.expectationFailed** : When sending documents in bulk, the bulk load operation failed.
- **.internalServerError** : The request was invalid, either because the supplied JSON was invalid, or invalid information was supplied as part of the request. # Perfect-LDAP

This project provides an express OpenLDAP class wrapper which enable access to OpenLDAP servers and Windows Active Directory server.

This package builds with Swift Package Manager and is part of the [Perfect](#) project.

Ensure you have installed and activated the latest Swift 3.0 tool chain.

*Caution:* for the reason that LDAP is widely using in many different operating systems with variable implementations, API marked with (⚠ EXPERIMENTAL ⚠) indicates that this method might not be fully applicable to certain context. However, as an open source software library, you may modify the source code to meet a specific requirement.

## Quick Start

Add the following dependency to your project's Package.swift file:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-LDAP.git", majorVersion: 1)
```

Then import PerfectLDAP to your source code:

```
import PerfectLDAP
```

## Connect to LDAP Server

You can create actual connections as need with or without login credential. The full API is `LDAP(url:String, loginData: Login?, codePage: Iconv.CodePage)`. The `codePage` option is for those servers applying character set other than .UTF8, e.g., set `codePage: .GB2312` to connect to LDAP server in Simplified Chinese.

## TLS Option

PerfectLDAP provides TLS options for network security considerations, i.e, you can choose either `ldap://` or `ldaps://` for connections, as demo below:

```
// this will connect to a 389 port without any encryption
let ld = try LDAP(url: "ldap://perfect.com")
```

or,

```
// this will connect to a 636 port with certificates
let ld = try LDAP(url: "ldaps://perfect.com")
```

## Connection Timeout

Once connected, LDAP object could be set with timeout option, and the timing unit is second:

```
// set the timeout for communication. In this example, connection will be timeout in ten seconds.
connection.timeout = 10
```

## Login or Anonymous

Many servers mandate login before performing any actual LDAP operations, so PerfectLDAP provides multiple login options as demo below:

```
// this snippet demonstrate how to connect to LDAP server with a login credential
// NOTE: this kind of connection will block the thread until server return or timeout.
// create login credential
let credential = LDAP.login( ... )
let connection = try LDAP(url: "ldaps://...", loginData: login)
```

Aside from the above synchronous login option, a two-phased threaded login process could also bring more controls to the application:

```
// first create a connection
let connection = try LDAP(url: "ldaps:// ...")

// setup login info
let credential = LDAP.login( ... )

// login in a separated thread
connection.login(info: credential) { err in
    // if err is not nil, then something must be wrong in the login process.
}
```

## Login Options

PerfectLDAP provides a special object called `LDAP.Login` to store essential account information for LDAP connections and the form of constructor is subject to the authentication types:

### Simple Login

To use simple login method, simply call `LDAP.login(binddn: String, password: String)`, as shown below:

```
let credential = LDAP.Login(binddn: "CN=judy,CN=Users,DC=perfect,DC=com", password: "0penLDAP")
```

## GSSAPI

To apply GSSAPI authentication, call `LDAP.login(user:String, mechanism: AuthType)` to construct a login credential (assuming the user has already acquired a valid ticket):

```
// this call will generate a GSSAPI login credential
let credential = LDAP.login(user: "judy", mechanism: .GSSAPI)
```

## GSS-SPNEGO and Digest-MD5 (⚠️EXPERIMENTAL⚠️)

To apply other SASL mechanisms, such as GSS-SPNEGO and Digest-MD5 interactive logins, call

`LDAP.login(authname: String, user: String, password: String, realm: String, mechanism: AuthType)` as shown below:

```
// apply DIGEST-MD5 mechanism.
let credential = LDAP.Login(authname: "judy", user: "DN:CN=judy,CN=Users,DC=perfect,DC=com", password: "OpenLDAP", realm: "PERFECT.COM", mechanism: .DIGEST)
```

⚠️ **NOTE** ⚠️ The `authname` is equivalent to `SASL_CB_AUTHNAME` and `user` is actually the macro of `SASL_CB_USER`. If any parameter above is not applicable to your case, simply assign an empty string "" to ignore it.

## Search

PerfectLDAP provides asynchronous and synchronous version of searching API with the same parameters:

### Synchronous Search

Synchronous search will block the thread until server returns, the full api is

`LDAP.search(base:String, filter:String, scope:Scope, attributes: [String], sortBy: String) throws -> [String:[String:Any]]`. Here is an example:

```
// perform an ldap search synchronously, which will return a full set of attributes
// with a natural (unsorted) order, in form of a dictionary.
let res = try connection.search(base: "CN=Users,DC=perfect,DC=com", filter:"(objectclass=*)")

print(res)
```

### Asynchronous Search

Asynchronous search allows performing search in an independent thread. Once completed, the thread will call back with the result set in a dictionary. Full api of asynchronous search is

`LDAP.search(base:String, filter:String, scope:Scope, attributes: [String], sortBy: String, completion: @escaping ([String:[String:Any]])->`

The equivalent example is:

```
// perform an ldap search asynchronously, which will return a full set of attributes
// with a natural (unsorted) order, in form of a dictionary.
connection.search(base: "CN=Users,DC=perfect,DC=com", filter:"(objectclass=*)") {
    res in
    print(res)
}
```

### Parameters of Search

- `base: String`, search base domain (dn), default = ""
- `filter: String`, the filter of query, default is `"(objectclass=*)"`, means all possible results
- `scope: Searching Scope`, i.e., `.BASE`, `.SINGLE_LEVEL`, `.SUBTREE` or `.CHILDREN`
- `sortBy`: a sorting string, may also be generated by `LDAP.sortingString()`
- `completion`: callback with a parameter of dictionary, empty if failed

### Server Side Sort (⚠️ EXPERIMENTAL ⚠️)

The `sortBy` parameter is a string that instructs the remote server to perform search with a sorted set. PerfectLDAP provides a more verbal way to build such a string, i.e, an array of tuples to describe what attributes would control the result set:

```
// each tuple consists two parts: the sorting field and its order - .ASC or .DSC
let sort = LDAP.sortingString(sortBy: [("description", .ASC)])
```

### Limitation of Searching Result

Once connected, LDAP object could be set with an limitation option - `LDAP.limitation`. It is an integer which specifies the maximum number of entries that can be returned on a search operation.

```
// set the limitation for searching result set. In this example, only the first 1000 entries will return.
connection.limitation = 1000
```

## Attribute Operations

PerfectLDAP provides add() / modify() and delete() for attributes operations with both synchronous and asynchronous options.

## Add Attributes (⚠️EXPERIMENTAL⚠️)

Function `LDAP.add()` can add attributes to a specific DN with parameters below:

- distinguishedName: String, specific DN
- attributes:[String:[String]], attributes as an dictionary to add. In this dictionary, every attribute, as a unique key in the dictionary, could have a series of values as an array.

Both asynchronous add() and synchronous add() share the same parameters above, for example:

```
// try an add() synchronously.
do {
  try connection.add(distinguishedName: "CN=judy,CN=User,DC=perfect,DC=com", attributes: ["mail":["judy@perfect.com", "judy@perfect.org"]])
}catch (let err) {
  // failed for some reason
}

// try and add() asynchronously:
connection.add(distinguishedName: "CN=judy,CN=User,DC=perfect,DC=com", attributes: ["mail":["judy@perfect.com", "judy@perfect.org"]]) { err in
  // if nothing wrong, err will be nil
}
```

## Modify Attributes

Function `LDAP.modify()` can modify attributes from a specific DN with parameters below:

- distinguishedName: String, specific DN
- attributes:[String:[String]], attributes as an dictionary to modify. In this dictionary, every attribute, as a unique key in the dictionary, could have a series of values as an array.

Both asynchronous modify() and synchronous modify() share the same parameters above, for example:

```
// try an modify() synchronously.
do {
  try connection.modify(distinguishedName: "CN=judy,CN=User,DC=perfect,DC=com", attributes: ["codePage":["437"]])
}catch (let err) {
  // failed for some reason
}

// try and modify() asynchronously:
connection.modify(distinguishedName: "CN=judy,CN=User,DC=perfect,DC=com", attributes:["codePage":["437"]]) { err in
  // if nothing wrong, err will be nil
}
```

## Delete Attributes (⚠️EXPERIMENTAL⚠️)

Function `LDAP.delete()` can delete attributes from a specific DN with only one parameter:

- distinguishedName: String, specific DN

Both asynchronous delete() and synchronous delete() share the same parameter above, for example:

```
// try an delete() synchronously.
do {
  try connection.delete(distinguishedName: "CN=judy,CN=User,DC=perfect,DC=com")
}catch (let err) {
  // failed for some reason
}

// try and delete() asynchronously:
connection.delete(distinguishedName: "CN=judy,CN=User,DC=perfect,DC=com") { err in
  // if nothing wrong, err will be nil
}
```

# Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache, and message broker.

More info at <http://redis.io>

## Quick Start

---

Get a redis client with defaults:

```
RedisClient.getClient(withIdentifier: RedisClientIdentifier()) {
    c in
    do {
        let client = try c()
        ...
    } catch {
        ...
    }
}
```

Ping the server:

```
client.ping {
    response in
    defer {
        RedisClient.releaseClient(client)
    }
    guard case .simpleString(let s) = response else {
        ...
        return
    }
    XCTAssert(s == "PONG", "Unexpected response \(response)")
}
```

Set/get a value:

```
let (key, value) = ("mykey", "myvalue")
client.set(key: key, value: .string(value)) {
    response in
    guard case .simpleString(let s) = response else {
        ...
        return
    }
}
client.get(key: key) {
    response in
    defer {
        RedisClient.releaseClient(client)
    }
    guard case .bulkString = response else {
        ...
        return
    }
    let s = response.toString()
    XCTAssert(s == value, "Unexpected response \(response)")
}
```

Pub/sub:

```

RedisClient.getClient(withIdentifier: RedisClientIdentifier()) {
    c in
    do {
        let client1 = try c()
        RedisClient.getClient(withIdentifier: RedisClientIdentifier()) {
            c in
            do {
                let client2 = try c()
                client1.subscribe(channels: ["foo"]) {
                    response in
                    client2.publish(channel: "foo", message: .string("Hello!")) {
                        response in
                        client1.readPublished(timeoutSeconds: 5.0) {
                            response in
                            guard case .array(let array) = response else {
                                ...
                                return
                            }
                            XCTAssert(array.count == 3, "Invalid array elements")
                            XCTAssert(array[0].toString() == "message")
                            XCTAssert(array[1].toString() == "foo")
                            XCTAssert(array[2].toString() == "Hello!")
                        }
                    }
                }
            } catch {
                ...
            }
        } catch {
            ...
        }
    }
}

```

## Building

Add this project as a dependency in your Package.swift file.

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-Redis.git", Version(0,0,0)..Version\(10,0,0\))
```

## FileMaker

The FileMaker Server connector provides access to FileMaker Server databases using the XML Web publishing interface. It allows you to search for, add, update and delete records in a hosted FileMaker database.

## System Requirements

This module uses libxml2 and libcurl along with the [Perfect-XML](#) and [Perfect-CURL](#) packages.

### macOS

macOS currently includes suitable versions of the dependent libraries and no manual installation is necessary.

### Linux

Install the libcurl4-openssl-dev & libxml2-dev packages through apt.

```
sudo apt-get install libcurl4-openssl-dev libxml2-dev
```

## Setup

Add the "Perfect-FileMaker" project as a dependency in your Package.swift file:

```
.Package(  
    url:"https://github.com/PerfectlySoft/Perfect-FileMaker.git",  
    majorVersion: 2  
)
```

## Import

In any of the source files where you intend to use with this module, add the following import:

```
import PerfectFileMaker
```

## Overview

The main objects that you will be working with when accessing FileMaker databases are described here.

### struct FileMakerServer

This is the main interface to a FileMaker Server. Before accessing any database you will need to create an instance of this struct and provide a server host name or IP address, a port, and a valid username and password for the server. If the given port is 443 then all requests will be encrypted HTTPS.

Once the server connection is instantiated you can perform four different operations. These are:

- list available databases
- list a database's layouts
- list a layout's fields
- perform a query

In all cases you provide a callback and the operation occurs asynchronously. When the operation has completed the callback is called and given a closure which, when executed, will either return the operation response object or throw an error. The type of the operation response object will differ depending on the type of operation. The exception error object provides access to the error code and accompanying message string. It will be of the following type:

```
public enum FMPErrors: Error {  
    /// An error code and message.  
    case serverError(Int, String)  
}
```

The relevant portions of the FileMakerServer struct are defined as follows:

```
/// A connection to a FileMaker Server instance.  
/// Initialize using a host, port, username and password.  
public struct FileMakerServer {  
    /// Initialize using a host, port, username and password.  
    public init(host: String, port: Int, userName: String, password: String)  
    /// Retrieve the list of database hosted by the server.  
    public func databaseNames(completion: @escaping () throws -> [String]) -> ()  
    /// Retrieve the list of layouts for a particular database.  
    public func layoutNames(database: String, completion: @escaping () throws -> [String]) -> ()  
    /// Get a database's layout information. Includes all field and portal names.  
    public func layoutInfo(database: String,  
                           layout: String,  
                           completion: @escaping () throws -> FMPLayoutInfo) -> ()  
    /// Perform a query and provide any resulting data.  
    public func query(_ query: FMPLQuery, completion: @escaping () throws -> FMPLResultSet) -> ()  
}
```

When listing database or layout names, the response object will simply be an array of strings. When retrieving layout information, the response object will be a FMPLLayoutInfo object. When performing a query the response object will be a FMPLResultSet.

### struct FMPLLayoutInfo

FMPLLayoutInfo is defined as follows:

```

/// Represents meta-information about a particular layout.
public struct FMPLayoutInfo {
    /// Each field or related set as a list.
    public let fields: [FMPMetaDataItem]
    /// Each field or related set keyed by name.
    public let fieldsByName: [String:FMPFieldType]
}

```

This object contains all of the information pertaining to the fields on a layout. This includes the field names and types and also indicates which portals/relations are on the layout and which fields are contained within them.

Field information is provided in two different ways. The first is represented by an array of FMPMetaDataItems. This enum value indicates if the item is a regular field or a portal/relation. The second is a dictionary containing the full field name and the field type. If the field is in a relation it will be named with the standard FileMaker PortalName::FieldName syntax.

What follows are the definitions for FMPMetaDataItem, FMPFieldDefinition & FMPFieldType.

```

/// Represents either an individual field definition or a related (portal) definition.
public enum FMPMetaDataItem {
    /// An individual field.
    case fieldDefinition(FMPFieldDefinition)
    /// A related set. Indicates the portal name and its contained fields.
    case relatedSetDefinition(String, [FMPFieldDefinition])
}

```

```

/// A FileMaker field definition. Indicates a field name and type.
public struct FMPFieldDefinition {
    /// The field name.
    public let name: String
    /// The field type.
    public let type: FMPFieldType
}

```

```

/// One of the possible FileMaker field types.
public enum FMPFieldType {
    /// A text field.
    case text
    /// A numeric field.
    case number
    /// A container field.
    case container
    /// A date field.
    case date
    /// A time field.
    case time
    /// A timestamp field.
    case timestamp
}

```

## struct FMResultSet

An FMResultSet object contains the result of a FileMaker query. This includes database & layout meta information, as well as the number of records which were found in total and a each record in the found set.

```

/// The result set produced by a query.
public struct FMResultSet {
    /// Database meta-info.
    public let databaseInfo: FMPDatabaseInfo
    /// Layout meta-info.
    public let layoutInfo: FMPLayoutInfo
    /// The number of records found by the query.
    public let foundCount: Int
    /// The list of records produced by the query.
    public let records: [FMPRecord]
}

```

In addition to the previously described FMPLayoutInfo struct, a result set also contains the following bits of database related information:

```

/// Meta-information for a database.
public struct FMPDatabaseInfo {
    /// The date format indicated by the server.
    public let dateFormat: String
    /// The time format indicated by the server.
    public let timeFormat: String
    /// The timestamp format indicated by the server.
    public let timeStampFormat: String
    /// The total number of records in the database.
    public let recordCount: Int
}

```

The found record data is accessed through the `FMPResultSet.records` property. This array of `FMPRecords` will contain one element for each returned record.

```

/// An individual result set record.
public struct FMPRecord {
    /// A type of record item.
    public enum RecordItem {
        /// An individual field.
        case field(String, FMPFieldValue)
        /// A related set containing a list of related records.
        case relatedSet(String, [FMPRecord])
    }
    /// The record id.
    public let recordId: Int
    /// The contained record items keyed by name.
    public let elements: [String:RecordItem]
}

```

A record holds each field or portal item keyed by name in its `elements` property. The values in this dictionary are `FMPRecord.RecordItem` enum objects. These objects indicate if the item is a regular field, in which case its name and value can be directly accessed, or if the item is a related record set. If the item is a related record set then the portal name and an array of nested records are provided.

In either case a field's value is represented through the `FMPFieldValue` struct.

```

/// A returned FileMaker field value.
public enum FMPFieldValue: CustomStringConvertible {
    /// A text field.
    case text(String)
    /// A numeric field.
    case number(Double)
    /// A container field.
    case container(String)
    /// A date field.
    case date(String)
    /// A time field.
    case time(String)
    /// A timestamp field.
    case timestamp(String)
}

```

## Queries

The `FileMakerServer.query` function lets you search for sets of records or manipulate individual records. You specify a query using a `FMPQuery` struct. This struct is then formatted as a FileMaker XML query string and sent to the FileMaker Server. The server returns its response as XML. This XML is parsed and converted to a `FMPResultSet` object.

The query strings generated by the `FMPQuery` object correspond to what's defined in the "FileMaker® Server 12 Custom Web Publishing with XML" document. [PDF Doc.](#)

A query is created by instantiating a `FMPQuery` struct and then adding options to it. Each option add will return a new modified object. In this manner options can be "chained" to construct the desired final query object. The query object is then given to the `FileMakerServer.query` function and the results are generated.

`FMPQuery` is as follows:

```

/// An individual query & database action.
public struct FMPQuery: CustomStringConvertible {
    /// Initialize with a database name, layout name & database action.
    public init(database: String, layout: String, action: FMPAction)
    /// Sets the record id and returns the adjusted query.
    public func recordId(_ recordId: Int) -> FMPQuery
    /// Adds the query fields and returns the adjusted query.
    public func queryFields(_ queryFields: [FMPQueryFieldGroup]) -> FMPQuery
    /// Adds the query fields and returns the adjusted query.
    public func queryFields(_ queryFields: [FMPQueryField]) -> FMPQuery
    /// Adds the sort fields and returns the adjusted query.
    public func sortFields(_ sortFields: [FMPSortField]) -> FMPQuery
    /// Adds the indicated pre-sort scripts and returns the adjusted query.
    public func preSortScripts(_ preSortScripts: [String]) -> FMPQuery
    /// Adds the indicated pre-find scripts and returns the adjusted query.
    public func preFindScripts(_ preFindScripts: [String]) -> FMPQuery
    /// Adds the indicated post-find scripts and returns the adjusted query.
    public func postFindScripts(_ postFindScripts: [String]) -> FMPQuery
    /// Sets the response layout and returns the adjusted query.
    public func responseLayout(_ responseLayout: String) -> FMPQuery
    /// Adds response fields and returns the adjusted query.
    public func responseFields(_ responseFields: [String]) -> FMPQuery
    /// Sets the maximum records to fetch and returns the adjusted query.
    public func maxRecords(_ maxRecords: Int) -> FMPQuery
    /// Sets the number of records to skip in the found set and returns the adjusted query.
    public func skipRecords(_ skipRecords: Int) -> FMPQuery
    /// Returns the formulated query string.
    /// Useful for debugging purposes.
    public var queryString: String
}

```

A FMPQuery is first instantiated with a database name, layout name and action. The possible actions are:

```

/// A database action.
public enum FMPAction: CustomStringConvertible {
    /// Perform a search given the current query.
    case find
    /// Find all records in the database.
    case findAll
    /// Find and retrieve a random record.
    case findAny
    /// Create a new record given the current query data.
    case new
    /// Edit (update) the record indicated by the record id with the current query fields/values.
    case edit
    /// Delete the record indicated by the current record id.
    case delete
    /// Duplicate the record indicated by the current record id.
    case duplicate
}

```

Many of the FMPQuery functions accepts Strings or Ints and are self-explanatory. Any function which accepts an array can be called multiple times and the new values will be appended to the existing values.

When performing an `.edit`, `.delete` or `.duplicate` action a record id must be set by calling `FMPQuery.recordId(_ recordId: Int)`. Setting a record id when performing a `.find` will retrieve only the indicated record.

It is possible to search on one layout but return fields from another. Call `FMPQuery.responseLayout(_ responseLayout: String)` to set the response layout. By default the layout which is searched on will be the response layout.

It is possible and advisable, for performance reasons, to return only the minimum of required fields in a result. The names of the fields you want returned in the result can be set through the `FMPQuery.responseFields(_ responseFields: [String])` function.

Sorting and the usage of query fields are detailed below.

## Sorting

Records in a result set can be returned sorted by indicating one or more sort fields and sort orders. This is accomplished by calling the `FMPQuery.sortFields` function and passing the desired fields and orders as an array of FMPSortField objects. FMPSortField uses the FMPSortOrder enum to indicate the sort order. Both are defined as follows:

```

/// A record sort order.
public enum FMPSortOrder: CustomStringConvertible {
    /// Sort the records by the indicated field in ascending order.
    case ascending
    /// Sort the records by the indicated field in descending order.
    case descending
    /// Sort the records by the indicated field in a custom order.
    case custom
}

/// A sort field indicator.
public struct FMPSortField {
    /// The name of the field on which to sort.
    public let name: String
    /// A field sort order.
    public let order: FMPSortOrder
    /// Initialize with a field name and sort order.
    public init(name: String, order: FMPSortOrder)
    /// Initialize with a field name using the default FMPSortOrder.ascending sort order.
    public init(name: String)
}

```

## Query Fields

Query fields are added to a FMPQuery to indicate either fields which should be modified, in the case of an `.edit` action or fields which should be searched on, in the case of a `.find` action. Query fields hold a field name along with a corresponding value. In the case of the `.find` action, query fields also hold a field level operator. These operators represent a relation of a field to the indicated value. For example, a field operator could indicate that a field's contents should be greater-than-or-equal to the value when performing a search. The default field level operator is begins-with (as is standard for FileMaker database searches).

Individual query fields are represented by FMPQueryField objects. Field level operators are represented by FMPPFieldOp. These are defined as follows:

```

/// An individual field search operator.
public enum FMPPFieldOp {
    case equal
    case contains
    case beginsWith
    case endsWith
    case greaterThan
    case greaterThanEqual
    case lessThan
    case lessThanEqual
}

/// An individual query field.
public struct FMPQueryField {
    /// The name of the field.
    public let name: String
    /// The value for the field.
    public let value: Any
    /// The search operator.
    public let op: FMPPFieldOp
    /// Initialize with a name, value and operator.
    public init(name: String, value: Any, op: FMPPFieldOp = .beginsWith)
}

```

When performing an `.edit` action, FMPQueryFields can be added to a FMPQuery as an array through the `FMPQuery.queryFields(_ queryFields: [FMPQueryField])` function. Any field level operators are ignored in an `.edit`.

When performing a `.find` query fields are grouped by logical operator. Logical operators indicate how the fields should be treated together in a query. The possible logical operators are: and, or, not. Their meanings, when performing a search and selecting records for the result set, are:

- and: All query fields in the group must match for the record to be selected.
- or: Any field in the group can match and the record will be selected.
- not: If all query fields in the group match then the record will be omitted from the result set.

Multiple query field groups can be added to a FMPQuery. Each group will be considered in-order when FileMaker performs the search.

Logical operators and query field groups are represented by FMPLLogicalOp & FMPQueryFieldGroup, respectively.

```

/// A logical operator used with query field groups.
public enum FMPLogicalOp {
    case and, or, not
}

/// A group of query fields.
public struct FMPQueryFieldGroup {
    /// The logical operator for the field group.
    public let op: FMPLogicalOp
    /// The list of fields in the group.
    public let fields: [FMPQueryField]
    /// Initialize with an operator and field list.
    /// The default logical operator is FMPLogicalOp.and.
    public init(fields: [FMPQueryField], op: FMPLogicalOp = .and)
}

```

Query field groups are added through the `FMPQuery.queryFields(_ queryFields: [FMPQueryFieldGroup])` function.

## Examples

The following code snippets illustrate the basic activities that one would perform against FileMaker databases.

### List Available Databases

This snippet connects to the server and has it list all of the hosted databases.

```

let fms = FileMakerServer(host: testHost, port: testPort, userName: testUserName, password: testPassword)
fms.databaseNames {
    result in
    do {
        // Get the list of names
        let names = try result()
        for name in names {
            print("Got a database name \(name)")
        }
    } catch FMPErrors.serverError(let code, let msg) {
        print("Got a server error \(code) \(msg)")
    } catch let e {
        print("Got an unexpected error \(e)")
    }
}

```

### List Available Layouts

List all of the layouts in a particular database.

```

let fms = FileMakerServer(host: testHost, port: testPort, userName: testUserName, password: testPassword)
fms.layoutNames(database: "FMServer_Sample") {
    result in
    guard let names = try? result() else {
        return // got an error
    }
    for name in names {
        print("Got a layout name \(name)")
    }
}

```

### List Field On Layout

List all of the field names on a particular layout.

```

let fms = FileMakerServer(host: testHost, port: testPort, userName: testUserName, password: testPassword)
fms.layoutInfo(database: "FMServer_Sample", layout: "Task Details") {
    result in
    guard let layoutInfo = try? result() else {
        return // error
    }
    let fieldsByName = layoutInfo.fieldsByName
    for (name, value) in fieldsByName {
        print("Field \(name) = \(value)")
    }
}

```

## Find All Records

Perform a findAll and print all field names and values.

```

let query = FMPQuery(database: "FMServer_Sample", layout: "Task Details", action: .findAll)
let fms = FileMakerServer(host: testHost, port: testPort, userName: testUserName, password: testPassword)
fms.query(query) {
    result in
    guard let resultSet = try? result() else {
        return // error
    }
    let fields = resultSet.layoutInfo.fields
    let records = resultSet.records
    let recordCount = records.count
    for i in 0..

```

## Find All Records With Skip & Max

To add skip and max, the query above would be amended as follows:

```

// Skip two records and return a max of two records.
let query = FMPQuery(database: "FMServer_Sample", layout: "Task Details", action: .findAll)
    .skipRecords(2).maxRecords(2)
...

```

## Find Records Where "Status" Is "In Progress"

Find all records where the field "Status" has the value of "In Progress".

```

let qfields = [FMPQueryFieldGroup(fields: [FMPQueryField(name: "Status", value: "In Progress")])]
let query = FMPQuery(database: "FMServer_Sample", layout: "Task Details", action: .find)
    .queryFields(qfields)
let fms = FileMakerServer(host: testHost, port: testPort, userName: testUserName, password: testPassword)
fms.query(query) {
    result in
    guard let resultSet = try? result() else {
        return // error
    }
    let fields = resultSet.layoutInfo.fields
    let records = resultSet.records
    let recordCount = records.count
    for i in 0..

```

## iOS Notifications

...

## Deployment Options

Deployment options for Perfect.

## Producing an Ubuntu 15.10 Base Image for Swift 3 and Perfect 2

This guide will share steps that you can use to produce an Ubuntu 15.10 environment, which is suitable for use with the Swift 3 programming language, and with PerfectSoft's Perfect 2 application framework.

### Running as the Root User

Begin by installing, deploying, or otherwise utilizing an Ubuntu 15.10 system. For ease of use, you may use the following command to perform the steps as the **root** user. You'll need to be using an administrative account, and you'll need its password. If you choose to skip this step, you'll need to prefix many of the steps in this guide with the **sudo** command.

```
sudo su
```

### Running a Screen Session

If you are connected to the Ubuntu system via **SSH**, you might wish to establish a **screen** session for these steps in case you're disconnected. This section is not mandatory. You can skip to the next section of steps. To establish a **screen** session called "perfect," use:

```
screen -S perfect
```

If **screen** isn't installed, you'll need to install it:

```
apt-get install screen
```

To intentionally disconnect from a **screen** session, you can use **Control-A, D**. To reconnect to the intentionally disconnected **screen** session:

```
screen -r perfect
```

If you were unintentionally disconnected from your **screen** session, you might need to use:

```
screen -D -R perfect
```

## Install Some Dependencies via APT

You can install the majority of dependencies for Swift 3 and for Perfect 2 via the APT system:

```
apt-get install make git clang libicu-dev libmysqlclient-dev libpq-dev sqlite3 libsqlite3-dev apache2-dev pkg-config libssl-dev libsasl2-dev libcurl4-openssl-dev uuid-dev wget
```

## Install MongoDB from Source Code

Download the MongoDB source code:

```
cd /usr/src/  
wget https://github.com/mongodb/mongo-c-driver/releases/download/1.3.5/mongo-c-driver-1.3.5.tar.gz
```

Decompress the MongoDB source code:

```
gunzip mongo-c-driver-1.3.5.tar.gz
```

Extract the MongoDB source code:

```
tar -xvf mongo-c-driver-1.3.5.tar
```

Remove the MongoDB source code archive:

```
rm mongo-c-driver-1.3.5.tar
```

Configure the MongoDB source code for compilation:

```
cd mongo-c-driver-1.3.5/  
./configure --enable-sasl=yes
```

Compile MongoDB:

```
make
```

Install MongoDB:

```
make install
```

## Finished

Congratulations! You now have an environment suitable for Swift 3 and Perfect 2.

## Trying It Out

If you wish to install a "development snapshot" of Swift before Swift 3 has been released, you can download one:

```
cd /usr/src/  
wget https://swift.org/builds/swift-3.0-preview-3/ubuntu1510/swift-3.0-PREVIEW-3/swift-3.0-PREVIEW-3-ubuntu15.10.tar.gz
```

To install it:

```
gunzip < swift-3.0-PREVIEW-3-ubuntu15.10.tar.gz | tar -C / -xv --strip-components 1
```

And to remove the compressed archive:

```
rm swift-3.0-PREVIEW-3-ubuntu15.10.tar.gz
```

Now you can copy the Perfect 2 template project:

```
git clone https://github.com/PerfectlySoft/PerfectTemplate
```

And build it:

```
cd PerfectTemplate  
git checkout d13e8dd8eb4868fea36468758604fe05a48b9aa2  
swift build
```

Once built, this application will be available inside the `.build/debug/` directory. You can copy it to somewhere more convenient, such as the `/root/` directory:

```
cp .build/debug/PerfectTemplate /root/
```

Now you can run this application from the `/root/` directory. Be careful: If you are still using the **root** user, the application will have the **root** user's privileges!

```
/root/PerfectTemplate --port 80
```

You can test this application from a different shell with the `curl` command:

```
curl http://127.0.0.1
```

If you find an issue, [please report it](#).

## Digital Ocean Deployment

This guide will walk you through setting up a Digital Ocean Droplet that will run your compiled Swift application. You can actually do all of these steps on any other Linux server running Ubuntu 15 or 16. Let's start...

First create your droplet, select Ubuntu 16 (if 15 is not available), and let it finish. Once the droplet is created you can ssh into it as root. `ssh root@YOUR_DROPLET_IP -v` This should use the private key you have created earlier and it should not ask you for a password or anything.

### Setting up the server - Part 1

The next step is creating a user to run all commands through and also run the final app since it's not that nice to do all as root. Since we're still connected as root we're going to skip the `sudo` commands since there's no point in using it. Please adjust the `USERNAME` and `USER_PASSWORD` variables in the command below.

```
useradd -d /home/USERNAME -m -s /bin/bash USERNAME echo "USERNAME ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers
```

The last command will add your new `USERNAME` to the list of accounts allowed to run commands using `sudo`. The spectrum used here, `ALL`, is a bit unrestrictive and it is advised to limit the commands to just a few.

Once the user is created, you have two options: 1. Generate a password for this user and use the password when you want to connect to the server, or 2. Generate a SSH key and use that key to connect to the server - we're going to take a look at both options below (personally I prefer the SSH key over the password).

### Option 1 - Generate a password

Since we're still connected as root we're going to set the password for our user using `passwd USERNAME`

Once that is done we can drop the connection as root (Ctrl-D) and reconnect as USERNAME.

### Option 2 - Generate a SSH key

This option is a bit more complicated to set up, but in the long run it will be easier to connect since we don't have to remember the password we've set or risk that the password isn't secure enough, etc...

There are again, two options here: 1. Use the same key as root, or 2. Generate a new key for USERNAME.

### Sub-Option 1

To use the same key you've set up when the droplet was created you can just run

```
mkdir /home/USERNAME/.ssh cp /root/.ssh/authorized_keys /home/USERNAME/.ssh chown -R USERNAME.USERNAME /home/USERNAME/.ssh
```

### Sub-Option 2

First we need to generate a new SSH key on our local environment - if you are running Windows as your OS, please google how you can do it and continue after this step. Remember, this step is to be done on your local environment and not on the droplet... `ssh-keygen -t rsa` The key generator will ask you where do you want to save the new key and what will be the key name. For this example, I have saved my key to `/Users/rb/.ssh/id_rsa_perfectapp` - We'll call this `LOCAL_KEY_PATH` from now on.

Once it's finished generating the key, you need to extract the contents of the public key and prepare to use it on the droplet by using

```
cat /Users/rb/.ssh/id_rsa_perfectapp.pub
```

 which will result in something like this

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQ3Mzd0naYx59Xfi0buSEphNTupfNqHGFSRExAl+WS0yiA8j8Kja964qTg0Bs2DUD9OoazaAIUTrt+GoQFUSlP37a4XqUxu4Pm8LyDy
```

Next, moving back to the droplet server, we need to set up the public key we've just created and set it up so the server can identify us with it.

```
su - USERNAME mkdir .ssh chmod 700 .ssh echo "THE_PUBLIC_KEY_STRING" >> .ssh/authorized_keys chmod 600 .ssh/authorized_keys exit
```

 Long story short, we've switched to the new user's home folder, we've created the folder `.ssh` and inside of that folder we've created a file called `authorized_keys` in which we've pasted the content of the public key created earlier. The last two lines are going back up a level in the folder structure and change the permissions of the `.ssh` folder and its contents to the `USERNAME` user.

To test that everything works, and the key was added successfully, on your local environment run the following `ssh -i LOCAL_KEY_PATH USERNAME@YOUR_DROPLET_IP -v` You should be logged into your droplet now, under the USERNAME account.

## Setting up the server - Part 2

Now that we are not longer connected to the droplet as `root` we can continue setting it up and preparing it for running/compiling Swift code and the arrival of you app.

### Updating packages installed by DigitalOcean

```
export LC_ALL="en_US.UTF-8"
export LC_CTYPE="en_US.UTF-8"
sudo dpkg-reconfigure locales
sudo apt-get update
sudo apt-get upgrade -y
```

This will configure the locales installed on the server, update the packages list to the latest available and upgrade (-y without asking you anything) the installed packages - it's always a good practice to have the latest stable versions up and running.

Next, lets installed some of the packages needed by Swift to compile, and a few other packages needed in the process. You can do this by running

```
sudo apt-get install make clang libicu-dev pkg-config libssl-dev libsasl2-dev libcurl4-openssl-dev uuid-dev git curl wget unzip -y
```

Next, we will install Swift so the whole system has access to the binaries. At the moment, the latest available version of Swift binaries/libraries is `Swift 3.0 GM Candidate` - If you are not sure what's the latest version, head on to <https://swift.org/download/> and check there. Because we're trying to make Perfect2.0 compile on this server we need to use Swift3.

```
cd /usr/src
sudo wget https://swift.org/builds/swift-3.0-GM-CANDIDATE/ubuntu1510/swift-3.0-GM-CANDIDATE/swift-3.0-GM-CANDIDATE-ubuntu15.10.tar.gz
sudo gunzip < swift-3.0-GM-CANDIDATE-ubuntu15.10.tar.gz | sudo tar -C / -xv --strip-components 1
sudo rm -f swift-3.0-GM-CANDIDATE-ubuntu15.10.tar.gz
```

You can test if Swift was copied/installed correctly by running `swift --version` in any location on the server. You should get back something like

```
Swift version 3.0 (swift-3.0-GM-CANDIDATE) Target: x86_64-unknown-linux-gnu
```

At this point we have another fork in the road - What database system (if needed) will we use?

## MySQL

If you are going to use a database provider as a service (RDS from Amazon, etc) you can run `sudo apt-get install libmysqlclient-dev -y` but if you also want to host the database on the same server, you can run `sudo apt-get install mysql-client libmysqlclient-dev -y`

Next, you need to edit the file `mysqlclient.pc` located in the folder `/usr/lib/x86_64-linux-gnu/pkgconfig` . If you can't find the file there, please run a

```
find / -name mysqlclient.pc
```

 . Once you find the file and edit it, you will need to remove any occurrence of "-fno-omit-frame-pointer" and "-fabi-version=2".

```
sudo nano /usr/lib/x86_64-linux-gnu/pkgconfig/mysqlclient.pc
```

For example, my file looks like this: `` prefix=/usr includedir=\${prefix}/include/mysql libdir=\${prefix}/lib/x86\_64-linux-gnu

Name: mysqlclient Description: MySQL client library Version: 20.3.0 Cflags: -I\${includedir} -fabi-version=2 -fno-omit-frame-pointer Libs: -L\${libdir} -lmysqlclient Libs.private: -lpthread -lz -lm -lrt -ldl ``

and after the edit, the file looks like this: `` prefix=/usr includedir=\${prefix}/include/mysql libdir=\${prefix}/lib/x86\_64-linux-gnu

Name: mysqlclient Description: MySQL client library Version: 20.3.0 Cflags: -I\${includedir} Libs: -L\${libdir} -lmysqlclient Libs.private: -lpthread -lz -lm -lrt -ldl ``

If you chose to host your database on the same server as the Swift app, you will have to do a few more steps in order to configure the MySQL server.

```
sudo mysql_secure_installation
```

This will prompt you for the root password you created in step one. You can press ENTER to accept the defaults for all the subsequent questions, with the exception of the one that asks if you'd like to change the root password.

Next, we'll initialize the MySQL data directory, which is where MySQL stores its data. How you do this depends on which version of MySQL you're running. You can check your version of MySQL with the following command.

```
mysql --version
```

You'll see some output like this: `mysql Ver 14.14 Distrib 5.7.11, for Linux (x86_64) using EditLine wrapper` If you're using a version of MySQL earlier than 5.7.6, you should initialize the data directory by running `mysql_install_db . sudo mysql_install_db`

Please note that in MySQL 5.6, you might get an error that says FATAL ERROR: Could not find my-default.cnf. If you do, copy the `/usr/share/my.cnf` configuration file into the location that `mysql_install_db` expects, then rerun it. `sudo cp /etc/mysql/my.cnf /usr/share/mysql/my-default.cnf sudo mysql_install_db` This is due to some changes made in MySQL 5.6 and a minor error in the APT package.

The `mysql_install_db` command is deprecated as of MySQL 5.7.6. If you're using version 5.7.6 or later, you should use `mysqld --initialize` instead.

However, if you installed version 5.7 from the Debian distribution, the data directory was initialized automatically, so you don't have to do anything. If you try running the command anyway, you'll see the following error:

```
2016-03-07T20:11:15.998193Z 0 [ERROR] --initialize specified but the data directory has files in it. Aborting.
```

## Sequel3

Coming soon...

## PostgreSQL

Coming soon...

## MongoDB

Coming soon...

## Setting up the app folders, automate compiling and all that good stuff

Now that we're finished with the server setup, and all prerequisites are installed, it is the time to move forward with the app setup, its folder structure and compile automation.

The main plan is to have the app delivered to the server using `git`. If you noticed, the earlier package installation, did add `git` as an app to the server. Okay, so, `git` will deliver the raw Swift files to the server, we will need to compile the app, move it to another folder (for a clear structure) and run it from there.

To accomplish this we will use the `post-receive` git hook, which will run all the needed commands to compile. Move and restart the app. Yes, restart it, because we will also make use of the `supervisor` app to keep our Swift compiled application up and running 24/7.

First, let's setup the folder structure for our app.

**Note:** At this point you should be logged into your droplet server as your USERNAME, and be located in the home folder of this user - If you are logged as USERNAME, just `cd ~` (or `cd` for short) and hit Enter.

```
mkdir -p {running,app/.git} && cd app/.git
git init . --bare
cd hooks && rm -rf *.sample
```

This will create two folders in the USERNAME's home folder: `running` which will host your final compiled app, and `app` which will host your raw Swift files, compile by-products, and your Git repo. On the same line we're also changing location to the `app/.git` folder which will host the app's repo. `git init . --bare` will initialize a bare git repo in this folder and create a very basic repo structure. Next, we're again changing location in the hooks folder and removing all the sample files there.

Next we will need to create the `post-receive` hook that will do all the heavy lifting for us. `nano post-receive`

You can copy/paste the following block as your file contents, but be sure you change the following variables with their correct values: - `USERNAME` your current username - `THE_APP_NAME` the actual name of the compiled app - this is the name of your app as well written in the Package.swift file - `.build/debug/*` - if your app was compiled as release, please change it to `.build/release/*` ```

## #!/bin/bash

```
APPNAME="THEAPPNAME" USERNAME="USERNAME" HOME="/home/$USERNAME" APPFOLDER="$HOME/app" RUNNINGFOLDER="$HOME/running"
GITFOLDER="$APPFOLDER/.git" echo "" echo "-----" echo "Running with the following config:" echo "Owner: $USERNAME" echo "Home folder: $HOME" echo
"Application name: $APPNAME" echo "Application folder: $APPFOLDER" echo "Git folder: $GITFOLDER" echo "Running folder: $RUNNING_FOLDER" echo "-----"
-----" echo "" echo "Cleaning up previous commit" sudo rm -f $HOME/.build sudo rm -f $HOME/Packages echo ""
```

## Move files from commit to the main folder

```
echo "Extracting files from current commit" git --work-tree=$APPFOLDER --git-dir=$GITFOLDER checkout -f
```

## Switch to app folder

```
echo "Switching to $APPFOLDER" cd $APPFOLDER
```

## Compile the app and move it to the correct folder

```
echo "Starting to build Swift app" sudo swift build sudo chown -R $USERNAME.$USERNAME .build/ Packages echo ""
```

```
if [ -f $APPFOLDER/.build/debug/$APPNAME ]; then echo "Copying $APPFOLDER/.build/debug/* app to $RUNNINGFOLDER/" cp -f $APPFOLDER/.build/debug/* $RUNNINGFOLDER/
fi if [ -d "$DIRECTORY" ]; then echo "Copying $APPFOLDER/webroot folder to $RUNNINGFOLDER/webroot" cp -r $APPFOLDER/webroot $RUNNINGFOLDER/webroot else mkdir
$RUNNING_FOLDER/webroot fi
```

```
echo "" echo "Setting permissions" cd $HOME sudo chown -R $USERNAME.$USERNAME running
```

## Find and restart the previous process

```
echo "" echo "Restarting $APPNAME" sudo supervisorctl restart $APPNAME ```
```

Once the file was saved you will need to make it executable by executing `chmod +x post-receive`

Long story short, the `post-receive` file will do the following: - Extract the last commit files and move them to the `app` folder located in your USERNAME's home folder - Switch working folders to the `app` folder - Start building your app - Copy the compiled app to the `running` folder - Copy the `webroot` folder to the `running` folder as well - Try to find the PID of the compiled app (in case this is not the first commit) - If the app was already running, the script will kill it - at this point, the `supervisor` app (which we haven't installed yet) will kick in and restart it, but it will use the new compiled version - If the app was not running, like the first time we will deploy it, the script will start it and `supervisor` will keep an eye on it.

At this point the app compiling automated process should be all set up and ready to receive files. Next we'll setup the `supervisor` process that will watch over our app and keep it running.

### Setting up supervisor

Before installing it, make sure your system isn't running it already. You can run `supervisord --version` to make sure if `supervisor` is installed or not. If it's not, continue with `sudo apt-get install supervisor -y service supervisor restart` and that should be all, as far as installation is required.

Next, we'll need to set the init script for your app. All the init scripts are located in the `/etc/supervisor/conf.d` folder.

```
cd /etc/supervisor/conf.d
sudo nano THE_APP_NAME.conf
```

In order for the git hook to do its job, you will need to name the init file with the same name you have used for the `$APP_NAME` variable in the hook file. Use the below code block to fill in this file. Be sure to replace `USERNAME` and `APP_NAME` with the correct values.

```
[program:APP_NAME] command=/home/USERNAME/running/APP_NAME autostart=true autorestart=true stderr_logfile=/var/log/APP_NAME.err.log stdout_logfile=/var/log/APP_NAME.out.log
```

Save the file and quit the editor.

Once our configuration file is created and saved, we can inform Supervisor of our new program through the `supervisorctl` command. First we tell Supervisor to look for any new or

changed program configurations in the `/etc/supervisor/conf.d` directory with: `supervisorctl reread`

Right now, all that is left to do is create our app on our local environment and add the server as a git remote so we deploy our code. The second you will commit your files, the server will take care of everything else, leaving you to continue coding and not worrying about deployment and other configurations.

## Setting up your local environment

For the sake of this example, I'm going to use the Perfect Template repo, made available to us by the people working on the Perfect Framework (Thank you all @Perfect for the great work!!!)

```
cd ~
git clone https://github.com/PerfectlySoft/PerfectTemplate.git
cd PerfectTemplate
swift build
```

First we'll clone the PerfectTemplate repo, switch into the folder and build the project (just to make sure everything works). If the build process was a success you can run the app locally to test further (first time building an app will take you a bit longer because of all packages being downloaded and compiled). `./build/debug/PerfectTemplate`

You should see something like `[INFO] Starting HTTP server on 0.0.0.0:8181 with document root ./webroot` which means the app is running on your localhost on port 8181. You can open another terminal window and run `curl localhost:8181` and see the response, which in the case of the PerfectTemplate will look something like `<html><title>Hello, world!</title><body>Hello, world!</body></html>`

Ok, back to the terminal window running the app. Press `Ctrl+C` to stop the execution - we need to add our server as a remote to git and deploy on our server.

In the folder hosting your Swift app (PerfectTemplate if you're following my example) you need to run the following (please adjust the variables):

```
git remote add REMOTE_NAME ssh://USERNAME@YOUR_DROPLET_IP/home/USERNAME/app/.git
```

In my case, the line looks like this: `git remote add live ssh://perfectapp@146.185.134.227/home/perfectapp/app/.git` - `REMOTE_NAME` is the name you would like to give this remote address - `USERNAME` is the username created on the server - `YOUR_DROPLET_IP` is the IP address of your droplet (or domain name if you've already added a DNS entry for it)

Once the new remote is added, all you need to do is push your repo to this new remote. `git push live master` A.K.A. `git push REMOTE_NAME BRANCH`

**!!! Panic ensues !!!** But, but ... it's asking me for a password - WTF!?!?!

It has time to setup SSH on your local environment to use the key you have created earlier (if you created a key that is...)

If you have chosen the easy way out, just type/paste your chosen password and press Enter. If not, let's quickly tell your local SSH how to handle the key.

We will need to edit a file located in your local `.ssh` folder, the file's name is `config`. You might have it or not, so let's make sure it's there...

```
echo >> ~/.ssh/config
nano ~/.ssh/config
```

Add the following to the end of the file (those of you that set up the key on your local environment should know the variables below):

```
Host YOUR_DROPLET_IP IdentityFile LOCAL_KEY_PATH
```

Once the file is saved try to push to your server's git repo again... Watch the magic happen!

## Getting your app to meet the internet

Well, at this step, we have everything up and running, everything compiles successfully (at least it should), but still, the app is bound to port 8181 and on localhost. You would need root account privileges in order to bind the app on a lower port value than 1024, but we will correct that by using `nginx` and `proxy passing`.

First, lets install nginx: `sudo apt-get install nginx -y`

Once installed, lets configure our virtual host `cd /etc/nginx/sites-available sudo rm -rf default ../sites-enabled/default sudo nano app.vhost`

You can use the below code block to setup your virtual host. We'll use the following variables in the code block: - `SERVER_NAME` The name of the domain your app will have `` server { listen 80; servername *SERVERNAME*; root /home/USERNAME/running/webroot; accesslog /var/log/nginx/*SERVERNAME*.access.log; errorlog /var/log/nginx/*SERVERNAME*.error.log;

```
location ^~ / {
    proxy_set_header HOST $host;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    proxy_pass http://127.0.0.1:8181;
    proxy_redirect off;
}
```

} ``

Once the file is saved, you will need to tell nginx that this vhost should be enabled...

```
sudo ln -s /etc/nginx/sites-available/app.vhost /etc/nginx/sites-enabled/app.vhost
```

Next, lets change the config of nginx itself... `cd /etc/nginx sudo mv nginx.conf nginx.conf.backup sudo nano nginx.conf`

You can use the block below to paste in the new config file. If you want you can tweak some of the settings like `worker_processes` which should be equal to the number of processors of the server. `` user www-data; worker\_processes 1; pid /run/nginx.pid;

```
events { worker_connections 768; }
```

```
http { include mime.types; default_type application/octet-stream;
```

}'''

Save the file, and restart the nginx server (yes, it is running since the installation time). `sudo /etc/init.d/nginx restart`

Head on over to your local environment and try `curl http://YOUR_DROPLET_IP`

The output should be `<html><title>Hello, world!</title><body>Hello, world!</body></html>` provided you've used the same code as me.

## Success!

The app is now running in the background, nginx is providing a way for the app to be reached from the outside and you are done with all the complicated work - all you need to do now is change something in your app and commit the changes to the live remote.

# Perfect SysInfo

This project provides a Swift library to monitor system performance.

This package builds with Swift Package Manager and is part of the [Perfect](#) project but can also be used as an independent module.

Ensure you have installed and activated the latest Swift 3.1 tool chain.

## Quick Start

Add Perfect SysInfo library to your Package.swift:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-SysInfo.git", majorVersion: 1)
```

Add library header to your source code:

```
import PerfectSysInfo
```

Now SysInfo class is available to call.

## CPU Usage

Call static variable `SysInfo.CPU` will return a dictionary `[String: [String: Int]]` of all CPU usage, for example:

```
print(SysInfo.CPU)

//here is a typical return of single CPU (from Linux):

[
  "cpu0":
    [ "nice": 1201, "system": 3598, "user": 8432, "idle": 8657606],
  "cpu":
    [ "nice": 1201, "system": 3598, "user": 8432, "idle": 8657606]
]

// and the following is another example with 8 cores (from Mac):
[
  "cpu3":
    [ "user": 18095, "idle": 9708265, "nice": 0, "system": 16177],
  "cpu5":
    [ "user": 18032, "idle": 9708329, "nice": 0, "system": 16079],
  "cpu7":
    [ "user": 18186, "idle": 9707892, "nice": 0, "system": 16285],
  "cpu":
    [ "user": 344301, "idle": 9201762, "nice": 0, "system": 196763],
  "cpu0":
    [ "user": 730263, "idle": 8387000, "nice": 0, "system": 626684],
  "cpu2":
    [ "user": 648287, "idle": 8799969, "nice": 0, "system": 294749],
  "cpu1":
    [ "user": 17708, "idle": 9708996, "nice": 0, "system": 15950],
  "cpu4":
    [ "user": 647701, "idle": 8800643, "nice": 0, "system": 294544],
  "cpu6": [ "user": 656136, "idle": 8793002, "nice": 0, "system": 293640]
]
```

The record is a structure of `N+1` entries, where `N` is the number of CPU and `1` is the summary, so the each record will be tagged with "cpu0" ... "cpuN-1" and the tag "cpu" represents the average of overall. Each entries will contain `idle`, `user`, `system` and `nice` to represent the cpu usage time. In a common sense, `idle` shall be as large as possible to indicate the CPU is not busy.

## Memory Usage

Call static property `SysInfo.Memory` will return a dictionary `[String: Int]` with memory metrics in \*\* MB \*\*:

```
print(SysInfo.Memory)
```

**\*\* Note \*\*** Since system information is subject to operating system type, so please use directive `#if os(Linux) #else #endif` determine OS type before reading system metrics; The definition of each counter is out of the scope of this document, please see OS manual for detail.

Typical Linux memory looks like this ( 1G total memory with about 599MB available):

```
[
  "Inactive": 283, "MemTotal": 992, "CmaFree": 0,
  "VmallocTotal": 33554431, "CmaTotal": 0, "Mapped": 74,
  "SUnreclaim": 14, "Writeback": 0, "Active(anon)": 98,
  "Shmem": 26, "PageTables": 7, "VmallocUsed": 0,
  "MemFree": 98, "Inactive(file)": 179, "SwapCached": 0,
  "HugePages_Total": 0, "Inactive(anon)": 104, "HugePages_Rsvd": 0,
  "Buffers": 21, "SReclaimable": 39, "Cached": 613,
  "Mlocked": 3, "SwapTotal": 1021, "NFS_Unstable": 0,
  "CommitLimit": 1518, "Hugepagesize": 2, "SwapFree": 1016,
  "WritebackTmp": 0, "Committed_AS": 1410, "AnonHugePages": 130,
  "DirectMap2M": 966, "Unevictable": 3, "HugePages_Surp": 0,
  "Dirty": 3, "HugePages_Free": 0, "MemAvailable": 599,
  "Active(file)": 426, "Slab": 54, "Active": 525,
  "KernelStack": 2, "VmallocChunk": 0, "AnonPages": 177,
  "Bounce": 0, "HardwareCorrupted": 0, "DirectMap4k": 57
]
```

And here is a typical mac OS X memory summary, which indicates that there is about 4.5GB free memory:

```
[
  "hits": 0, "faults": 3154324, "cow": 31476,
  "wired": 3576, "reactivations": 366, "zero_filled": 2296248,
  "pageins": 13983, "lookups": 1021, "pageouts": 0,
  "active": 6967, "free": 4455, "inactive": 1008
]
```

## Network Traffic

Call static property `SysInfo.Net` will return total traffic summary from all interfaces as a dictionary `[String: [String: Int]]` where the key represents the network interface name, and the value is a detailed dictionary with two key-value pairs - `i` stands for receiving and `o` for transmitting, both in KB:

```
if let net = SysInfo.Net {
  print(net)
}
```

If success, it will print something like these mac / linux outputs:

```
// typical mac os x network summary, where the only physical network
// adapter "en0" has 1MB incoming data totally.

[
    "p2p0": ["o": 0, "i": 0],
    "stf0": ["o": 0, "i": 0],
    "vboxnet0": ["o": 0, "i": 1],
    "gif0": ["o": 0, "i": 0],
    "lo0": ["o": 0, "i": 887],
    "bridge0": ["o": 0, "i": 0],
    "utun0": ["o": 0, "i": 0],
    "awdl0": ["o": 0, "i": 318],
    "en1": ["o": 0, "i": 0],
    "en0": ["o": 0, "i": 1063],
    "en2": ["o": 0, "i": 0]
]

// typical linux network summary, where the only physical network
// adapter "enp0s3" has received 0.6MB data and sent out 506KB in the same time.

[
    "virbr0": ["o": 0, "i": 0],
    "enp0s8": ["o": 506, "i": 614],
    "virbr0-nic": ["o": 0, "i": 0],
    "lo": ["o": 1804, "i": 1804],
    "enp0s3": ["o": 158, "i": 7594]
]
```

## Disk IO

Call static method `SysInfo.Disk` may inspect disk i/o activity statistics in real time. It will return a `[String:[String: UInt64]]` dictionary with metrics as sample below. For more information of these counters, please refer to the operating system manual.

## Linux Release Notes

```
print(SysInfo.Disk)

// here is a sample output from Linux:
[
    "sda":
    [
        "io_ms": 7516, "reads_merged": 17, "reads_completed": 14993,
        "writing_ms": 9772, "io_in_progress": 0, "writes_completed": 4921,
        "sectors_read": 1292762, "reading_ms": 11952, "writes_merged": 5738,
        "sectors_written": 969480, "weighte_io_ms": 21636
    ],
    "sda2":
    [
        "io_ms": 0, "reads_merged": 0, "reads_completed": 4,
        "writing_ms": 0, "io_in_progress": 0, "writes_completed": 0,
        "sectors_read": 8, "reading_ms": 0, "writes_merged": 0,
        "sectors_written": 0, "weighte_io_ms": 0
    ],
    "sda1":
    [
        "io_ms": 7252, "reads_merged": 7, "reads_completed": 14817,
        "writing_ms": 9520, "io_in_progress": 0, "writes_completed": 3971,
        "sectors_read": 1281954, "reading_ms": 11908, "writes_merged": 5426,
        "sectors_written": 966696, "weighte_io_ms": 21340
    ]
]
```

## Mac OS X Release Notes

Please note that if using `SysInfo.Disk` in a loop, then an `autoreleasepool{ }` is strongly recommend to avoid unnecessary memory caching on such objects:

```

autoreleasepool(invoking: {
    let io = SysInfo.Disk
    print(io)
})

// here is the sample output from macOS:
[
    "disk0":
    [
        "operations_read": 501077, "latency_time_read": 0,
        "bytes_written": 21265645056, "bytes_read": 25022815232,
        "operations_written": 360598, "latency_time_written": 0
    ]
]

```

## Perfect New Relic Library for Linux

This project provides a Swift Agent SDK for New Relic.

This package builds with Swift Package Manager and is part of the [Perfect](#) project but can be used as an independent module.

## Release Note

This project is only compatible with Ubuntu 16.04 and Swift 3.1 Tool Chain.

## Quick Start

Please use [Perfect Assistant](#) to import this project, otherwise an install script is available for Ubuntu 16.04:

```

$ git clone https://github.com/PerfectlySoft/Perfect-NewRelic-linux.git
$ cd Perfect-libNewRelic-linux
$ sudo ./install.sh

```

During the installation, it will automatically ask for **license key**, **application name**, language and its version. Then it will install the command line `newrelic-collector-client-daemon` as a service which you can find the configuration on `/usr/local/etc/newrelic.service`.

Configure Package.swift:

```

.Package(url: "https://github.com/PerfectlySoft/Perfect-NewRelic-linux.git", majorVersion: 1)

```

Import library into your code ( **NOTE** Since Swift 3.1 on linux has a significant linker issue, so module `PerfectNewRelic` must be accompanied with `Foundation` ):

```

import PerfectNewRelic
import Foundation

```

Aside of Swift - C syntax conversion differences, document can be found on [New Relic Agent SDK](#)

## Configuration

Post Installation & Configuration can be found on [New Relic - Configuring the Agent SDK](#), please note that NewRelic instance is configured as **Daemon Mode** which is highly recommended because the *Embedded-mode* is still experimental.

If success, you can create NewRelic instance easily:

```

let nr = try NewRelic()

```

- Create a function to receive status change notifications:

```
nr.registerStatus { code in
    guard let status = NewRelic.Status(rawValue: code) else {
        // something wrong here
    } //end guard
    switch status {
        case .STARTING: // New Relic Daemon Service is starting
        case .STARTED: // New Relic Daemon Service is started
        case .STOPPING: // New Relic Daemon Service is stopping
        default: // shutdown already
    } //end case
} //end callback
```

## Limiting or disabling Agent SDK settings

According to [New Relic Limiting or disabling Agent SDK Settings](#), the following settings are available in Perfect NewRelic:

| If you want to ...                                                      | Use this setting ...                                                                                   |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Disable data collection during a transaction                            | <code>nr.enableInstrumentation(false)</code>                                                           |
| Configure the number of trace segments collected in a transaction trace | <code>let t = try Transaction(nr, maxTraceSegments: 50) // Only collect up to 50 trace segments</code> |

## API Quick Help

Base on [New Relic's Document of Using the Agent SDK](#), Perfect NewRelic library provides identically the same functions of New Relic Agent SDK in Swift:

### Instruments & Profiling

| Function    | <code>recordMetric()</code>                                      |
|-------------|------------------------------------------------------------------|
| Demo        | <code>try nr.recordMetric(name: "ActiveUsers", value: 25)</code> |
| Description | Record a custom metric.                                          |
| Parameters  | - name: name of the metric.<br>- value: value of the metric.     |

| Function    | <code>recordCPU()</code>                                                                                                                               |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| Demo        | <code>try nr.recordCPU(timeSeconds: 5.0, usagePercent: 1.2)</code>                                                                                     |
| Description | Record CPU user time in seconds and as a percentage of CPU capacity.                                                                                   |
| Parameters  | - timeSeconds: Double, number of seconds CPU spent processing user-level code<br>- usagePercent: Double, CPU user time as a percentage of CPU capacity |

| Function    | <code>recordMemory()</code>                                |
|-------------|------------------------------------------------------------|
| Demo        | <code>try nr.recordMemory(megabytes: 32)</code>            |
| Description | Record the current amount of memory being used.            |
| Parameters  | - megabytes: Double, amount of memory currently being used |

### Transaction

Transaction in Perfect NewRelic has been defined as a class, with constructor as below:

```
public init(_ instance: NewRelic,
    webType: Bool? = nil,
    category: String? = nil,
    name: String? = nil,
    url: String? = nil,
    attributes: [String: String],
    maxTraceSegments: Int? = nil
) throws
```

## Constructor parameters:

- instance: NewRelic instance, **required**.
- webType: **optional**. true for WebTransaction and false for other. default is true.
- category: **optional**. name of the transaction category, default is 'Uri'
- name: **optional**. transaction name
- url: **optional**. request url for a web transaction
- attributes: **optional**. transaction attributes, pair of "name: value"
- maxTraceSegments: **optional**. Set the maximum number of trace segments allowed in a transaction trace. By default, the maximum is set to 2000, which means the first 2000 segments in a transaction will create trace segments if the transaction exceeds the trace threshold (4 x apdex\_t).

## Demo of Transaction Class Initialization:

```
let nr = NewRelic()
let t = try Transaction(nr, webType: false,
    category: "my-class-1", name: "my-transaction-name",
    url: "http://localhost",
    attributes: ["tom": "jerry", "pros": "cons", "muddy": "puddels"],
    maxTraceSegments: 2000)
```

## Error Notice

Perfect NewRelic provides `setErrorNotice()` function for transactions:

```
try t.setErrorNotice(
    exceptionType: "my-panic-type-1",
    errorMessage: "my-notice",
    stackTrace: "my-stack",
    stackFrameDelimiter: "<frame>")
```

Parameters of `setErrorNotice()`:

- exceptionType: type of exception that occurred
- errorMessage: error message
- stackTrace: print stack trace when error occurred
- stackFrameDelimiter: delimiter to split stack trace into frames

## Segments

Segments in a transaction can be either Generic, DataStore or External, see demo below:

```
// assume t is a transaction
try t.setErrorNotice(exceptionType: "my-panic-type-1", errorMessage: "my-notice", stackTrace: "my-stack", stackFrameDelimiter: "<frame>")

let root = try t.segBeginGeneric(name: "my-segment")
// do some generic operations in this transaction
try t.segEnd(root)

// NOTE: it will automatically obfuscate the sql input and rollup if failed as well
let sub = try t.segBeginDataStore(table: "my-table", operation: .INSERT, sql: "INSERT INTO table(field) value('000-000-0000')")
// do some data operations in this transaction
try t.segEnd(sub)

let s2 = try t.segBeginExternal(host: "perfect.org", name: "my-seg")
// do some external operations in this transaction
try t.segEnd(s2)
```

Parameters:

- parentSegmentId: id of parent segment, root segment by default, i.e., `NewRelic.ROOT_SEGMENT`.
- name: name to represent segment

# Perfect-Kafka

This project provides an express Swift wrapper of librdkafka.

This package builds with Swift Package Manager and is part of the [Perfect](#) project but can also be used as an independent module.

## Release Notes for MacOS X

---

Before importing this library, please install librdkafka first:

```
$ brew install librdkafka
```

Please also note that a proper pkg-config path setting is required:

```
$ export PKG_CONFIG_PATH="/usr/local/lib/pkgconfig"
```

## Release Notes for Linux

---

Before importing this library, please install librdkafka-dev first:

```
$ sudo apt-get install librdkafka-dev
```

## Quick Start

---

### Kafka Client Configurations

Before starting any stream operations, it is necessary to apply settings to clients, i.e., producers or consumers.

Perfect Kafka provides two different categories of configuration, i.e. `Kafka.Config()` for global configurations and `Kafka.TopicConfig()` for topic configurations.

#### Initialization of Global Configurations

To create a configuration set with default value settings, simple call:

```
let conf = try Kafka.Config()
```

or, if another configuration based on an existing one can be also duplicated in such a form:

```
let conf = try Kafka.Config()  
// this will keep the original settings and duplicate a new one  
let conf2 = try Kafka.Config(conf)
```

#### Initialization of Topic Configurations

Topic configuration shares the same initialization fashion with global configuration.

To create a topic configuration with default settings, call:

```
let conf = try Kafka.TopicConfig()
```

or, if another configuration based on an existing one can be also duplicated in such a form:

```
let conf = try Kafka.TopicConfig()  
// this will keep the original settings and duplicate a new one  
let conf2 = try Kafka.TopicConfig(conf)
```

### Access Settings of Configuration

Both `Kafka.Config` and `Kafka.TopicConfig` have the same api of accessing settings.

#### List All Variables with Value

`Kafka.Config.properties` and `Kafka.TopicConfig.properties` provides dictionary type settings:

```
// this will print out all variables in a configuration
print(conf.properties)
// for example, it will print out something like:
// ["topic.metadata.refresh.fast.interval.ms": "250",
// "receive.message.max.bytes": "100000000", ...]
```

## Get a Variable Value

Call `get()` to retrieve the value from a specific variable:

```
let maxBytes = try conf.get("receive.message.max.bytes")
// maxBytes would be "100000000" by default
```

## Set a Variable with New Value

Call `set()` to save settings for a specific variable:

```
// this will restrict message receiving buffer to 1MB
try conf.set("receive.message.max.bytes", "1048576")
```

## Producer

Perfect-Kafka provides a `Producer` class to send data / message to Kafka hosts. `Producer` can send a message one at a time, or sent multiple messages in a batch. Messages can be either text string or binary bytes.

```
let producer = try Producer("VideoTest")
let brokers = producer.connect(brokers: "host:9092")
if brokers > 0 {
    let _ = try producer.send(message: "hello, world!")
}
```

Before sending any actual messages, a few steps are required to setup the connection to Kafka hosts.

### Producer Instance with a Topic

To initialize a `Producer` instance, a topic name is required no matter whether this topic exists in the Kafka hosts or not.

If the topic didn't exist when connected to Kafka hosts / brokers, `Producer()` would try to create a new one; Otherwise it would use the existing topic for further operations.

For example, the demo below shows how to start a producer with a topic named "VideoTest":

```
let producer = try Producer("VideoTest")
```

### Connect to Brokers

Use method `connect()` to connect to one or more message brokers, i.e., Kafka hosts ( host and port ):

```
let brokers = producer.connect(brokers: "host1:9092,host2:9092,host3:9092")
```

If success, it will return the number of hosts that connected.

Alternatively, it is also possible to connect to brokers by different parameter fashions, take example, hosts can be an array of string:

```
let brokers = producer.connect(brokers: ["host1:9092", "host2:9092", "host3:9092"])
```

or dictionary: `swift let brokers = producer.connect(brokers: ["host1": 9092, "host2": 9092, "host3": 9092])`

## Send Messages

Perfect Kafka allows to send either text or binary messages to brokers one at a time or in a batch.

| Method                                                 | Description                                    | Returns                              |
|--------------------------------------------------------|------------------------------------------------|--------------------------------------|
| <code>send(message: String, key: String? = nil)</code> | a text message with an optional key to send    | an Int64 message id                  |
| <code>send(message: [Int8], key: [Int8] = [])</code>   | a binary message with an optional key to send  | an Int64 message id                  |
| <code>send(messages: [(String, String?)])</code>       | text messages with optional keys in an array   | [Int64] message IDs for each message |
| <code>send(messages: [[[Int8], [Int8]])]</code>        | binary messages with optional keys in an array | [Int64] message IDs for each message |

## Sent or Not

Perfect Kafka `send()` is asynchronous function so the library provides a few extra methods to determine the sending status of each message.

- `OnSent()` callback. If set properly, each message will call this event once actually sent. For example: `producer.OnSent = { print("msg #\($0) was sent") }`. The only parameter of this event is the Int64 message id returned by `send()`.
- `producer.outbox` is an [Int64] array to indicate the messages in sending queue. *NOTE* As a high performant streaming platform, the existence of messages in outbox doesn't mean that they were failed to send, so don't try to resend these message unless it was explicitly confirmed that they were failed to send.
- `OnError()` callback. Producer will call this event if something wrong, e.g., `producer.OnError = { print("error: \($0)") }` will print out the error message if happen.
- `flush(_ seconds: Int)` method can help wait seconds for clearing the message queue and flushing the outbox.

## Consumer

Before actually receiving messages from Kafka with a specific topic, a few procedures are required to initialize a Consumer instance:

```
let consumer = try Consumer("VideoTest")
let brokers = consumer.connect(brokers: ["host1": 9092, "host2": 9092, "host3": 9092])
guard brokers > 0 else {
    // connection failed
} //end guard
```

## Partitions

Once connected, it is a good idea to get the information from the brokers to see if there are sufficient resources, i.e., partitions, for further operations:

```
let info = try consumer.brokerInfo()
print(info)
```

The above variable `info` is a `MetaData` structure as reference below:

| Member  | Type     | Description                  |
|---------|----------|------------------------------|
| brokers | [Broker] | An array of Broker structure |
| topics  | [Topic]  | An array of Topic structure  |

Structure `Broker` stores the information of a broker:

| Member | Type   | Description             |
|--------|--------|-------------------------|
| id     | Int    | Broker Id               |
| host   | String | Host name of the broker |
| port   | Int    | Host port that listens  |

The major content of `Topic` structure is to record how many partitions are using in such a topic:

| Member     | Type        | Description                    |
|------------|-------------|--------------------------------|
| name       | String      | Topic name                     |
| err        | Exception   | Topic error reported by broker |
| partitions | [Partition] | Partitions of this topic       |

Data structure `Partition` is vitally important to indicate the partition id for messaging:

| Member   | Type      | Description                                       |
|----------|-----------|---------------------------------------------------|
| id       | Int       | Partition Id - use this to start / stop messaging |
| err      | Exception | Partition error reported by broker                |
| leader   | Int       | Leader broker                                     |
| replicas | [Int]     | Replica brokers                                   |
| isrs     | [Int]     | In-Sync-Replica brokers                           |

Practically, partition info could be acquired by way below:

```
let consumer = try Consumer("VideoTest")
let brokers = consumer.connect(brokers: ["host1": 9092, "host2": 9092, "host3": 9092])
guard brokers > 0 else {
    // connection failed
}
consumer.OnArrival = { m in print("message : #\{m.offset} \{m.text}")}
let info = try consumer.brokerInfo()
guard info.topics.count > 0 else {
    // no topic found
}
guard info.topics[0].name == "VideoTest" else {
    // it is not the topic we want
}
let partitions = info.topics[0].partitions
```

## Download Messages From A Partition

Code below shows how to download messages from a partition. In this demo, we assume `let partId = partitions[0].id`:

```
consumer.OnArrival = { m in
    print("message #\{m.offset} : \{m.text}")
}
// start downloading
try consumer.start(partition: partId)
// run until end of program
while(notEndOfProgram) {
    let total = try consumer.poll(partition: partId)
    print("\{total} messages arrived in this moment")
}
consumer.stop(partId)
```

Now we take a walk through:

Firstly, `OnArrival()` event is a callback with a `Message` data structure:

| Member    | Type      | Description                                                    |
|-----------|-----------|----------------------------------------------------------------|
| err       | Exception | Error: if the message is good or not                           |
| topic     | String    | topic name of the message                                      |
| partition | Int       | partition of the message                                       |
| isText    | Bool      | if the message is a valid UTF-8 text or not                    |
| data      | [Int8]    | the original binary data of message body                       |
| text      | String    | decoded message body in a UTF-8 string, if <code>isText</code> |
| keyIsText | Bool      | if the key is a valid UTF-8 text                               |
| keybuf    | [Int8]    | the original binary data of optional key                       |
| key       | String    | decoded key in a UTF-8 string, if <code>keyIsText</code>       |
| offset    | Int64     | offset inside the topic                                        |

Secondly, call `start()` to start download messages: `func start(_ from: Position = .BEGIN, partition: Int32 = RD_KAFKA_PARTITION_UA)`, here are the parameter details: - from: Position, from which position of the messages in the partition to download. Valid value can be `.BEGIN` to indicate downloading every messages from the very beginning, or `.END` to download the most reason one, or `.STORED` to download the previous stored messages in case of failure, or `.SPECIFY(Int64)` to start downloading from a specific location. *NOTE* use `func store(_ offset: Int64, partition: Int32 = RD_KAFKA_PARTITION_UA)` to store a specific message if `.STORED` is needed. - partition: Int32, the partition id.

Then Perfect Kafka provides the `poll()` function to wait a short while to listen the activity of a specific partition: `func poll(_ timeout: UInt = 10, partition: Int32 = RD_KAFKA_PARTITION_UA)`. The `timeout` is the milliseconds to wait for polling.

Finally, call `stop()` to end the messaging.

## Perfect-Mosquitto

This project provides a Swift class wrapper of `mosquitto` client library which implements the MQTT protocol version 3.1 and 3.1.1.

This package builds with Swift Package Manager and is part of the [Perfect](#) project, however, it can work independently as a Server Side Swift component.

Ensure you have installed and activated the latest Swift 3.0 tool chain.

## OS X Notes

### Homebrew Installation

This project depends on mosquitto library. To install on mac OS, try command `brew`:

```
$ brew install mosquitto
```

### PC File

A package configuration file is needed, for example, `/usr/local/lib/pkgconfig/mosquitto.pc` as below:

```
Name: mosquitto
Description: Mosquitto Client Library
Version: 1.4.11
Requires:
Libs: -L/usr/local/lib -lmosquitto
Cflags: -I/usr/local/include
```

Please also export an environmental variable called `$PKG_CONFIG_PATH`:

```
$ export PKG_CONFIG_PATH="/usr/local/lib/pkgconfig:/usr/lib/pkgconfig"
```

## Linux Notes

This project depends on Ubuntu 16.04 library `libmosquitto-dev`:

```
$ apt-get libmosquitto-dev
```

## Quick Start

### Library Open / Close

Before using any functions of Perfect Mosquitto, please initialize the function set by calling `Mosquitto.OpenLibrary()`. Also, `Mosquitto.CloseLibrary()` is highly recommended once program quitted.

*Note* Both operations are not thread safe, so please perform the operation prior to any threads.

### Initialize a Mosquitto Client

Mosquitto instance can be constructed by calling with or without any parameters. The simplest form can be:

```
let m = Mosquitto()
```

Which means assigning a random client id to this new instance, while all messages and subscriptions will be cleared once disconnected.

However, you can also assign it with a customized client id with instruction of keeping all messages and subscription by this specific name. This is useful for resuming work in case of connection loss.

```
let mosquitto = Mosquitto(id: "myInstanceId", cleanSession: false)
```

### Connect to a Message Broker

A message broker is a server that implements MQTT protocol and serves all clients in terms of messaging - receiving messages from producer and dispatching them to message subscribers.

Although connection to a message broker can be asynchronous, keeping alive or binding to a specific network address, api `connect()` can be as express as demo below - only a host name and a port (usually 1883) are required:

```
try mosquitto.connect(host: "mybroker.com", port: 1883)
```

Although the instance can `disconnect()` from a broker automatically when no longer uses the object, it is recommended to call this function explicitly for a better practice. Besides, a `reconnect()` function is available for the same instance.

### Threading Model

#### Start / Stop

Perfect Mosquitto is flexible in dealing with threads. Clients can call `start()` to run the mosquitto thread in background, which will automatically execute message publishing / receiving without any extra operations in the main thread, i.e., the thread will do the actual sending after calling `publish()` and activate callbacks for incoming messages. If no longer running, you can also stop the service thread at any time by calling `stop()`

```
// start the messaging thread as a background service, it will not block the main thread and will return immediately.
try mosquitto.start()

// do your other work in the main thread, such as publishing etc., and the messages received will go to the callbacks

// stop the background messaging service if no longer need.
try mosquitto.stop()
```

#### Wait for Event

Or, alternatively, you can process events in the main thread, by calling `wait()` method frequently in a message polling style:

```
// wait a minimal while for events.
// In this specific moment, mosquitto will perform actual message sending,
// and pull messages from broker / run call backs.
try mosquitto.wait(0)
```

The only parameter of `wait()` is the timeout value for message polling in milliseconds. Zero represents the minimal period of the system wait and negative value will be treated the same as 1000 (1 second).

⚠️ **NOTE** ⚠️ **DO NOT MIX** `start()` / `stop()` with `wait()`

## Publish a Message

Once connected to a broker, you can send messages at any time you want:

```
var msg = Mosquitto.Message()
msg.id = 100 // input your message id here
msg.topic = "publish/test"
msg.string = "publication test 🇨🇦"

let mid = try mosquitto.publish(message: msg)
```

As demo above, firstly setup an empty message structure, then assign an message id (integer), a topic for this message and the message content; Then call `publish()` method to send it to the broker with a returned message id.

*Note* Message content can also be a binary buffer, for example: `swift // send a [Int8] array msg.payload = [40, 41, 42, 43, 44, 45]`

Once published, call `start()` or `wait()` to perform the actual message sending as described in the Thread Model.

## Message Subscription and Receiving

The only way to receive a MQTT message in Perfect Mosquitto is messaging callback:

```
mosquitto.OnMessage = { msg in

    // print out message id
    print(msg.id)

    // print out message topic
    print(msg.topic)

    // print out message content
    print(msg.string)

    // print out message body, in a binary array form
    print(msg.payload)
} //end on Message
```

Once set the callback, you can call `subscribe()` to complete the message subscription on client side:

```
try mosquitto.subscribe(topic: "publish/test")
```

Once subscribed, call `start()` or `wait()` to perform actual receiving process as described in the Thread Model.

## More API

Perfect Mosquitto also provides a rich set of functions beside the above ones, please check the project references for detail information.

### Event Callbacks

Please set the following event callbacks for your mosquitto objects if need:

| API                                       | Parameters                      | Description                  |
|-------------------------------------------|---------------------------------|------------------------------|
| <code>OnConnect { status in }</code>      | <code>ConnectionStatus</code>   | Triggered on connection      |
| <code>OnDisconnected { status in }</code> | <code>ConnectionStatus</code>   | Triggered on disconnection   |
| <code>OnPublish { msg in }</code>         | <code>Message</code>            | Triggered when message sent  |
| <code>OnMessage { msg in }</code>         | <code>Message</code>            | Triggered on message arrival |
| <code>OnSubscribe { id, qos in }</code>   | <code>(Int32, [Int32])</code>   | Triggered on subscription    |
| <code>OnUnsubscribe { id in }</code>      | <code>Int32</code> (message id) | Triggered on unsubscription  |
| <code>OnLog { level, content in }</code>  | <code>(LogLevel, String)</code> | Triggered on log output      |

## TLS Configuration

- Set TLS certification file: `func setTLS(caFile: String, caPath: String, certFile: String? = nil, keyFile: String? = nil, keyPass: String? = nil) throws`
- Set TLS verification method: `func setTLS(verify: SSLVerify = .PEER, version: String? = nil, ciphers: String? = nil) throws`
- Set pre shared key: `func setTLS(psk: String, identity: String, ciphers: String? = nil) throws`

## Miscellaneous Functions

- Configure will information for a mosquitto instance: `func setConfigWill(message: Message?) throws`
- Configure username and password for a mosquitton instance: `func login(username: String? = nil, password: String? = nil) throws`
- Set the number of seconds to wait before retrying messages: `func setMessageRetry(max: UInt32 = 20)`
- Set the number of QoS 1 and 2 messages that can be “in flight” at one time: `func setInflightMessages(max: UInt32 = 20) throws`
- Control the behaviour of the client when it has unexpectedly disconnected: `func reconnectSetDelay(delay: UInt32 = 2, delayMax: UInt32 = 10, backOff: Bool = false) throws`
- Reconnecting to a broker after a connection has been lost: `func reconnect(_ asynchronous: Bool = true) throws`
- Reuse an existing mosquitto instance: `func reset(id: String? = nil, cleanSession: Bool = true) throws`
- Set MQTT Version: `func setClientOption(_ version: MQTTVersion = .V31, value: UnsafeMutableRawPointer) throws`
- Explain an exception (in English): `static func Explain(_ fault: Exception) -> String`

# Perfect-ZooKeeper

This project implements an express Swift library of ZooKeeper

This package builds with Swift Package Manager and is part of the [Perfect](#) project.

## Release Note

This project can only be built on  Ubuntu 16.04  . Mac OS X doesn't support it.

## Quick Start

### Swift Package Manager

Add Perfect-ZooKeeper to your project's Package.swift:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-ZooKeeper.git", majorVersion: 1)
```

### Import Library

Import Perfect-ZooKeeper library to your source code:

```
import PerfectZooKeeper
```

### Debug

To debug your ZooKeeper application, Perfect-ZooKeeper provides a static method called `debug` to set the debug level, for example:

```
// this will set debug level to the whole application
ZooKeeper.debug()
```

You can also adjust the debug level to method `debug(_ level: LogLevel = .DEBUG)` by a parameter:

- level: LogLevel, the debug level, could be .ERROR, .WARN, .INFO, or .DEBUG by default

## Log

To trace and log your ZooKeeper application, Perfect-ZooKeeper provides a static method called `log`, for example:

```
// this will redirect the debug information to standard error stream
ZooKeeper.log()
```

The only parameter of `log(_ to: UnsafeMutablePointer<FILE> = stderr)` is `to`, a `FILE` pointer as in C stream, it is `stderr` by default but you can redirect it to any available `FILE` streams.

## Using ZooKeeper Object

Before performing any actual connections, it is necessary to construct a `ZooKeeper` object:

```
let z = ZooKeeper()
```

Or alternatively, you can also add a timeout setting to such an object, which defines the maximal milliseconds to wait for connection:

```
// indicates that the connection attempt will be treated as broken in eight seconds.
let z = ZooKeeper(8192)
```

## Connect to ZooKeeper Hosts

Use `ZooKeeper.connect()` to connect to specified hosts. Take example, the demo below shows how to connect to a ZooKeeper host, and how the program invokes your callback once connected:

```
try z.connect("servername:2181") { connect in
    switch(connect) {
    case .CONNECTED:
        // connection is made
    case .EXPIRED:
        // connection is expired
    default:
        // connection is broken
    }
}
```

⚠ NOTE ⚠, you may also connect to a cluster of host by replacing the above connection string into a string of multiple hosts in such an expression:

"server1:2181,server2:2181,server3:2181", but this may be subject to the ZooKeeper version that you are connecting with. For more information of ZooKeeper connection string, see [ZooKeeper Programmer's Guide](#)

## Existence of a ZNode

Once connected, you may check a specific ZNode by calling `exists()`, for example:

```
let a = try z.exists("/path/to")
print(a)
```

This function will return a `Stat()` structure if nothing wrong, for example:

```
swift // this is a sample result of calling print(try z.exists("/path/to")) Stat(czxid: 0, mxzid: 0, ctime: 0, mtime: 0, version: 0, cversion: 0)
```

## List Children of a ZNode

Method `children()` may list all available direct sub nodes under the objective and put them into an array of string.

```
let kids = try z.children("/path/to")
// if success, it will list all sub nodes under /path/to in an array.
// for example, if there is /path/to/a and /path/to/b,
// then the result is probably ["a", "b"]
print(kids)
```

## Save Data to a ZNode

As a key-value directory, each ZNode may contain a small amount of data in form of a string, usually not exceed to 10k. You can save your own configuration data into a ZNode, synchronously or asynchronously, as demanded.

## Save Data Synchronously

Synchronous version of `save()` will return a `Stat()` structure if success:

```
let stat = try z.save("/path/to/key", data: "my configuration value of key")
print(stat)
```

Parameters of `func save(_ path: String, data: String, version: Int = -1) throws -> Stat` :- path: String, the absolute full path of the node to access - data: String, the data to save - version: Int, version of data, default is -1 which indicates ignoring the version info

## Save Data Asynchronously

Asynchronous version of `save()` has all the same parameters with an extra `StatusCallback` but without returning value:

```
try z.save("/path/to/key", data: "my configuration value of key") { err, stat in
    guard err == .ZOK else {
        // something wrong
    }
    guard let st = stat else {
        // async save() returns a null status
    }
    // print the status after saving
    print(st)
}
```

## Load Data from a ZNode

Similar to `save()`, the ZooKeeper `load()` also has both synchronous version and asynchronous version as well:

### Load Data Synchronously

To load data from a ZNode synchronously, simply call `load("/path/to")`, and it will return a tuple of `(value: String, stat: Stat)`, which stands for data value and status of the node:

```
let (value, stat) = try z.load("/path/to")
```

### Load Data Asynchronously

The data loading from a ZNode asynchronously will require an extra callback with a parameter of (error: Exception, value: String, Stat) as demo below:

```
try z.load(path) { err, value, stat in
    guard err == .ZOK else {
        // something wrong
    } //end guard
    guard let st = stat else {
        // there is no status information of node
    } //end guard
    print(st)
    // this is the actual data value as a String
    print(value)
} //end load
```

## Make a Node

Function `func make(_ path: String, value: String = "", type: NodeType = .PERSISTENT, acl: ACLTemplate = .OPEN) throws -> String` can build different type of nodes, with writing data value and set ACL (Access Control List) info for this node in the same moment. Here are the parameters:

- path: String, the absolute full path of the node to make
- value: String, the value to store into node
- type: NodeType, i.e., .PERSISTENT, .EPHEMERAL, .SEQUENTIAL, or .LEADERSHIP, which means ephemeral + sequential. Default type is .PERSISTENT
- acl: ACLTemplate, basic ACL template to apply in this incoming node, i.e., .OPEN, .READ or .CREATOR. Default is .OPEN, which means nothing to restrict

⚠ Note ⚠ The return value will be the newly created pat, i.e., the same one as input if type is .PERSISTENT or .EPHEMERAL, and will be appended with a serial number only if the node type is .SEQUENTIAL or .LEADERSHIP.

### Make a Persistent Node

The following code demonstrates how to create a persistent node with data:

```
let _ = try z.make("/path/to/key", value: "my config data value for this key")
```

### Make a Temporary Node

A temporary ZNode means it will automatically disappear once the session was over (usually after a few seconds of disconnection). To create such a node, simply add a node type parameter to call:

```
let _ = try z.make("/path/to/tempKey", value: "data for this temporary key", type: .EPHEMERAL)
```

### Make a Sequential Node

A Sequential ZNode means if you want to create a `/path/to/key` node, it will return a `/path/to/key0123456789`, i.e., a 10 digit number will be added to the node you named.

```
let path = try z.make("/path/to/myApplication", type: .SEQUENTIAL)
print(path)
// if success, the path will be something like `/path/to/myApplication0000000123`
```

⚠️ Note ⚠️ Sequential node is persistent and can be removed only by calling `remove()` method explicitly.

### Make a Leadership Node

The purpose of leadership node is to select a leadership server among all candidates. Similar to `.SEQUENTIAL`, the leadership node is also a temporary node, which help all clustered backups to determine who shall be the leader among the cluster by checking whose serial number is the minimal one, which means who is the first available.

```
let path = try z.make("/path/to/myApplication", type: .LEADERSHIP)
print(path)
// if success, the path will be something like `/path/to/myApplication0000000123`
// and will be deleted automatically once disconnected.
```

### Remove a Node

Method `func remove(_ path: String, version: Int32 = -1)` enables the function to delete an existing node:

```
// this action will result in a removal regardless node versions.
try z.remove("/path/to/uselessNode")
```

### Watch for Changes

Perfect-ZooKeeper provides a useful function `watch()` to monitor other instance operations against a specific node. The full API of `watch()` method is

`func watch(_ path: String, eventType: EventType = .BOTH, renew: Bool = true, onChange: @escaping WatchCallback)` with parameter explained below:

- path: String, the absolute full path of the node to watch
- eventType: watch for `.DATA` or `.CHILDREN`, or `.BOTH`
- renew: watch the event for once or for ever, false for once and true for ever.
- onChange: WatchCallback, callback once something changed

For example:

```
try z.watch("/path/to/myCheese") { event in
    switch(event) {
    case CONNECTED:
        // me myself just connected to this node???
    case DISCONNECTED:
        // connection is broken
    case EXPIRED:
        // connection is expired
    case CREATED:
        // this shall never happen - just created for the node me watch?
    case DELETED:
        // the node has been deleted by someone else
    case DATA_CHANGED:
        // someone just touched my cheese
    case CHILD_CHANGED:
        // children were changed
    default:
        // unexpected here
    }
}
} //end watch
```

## Election in a Cluster

Perfect ZooKeeper provides a convenient way of choosing a leader / master from a cluster by calling method `elect()` :

```
let (me, leader, candidates) = try z.elect("/path/to")
```

If success, the return value of `elect()` function is a tuple of `(me: Int, leader: Int, candidates: [Int])`, which means every candidate, include the current instance, will be included in the `candidates` array as an integer. Result `me` is the current instance's election serial number and the result of `leader` is the number represents the final leader in this election. If `me == leader`, then congratulations - the current instance of ZooKeeper just won the election. In such a case, further actions should be done to upgrade current instance into the master of cluster.

## ACL Operations

ACL, the access control list, can be manipulated in ZooKeeper API by core data structure `ACL_vector`, i.e., a C based pointer array. Content of such a structure could be checked by similar operations as below:

```
func show(_ aclArray: ACL_vector) {
    guard let pAcl = aclArray.data else {
        // the array doesn't contain any valid data
        return
    }
    var i = 0
    while (Int32(i) < aclArray.count) {
        let cursor = pAcl.advanced(by: i)
        let acl = cursor.pointee
        let scheme = String(cString: acl.id.scheme)
        let id = String(cString: acl.id.id)
        // id is subject to scheme
        // if scheme is "world", then id shall be "anyone",
        // scheme "auth" doesn't use any id.
        // scheme "ip" uses host ip as id, such as "1.2.3.4/5"
        // scheme "x509" uses client X500 principal as id.
        // scheme "digest" use name:password to generate MD5 as id:
        // `username:base64 encoded SHA1 password digest.`
        print("id: \(id)")
        print("scheme: \(scheme)")
        // permission is a combination of :
        // ZOO_PERM_READ | ZOO_PERM_WRITE | ZOO_PERM_CREATE
        // ZOO_PERM_DELETE | ZOO_PERM_ADMIN | ZOO_PERM_ALL
        let perm = String(format: "%8X", acl.perms)
        print("permissions: \(perm)")
        i += 1
    }
}
```

⚠ CAUTION ⚠ To manipulate `ACL_vector` with permission options, please make sure to import the C ZooKeeper library:

```
import czookeeper
```

## Get ACL Info

Method `getACL()` can retrieve ACL info from a ZNode:

```
let (acl, stat) = try z.getACL("/path/to")
```

The return value is a tuple of `(acl: ACL_vector, stat: Stat)` stands for the acl info as an `ACL_vector` pointer array and status of the node.

## Set ACL Info

Method `func setACL(_ path: String, version: Int32 = -1, acl: ACL_vector) throws` may save an ACL setting to a ZNode, where the version could be skipped by default and providing a valid `ACL_vector` pointer array:

```
var acl = ACL_vector()  
// do some modification to acl variable  
try z.setACL("/path/to", acl:acl)
```

However, there is also another alternative form of `setACL()` by replacing the complicated `ACL_vector` to preset ACL template:

```
// available templates include .OPEN, .READ and .CREATOR  
// default is .OPEN, which means nothing to restrict  
try z.setACL("/path/to", aclTemplate: .READ)
```

# Perfect TensorFlow

This project is an experimental wrapper of TensorFlow C API which enables Machine Learning in Server Side Swift.

## Development Notes

These files are the key part of Perfect-TensorFlow:

```
Sources  
├─ PerfectTensorFlow  
│   ├── APILoader.swift (886 lines, translated from tensorflow/c/c_api.h)  
│   └── PerfectTensorFlow.swift (2277 lines)  
└─ TensorFlowAPI  
    ├── TensorFlowAPI.c (72 lines)  
    └── include  
        └── TensorFlowAPI.h (137 lines)
```

All other Swift sources named as 'pb.\*.swift', which is totally up to 45,000+ lines of code, are automatically generated by `updateprotos.sh` in the root directory. Unfortunately, if using such a script, you still need to manually edit the `public typealias` part listed in the **PerfectTensorFlow.swift**.

Up to now there is no such a plan to generate these protocol buffer files dynamically in the Swift Source since Perfect-TensorFlow is a part of Perfect, although it can run independently, all features of Perfect framework are built by Swift Package Manager for consistency consideration. However, since the project is also fast growing, all pull request, ideas, suggestions and comments are welcome!

## Quick Start

### Installation

Perfect-TensorFlow is based on TensorFlow C API, i.e., `libtensorflow.so` on runtime. This project contains an express CPU v1.1.0 installation script for this module on both macOS / Ubuntu Linux, and will install the dynamic library into path `/usr/local/lib/libtensorflow.so`. You can download & run [install.sh](#). Before running this script, please make sure that `wget` has been installed onto your computer, otherwise please run either `brew install wget` for macOS or `sudo apt-get install wget` for Ubuntu first.

For more installation options, such as GPU/CPU and multiple versions on the same machine, please check TensorFlow website: [Installing TensorFlow for C](#)

### Perfect TensorFlow Application

To use this library, add dependencies to your project's Package.swift with the **LATEST TAG**:

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-TensorFlow.git", majorVersion: 1)
```

Then declare the library:

```
// TensorFlowAPI contains most API functions defined in libtensorflow.so
import TensorFlowAPI

// This is the Swift version of TensorFlow classes and objects
import PerfectTensorFlow

// To keep the naming consistency with TensorFlow in other languages such as
// Python or Java, making an alias of `TensorFlow` Class is a good idea:
public typealias TF = TensorFlow
```

## Library Activation

⚠️ NOTE ⚠️ Prior to use **ANY ACTUAL FUNCTIONS** of Perfect TensorFlow framework, `TF.Open()` must be called first:

```
// this action will load all api functions defined
// in /usr/local/lib/libtensorflow.so
try TF.Open()
```

Please also note that you can active the library with a specific path, alternatively, especially in case of different versions or CPU/GPU library adjustment required:

```
// this action will load the library with the path
try TF.Open("/path/to/DLL/of/libtensorflow.so")
```

## "Hello, Perfect TensorFlow!"

Here is the Swift version of "Hello, TensorFlow!":

```
// define a string tensor
let tensor = try TF.Tensor.Scalar("Hello, Perfect TensorFlow! 🇨🇦")

// declare a new graph
let g = try TF.Graph()

// turn the tensor into an operation
let op = try g.const(tensor: tensor, name: "hello")

// run a session
let o = try g.runner().fetch(op).addTarget(op).run()

// decode the result
let decoded = try TF.Decode(strings: o[0].data, count: 1)

// check the result
let s2 = decoded[0].string
print(s2)
```

## Matrix Operations

As you can see, Swift version of TensorFlow keeps the same principals of the original one, i.e., create tensors, save tensors into graph, define the operations and then run the session & check the result.

Here is an other simple example of matrix operations in Perfect TensorFlow:

```

/* Matrix Multiply:
| 1 2 |   | 0 1 |   | 0 1 |
| 3 4 | * | 0 0 | = | 0 3 |
*/
// input the matrix.
let tA = try TF.Tensor.Matrix([[1, 2], [3, 4]])
let tB = try TF.Tensor.Matrix([[0, 0], [1, 0]])

// adding tensors to graph
let g = try TF.Graph()
let A = try g.const(tensor: tA, name: "Const_0")
let B = try g.const(tensor: tB, name: "Const_1")

// define matrix multiply operation
let v = try g.matMul(l: A, r: B, name: "v", transposeB: true)

// run the session
let o = try g.runner().fetch(v).addTarget(v).run()
let m:[Float] = try o[0].asArray()
print(m)
// m shall be [0, 1, 0, 3]

```

## Load a Saved Artificial Neural Network Model

Besides building graph & sessions in code, Perfect TensorFlow also provides a handy method to load models into runtime, i.e, generate a new session by loading a model file:

```

let g = try TF.Graph()

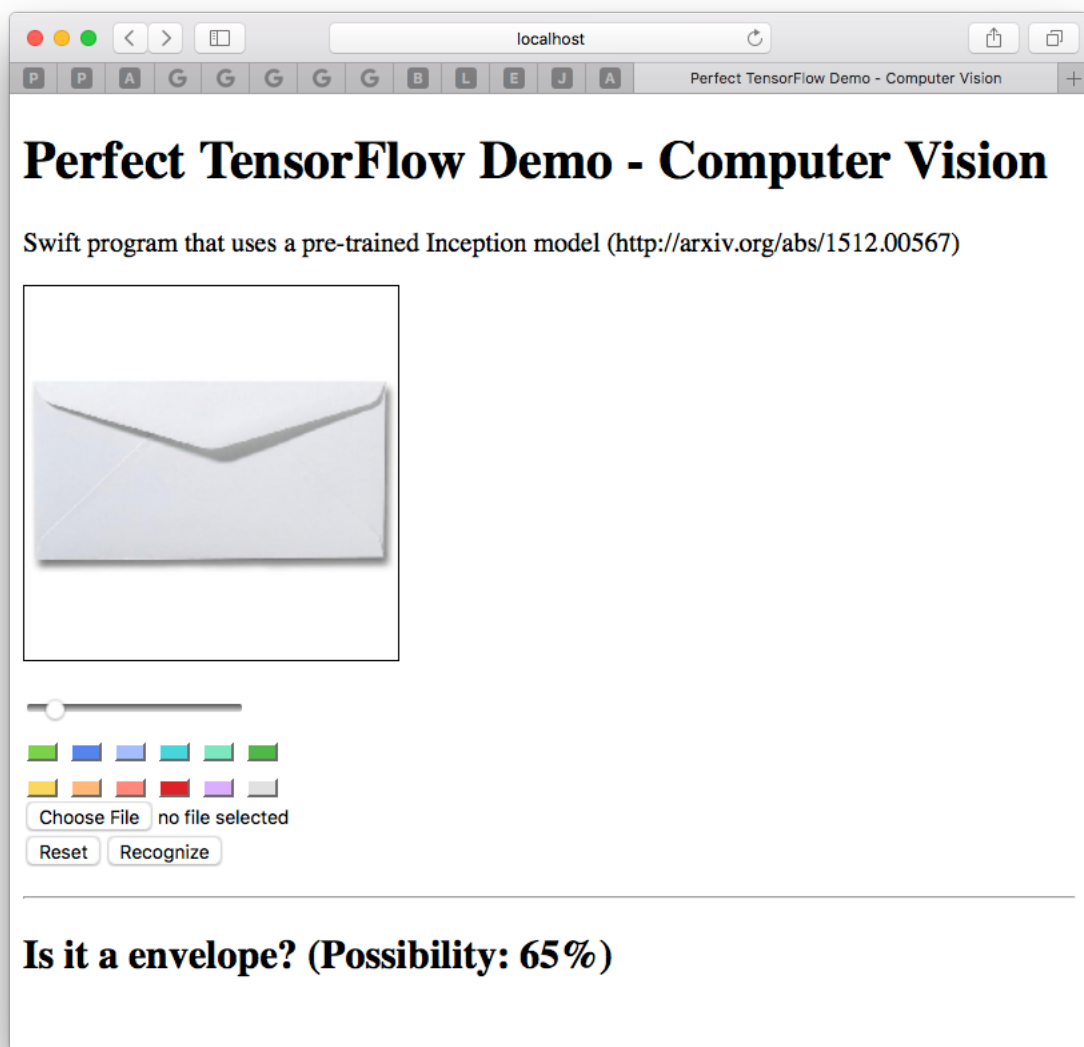
// the meta signature info defined in a saved model
let metaBuf = try TF.Buffer()

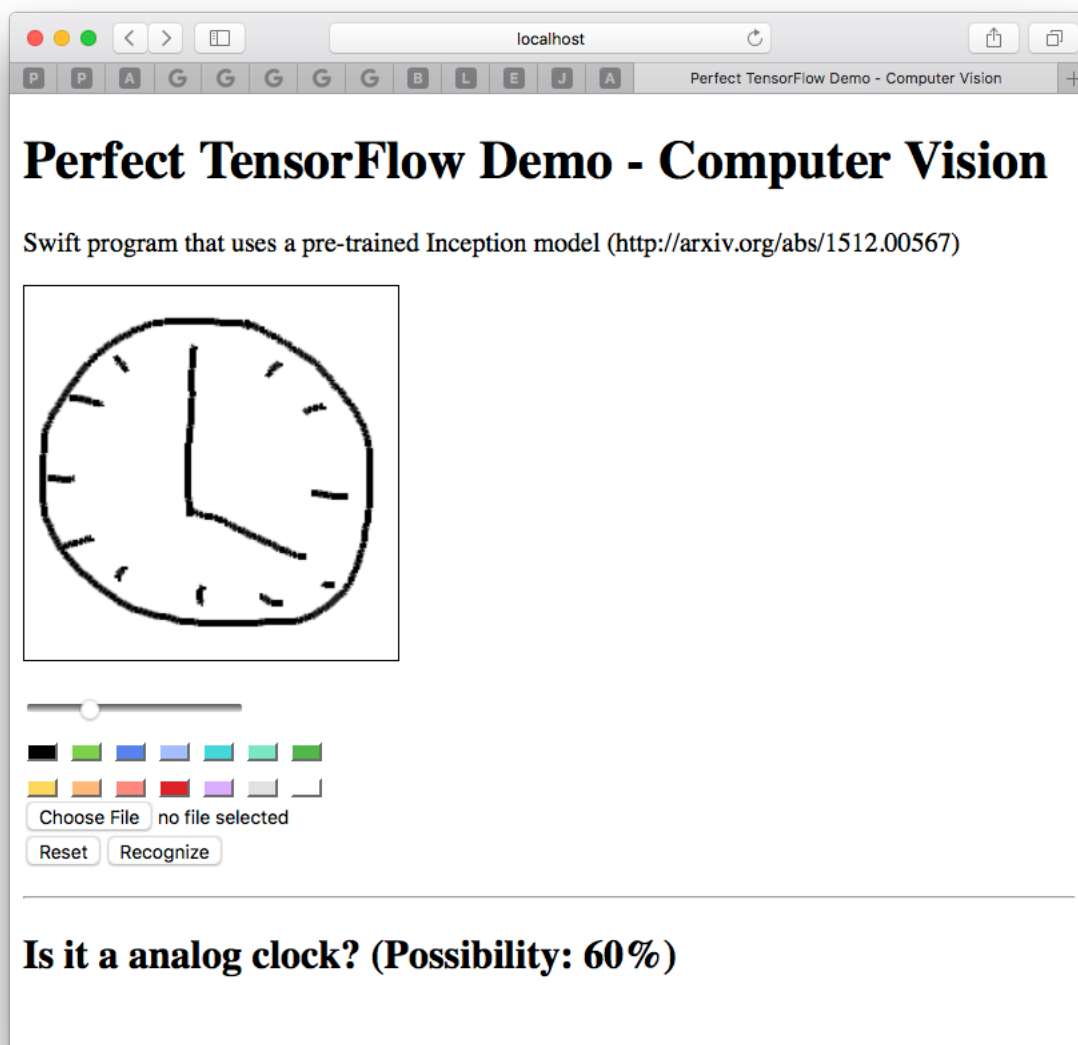
// load the session
let session = try g.load(
    exportDir: "/path/to/saved/model",
    tags: ["tag1", "tag2", ...],
    metaGraphDef: metaBuf)

```

## Computer Vision Demo

A detailed example of Perfect TensorFlow for Computer Vision can be found in this repo: [Perfect TensorFlow Demo](#), where you can upload any local images or draw a scribble online to test if the server can recognize the picture content:





## API Guide

Perfect TensorFlow is a Swift Class Wrapper of TensorFlow's C API to build, save, load and execute TensorFlow models on Server Side Swift.

**⚠ WARNING ⚠**: The API is currently experimental and is not covered by TensorFlow API stability guarantees.

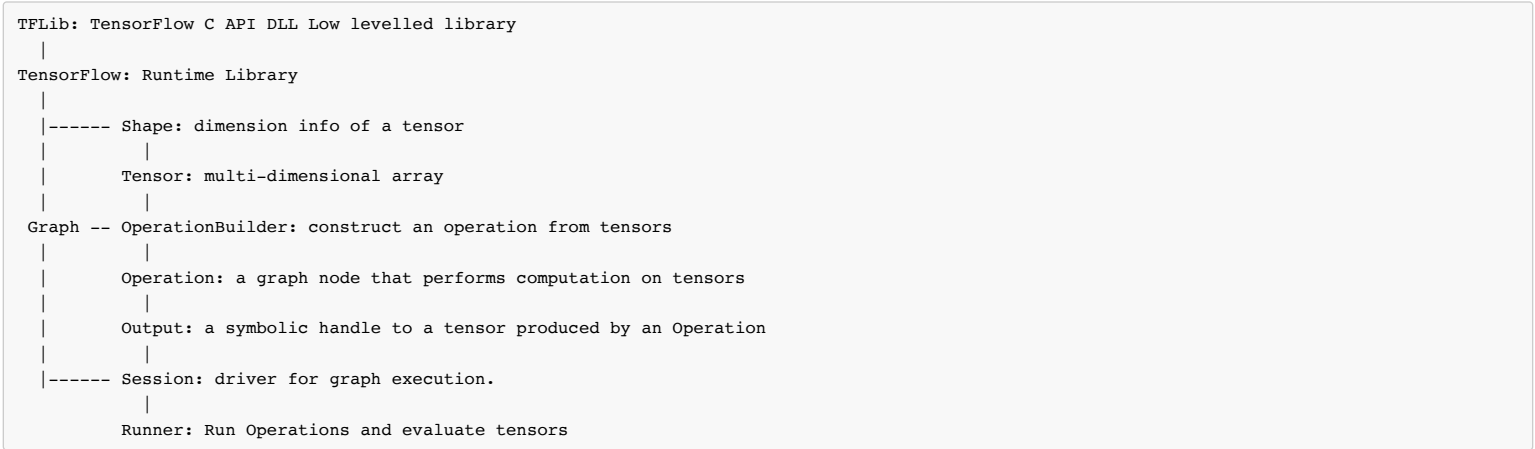
The [Perfect TensorFlow Computer Vision Server](#) example demonstrates use of this API to classify images using a pre-trained Inception architecture convolutional neural network. It demonstrates:

Graph construction: using the OperationBuilder class to construct a graph to decode, resize and normalize a JPEG image. Model loading: Using Graph.import() to load a pre-trained Inception model. Graph execution: Using a Session to execute the graphs and find the best label for an image.

Please [install Perfect TensorFlow library](#) before using.

## Classes

There are quite a few classes in Perfect-TensorFlow. Relationship between these classes can be described as the workflow below:



Content

| Class Name                       | Description                                                                                         |
|----------------------------------|-----------------------------------------------------------------------------------------------------|
| TFLib                            | TensorFlow C API low levelled DLL library - suggested for internal use only.                        |
| <a href="#">TensorFlow</a>       | namespace describing the TensorFlow runtime.                                                        |
| <a href="#">Shape</a>            | The possibly partially known shape of a tensor produced by an operation; create from an Int64 array |
| <a href="#">Tensor</a>           | A typed multi-dimensional array; can be created either from a scalar or a typed array.              |
| <a href="#">OperationBuilder</a> | A builder for Operations in a Graph.                                                                |
| <a href="#">Operation</a>        | A Graph node that performs computation on Tensors.                                                  |
| <a href="#">Output</a>           | A symbolic handle to a tensor produced by an Operation.                                             |
| <a href="#">Graph</a>            | A data flow graph representing a TensorFlow computation.                                            |
| <a href="#">Session Runner</a>   | Run Operations and evaluate Tensors.                                                                |

Protocol Buffers

Perfect TensorFlow depends heavily on Google Protocol Buffers, which means that you can easily save / load most of the objects from / into persistent files or database records in form of either binary bytes or JSON string.

Take Graph as a typical example, you can easily load a pre-trained model from a third party into your Swift program, as the digest source code in [Perfect TensorFlow Demo of Computer Vision](#):

```
import PerfectLib
import PerfectTensorFlow

typealias TF = TensorFlow

// load a third party graph protocol buffer file:
let fModel = File("/tmp/tensorflow_inception_graph.pb")
try fModel.open(.read)
let modelBytes = try fModel.readSomeBytes(count: fModel.size)
fModel.close()

// import buffer file into Graph Object:
let def = try TF.GraphDef(Data(bytes: modelBytes))
let g = try TF.Graph()
try g.import(definition: def)
```

⚠️NOTE⚠️ You can get the data buffer by `let data = try def.serializedData()` or `let json = try def.jsonString()` vice versa.

TensorFlow Runtime

Class TensorFlow is actually the namespace that represents the TensorFlow runtime.

To align with other language implementation such as Python or Java, make an alias of class TensorFlow is a good idea:

```
/// load the TensorFlow C API DLL library
import TensorFlowAPI
import PerfectTensorFlow
public typealias TF = TensorFlow
```

⚠️NOTE⚠️ Prior to use **ANY ACTUAL FUNCTIONS** of Perfect TensorFlow framework, `TF.Open()` must be called first:

```
// this action will load all api functions defined
// in /usr/local/lib/libtensorflow.so
try TF.Open()
```

Please also note that you can active the library with a specific path, alternatively, especially in case of different versions or CPU/GPU library adjustment required:

```
// this action will load the library with the path
try TF.Open("/path/to/DLL/of/libtensorflow.so")
```

## Shape

In Perfect TensorFlow definition, Shape is a struct that holding dimensions for a future tensor:

```
/// Tensor Shape Object
public struct Shape {
    public var dimensions = [Int64]()
} //end class
```

## Tensor

A Tensor is defined as a typed multi-dimensional array.

Generally you don't have to call the initiator of Tensor object directly, but using two static methods will be more efficient: `Scalar` for one dimensional constant and `Array` for a multi-dimensional array.

### Scalar

Scalar Tensor is actually a one dimensional array (or zero dimensional in an other term). To create such a tensor, just pass the variable to the static method `Scalar<T>(value: T)`:

```
// create an Int32 scalar tensor;
// Please note that Int will be Int64 actually
let x = try TF.Tensor.Scalar(Int32(100))

// create a Float scalar tensor;
// Please note that constant without Float type cast will be Double indeed
let y = try TF.Tensor.Scalar(Float(1.1))

// create a String scalar tensor:
let s = try TF.Tensor.Scalar("Hello, TensorFlow! 🇺🇸")
```

To get the value of Tensor as a scalar, use `asScalar<T>()` method:

```
let x: Int32 = try tensor.asScalar()
```

### Array

Array Tensor requires shape information to create; Also please note that the array has to be flatten to generate the expected array:

```
/* here is the matrix to input:
| 1 2 |
| 3 4 |
*/
// flatten the matrix first
let m:[Float] = [[1, 2], [3, 4]].flatMap { $0 }

// then turn it into a tensor with shape info.
let tensor = try TF.Tensor.Array(dimenisons: [2,2], value: m)
```

To get the value of Tensor as an array, use `asArray<T>()` method:

```
let y: [Float] = try tensor.asArray()
```

**⚠️NOTES⚠️** If the tensor is a string array, please directly call the `strings` property, which will actually return an array of `Data` :

```
// datastr is actually [Data]
let datastr = tensor.strings

// translate it into [String]
let str = datastr.map { $0.string }
```

## Matrix

Since Perfect-TensorFlow v1.2.1, however, you can apply a multi-dimensional array to a tensor as a matrix without considering the shape or dimensions. The equivalent example of above will look like:

```
let M = try TF.Tensor.Matrix([[1, 2], [3, 4]])
```

**⚠️NOTES⚠️** Element in a Matrix must be number!

## Raw Data of a Tensor

To access raw data of a tensor, you can use either `data` property, or `withDataPointer()` method with better performance but tricky pointer operations - and property of `bytesCount` and `type` will be useful in this case:

```
public class Tensor {
    /// get a buffer copy from the tensor value
    public var data: [Int8]

    /// get total size of memory in bytes
    public var bytesCount: Int

    /// check data type of the value / element of value array
    public var `type`: DataType?

    /// perform data pointer operations unsafely.
    public func withDataPointer<R>(body: (UnsafeMutableRawPointer) throws -> R) throws -> R
}
```

## Shape of a Tensor

To get the dimension of a tensor, use `dim` property:

```
public class Tensor {
    public var dim: [Int64]
}
```

## Operation and Its Builder

An operation is a graph node that performs computation on Tensors.

To create an operation in a graph, check the workflow below:

- export an `OperationBuilder` object instance from a graph
- set the operation name, type and device
- set the tensors of the operation
- add input / output
- set other attributes of the operation
- call `build()` method and return an expected operation.

Here is an example:

```

// make a tensor, e.g., Tensor.Scalar(100)
let tensor = try TF.Tensor ...

let g = try TF.Graph()

// export an instance from the graph and set the name / type
let operation = try g.opBuilder(name: "Const_0", type: `Const`)

    // set attributes
    .set(attributes: ["value": tensor, "dtype": tensor.type])

    // build the operation
    .build()

```

## OperationBuilder

OperationBuilder is defined as below:

```

public class OperationBuilder {
    /// constructor; using of graph.opBuilder() is more recommended.
    public init(graph: Graph, name: String, `type`: String) throws

    /// add an input to the operation
    public func add(input: Output) -> OperationBuilder

    /// add an input array to the operation
    public func add(inputs: [Output]) throws -> OperationBuilder

    /// add a control to the builder
    public func add(control: Operation) -> OperationBuilder

    /// build the operation
    public func build() throws -> Operation

    /// set the device
    public func `set`(device: String) -> OperationBuilder

    /// set attributes for the operation to build.
    /// parameter attributes: a dictionary of attributes of the operation,
    /// key for the attribute name.
    /// Valid attributes include Int64, [Int64], Float, [Float],
    /// Bool, [Bool], DataType, [DataType], String, [String],
    /// Shape, [Shape], Tensor, [Tensor],
    /// TensorProto, [TensorProto], Data
    public func `set`(attributes: [String: Any] = [:])

```

## Operation

Once built by a OperationBuilder, all preset attributes and values of an operation can be read by the following properties and methods below:

| Instance Interface                      | Type                      | Description                                             |
|-----------------------------------------|---------------------------|---------------------------------------------------------|
| fun attribute(forKey: String)           | <a href="#">AttrValue</a> | lookup an attribute in the current operation            |
| var NodeDefinition                      | <a href="#">NodeDef</a>   | get node definition                                     |
| var name                                | String                    | get operation name                                      |
| var type                                | String                    | get operation type                                      |
| var device                              | String                    | get operation device                                    |
| var numberOfInputs                      | Int                       | get number of inputs of the operation                   |
| var numberOfOutputs                     | Int                       | get number of outputs of the operation                  |
| func sizeOfInputList(argument: String)  | Int                       | get size of the input list array by an argument string  |
| func sizeOfOutputList(argument: String) | Int                       | get size of the output list array by an argument string |
| var controllInputs                      | [Operation]               | get control inputs as an operation array                |
| func asInput(_ index: Int)              | Input                     | generate an Input from the current operation            |
| func asOutput(_ index: Int)             | Output                    | generate an Output from the current operation           |

Besides, Operation Class also have three different static methods:

| Static Method                             | Description                    |
|-------------------------------------------|--------------------------------|
| func Consumers(output: Output) -> [Input] | get the consumers of an output |
| func TypeOf(input: Input) -> DataType?    | get type of the input          |
| func TypeOf(output: Output) -> DataType?  | get type of the output         |

⚠ **NOTES** ⚠ More convenient methods to create an operation from a tensor in a graph can be found in the [Chapter Graph](#)

## Output

Output is a symbolic handle using in TensorFlow and can be generate from Operation, you can simply treat an Output as "an Operation with an index number":

```
let output = op.asOutput(0)
```

## Graph

Graph is a data flow graph representing a TensorFlow computation. There are several different approaches to create a Graph object:

```
public class Graph {

    /// create a blank graph, just `let x = try TF.Graph()`
    public init () throws

    /// create a graph from a reference pointer,
    /// usually take place in array operation.
    public init(handle: OpaquePointer)

}
```

Besides, you can also import a pre-trained model into the graph, where `definition` is a [GraphDef Protobuf](#) which can be read / write from a binary file:

```
let g = try TF.Graph()
try g.import(definition: def)
```

### Operations in a Graph

You can get all operations in a graph by accessing property `operations`, or using `searchOperation(forName)` to search an operation by name:

```
// search an operation in a graph instance:
let placeholder = try graph.searchOperation(forName: "placeholder")

// retrieve all operations into an array:
let list = graph.operations
// list is [Operation] now.
```

## Common Operations

There is also a rich and growing set of methods for construct operations from different tensors in a graph, check the table below:

| Name          | Description                                                     | Example                                                                                                                                             |
|---------------|-----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| const         | create an operation from a constant tensor                      | <code>let x = try graph.const(tensor: t, name: "Const_0")</code>                                                                                    |
| placeholder   | create a placeholder operation                                  | <code>let feed = try graph.placeholder(name: "feed")</code>                                                                                         |
| binaryOp      | any binary operations between two outputs                       | <code>func binaryOp(_ type : String, _ in1: Output, _ in2: Output, name: String = "", index: Int = 0) throws -&gt; Output</code>                    |
| scalar        | create an Int32 scalar                                          | <code>let ten = try graph.scalar(10, name: "ten")</code>                                                                                            |
| scalar        | create an Float32 scalar                                        | <code>let point = try graph.scalar(0.1, name: "point")</code>                                                                                       |
| constantArray | create an array tensor and attach it to the graph               | <code>let size = try g.constantArray(name: "size", value: [1024,768])</code>                                                                        |
| add           | add two outputs                                                 | <code>let sum = try graph.add(left:lOutput, right:rOutput, name: "sum")</code>                                                                      |
| add           | add two operations (indices are automatically assuming to zero) | <code>let sum = try graph.add(left:lOp, right:rOp, name: "sumOp")</code>                                                                            |
| neg           | get the negative operation                                      | <code>let neg = try graph.neg(op, name: "MyNeg")</code>                                                                                             |
| matMul        | Matrix Multiply                                                 | <code>let m = try graph.matMul(l: aOp, r: bOp, name: "m0", transposeA:false, transposeB: true)</code> will perform an $A \times B^A$ multiplication |
| div           | y Output divided by x Output                                    | <code>let z = try graph.div(x: x, y: y, name: "Div0")</code>                                                                                        |
| sub           | y Output subtracted by x Output                                 | <code>let z = try graph.sub(x: x, y: y, name: "Div0")</code>                                                                                        |
| decodeJpeg    | decode a JPEG picture                                           | <code>let decoded = try graph.decodeJpeg(content: input, channels: 3)</code>                                                                        |

|                |                                        |                                                                             |
|----------------|----------------------------------------|-----------------------------------------------------------------------------|
| resizeBilinear | resize a bilinear image                | <code>let resizes = try g.resizeBilinear(images: images, size: size)</code> |
| cast           | cast an output to a specific data type | <code>let cast = try g.cast(value: jpeg, dtype: TF.DataType.dtFloat)</code> |
| expandDims     | expand dimensions                      | <code>let images = try g.expandDims(input: cast, dim: batch)</code>         |

A Full demo of almost all the express operation creations above can be found on the source code of [Perfect TensorFlow Computer Vision Demo](#)

⚠️ **NOTE** ⚠️ To load a graph from a saved model, use `graph.load()` method and will get a session object, see [Load A Session](#)

## Session

A session object represents a round of execution of all computations defined in a graph.

The best practice to create a session is using a chain of operations of `Runner` object. Here is an example:

```
let results = try graph.runner()
    .feed("input", tensor: image)
    .fetch("output")
    .run()
```

Let's break it down a bit. The first step is to get a session runner instance: `let r = try graph.runner()`.

Then you can feed some inputs into the runner:

```
class Runner {
    /// feed the session by using an output as input with a tensor
    public func feed(_ output: Output, tensor: Tensor) -> Runner

    /// feed the session by an operation with a tensor
    public func feed(_ operation: Operation, index: Int = 0, tensor: Tensor) -> Runner

    /// feed the session by an operation name with a tensor,
    /// in this case the graph will search for the operation by name first
    public func feed(_ operation: String, index: Int = 0, tensor: Tensor) throws -> Runner
}
```

Similar to `feed`, you can also indicate the session runner to fetch something out after the running:

```
class Runner {
    /// fetch the session to a specific output
    public func fetch(_ output: Output) -> Runner

    /// fetch the session to a specific operation.
    /// if index = 0, than it will be equivalent to fetch(output)
    public func fetch(_ operation: Operation, index: Int = 0) -> Runner

    /// fetch the session to a specific operation by its name
    public func fetch(_ operation: String, index: Int = 0) throws -> Runner
}
```

Then you can add target operations to the session as well:

```
class Runner {
    /// add an operation to the target
    public func addTarget(_ operation: Operation) -> Runner

    /// add an operation to the target by its name
    public func addTarget(_ operation: String) throws -> Runner
}
```

The final step of runner creation is to call the `run()` method and the result is an array of tensor:

```
class Runner {
    /// Execute the graph fragments necessary to compute all requested fetches.
    public func run() throws -> [Tensor]
}
```

Since you may want to feed / fetch / add targets to the session with more than one time, you can chain everything up like this:

```
let r = try graph
    .feed(op1, tensor: t1).feed(op2, tensor: t2)....
    .fetch(out1).fetch(out2).fetch(out3)....
    .addTarget("opA").addTarget("opB").addTarget("opC")...
    .run()
```

## Load A Session Runner

To import a session object from a previously saved model, firstly create a blank graph, then use `import` method to load it back:

```
let g = try TF.Graph()

// the meta signature info defined in a saved model
let metaBuf = try TF.Buffer()

// load the session runner
let runner = try g.load(
    exportDir: "/path/to/saved/model",
    tags: ["tag1", "tag2", ...],
    metaGraphDef: metaBuf)
```

In this case, you can check the meta data stored in this previously saved model, if available, for more information:

```
if let data = metaBuf.data {
    let meta = try TF.MetaGraphDef(serializedData: data)
    let signature_def = meta.signatureDef["some signatures ..."]
    let input_name = signature_def.inputs["name of inputs ..."]?.name
    let output_name = signature_def.outputs["name of outputs"]?.name
    ...
}
```

If ready, you can call `runner.run()` as the previous chapter.

## Device List of a Session

Since TensorFlow 1.3.0+, you can access device information by calling `session.devices` property, which will return a tuple array:

```
let dev = try g.runner().session.devices
print(dev)
// sample output:
// ["/job:localhost/replica:0/task:0/cpu:0": (type: "CPU", memory: 268435456)]
```# PerfectHadoop
```

This project provides a set of Swift classes which enable access to Hadoop servers.

This package builds with Swift Package Manager and is part of the [Perfect](https://github.com/PerfectlySoft/Perfect) project. It was written to be stand-alone and so does not require PerfectLib or any other components.

Ensure you have installed and activated the latest Swift 3.0 tool chain.

## Release Note

PerfectHadoop supports Hadoop 3.0.0 with a limitation on 2.7.3.

## Building

Add this project as a dependency in your Package.swift file.

```
``` swift
.Package(url:"https://github.com/PerfectlySoft/Perfect-Hadoop.git", majorVersion: 1)
```

Then please add the following line to the beginning part of swift sources: `swift import PerfectHadoop`

## Error Handle - `Exception`

In case of operation failure, an exception might be thrown out. In most cases of Perfect-Hadoop, the library would probably throw a `Exception` object. User can catch it and check a tuple `(url, header, body)` of the failure, as demo below:

```
do {
    // some Perfect Hadoop operations, including WebHDFS / MapReduce / YARN, all of them:
    ...
}
catch(Exception.unexpectedResponse(let (url, header, body))) {
    print("Exception: \"(url)\\n\\(header)\\n\\(body)\"")
}
catch (let err){
    print("Other Error:\\(err)")
}
```

## User Manual

- WebHDFS: [Perfect-HDFS](#)
- MapReduce:
  - [Perfect-MapReduce Application Master API](#) ⚠ Experimental ⚠
  - [Perfect-MapReduce History Server API](#)
- YARN:
  - [Perfect-YARN Node Manager](#)
  - [Perfect-YARN Resource Manager](#)

## PerfectHadoop: WebHDFS

This project provides a Swift wrapper of WebHDFS API

## Quick Start

### Connect to Hadoop

To connect to your HDFS server by WebHDFS, initialize a WebHDFS object with sufficient parameters:

```
// this connection could possibly do some basic operations
let hdfs = WebHDFS(host: "hdfs.somedomain.com", port: 9870)
```

or connect to Hadoop with a valid user name:

```
// add user name to do more operations such as modification of file or directory
let hdfs = WebHDFS(host: "hdfs.somedomain.com", port: 9870, user: "username")
```

### Authentication

If using Kerberos to authenticate, please try codes below:

```
// set auth to kerberos
let hdfs = WebHDFS(host: "hdfs.somedomain.com", port: 9870, user: "username", auth: .krb5)
```

### Parameters of WebHDFS Object

- `service` :String, the service protocol of web request - http / https / webhdfs / hdfs
- `host` :String, the hostname or ip address of the webhdfs host
- `port` :Int, the port of webhdfs host, default is 9870
- `auth` : Authorization Model, .off or .krb5. Default value is .off
- `proxyUser` :String, proxy user, if applicable
- `apibase` :String, use this parameter *ONLY* the target server has a different api routine other than /webhdfs/v1
- `timeout` :Int, timeout in seconds, zero means never timeout during transfer

## Get Home Directory

Call `getHomeDirectory()` to get the home directory for current user.

```
let home = try hdfs.getHomeDirectory()
print("the home is \" + home + ")
```

## Get File Status

`getFileStatus()` will return a `FileStatus` structure with properties below:

### Properties of `FileStatus` Structure

- `accessTime` : Int, unix time for last access
- `pathSuffix` : String, file suffix / extension - type
- `replication` : Int, replicated nodes count
- `type` : String, node type: directory or file
- `blockSize` : Int, storage unit, default = 128M, min = 1M
- `owner` : String, user name of the node owner
- `modificationTime` : Int, last modification in unix epoch time format
- `group` : String, group name of the node
- `permission` : Int, node permission, (u)rwx (g)rwx (o)rwx
- `length` : Int, file length

To get status info from a file or a directory, call `getFileStatus()` as example below:

```
let fs = try hdfs.getFileStatus(path: "/")
if fs.length > 0 {
    ...
}
```

## List Status

Method `listStatus()` will return an array of `[FileStatus]`, i.e., a list of all files with status under a specific directory. For example,

```
let list = try hdfs.listStatus(path: "/")
for file in list {
    // print the ownership of a file in the list
    print(file.owner)
}
```

The structure of item listed is the same with `getFileStatus()`.

## Create Directory

Basic HDFS directory operations include `mkdir` and `delete`. To create a new directory named `/demo` with a permission 754, i.e., `rwxr-xr--` (read/write/execute for user, read/execute for group and read only for others), try the line of code below:

```
try hdfs.mkdir(path: "/demo", permission: 754)
```

## Summary of Directory

WebHDFS provides a `getDirectoryContentSummary()` method to developers and will return detail info as defined below:

### Properties of `ContentSummary` Structure

- `directoryCount` : Int, how many sub folders does this node have
- `fileCount` : Int, file count of the node
- `length` : Int, length of the node
- `quota` : Int, quota of the node
- `spaceConsumed` : Int, blocks that node consumed
- `spaceQuota` : Int, block quota
- `typeQuota` : Three Quota Structures, with two properties of each: `consumed` and `quota`, both properties are integers:
  - `ARCHIVE` : Quota, quota info about data stored in archived files

- `DISK` : Quota, quota info about data stored in hard disk
- `SSD` : Quota, quota info about data stored in SSD

To get this summary, call `getDirectoryContentSummary()` with path info:

```
let sum = try hdfs.getDirectoryContentSummary(path: "/")
print(sum.length)
print(sum.spaceConsumed)
print(sum.typeQuota.SSD.consumed)
print(sum.typeQuota.SSD.quota)
print(sum.typeQuota.DISK.consumed)
print(sum.typeQuota.DISK.quota)
print(sum.typeQuota.ARCHIVE.consumed)
print(sum.typeQuota.ARCHIVE.quota)
...
```

## Checksum

Checksum method `getFileChecksum()` helps user check integrity of file by three properties of `FileChecksum` Structure:

### Property of `FileChecksum` Structure

- `algorithm` : String, algorithm information of this checksum
- `bytes` : String, checksum string result
- `length` : Int, length of the string

Here is an example of checksum:

```
let checksum = try hdfs.getFileChecksum(path: "/book/chickenrun.txt")
// checksum is a struct:
// algorithm information of this checksum
print(checksum.algorithm)
// checksum string
print(checksum.bytes)
// string length
print(checksum.length)
```

## Delete

To delete a directory or a file, simply call `delete()` . If the object to remove is a directory, users can also apply another parameter of `recursive` . If set to true, the directory will be removed with all sub folders.

```
// remove a file
try hdfs.delete(path "/demo/boo.txt")

// remove a directory, recursively
try hdfs.delete(path:"/demo", recursive: true)
```

## Upload

To upload a file, call `create()` method, with two parameters essentially, i.e., local file to upload and the expected remote file path, as below:

```
try hdfs.create(path: "/destination", localFile: "/tmp/afile.txt")
```

Considering it is a time consuming operation, please consider to call this function in a threading way practically.

### Parameters

Parameters of `create()` include:

- `path` :String, full path of the remote file / directory.
- `localFile` :String, full path of file to upload
- `overwrite` :Bool, If a file already exists, should it be overwritten?
- `permission` : Int, unix style file permission (u)rxw (g)rxw (o)rxw. Default is 755, i.e., rwxr-xr-x
- `blocksize` :Int, size of per block unit. default 128M, min = 1M
- `replication` :Int, The number of replications of a file.
- `buffersize` : The size of the buffer used in transferring data.

## Symbol Link

The same as Unix system, HDFS provides a method called `createSymLink` to create a symbolic link to another file or directory:

```
try hdfs.createSymLink(path: "/book/longname.txt", destination:"/my/recent/quick.lnk", createParent: true)
```

Please note that there is a parameter called `createParent`, which means if there is no such a path, the system will automatically create a full path as demand, i.e, if there is no such a path of "recent" under folder of "my", then it will be automatically created.

## Download

To download a file, call `openFile()` method as below:

```
let bytes = try hdfs.openFile(path: "/books/bedtimestory.txt")
print(bytes.count)
```

In this example, the content of "bedtimestory.txt" will be save to an binary byte array called `bytes`

Considering it is a time consuming operation, please consider to call this function in a threading way practically. In this case, please also consider to call `openFile()` for severral times to get the downloading process, as indicated by the parameters below, which means you can download the file by pieces, and if something wrong, you can also re-download the failure parts:

### Parameters

- `path` :String, full path of the remote file / directory.
- `offset` :Int, The starting byte position.
- `length` :Int, The number of bytes to be processed.
- `bufferSize` :Int, The size of the buffer used in transferring data.

## Append

Append operation is similar to `create`, instead of overwriting, it will append the local file content to the end of the remote file:

```
try hdfs.append(path: "/remoteFile.txt", localFile: "/tmp/b.txt")
```

### Parameters

- `path` :String, full path of the remote file / directory.
- `localFile` :String, full path of file to upload
- `bufferSize` :Int, The size of the buffer used in transferring data.

## Merge Files

HDFS allows user to concat two or more files into one, for example:

```
try hdfs.concat(path:"/tmp/1.txt", sources:["/tmp/2.txt", "/tmp/3.txt"])
```

Then file 2.txt and 3.txt will all append to 1.txt

## Truncate

File on an HDFS could be truncated into expected length as below:

```
try hdfs.truncate(path: "/books/LordOfRings.txt", newlength: 1024)
```

The above example will trim the file into 1k.

## Set Permission

HDFS file permission can be set by method of `setPermission`. The example below demonstrates how to set "/demo" directory with a permission of 754, i.e., rwxr-xr-- (read/write/execute for user, read/execute for group and read only for others):

```
try hdfs.setPermission(path: "/demo", permission: 754)
```

## Set Owner

Ownership of a file or a directory can be transferred by a method called `setOwner` :

```
try hdfs.setOwner(path: "/book/chickenrun.html", name:"NewOwnerName", group: "NewGroupName")
```

## Set Replication

Files on HDFS system can be replicated on more than one node. Use `setReplication` do this job:

```
try hdfs.setReplication(path: "/book/twins.txt", factor: 2)
// if success, twins.txt will have two replications
```

## Access & Modification Time

HDFS accepts changing the access or modification time info of a file. The time is in Epoch / Unix timestamp format. The example below shows a similar operation of unix command `touch` :

```
let now = time(nil)
try hdfs.setTime(path: "/tmp/touchable.txt", modification: now, access: now)
// if success, the time info of the file will be updated.
```

## Access Control List

Access control list of HDFS file system can be operated by the following methods:

- `getACL` : retrieve the ACL info
- `setACL` : set the ACL info
- `modifyACL` : modify the ACL entries
- `removeACL` : remove one or more ACL entries, or remove all entries by default.

The `getACL()` method will return an `AclStatus` structure, with properties below:

- `entries` : [String], an array of ACL entry strings.
- `owner` : String, the user who is the owner
- `group` : String, the group owner
- `permission` : Int, permission code in unix style
- `stickyBit` : Bool, true if the sticky bit is on

The following example demonstrates all basic ACL operations:

```
let hdfs = WebHDFS(auth:.byUser(name: defaultUserName))
let remoteFile = "/acl.txt"
do {
    // get access control list
    var acl = try hdfs.getACL(path: remoteFile)
    print("group info: \(acl.group)")
    print("owner info: \(acl.owner)")
    print("entry info: \(acl.entries)")
    print("permission info: \(acl.permission)")
    print("stickyBit info: \(acl.stickyBit)")

    try hdfs.setACL(path: remoteFile, specification: "user::rw-,user:hadoop:rw-,group::r--,other::r--")

    try hdfs.modifyACL(path: remoteFile, entries: "user::rwx,user:hadoop:rwx,group::rwx,other::---")

    try hdfs.removeACL(path: remoteFile, defaultACL: false)

    try hdfs.removeACL(path: remoteFile)

    try hdfs.removeACL(path: remoteFile, entries: "", defaultACL: false)
```

## Check Access

Method `checkAccess()` is for checking whether a specific action is accessible or not. Typical Usage of this method is:

```
let b = try hdfs.checkAccess(path: "/", fsaction: "mkdir")
// true value means user can perform mkdir() on the root folder
if b {
    print("mkdir: Access Granted")
} else {
    print("mkdir: Access Denied")
}
```

## Extension Attributes

Besides the traditional file attributes, HDFS also provides an extension method for more customizable attributes, which is named as XAttr. XAttr operations include:

- `setXAttr`: set the attributes
- `getXAttr`: get one or more attributes' value
- `listXAttr`: list all attributes
- `removeXAttr`: remove one or more attributes.

Besides, there are two flags for XAttr operations: `CREATE` and `REPLACE`. The default flag is `CREATE` when setting an XAttr.

```
public enum XAttrFlag:String {
    case CREATE = "CREATE"
    case REPLACE = "REPLACE"
}
```

Please check the code below:

```
let remoteFile = "/book/a.txt"

try hdfs.setXAttr(path: remoteFile, name: "user.color", value: "red")
// if success, an attribute called 'user.color' with a value of 'red' will be added to the file 'a.txt'

try hdfs.setXAttr(path: remoteFile, name: "user.size", value: "small")
// if success, an attribute called 'user.size' with a value of 'small' will be added to the file 'a.txt'

try hdfs.setXAttr(path: remoteFile, name: "user.build", value: "2016")
// if success, an attribute called 'user.build' with a value of '2016' will be added to the file 'a.txt'

try hdfs.setXAttr(path: remoteFile, name: "user.build", value: "2017", flag:.REPLACE)
// please note the flag of REPLACE. if true, an attribute called 'user.build' will be replaced with the new value of 2017 from 2016

// list all attributes
let list = try hdfs.listXAttr(path: remoteFile)
list.forEach {
    item in
    print(item)
}

// retrieve specific attributes
var a = try hdfs.getXAttr(path: remoteFile, name: ["user.color", "user.size", "user.build"])
// print the attributes with value
a.forEach{
    x in
    print("\(x.name) => \(x.value)")
}

try hdfs.removeXAttr(path: remoteFile, name: "user.size")
// if success, the attribute of user.size will be removed
```

## Snapshots

HDFS provides snapshots functions for directories.

- `CreateSnapshot()`

If success, function `createSnapshot()` will return a tuple `(longname, shortname)`. The long name is the full path of the snapshot, and the short name is the snapshot's own name. Check the codes below:

```
let (fullpath, shortname) = try hdfs.createSnapshot(path: "/mydata")
print(fullpath)
print(shortname)
```

- `renameSnapshot()` This function can rename the snapshot from its short name to a new one:

```
try hdfs.renameSnapshot(path: "/mydata", from: shortname, to: "snapshotNewName")
```

- `deleteSnapshot()`

Once having the short name of snapshot, `deleteSnapshot()` can be used to delete the snapshot:

```
try hdfs.deleteSnapshot(path: dir, name: shortname)
```

## PerfectHadoop: MapReduce Application

This project provides a Swift wrapper of MapReduce Application Master REST API:

- `MapReduceApplication()` : access a specific map / reduce application running on current server.

## RELEASE NOTE

⚠ This is an experimental module and is subject to change in future. ⚠

## Connect to Hadoop Map Reduce Application

To connect to a current active Hadoop Map / Reduce Application by Perfect, initialize a `MapReduceApplication()` object with an *APPLICATION ID*, and other essential parameters:

```
// this connection could possibly do some basic operations
let app = MapReduceApplication(applicationId: "application_12345678_xxxxxx", host: "mapReducer.somedomain.com")
```

you can also

```
// add user name if need
let app = MapReduceApplication(applicationId: "application_12345678_xxxxxx", host: "mapReducer.somedomain.com", user: "your user name")
```

### Authentication

If using Kerberos to authenticate, please try codes below:

```
// set auth to kerberos
let app = MapReduceApplication(applicationId: "application_12345678_xxxxxx", host: "mapReducer.somedomain.com", user: "your user name", auth: .krb5)
```

### Parameters of MapReduceApplication Object

| Item          | Data Type     | Description                                                                                                           |
|---------------|---------------|-----------------------------------------------------------------------------------------------------------------------|
| applicationId | String        | the id string of the application to control. <i>Required</i>                                                          |
| service       | String        | the service protocol of web request - http / https                                                                    |
| host          | String        | the hostname or ip address of the Hadoop Map Reduce Application Master host                                           |
| port          | Int           | the port of map reduce application master host, default is 8088                                                       |
| auth          | Authorization | .off or .krb5. Default value is .off                                                                                  |
| proxyUser     | String        | proxy user, if applicable                                                                                             |
| apibase       | String        | use this parameter <i>ONLY</i> the target server has a different api routine other than <code>/ws/v1/mapreduce</code> |
| timeout       | Int           | timeout in seconds, zero means never timeout during transfer                                                          |

## Get General Information

Call `checkInfo()` to get the general information of a Hadoop MapReduce Application Master in form of a `MapReduceApplication.Info` structure:

```
guard let inf = try app.checkInfo() else {
    // something goes wrong here
}
print(inf.startedOn)
print(inf.hadoopVersion)
print(inf.hadoopBuildVersion)
print(inf.hadoopVersionBuiltOn)
```

### Members of `MapReduceApplication.Info`

| Item        | Data Type | Description                                           |
|-------------|-----------|-------------------------------------------------------|
| appld       | Int       | The application id                                    |
| startedOn   | Int       | The time the application started (in ms since epoch)  |
| name        | string    | The name of the application                           |
| user        | string    | The user name of the user who started the application |
| elapsedTime | long      | The time since the application was started (in ms)    |

## MapReduce Jobs

Call `checkJobs()` to return an array of `Job` structure. The jobs resource provides a list of the MapReduce jobs that have finished. It does not currently return a full list of parameters. The simplest form is `checkJobs()`, which will return all jobs available:

```
let jobs = try app.checkJobs()
jobs.forEach { j in
    print(j.id)
    print(j.name)
    print(j.queue)
    print(j.state)
}
```

If succeeded, `checkJobs()` would probably return a job object as described below:

### Data Structure of Job

| Item                     | Data Type | Description                                                                                         |
|--------------------------|-----------|-----------------------------------------------------------------------------------------------------|
| id                       | String    | The job id                                                                                          |
| name                     | String    | The job name                                                                                        |
| queue                    | String    | The queue the job was submitted to                                                                  |
| user                     | String    | The user name                                                                                       |
| state                    | String    | the job state - valid values are: NEW, INITED, RUNNING, SUCCEEDED, FAILED, KILL_WAIT, KILLED, ERROR |
| startTime                | Int       | The time the job started (in ms since epoch)                                                        |
| finishTime               | Int       | The time the job finished (in ms since epoch)                                                       |
| elapsedTime              | Int       | The elapsed time since job started (in ms)                                                          |
| mapsTotal                | Int       | The total number of maps                                                                            |
| mapsCompleted            | Int       | The number of completed maps                                                                        |
| reducesTotal             | Int       | The total number of reduces                                                                         |
| reducesCompleted         | Int       | The number of completed reduces                                                                     |
| diagnostics              | String    | A diagnostic message                                                                                |
| uberized                 | Bool      | Indicates if the job was an uber job - ran completely in the application master                     |
| mapsPending              | Int       | The number of maps still to be run                                                                  |
| mapsRunning              | Int       | The number of running maps                                                                          |
| reducesPending           | Int       | The number of reduces still to be run                                                               |
| reducesRunning           | Int       | The number of running reduces                                                                       |
| failedReduceAttempts     | Int       | The number of failed reduce attempts                                                                |
| killedReduceAttempts     | Int       | The number of killed reduce attempts                                                                |
| successfulReduceAttempts | Int       | The number of successful reduce attempts                                                            |
| failedMapAttempts        | Int       | The number of failed map attempts                                                                   |
| killedMapAttempts        | Int       | The number of killed map attempts                                                                   |
| successfulMapAttempts    | Int       | The number of successful map attempts                                                               |
| acls                     | [ACL]     | A collection of acls objects                                                                        |

**Elements of the acls object**

| Item  | Data Type | Description   |
|-------|-----------|---------------|
| value | String    | The acl value |
| name  | String    | The acl name  |

## Check a Specific Job

It is also possible to check a specific job by a job id:

```
let job = try app.checkJob(jobId: "job_1484231633049_0005")
```

See [Data Structure of Job](#) as above.

## JobAttempt of Job

When you make a request for the list of job attempts, the information will be returned as an array of job attempt objects.

```
guard let attempts = try app.checkJobAttempts(jobId: "job_1484231633049_0005") else {
    // something goes wrong
}
attempts.forEach { attempt in
    print(attempt.id)
    print(attempt.containerId)
    print(attempt.nodeHttpAddress)
    print(attempt.nodeId)
    print(attempt.startTime)
}
```

JobAttempt Structure:

| Item            | Data Type | Description                                          |
|-----------------|-----------|------------------------------------------------------|
| id              | String    | The job attempt id                                   |
| nodeId          | String    | The node id of the node the attempt ran on           |
| nodeHttpAddress | String    | The node http address of the node the attempt ran on |
| logsLink        | String    | The http link to the job attempt logs                |
| containerId     | String    | The id of the container for the job attempt          |
| startTime       | Long      | The start time of the attempt (in ms since epoch)    |

## Counters of Job

With the job counters API, you can object a collection of resources that represent al the counters for that job.

```
guard let js = try app.checkJobCounters(jobId: "job_1484231633049_0005") else {
    // something wrong
}
js.counterGroup.forEach{ group in
    print(group.counterGroupName)
    group.counters.forEach { counter in
        print(counter.name)
        print(counter.mapCounterValue)
        print(counter.reduceCounterValue)
        print(counter.totalCounterValue)
    }
}
}
```

JobCounter Object

| Item         | Data Type      | Description                       |
|--------------|----------------|-----------------------------------|
| id           | String         | The job id                        |
| counterGroup | [CounterGroup] | An array of counter group objects |

CounterGroup Object

| Item             | Data Type | Description                   |
|------------------|-----------|-------------------------------|
| counterGroupName | string    | The name of the counter group |
| counter          | [Counter] | An array of counter objects   |

Counter Object

| Item               | Data Type | Description                       |
|--------------------|-----------|-----------------------------------|
| name               | String    | The name of the counter           |
| reduceCounterValue | Int       | The counter value of reduce tasks |
| mapCounterValue    | Int       | The counter value of map tasks    |
| totalCounterValue  | Int       | The counter value of all tasks    |

## Check Configuration of Job

Use `checkJobConfig()` to check configuration of a specific job:

```
gurard let config = try app.checkJobConfig(jobId: "job_1484231633049_0005") else {
    /// something wrong
}
// print the configuration path
print(config.path)
// check properties of configuration file
for p in config.property {
    print(p.name)
    print(p.value)
    print(p.source)
}
```

### JobConfig Object

A job configuration resource contains information about the job configuration for this job.

| Item     | Data Type  | Description                            |
|----------|------------|----------------------------------------|
| path     | String     | The path to the job configuration file |
| property | [Property] | an array of property object            |

### Property Object

| Item   | Data Type | Description                                                                                                                                              |
|--------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| name   | String    | The name of the configuration property                                                                                                                   |
| value  | String    | The value of the configuration property                                                                                                                  |
| source | String    | The location this configuration object came from. If there is more then one of these it shows the history with the latest source at the end of the list. |

## Tasks of a Job

Use `checkJobTasks()` to collect task information from a specific job:

```
// get all tasks information from a specific job
let tasks = try app.checkJobTasks(jobId: "job_1484231633049_0005")

// print properties of each task
for t in tasks {
    print(t.progress)
    print(t.elapsedTime)
    print(t.state)
    print(t.startTime)
    print(t.id)
    print(t.type)
    print(t.successfulAttempt)
    print(t.finishTime)
} //next t
```

### Parameters of checkJobTasks()

Calling `checkJobTasks()` requires a jobId all the time, however, you can also specify an extra parameter to indicate what type of tasks shall be listed in the query results:

- jobId: a string represents the id of the job
- taskType: optional, could be `.MAP` or `.REDUCE`

### JobTask Object

The `checkJobTasks()` will return an array of `JobTask` data structure, as described below:

| Item              | Data Type | Description                                                                                            |
|-------------------|-----------|--------------------------------------------------------------------------------------------------------|
| id                | string    | The task id                                                                                            |
| state             | String    | The state of the task - valid values are: NEW, SCHEDULED, RUNNING, SUCCEDED, FAILED, KILL_WAIT, KILLED |
| type              | String    | The task type - MAP or REDUCE                                                                          |
| successfulAttempt | String    | The id of the last successful attempt                                                                  |
| progress          | Double    | The progress of the task as a percent                                                                  |
| startTime         | Int       | The time in which the task started (in ms since epoch) or -1 if it was never started                   |
| finishTime        | Int       | The time in which the task finished (in ms since epoch)                                                |
| elapsedTime       | Int       | The elapsed time since the application started (in ms)                                                 |

### Query a Specific JobTask

If both `jobId` and `jobTaskId` are available, you can use `checkJobTask()` to perform a query on a specific job task:

```
guard let task = try app.checkJobTask(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0") else {
    // something wrong
}
```

```
print(task.progress)
```

The return value of `checkJobTask` will be a [JobTask](#) object, as described above.

### Task Counters of Job

Method `checkJobTaskCounters()` can return counters of a specific task, as the example below:

```
guard let js = try app.checkJobTaskCounters(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0") else {
    // something wrong
}
```

```
// print the id of the job task counters
print(js.id)
// check out each counter group
js.taskCounterGroup.forEach{ group in
    print(group.counterGroupName)
    // print counters in each group.
    group.counters.forEach { counter in
        print(counter.name)
        print(counter.value)
    }
}
```

### JobTaskCounter Object

| Item             | Data Type      | Description                                                         |
|------------------|----------------|---------------------------------------------------------------------|
| id               | String         | The job id                                                          |
| taskCounterGroup | [CounterGroup] | An array of counter group objects, See <a href="#">CounterGroup</a> |

### Task Attempts

Function `checkJobTaskAttempts()` can return attempts from a specific job task.

```
let jobTaskAttempts = try app.checkJobTaskAttempts(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0")
for attempt in jobTaskAttempts {
    print(attempt.id)
    print(attempt.diagnostics)
    print(attempt.assignedContainerId)
    print(attempt.rack)
    print(attempt.state)
    print(attempt.progress)
} //next
```

Once done, the `checkJobTaskAttempts` will return an array of `TaskAttempt` object, as described below:

TaskAttempt Object

Elements of the TaskAttempt object:

| Item                | Data Type         | Description                                                                                                                                                                                                                            |
|---------------------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| id                  | String            | The task id                                                                                                                                                                                                                            |
| rack                | String            | The rack                                                                                                                                                                                                                               |
| state               | TaskAttempt.State | The state of the task attempt - valid values are: NEW, UNASSIGNED, ASSIGNED, RUNNING, COMMITPENDING, SUCCESSCONTAINERCLEANUP, SUCCEEDED, FAILCONTAINERCLEANUP, FAILTASKCLEANUP, FAILED, KILLCONTAINERCLEANUP, KILLTASK_CLEANUP, KILLED |
| type                | TaskType          | The type of task - MAP or REDUCE                                                                                                                                                                                                       |
| assignedContainerId | String            | The container id this attempt is assigned to                                                                                                                                                                                           |
| nodeHttpAddress     | String            | The http address of the node this task attempt ran on                                                                                                                                                                                  |
| diagnostics         | String            | A diagnostics message                                                                                                                                                                                                                  |
| progress            | Double            | The progress of the task attempt as a percent                                                                                                                                                                                          |
| startTime           | Int               | The time in which the task attempt started (in ms since epoch)                                                                                                                                                                         |
| finishTime          | Int               | The time in which the task attempt finished (in ms since epoch)                                                                                                                                                                        |
| elapsedTime         | Int               | The elapsed time since the task attempt started (in ms)                                                                                                                                                                                |

Besides, For reduce task attempts you also have the following fields:

| Item               | Data Type | Description                                                                                                  |
|--------------------|-----------|--------------------------------------------------------------------------------------------------------------|
| shuffleFinishTime  | Int       | The time at which shuffle finished (in ms since epoch)                                                       |
| mergeFinishTime    | Int       | The time at which merge finished (in ms since epoch)                                                         |
| elapsedShuffleTime | Int       | The time it took for the shuffle phase to complete (time in ms between reduce task start and shuffle finish) |
| elapsedMergeTime   | Int       | The time it took for the merge phase to complete (time in ms between the shuffle finish and merge finish)    |
| elapsedReduceTime  | Int       | The time it took for the reduce phase to complete (time in ms between merge finish to end of reduce task)    |

Query a Specific TaskAttempt

If a taskAttempt id is available, you can also check the specific one by `checkJobTaskAttempt()` , as demo below:

```
guard let jobTaskAttempts = try app.checkJobTaskAttempt(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0", "attempt_1326381300833_2_2_m_0_0") else {
    // something wrong
    print("there is no such a task attempt")
    return
} //end guard
print(attempt.id)
print(attempt.diagnostics)
print(attempt.assignedContainerId)
print(attempt.rack)
print(attempt.state)
print(attempt.progress)
```

The `checkJobTaskAttempt()` will return such an [TaskAttempt object](#) as described above.

## Attempt State Control

Currently Hadoop MapReduce Application Master has an experimental attempt state control.

### Check Attempt

To check a running attempt state, use `checkJobTaskAttemptState()` :

```
swift guard let state = try checkJobTaskAttemptState(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0", "attempt_1326381300833_2_2_m_0_0")
```

### Kill Attempt

To terminate an attempt, call `killTaskAttempt()` :

```
try killTaskAttempt(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0", "attempt_1326381300833_2_2_m_0_0")
```

## Attempt Counters of Task

Method `checkJobTaskAttemptCounters()` can return counters of a specific task attempt, as the example below:

```
guard let counters = try app.checkJobTaskAttemptCounters(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0", "attempt_1326381300833_2_2_m_0_0") else {
    // something wrong
    print("Task Attempt Counters Acquiring failed")
    return
}
// print the counters' id
print(counters.id)
// enumerate all groups
for group in counters.taskAttemptCounterGroup {
    // print group name
    print(group.counterGroupName)
    // enumerate all counters in this group
    for counter in group.counters {
        print(counter.name)
        print(counter.value)
    } //next counter
} //next group
}
```

### JobTaskAttemptCounters Object

| Item                    | Data Type      | Description                                                         |
|-------------------------|----------------|---------------------------------------------------------------------|
| id                      | String         | The job id                                                          |
| taskAttemptcounterGroup | [CounterGroup] | An array of counter group objects, See <a href="#">CounterGroup</a> |

## PerfectHadoop: MapReduce History Server

This project provides a Swift wrapper of MapReduce History Server REST API:

- `MapReduceHistory()` : access to all applications on map / reduce history server.

## Connect to Hadoop Map Reduce History Server

To connect to your Hadoop Map / Reduce History server by Perfect, initialize a `MapReduceHistory()` object with sufficient parameters:

```
// this connection could possibly do some basic operations
let history = MapReduceHistory(host: "mapReduceHistory.somedomain.com", port: 19888)
```

or connect to Hadoop Map / Reduce History server with a valid user name:

```
// add user name if need
let history = MapReduceHistory(host: "mapReduceHistory.somedomain.com", port: 19888, user: "your user name")
```

### Authentication

If using Kerberos to authenticate, please try codes below:

```
// set auth to kerberos
let history = MapReduceHistory(host: "mapReduceHistory.somedomain.com", port: 19888, user: "username", auth: .krb5)
```

### Parameters of MapReduceHistory Object

| Item      | Data Type     | Description                                                                                                         |
|-----------|---------------|---------------------------------------------------------------------------------------------------------------------|
| service   | String        | the service protocol of web request - http / https                                                                  |
| host      | String        | the hostname or ip address of the Hadoop Map Reduce history server host                                             |
| port      | Int           | the port of host, default is 19888                                                                                  |
| auth      | Authorization | .off or .krb5. Default value is .off                                                                                |
| proxyUser | String        | proxy user, if applicable                                                                                           |
| apibase   | String        | use this parameter <i>ONLY</i> the target server has a different api routine other than <code>/ws/v1/history</code> |
| timeout   | Int           | timeout in seconds, zero means never timeout during transfer                                                        |

## Get General Information

Call `checkInfo()` to get the general information of a Hadoop MapReduce History Server in form of a `MapReduceHistory.Info` structure:

```
guard let inf = try history.checkInfo() else {
    // something goes wrong here
}
print(inf.startedOn)
print(inf.hadoopVersion)
print(inf.hadoopBuildVersion)
print(inf.hadoopVersionBuiltOn)
```

### Members of `MapReduceHistory.Info`

| Item                 | Data Type | Description                                                       |
|----------------------|-----------|-------------------------------------------------------------------|
| startedOn            | Int       | The time the history server was started (in ms since epoch)       |
| hadoopVersion        | String    | Version of hadoop common                                          |
| hadoopBuildVersion   | String    | Hadoop common build string with build version, user, and checksum |
| hadoopVersionBuiltOn | String    | Timestamp when hadoop common was built                            |

## Historical MapReduce Jobs

Call `checkJobs()` to return an array of `Job` structure. The jobs resource provides a list of the MapReduce jobs that have finished. It does not currently return a full list of

parameters. The simplest form is `checkJobs()` , which will return all jobs available:

```
let jobs = try history.checkJobs()  
// or you can narrow the query result by set some parameters:  
// let jobs = try his.checkJobs(state: .SUCCEEDED, queue: "default", limit: 10)  
jobs.forEach { j in  
    print(j.id)  
    print(j.name)  
    print(j.queue)  
    print(j.state)  
}
```

Parameters of checkJobs()

| Item              | Data Type       | Description                                                                 |
|-------------------|-----------------|-----------------------------------------------------------------------------|
| user              | String          | user name                                                                   |
| state             | APP.FinalStatus | the job state, i.e, UNDEFINED, SUCCEEDED, FAILED and KILLED                 |
| queue             | String          | queue name                                                                  |
| limit             | Int             | total number of app objects to be returned                                  |
| startedTimeBegin  | Int             | jobs with start time beginning with this time, specified in ms since epoch  |
| startedTimeEnd    | Int             | jobs with start time ending with this time, specified in ms since epoch     |
| finishedTimeBegin | Int             | jobs with finish time beginning with this time, specified in ms since epoch |
| finishedTimeEnd   | Int             | jobs with finish time ending with this time, specified in ms since epoch    |

Data Structure of Job

| Item                     | Data Type | Description                                                                                         |
|--------------------------|-----------|-----------------------------------------------------------------------------------------------------|
| id                       | String    | The job id                                                                                          |
| name                     | String    | The job name                                                                                        |
| queue                    | String    | The queue the job was submitted to                                                                  |
| user                     | String    | The user name                                                                                       |
| state                    | String    | the job state - valid values are: NEW, INITED, RUNNING, SUCCEEDED, FAILED, KILL_WAIT, KILLED, ERROR |
| diagnostics              | String    | A diagnostic message                                                                                |
| submitTime               | Int       | The time the job submitted (in ms since epoch)                                                      |
| startTime                | Int       | The time the job started (in ms since epoch)                                                        |
| finishTime               | Int       | The time the job finished (in ms since epoch)                                                       |
| mapsTotal                | Int       | The total number of maps                                                                            |
| mapsCompleted            | Int       | The number of completed maps                                                                        |
| reducesTotal             | Int       | The total number of reduces                                                                         |
| reducesCompleted         | Int       | The number of completed reduces                                                                     |
| uberized                 | Boolean   | Indicates if the job was an uber job - ran completely in the application master                     |
| avgMapTime               | Int       | The average time of a map task (in ms)                                                              |
| avgReduceTime            | Int       | The average time of the reduce (in ms)                                                              |
| avgShuffleTime           | Int       | The average time of the shuffle (in ms)                                                             |
| avgMergeTime             | Int       | The average time of the merge (in ms)                                                               |
| failedReduceAttempts     | Int       | The number of failed reduce attempts                                                                |
| killedReduceAttempts     | Int       | The number of killed reduce attempts                                                                |
| successfulReduceAttempts | Int       | The number of successful reduce attempts                                                            |
| failedMapAttempts        | Int       | The number of failed map attempts                                                                   |
| killedMapAttempts        | Int       | The number of killed map attempts                                                                   |
| successfulMapAttempts    | Int       | The number of successful map attempts                                                               |
| acls                     | [ACL]     | A collection of acls objects                                                                        |

**Elements of the acls object**

| Item  | Data Type | Description   |
|-------|-----------|---------------|
| value | String    | The acl value |
| name  | String    | The acl name  |

## Check a Specific Job

It is also possible to check a specific historical job by a job id:

```
let job = try history.checkJob(jobId: "job_1484231633049_0005")
```

See [Data Structure of Job](#) as above.

## JobAttempt of Job

When you make a request for the list of job attempts, the information will be returned as an array of job attempt objects.

```
guard let attempts = try history.checkJobAttempts(jobId: "job_1484231633049_0005") else {
    // something goes wrong
}
attempts.forEach { attempt in
    print(attempt.id)
    print(attempt.containerId)
    print(attempt.nodeHttpAddress)
    print(attempt.nodeId)
    print(attempt.startTime)
}
```

JobAttempt Structure:

| Item            | Data Type | Description                                          |
|-----------------|-----------|------------------------------------------------------|
| id              | String    | The job attempt id                                   |
| nodeId          | String    | The node id of the node the attempt ran on           |
| nodeHttpAddress | String    | The node http address of the node the attempt ran on |
| logsLink        | String    | The http link to the job attempt logs                |
| containerId     | String    | The id of the container for the job attempt          |
| startTime       | Long      | The start time of the attempt (in ms since epoch)    |

## Counters of Job

With the job counters API, you can object a collection of resources that represent al the counters for that job.

```
guard let js = try his.checkJobCounters(jobId: "job_1484231633049_0005") else {
    // something wrong
}
js.counterGroup.forEach{ group in
    print(group.counterGroupName)
    group.counters.forEach { counter in
        print(counter.name)
        print(counter.mapCounterValue)
        print(counter.reduceCounterValue)
        print(counter.totalCounterValue)
    }
}
}
```

JobCounter Object

| Item         | Data Type      | Description                       |
|--------------|----------------|-----------------------------------|
| id           | String         | The job id                        |
| counterGroup | [CounterGroup] | An array of counter group objects |

CounterGroup Object

| Item             | Data Type | Description                   |
|------------------|-----------|-------------------------------|
| counterGroupName | string    | The name of the counter group |
| counter          | [Counter] | An array of counter objects   |

Counter Object

| Item               | Data Type | Description                       |
|--------------------|-----------|-----------------------------------|
| name               | String    | The name of the counter           |
| reduceCounterValue | Int       | The counter value of reduce tasks |
| mapCounterValue    | Int       | The counter value of map tasks    |
| totalCounterValue  | Int       | The counter value of all tasks    |

## Check Configuration of Job

Use `checkJobConfig()` to check configuration of a specific job:

```
guard let config = try his.checkJobConfig(jobId: "job_1484231633049_0005") else {
    /// something wrong
}
// print the configuration path
print(config.path)
// check properties of configuration file
for p in config.property {
    print(p.name)
    print(p.value)
    print(p.source)
}
```

### JobConfig Object

A job configuration resource contains information about the job configuration for this job.

| Item     | Data Type  | Description                            |
|----------|------------|----------------------------------------|
| path     | String     | The path to the job configuration file |
| property | [Property] | an array of property object            |

### Property Object

| Item   | Data Type | Description                                                                                                                                              |
|--------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| name   | String    | The name of the configuration property                                                                                                                   |
| value  | String    | The value of the configuration property                                                                                                                  |
| source | String    | The location this configuration object came from. If there is more then one of these it shows the history with the latest source at the end of the list. |

## Tasks of a Job

Use `checkJobTasks()` to collect task information from a specific job:

```
// get all tasks information from a specific job
let tasks = try his.checkJobTasks(jobId: "job_1484231633049_0005")

// print properties of each task
for t in tasks {
    print(t.progress)
    print(t.elapsedTime)
    print(t.state)
    print(t.startTime)
    print(t.id)
    print(t.type)
    print(t.successfulAttempt)
    print(t.finishTime)
} //next t
```

### Parameters of checkJobTasks()

Calling `checkJobTasks()` requires a jobId all the time, however, you can also specify an extra parameter to indicate what type of tasks shall be listed in the query results:

- jobId: a string represents the id of the job
- taskType: optional, could be `.MAP` or `.REDUCE`

### JobTask Object

The `checkJobTasks()` will return an array of `JobTask` data structure, as described below:

| Item              | Data Type | Description                                                                                            |
|-------------------|-----------|--------------------------------------------------------------------------------------------------------|
| id                | string    | The task id                                                                                            |
| state             | String    | The state of the task - valid values are: NEW, SCHEDULED, RUNNING, SUCCEDED, FAILED, KILL_WAIT, KILLED |
| type              | String    | The task type - MAP or REDUCE                                                                          |
| successfulAttempt | String    | The id of the last successful attempt                                                                  |
| progress          | Double    | The progress of the task as a percent                                                                  |
| startTime         | Int       | The time in which the task started (in ms since epoch) or -1 if it was never started                   |
| finishTime        | Int       | The time in which the task finished (in ms since epoch)                                                |
| elapsedTime       | Int       | The elapsed time since the application started (in ms)                                                 |

### Query a Specific JobTask

If both `jobId` and `jobTaskId` are available, you can use `checkJobTask()` to perform a query on a specific job task:

```
guard let task = try his.checkJobTask(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0") else {
    // something wrong
}
print(task.progress)
```

The return value of `checkJobTask` will be a [JobTask](#) object, as described above.

### Task Counters of Job

Method `checkJobTaskCounters()` can return counters of a specific task, as the example below:

```
guard let js = try his.checkJobTaskCounters(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0") else {
    // something wrong
}
// print the id of the job task counters
print(js.id)
// check out each counter group
js.taskCounterGroup.forEach{ group in
    print(group.counterGroupName)
    // print counters in each group.
    group.counters.forEach { counter in
        print(counter.name)
        print(counter.value)
    }
}
```

### JobTaskCounter Object

| Item             | Data Type      | Description                                                         |
|------------------|----------------|---------------------------------------------------------------------|
| id               | String         | The job id                                                          |
| taskCounterGroup | [CounterGroup] | An array of counter group objects, See <a href="#">CounterGroup</a> |

### Task Attempts

Function `checkJobTaskAttempts()` can return attempts from a specific job task.

```
let jobTaskAttempts = try his.checkJobTaskAttempts(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0")
for attempt in jobTaskAttempts {
    print(attempt.id)
    print(attempt.diagnostics)
    print(attempt.assignedContainerId)
    print(attempt.rack)
    print(attempt.state)
    print(attempt.progress)
} //next
```

Once done, the `checkJobTaskAttempts` will return an array of `TaskAttempt` object, as described below:

TaskAttempt Object

Elements of the TaskAttempt object:

| Item                | Data Type         | Description                                                                                                                                                                                                                            |
|---------------------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| id                  | String            | The task id                                                                                                                                                                                                                            |
| rack                | String            | The rack                                                                                                                                                                                                                               |
| state               | TaskAttempt.State | The state of the task attempt - valid values are: NEW, UNASSIGNED, ASSIGNED, RUNNING, COMMITPENDING, SUCCESSCONTAINERCLEANUP, SUCCEEDED, FAILCONTAINERCLEANUP, FAILTASKCLEANUP, FAILED, KILLCONTAINERCLEANUP, KILLTASK_CLEANUP, KILLED |
| type                | TaskType          | The type of task - MAP or REDUCE                                                                                                                                                                                                       |
| assignedContainerId | String            | The container id this attempt is assigned to                                                                                                                                                                                           |
| nodeHttpAddress     | String            | The http address of the node this task attempt ran on                                                                                                                                                                                  |
| diagnostics         | String            | A diagnostics message                                                                                                                                                                                                                  |
| progress            | Double            | The progress of the task attempt as a percent                                                                                                                                                                                          |
| startTime           | Int               | The time in which the task attempt started (in ms since epoch)                                                                                                                                                                         |
| finishTime          | Int               | The time in which the task attempt finished (in ms since epoch)                                                                                                                                                                        |
| elapsedTime         | Int               | The elapsed time since the task attempt started (in ms)                                                                                                                                                                                |

Besides, For reduce task attempts you also have the following fields:

| Item               | Data Type | Description                                                                                                  |
|--------------------|-----------|--------------------------------------------------------------------------------------------------------------|
| shuffleFinishTime  | Int       | The time at which shuffle finished (in ms since epoch)                                                       |
| mergeFinishTime    | Int       | The time at which merge finished (in ms since epoch)                                                         |
| elapsedShuffleTime | Int       | The time it took for the shuffle phase to complete (time in ms between reduce task start and shuffle finish) |
| elapsedMergeTime   | Int       | The time it took for the merge phase to complete (time in ms between the shuffle finish and merge finish)    |
| elapsedReduceTime  | Int       | The time it took for the reduce phase to complete (time in ms between merge finish to end of reduce task)    |

Query a Specific TaskAttempt

If a taskAttempt id is available, you can also check the specific one by `checkJobTaskAttempt()` , as demo below:

```
guard let jobTaskAttempts = try his.checkJobTaskAttempt(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0", "attempt_1326381300833_2_2_m_0") else {
    // something wrong
    print("there is no such a task attempt")
    return
} //end guard
print(attempt.id)
print(attempt.diagnostics)
print(attempt.assignedContainerId)
print(attempt.rack)
print(attempt.state)
print(attempt.progress)
```

The `checkJobTaskAttempt()` will return such an [TaskAttempt object](#) as described above.

## Attempt Counters of Task

Method `checkJobTaskAttemptCounters()` can return counters of a specific task attempt, as the example below:

```
guard let counters = try his.checkJobTaskAttemptCounters(jobId: "job_1484231633049_0005", taskId: "task_1326381300833_2_2_m_0", "attempt_1326381300833_2_2_m_0") else {
    // something wrong
    print("Task Attempt Counters Acquiring failed")
    return
}
// print the counters' id
print(counters.id)
// enumerate all groups
for group in counters.taskAttemptCounterGroup {
    // print group name
    print(group.counterGroupName)
    // enumerate all counters in this group
    for counter in group.counters {
        print(counter.name)
        print(counter.value)
    } //next counter
} //next group
}
```

### JobTaskAttemptCounters Object

| Item                    | Data Type      | Description                                                         |
|-------------------------|----------------|---------------------------------------------------------------------|
| id                      | String         | The job id                                                          |
| taskAttemptcounterGroup | [CounterGroup] | An array of counter group objects, See <a href="#">CounterGroup</a> |

## PerfectHadoop: YARN Node Manager

This project provides a Swift wrapper of YARN Node Manager REST API:

- `YARNNodeManager()`: access to all Applications and Containers information of a YARN manager

### Connect to YARN Node Manager

To connect to your Hadoop YARN Node Manager by Perfect, initialize a `YARNNodeManager()` object with sufficient parameters:

```
// this connection could possibly do some basic operations
let nm = YARNNodeManager(host: "yarnNodeManager.somehadoopdomain.com", port: 8042)
```

or connect to Hadoop YARN Node Manager with a valid user name:

```
// add user name if need
let nm = YARNNodeManager(host: "yarnNodeManager.somehadoopdomain.com", port: 8042, user: "your user name")
```

## Authentication

If using Kerberos to authenticate, please try codes below:

```
// set auth to kerberos
let yarn = YARNNodeManager(host: "yarnNodeManager.somehadoopdomain.com", port: 8042, user: "username", auth: .krb5)
```

## Parameters of YARNNodeManager Object

| Item      | Data Type     | Description                                                                                                      |
|-----------|---------------|------------------------------------------------------------------------------------------------------------------|
| service   | String        | the service protocol of web request - http / https                                                               |
| host      | String        | the hostname or ip address of the Hadoop YARN node manager                                                       |
| port      | Int           | the port of yarn host, default is 8042                                                                           |
| auth      | Authorization | .off or .krb5. Default value is .off                                                                             |
| proxyUser | String        | proxy user, if applicable                                                                                        |
| apibase   | String        | use this parameter <i>ONLY</i> the target server has a different api routine other than <code>/ws/v1/node</code> |
| timeout   | Int           | timeout in seconds, zero means never timeout during transfer                                                     |

## Get General Information

Call `checkOverall()` to get the general information of a YARN node manager in form of a `YARNNodeManager.Info` structure:

```
guard let inf = try yarn.checkOverall() else {
    // something goes wrong here
}
print(inf.id)
print(inf.nodeHostName)
print(inf.healthReport)
```

## Members of YARNNodeManager.NodeInfo

| Item                           | Data Type | Description                                                                    |
|--------------------------------|-----------|--------------------------------------------------------------------------------|
| id                             | String    | The NodeManager id                                                             |
| nodeHostName                   | String    | The host name of the NodeManager                                               |
| totalPmemAllocatedContainersMB | Int       | The amount of physical memory allocated for use by containers in MB            |
| totalVmemAllocatedContainersMB | Int       | The amount of virtual memory allocated for use by containers in MB             |
| totalVCoresAllocatedContainers | Int       | The number of virtual cores allocated for use by containers                    |
| lastNodeUpdateTime             | Int       | The last timestamp at which the health report was received (in ms since epoch) |
| healthReport                   | String    | The diagnostic health report of the node                                       |
| nodeHealthy                    | Bool      | true/false indicator of if the node is healthy                                 |
| nodeManagerVersion             | String    | Version of the NodeManager                                                     |
| nodeManagerBuildVersion        | String    | NodeManager build string with build version, user, and checksum                |
| nodeManagerVersionBuiltOn      | String    | Timestamp when NodeManager was built(in ms since epoch)                        |
| hadoopVersion                  | String    | Version of hadoop common                                                       |
| hadoopBuildVersion             | String    | Hadoop common build string with build version, user, and checksum              |
| hadoopVersionBuiltOn           | String    | Timestamp when hadoop common was built(in ms since epoch)                      |

## Applications

Call `checkApps()` to return an array of `APP` structure. With the Applications API, you can obtain a collection of resources, each of which represents an application.

```
let apps = try yarn.checkApps()
apps.forEach { a in
    print(a.id)
    print(a.containerids)
    print(a.state)
} //next
```

Data Structure of APP

| Item         | Data Type | Description                                                                                                                                                  |
|--------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| id           | String    | The application id                                                                                                                                           |
| user         | String    | The user who started the application                                                                                                                         |
| state        | APP.State | The state of the application - valid states are: NEW, INITING, RUNNING, FINISHING,CONTAINERSWAIT, APPLICATIONRESOURCESCLEANINGUP, FINISHED                   |
| containerids | [String]] | The list of containerids currently being used by the application on this node. If not present then no containers are currently running for this application. |

Check A Specific Application

If an application id is available, you can call `checkApp()` to query information on nodes:

```
guard let app = try yarn.checkApp("application_1326121700862_0005") else {
    // something wrong
    print("check app failed")
    return
}
// print out all container ids of the application
print(app.containerids)
```

`checkApp()` will return an `APP` object, as described above.

Check Containers

Call `checkContainers()` to check all containers on the current node:

```
// get all containers
let containers = try ynode.checkContainers()

// check how many containers on node
print(containers.count)

containers.forEach { container in
    print(container.id)
    print(container.state)
    print(container.nodeId)
    print(container.diagnostics)
    /// ...
}
```

`checkContainers()` returns an array of `Container` structure, as described below:

Container Object

| Item                | Data Type | Description                                                                                                                                                                                                              |
|---------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| id                  | String    | The container id                                                                                                                                                                                                         |
| state               | String    | State of the container - valid states are: <i>NEW, LOCALIZING, LOCALIZATIONFAILED, LOCALIZED, RUNNING, EXITEDWITHSUCCESS, EXITEDWITHFAILURE, KILLING, CONTAINERCLEANEDUPAFTERKILL, CONTAINERRESOURCECLEANINGUP, DONE</i> |
| nodeId              | String    | The id of the node the container is on                                                                                                                                                                                   |
| containerLogsLink   | String    | The http link to the container logs                                                                                                                                                                                      |
| user                | String    | The user name of the user which started the container                                                                                                                                                                    |
| exitCode            | Int       | Exit code of the container                                                                                                                                                                                               |
| diagnostics         | String    | A diagnostic message for failed containers                                                                                                                                                                               |
| totalMemoryNeededMB | Int       | Total amount of memory needed by the container (in MB)                                                                                                                                                                   |
| totalVCoresNeeded   | Int       | Total number of virtual cores needed by the container                                                                                                                                                                    |

## Check a Specific Container

If a container id is available, you can also use `checkContainer()` to check a specific container:

```
guard let container = try ynode.checkContainer(id: container.id) else {
    // checking container failed
}
print(container.id)
print(container.state)
print(container.nodeId)
print(container.diagnostics)
```

This method will return a `Container` object as described above.

## PerfectHadoop: YARN Resource Manager

This project provides a Swift wrapper of YARN Resource Manager REST API:

- `YARNResourceManager()` : access to cluster information of YARN, including cluster and its metrics, scheduler, application submit, etc.

## Connect to YARN Resource Manager

To connect to your Hadoop YARN Resource Manager by Perfect, initialize a `YARNResourceManager()` object with sufficient parameters:

```
// this connection could possibly do some basic operations
let yarn = YARNResourceManager(host: "yarn.somehadoopdomain.com", port: 8088)
```

or connect to Hadoop YARN Node Manager with a valid user name:

```
// add user name if need
let yarn = YARNResourceManager(host: "yarn.somehadoopdomain.com", port: 8088, user: "your user name")
```

### Authentication

If using Kerberos to authenticate, please try codes below:

```
// set auth to kerberos
let yarn = YARNResourceManager(host: "yarn.somehadoopdomain.com", port: 8088, user: "username", auth: .krb5)
```

### Parameters of YARNResourceManager Object

| Item      | Data Type     | Description                                                                                                         |
|-----------|---------------|---------------------------------------------------------------------------------------------------------------------|
| service   | String        | the service protocol of web request - http / https                                                                  |
| host      | String        | the hostname or ip address of the Hadoop YARN Resource Manager                                                      |
| port      | Int           | the port of yarn host, default is 8088                                                                              |
| auth      | Authorization | .off or .krb5. Default value is .off                                                                                |
| proxyUser | String        | proxy user, if applicable                                                                                           |
| apibase   | String        | use this parameter <i>ONLY</i> the target server has a different api routine other than <code>/ws/v1/cluster</code> |
| timeout   | Int           | timeout in seconds, zero means never timeout during transfer                                                        |

## Get General Information

Call `checkClusterInfo()` to get the general information of a YARN Resource Manager in form of a `ClusterInfo` structure:

```
guard let i = try yarn.checkClusterInfo() else {
    print("unable to check cluster info")
    return
} //end guard
print(i.startedOn)
print(i.state)
print(i.hadoopVersion)
print(i.resourceManagerVersion)
```

Once called, `checkClusterInfo()` would return a `ClusterInfo` object, as described below:

### ClusterInfo Object

| Item                          | Data Type | Description                                                                             |
|-------------------------------|-----------|-----------------------------------------------------------------------------------------|
| id                            | Int       | The cluster id                                                                          |
| startedOn                     | Int       | The time the cluster started (in ms since epoch)                                        |
| state                         | String    | The ResourceManager state - valid values are: NOTINITED, INITED, STARTED, STOPPED       |
| haState                       | String    | The ResourceManager HA state - valid values are: INITIALIZING, ACTIVE, STANDBY, STOPPED |
| resourceManagerVersion        | String    | Version of the ResourceManager                                                          |
| resourceManagerBuildVersion   | String    | ResourceManager build string with build version, user, and checksum                     |
| resourceManagerVersionBuiltOn | String    | Timestamp when ResourceManager was built (in ms since epoch)                            |
| hadoopVersion                 | String    | Version of hadoop common                                                                |
| hadoopBuildVersion            | String    | Hadoop common build string with build version, user, and checksum                       |
| hadoopVersionBuiltOn          | String    | Timestamp when hadoop common was built(in ms since epoch)                               |

## Cluster Metrics

Method `checkClusterMetrics()` returns a detailed info structure of cluster.

```
guard let m = try yarn.checkClusterMetrics() else {
    // something wrong
}
print(m.availableMB)
print(m.availableVirtualCores)
print(m.allocatedVirtualCores)
print(m.totalMB)
```

Once called, `checkClusterMetrics()` would return a ClusterMetrics object as listed below:

ClusterMetrics Object

| Item                  | Data Type | Description                           |
|-----------------------|-----------|---------------------------------------|
| appsSubmitted         | Int       | The number of applications submitted  |
| appsCompleted         | Int       | The number of applications completed  |
| appsPending           | Int       | The number of applications pending    |
| appsRunning           | Int       | The number of applications running    |
| appsFailed            | Int       | The number of applications failed     |
| appsKilled            | Int       | The number of applications killed     |
| reservedMB            | Int       | The amount of memory reserved in MB   |
| availableMB           | Int       | The amount of memory available in MB  |
| allocatedMB           | Int       | The amount of memory allocated in MB  |
| totalMB               | Int       | The amount of total memory in MB      |
| reservedVirtualCores  | Int       | The number of reserved virtual cores  |
| availableVirtualCores | Int       | The number of available virtual cores |
| allocatedVirtualCores | Int       | The number of allocated virtual cores |
| totalVirtualCores     | Int       | The total number of virtual cores     |
| containersAllocated   | Int       | The number of containers allocated    |
| containersReserved    | Int       | The number of containers reserved     |
| containersPending     | Int       | The number of containers pending      |
| totalNodes            | Int       | The total number of nodes             |
| activeNodes           | Int       | The number of active nodes            |
| lostNodes             | Int       | The number of lost nodes              |
| unhealthyNodes        | Int       | The number of unhealthy nodes         |
| decommissionedNodes   | Int       | The number of nodes decommissioned    |
| rebootedNodes         | Int       | The number of nodes rebooted          |

Cluster Scheduler

Method `checkSchedulerInfo()` returns a detailed info structure of scheduler.

```
guard let sch = try yarn.checkSchedulerInfo() else {
    // something wrong, must return
}
print(sch.capacity)
print(sch.maxCapacity)
print(sch.queueName)
print(sch.queues.count)
```

Once done, `checkSchedulerInfo()` would return a `SchedulerInfo` structure as listed below:

SchedulerInfo Object

| Item                   | Data Type | Description                                                                  |
|------------------------|-----------|------------------------------------------------------------------------------|
| availNodeCapacity      | Int       | The available node capacity                                                  |
| capacity               | Double    | Configured queue capacity in percentage relative to its parent queue         |
| maxCapacity            | Double    | Max capacity of the queue                                                    |
| maxQueueMemoryCapacity | Int       | Configured maximum queue capacity in percentage relative to its parent queue |
| minQueueMemoryCapacity | Int       | Minimum queue memory capacity                                                |
| numContainers          | Int       | The number of containers                                                     |
| numNodes               | Int       | The total number of nodes                                                    |
| qstate                 | QState    | State of the queue - valid values are: STOPPED, RUNNING                      |
| queueName              | String    | Name of the queue                                                            |
| queues                 | [Queue]   | A collection of queue resources                                              |
| rootQueue              | FairQueue | A collection of root queue resources                                         |
| totalNodeCapacity      | Int       | The total node capacity                                                      |
| type                   | String    | Scheduler type - capacityScheduler                                           |
| usedCapacity           | Double    | Used queue capacity in percentage                                            |
| usedNodeCapacity       | Int       | The used node capacity                                                       |

FairQueue Object

| Item             | Data Type     | Description                                                                     |
|------------------|---------------|---------------------------------------------------------------------------------|
| maxApps          | Int           | The maximum number of applications the queue can have                           |
| minResources     | ResourcesUsed | The configured minimum resources that are guaranteed to the queue               |
| maxResources     | ResourcesUsed | The configured maximum resources that are allowed to the queue                  |
| usedResources    | ResourcesUsed | The sum of resources allocated to containers within the queue                   |
| fairResources    | ResourcesUsed | The queue's fair share of resources                                             |
| clusterResources | ResourcesUsed | The capacity of the cluster                                                     |
| queueName        | String        | The name of the queue                                                           |
| schedulingPolicy | String        | The name of the scheduling policy used by the queue                             |
| childQueues      | FairQueue     | A collection of sub-queue information. Omitted if the queue has no childQueues. |
| type             | String        | type of the queue - fairSchedulerLeafQueueInfo                                  |
| numActiveApps    | Int           | The number of active applications in this queue                                 |
| numPendingApps   | Int           | The number of pending applications in this queue                                |

Queue Object

| Item                         | Data Type     | Description                                                                    |
|------------------------------|---------------|--------------------------------------------------------------------------------|
| absoluteCapacity             | Double        | Absolute capacity percentage this queue can use of entire cluster              |
| absoluteMaxCapacity          | Double        | Absolute maximum capacity percentage this queue can use of the entire cluster  |
| absoluteUsedCapacity         | Double        | Absolute used capacity percentage this queue is using of the entire cluster    |
| capacity                     | Double        | Configured queue capacity in percentage relative to its parent queue           |
| maxActiveApplications        | Int           | The maximum number of active applications this queue can have                  |
| maxActiveApplicationsPerUser | Int           | The maximum number of active applications per user this queue can have         |
| maxApplications              | Int           | The maximum number of applications this queue can have                         |
| maxApplicationsPerUser       | Int           | The maximum number of applications per user this queue can have                |
| maxCapacity                  | Double        | Configured maximum queue capacity in percentage relative to its parent queue   |
| numActiveApplications        | Int           | The number of active applications in this queue                                |
| numApplications              | Int           | The number of applications currently in the queue                              |
| numContainers                | Int           | The number of containers being used                                            |
| numPendingApplications       | Int           | The number of pending applications in this queue                               |
| queueName                    | String        | The name of the queue                                                          |
| queues                       | [Queue]       | A collection of sub-queue information. Omitted if the queue has no sub-queues. |
| resourcesUsed                | ResourcesUsed | The total amount of resources used by this queue                               |
| state                        | String        | The state of the queue                                                         |
| type                         | String        | type of the queue - capacitySchedulerLeafQueueInfo                             |
| usedCapacity                 | Double        | Used queue capacity in percentage                                              |
| usedResources                | String        | A string describing the current resources used by the queue                    |
| userLimit                    | Int           | The minimum user limit percent set in the configuration                        |
| userLimitFactor              | Double        | The user limit factor set in the configuration                                 |
| users                        | [User]        | A collection of user objects containing resources used, see below:             |

User Object

| Item                   | Data Type     | Description                                                                  |
|------------------------|---------------|------------------------------------------------------------------------------|
| username               | String        | The username of the user using the resources                                 |
| resourcesUsed          | ResourcesUsed | The amount of resources used by the user in this queue, see definition below |
| numActiveApplications  | Int           | The number of active applications for this user in this queue                |
| numPendingApplications | Int           | The number of pending applications for this user in this queue               |

ResourcesUsed Object

| Item   | Data Type | Description                               |
|--------|-----------|-------------------------------------------|
| memory | int       | Memory required for each container        |
| vCores | int       | Virtual cores required for each container |

# Cluster Nodes

Check All Nodes

Method `checkClusterNodes()` returns an array of node info of cluster.

```
let nodes = try yarn.checkClusterNodes()
nodes.forEach { node in
    print(node.rack)
    print(node.availableVirtualCores)
    print(node.availMemoryMB)
    print(node.healthReport)
    print(node.healthStatus)
    print(node.id)
    print(node.lastHealthUpdate)
    print(node.nodeHostName)
    print(node.nodeHTTPAddress)
```

Once done, the `checkClusterNodes()` would return an array of `Node` object, as described below:

Node Object

| Item                  | Data Type | Description                                                                                   |
|-----------------------|-----------|-----------------------------------------------------------------------------------------------|
| rack                  | String    | The rack location of this node                                                                |
| state                 | String    | State of the node - valid values are: NEW, RUNNING, UNHEALTHY, DECOMMISSIONED, LOST, REBOOTED |
| id                    | String    | The node id                                                                                   |
| nodeHostName          | String    | The host name of the node                                                                     |
| nodeHTTPAddress       | String    | The nodes HTTP address                                                                        |
| healthStatus          | String    | The health status of the node - Healthy or Unhealthy                                          |
| healthReport          | String    | A detailed health report                                                                      |
| lastHealthUpdate      | Int       | The last time the node reported its health (in ms since epoch)                                |
| usedMemoryMB          | Int       | The total amount of memory currently used on the node (in MB)                                 |
| availMemoryMB         | Int       | The total amount of memory currently available on the node (in MB)                            |
| usedVirtualCores      | Int       | The total number of vCores currently used on the node                                         |
| availableVirtualCores | Int       | The total number of vCores available on the node                                              |
| numContainers         | int       | The total number of containers currently running on the node                                  |

Check A Node

Method `checkClusterNode()` returns a detailed info structure of a node.

```
guard let n = try yarn.checkClusterNode(id: "host.domain.com:8041") else {
    // something wrong, must return
}
```

# Applications on Cluster

Check All Applications

Method `checkApps()` returns an array of APP structure. `` swift let apps = try yarn.checkApps() // or alternatively, you can filter out those APPs you want by setting query parameters: /// let apps = try yarn.checkApps(states: [APP.State.FINISHED, APP.State.RUNNING], finalStatus: APP.FinalStatus.SUCCEEDED)

```
apps.forEach{ a in print(a.allocatedMB) print(a.allocatedVCores) print(a.amContainerLogs) print(a.amHostHttpAddress) print(a.amNodeLabelExpression) print(a.amRPCAddress)
print(a.applicationPriority) print(a.applicationTags) }
```

Once done, the `checkApps()` would return an array of APP objects, as described below:

APP Object

| Item                   | Data Type | Description                                                                                                                                                                                 |
|------------------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| id                     | String    | The application id                                                                                                                                                                          |
| user                   | String    | The user who started the application                                                                                                                                                        |
| name                   | String    | The application name                                                                                                                                                                        |
| applicationType        | String    | The application type                                                                                                                                                                        |
| queue                  | String    | The queue the application was submitted to                                                                                                                                                  |
| state                  | String    | The application state according to the ResourceManager - valid values are members of the YarnApplicationState enum: NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED |
| finalStatus            | String    | The final status of the application if finished - reported by the application itself - valid values are: UNDEFINED, SUCCEEDED, FAILED, KILLED                                               |
| progress               | Double    | The progress of the application as a percent                                                                                                                                                |
| trackingUI             | String    | Where the tracking url is currently pointing - History (for history server) or ApplicationMaster                                                                                            |
| trackingUrl            | String    | The web URL that can be used to track the application                                                                                                                                       |
| diagnostics            | String    | Detailed diagnostics information                                                                                                                                                            |
| clusterId              | Int       | The cluster id                                                                                                                                                                              |
| startedTime            | Int       | The time in which application started (in ms since epoch)                                                                                                                                   |
| finishedTime           | Int       | The time in which the application finished (in ms since epoch)                                                                                                                              |
| elapsedTime            | Int       | The elapsed time since the application started (in ms)                                                                                                                                      |
| amContainerLogs        | String    | The URL of the application master container logs                                                                                                                                            |
| amHostHttpAddress      | String    | The nodes http address of the application master                                                                                                                                            |
| amRPCAddress           | String    | The RPC address of the application master                                                                                                                                                   |
| allocatedMB            | Int       | The sum of memory in MB allocated to the application's running containers                                                                                                                   |
| allocatedVCores        | Int       | The sum of virtual cores allocated to the application's running containers                                                                                                                  |
| runningContainers      | Int       | The number of containers currently running for the application                                                                                                                              |
| memorySeconds          | Int       | The amount of memory the application has allocated (megabyte-seconds)                                                                                                                       |
| vcoreSeconds           | Int       | The amount of CPU resources the application has allocated (virtual core-seconds)                                                                                                            |
| unmanagedApplication   | Bool      | Is the application unmanaged.                                                                                                                                                               |
| applicationPriority    | Int       | priority of the submitted application                                                                                                                                                       |
| appNodeLabelExpression | String    | Node Label expression which is used to identify the nodes on which application's containers are expected to run by default.                                                                 |
| amNodeLabelExpression  | String    | Node Label expression which is used to identify the node on which application's AM container is expected to run.                                                                            |

### Check A Specific Application

Method `checkApp()` returns a specific APP Object.

```
let a = try yarn.checkApp(id: "application_1484231633049_0025")
print(a.allocatedMB)
print(a.allocatedVCores)
print(a.amContainerLogs)
print(a.amHostHttpAddress)
print(a.amNodeLabelExpression)
print(a.amRPCAddress)
print(a.applicationPriority)
print(a.applicationTags)
```

The return value is an APP structure, please check the above APP object for detail.

## Apply A New Application

Method `newApplication()` returns a new application handler.

```
guard let a = try yarn.newApplication() else {  
    // cannot create a new application, must return  
}
```

Once completed, the `newApplication()` method will return a `NewApplication` object, as described below:

### NewApplication Object

| Item                      | Data Type     | Description                                                 |
|---------------------------|---------------|-------------------------------------------------------------|
| id                        | String        | The newly created application id                            |
| maximumResourceCapability | ResourcesUsed | The maximum resource capabilities available on this cluster |

### ResourcesUsed Object

The `NewApplication` object will contain a special object called `ResourceUsed`, as described below:

| Item   | Data Type | Description                                           |
|--------|-----------|-------------------------------------------------------|
| memory | Int       | The maximum memory available for a container          |
| vCores | Int       | The maximum number of cores available for a container |

## Submit Modification to An Application

Method `submit()` can submit modifications to a specific application.

**⚠️ Note ⚠️** Detail configuration of Yarn MapReduce Application is out of the range of this document, please get more information at [Hadoop MapReduce Next Generation - Writing YARN Applications https://wiki.apache.org/hadoop/WritingYarnApps](https://wiki.apache.org/hadoop/WritingYarnApps)

```

// create an empty application to fill in the blanks
let sum = SubmitApplication()

// *MUST* set the application id
sum.id = "application_1484231633049_0025"
// set the application name
sum.name = "test"

// allocate a blank local resource
let local = LocalResource(resource: "hdfs://localhost:9000/user/rockywei/DistributedShell/demo-app/AppMaster.jar", type: .FILE, visibility: .APPLICATION, size: 43004, timestamp: 1405452071209)

// assign the resource into an array
let localResources = Entries([Entry(key:"AppMaster.jar", value: local)])

// *MUST* fill in the field of map reduce command
let commands = Commands("/hdp/bin/hadoop jar /hdp/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.0.0-alpha1.jar grep input output 'dfs[a-z.]+'")

// setup environment as need
let environments = Entries([Entry(key:"DISTRIBUTEDSHELLSCRIPTTIMESTAMP", value: "1405459400754"), Entry(key:"CLASSPATH", value:"{{CLASSPATH}}<CPS>.*<CPS>{{HADOOP_CONF_DIR}}<CPS>{{HADOOP_COMMON_HOME}}/share/hadoop/common/*<CPS>{{HADOOP_COMMON_HOME}}/share/hadoop/common/lib/*<CPS>{{HADOOP_HDFS_HOME}}/share/hadoop/hdfs/*<CPS>{{HADOOP_HDFS_HOME}}/share/hadoop/hdfs/lib/*<CPS>{{HADOOP_YARN_HOME}}/share/hadoop/yarn/*<CPS>{{HADOOP_YARN_HOME}}/share/hadoop/yarn/lib/*<CPS>./log4j.properties"), Entry(key:"DISTRIBUTEDSHELLSCRIPTLEN", value:6), Entry(key:"DISTRIBUTEDSHELLSCRIPTLOCATION", value: "hdfs://localhost:9000/user/rockywei/demo-app/shellCommands")])

// specify the container info
sum.amContainerSpec = AmContainerSpec(localResources: localResources, environment: environments, commands: commands)

// set other information as need
sum.unmanagedAM = false
sum.maxAppAttempts = 2
sum.resource = ResourceRequest(memory: 1024, vCores: 1)
sum.type = "MapReduce"
sum.keepContainersAcrossApplicationAttempts = false

// set the log context
sum.logAggregationContext = LogAggregationContext(logIncludePattern: "file1", logExcludePattern: "file2", rolledLogIncludePattern: "file3", rolledLogExcludePattern: "file4", logAggregationPolicyClassName: "org.apache.hadoop.yarn.server.nodemanager.containermanager.logaggregation.AllContainerLogAggregationPolicy", logAggregationPolicyParameters: "")

// set the failures validity interval
sum.attemptFailuresValidityInterval = 3600000

// set the reservationId, if possible
sum.reservationId = "reservation_1454114874_1"
sum.amBlackListingRequests = AmBlackListingRequests(amBlackListingEnabled: true, disableFailureThreshold: 0.01)

try yarn.submit(application: sum)

```

## SubmitApplication Object

`SubmitApplication` class contains quite a few sub types, as described below:

| Item                                    | Data Type              | Description                                                                                                  |
|-----------------------------------------|------------------------|--------------------------------------------------------------------------------------------------------------|
| id                                      | String                 | The application id                                                                                           |
| name                                    | String                 | The application name                                                                                         |
| queue                                   | String                 | The name of the queue to which the application should be submitted                                           |
| priority                                | Int                    | The priority of the application                                                                              |
| amContainerSpec                         | AmContainerSpec        | The application master container launch context, described below                                             |
| unmanagedAM                             | Bool                   | Is the application using an unmanaged application master                                                     |
| maxAppAttempts                          | int                    | The max number of attempts for this application                                                              |
| resource                                | ResourceRequest        | The resources the application master requires, described below                                               |
| type                                    | String                 | The application type(MapReduce, Pig, Hive, etc)                                                              |
| keepContainersAcrossApplicationAttempts | Bool                   | Should YARN keep the containers used by this application instead of destroying them                          |
| tags                                    | [String]               | List of application tags, please see the request examples on how to specify the tags                         |
| logAggregationContext                   | LogAggregationContext  | Represents all of the information needed by the NodeManager to handle the logs for this application          |
| attemptFailuresValidityInterval         | Int                    | The failure number will no take attempt failures which happen out of the validityInterval into failure count |
| reservationId                           | string                 | Represent the unique id of the corresponding reserved resource allocation in the scheduler                   |
| amBlackListingRequests                  | AmBlackListingRequests | Contains blacklisting information such as “enable/disable AM blacklisting” and “disable failure threshold”   |

AmContainerSpec Object

| Item            | Data Type   | Description                                                                                                                         |
|-----------------|-------------|-------------------------------------------------------------------------------------------------------------------------------------|
| localResources  | Entries     | Object describing the resources that need to be localized, described below                                                          |
| environment     | Entries     | Environment variables for your containers, specified as key value pairs                                                             |
| commands        | Commands    | The commands for launching your container, in the order in which they should be executed                                            |
| serviceData     | Entries     | Application specific service data; key is the name of the auxiliary service, value is base-64 encoding of the data you wish to pass |
| credentials     | Credentials | The credentials required for your application to run, described below                                                               |
| applicationAcls | Entries     | ACLs for your application; the key can be “VIEWAPP” or “MODIFYAPP”, the value is the list of users with the permissions             |

LocalResource Object

| Item       | Data Type    | Description                                                                                 |
|------------|--------------|---------------------------------------------------------------------------------------------|
| resource   | String       | Location of the resource to be localized                                                    |
| type       | ResourceType | Type of the resource; options are “ARCHIVE”, “FILE”, and “PATTERN”                          |
| visibility | Visibility   | Visibility the resource to be localized; options are “PUBLIC”, “PRIVATE”, and “APPLICATION” |
| size       | Int          | Size of the resource to be localized                                                        |
| timestamp  | Int          | Timestamp of the resource to be localized                                                   |

Commands Object

Currently Commands object only one string field named `command` . **⚠️NOTE⚠️** According to Hadoop 3.0 alpha document, commands should be an array of command strings with a sequential meaning, however, the example given in Hadoop 3.0 alpha indicates only one command. This feature is experimental and subject to change in future.

Credentials Object

| Item    | Data Type       | Description                                                                                                                                                                                                   |
|---------|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| tokens  | [String:String] | Tokens that you wish to pass to your application, specified as key-value pairs. The key is an identifier for the token and the value is the token(which should be obtained using the respective web-services) |
| secrets | [String:String] | Secrets that you wish to use in your application, specified as key-value pairs. They key is an identifier and the value is the base-64 encoding of the secret                                                 |

ResourceRequest Object

| Item   | Data Type | Description                               |
|--------|-----------|-------------------------------------------|
| memory | int       | Memory required for each container        |
| vCores | int       | Virtual cores required for each container |

Entries Object

The `Entries` Object has only one field of element: `entry` , which is an array of `Entry` object or `EncryptedEntry` object. Each `Entry` has two elements: a string `key` and a `Any` type `value` .

The difference between `Entry` and `EncryptedEntry` is that the value of `EncryptedEntry` will be encoded in base64 format before submission, providing the type of value is either `String` or `[UInt8]` .

AmBlackListingRequests Object

| Item                    | Data Type | Description                               |
|-------------------------|-----------|-------------------------------------------|
| amBlackListingEnabled   | Bool      | Whether AM Blacklisting is enabled        |
| disableFailureThreshold | Float     | AM Blacklisting disable failure threshold |

LogAggregationContext Object

| Item                           | Data Type | Description                                                                                              |
|--------------------------------|-----------|----------------------------------------------------------------------------------------------------------|
| logIncludePattern              | String    | The log files which match the defined include pattern will be uploaded when the application finishes     |
| logExcludePattern              | String    | The log files which match the defined exclude pattern will not be uploaded when the application finishes |
| rolledLogIncludePattern        | String    | The log files which match the defined include pattern will be aggregated in a rolling fashion            |
| rolledLogExcludePattern        | String    | The log files which match the defined exclude pattern will not be aggregated in a rolling fashion        |
| logAggregationPolicyClassName  | String    | The policy which will be used by NodeManager to aggregate the logs                                       |
| logAggregationPolicyParameters | String    | The parameters passed to the policy class                                                                |

Application State Control

Method `getApplicationStatus()` returns the current state of application.

```
guard let state = try yarn.getApplicationStatus(id: "application_1484231633049_0025") else {
    // something wrong, must return
}
print(state)
```

Method `setApplicationStatus()` can set the current state of application to a designated one.

```
try yarn.setApplicationStatus(id: "application_1484231633049_0025", state: .KILLED)
```

Valid States includes:

```
NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED
```

Application Queue Control

Method `getApplicationQueue()` returns the current queue name of application.

```
guard let queue = try yarn.getApplicationQueue(id: "application_1484231633049_0025") else {
    // something wrong, must return
}
print(queue)
```

Method `setApplicationQueue()` can set the current queue name of application to a designated one.

```
try yarn.setApplicationQueue(id: "application_1484231633049_0025", queue:"ala")
```

## Application Priority Control

Method `getApplicationPriority()` returns the current priority of application.

```
guard let priority = try yarn.getApplicationPriority(id: "application_1484231633049_0025") else {
    // something wrong, must return
}
print(priority)
```

Currently the returned priority is an integer such as `0`.

Method `setApplicationPriority()` can set the current priority of application to a designated one.

```
try yarn.setApplicationPriority(id: "application_1484231633049_0025", priority: 1)
```

## Application Attempts

Method `checkAppAttempts()` returns an array of Attempt object of application.

```
guard let attempts = try yarn.checkAppAttempts(id: "application_1484231633049_0025") else {
    // something wrong, must return
}
attempts.forEach { attempt in
    print(attempt.containerId)
    print(attempt.id)
    print(attempt.nodeHttpAddress)
    print(attempt.nodeId)
    print(attempt.startTime)
} //next
```

Once done, `checkAppAttempts()` would return an array of AppAttempt object, as described below:

### AppAttempt Object

| Item            | Data Type | Description                                          |
|-----------------|-----------|------------------------------------------------------|
| id              | String    | The app attempt id                                   |
| nodeId          | String    | The node id of the node the attempt ran on           |
| nodeHttpAddress | String    | The node http address of the node the attempt ran on |
| logsLink        | String    | The http link to the app attempt logs                |
| containerId     | String    | The id of the container for the app attempt          |
| startTime       | Int       | The start time of the attempt (in ms since epoch)    |

## Application Statistics

Method `checkAppStatistics()` returns an array of statistic variables of application.

```
swift let sta = try yarn.checkAppStatistics(states: [APP.State.FINISHED, APP.State.RUNNING]) sta.forEach{ s in print(s.count) print(s.state)}
```

`checkAppStatistics()` allows user to perform a query with two additional parameters:

| Parameter        | Data Type   | Description                                                                                                                                                                                                                                                                                                                  |
|------------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| states           | [APP.State] | states of the applications. If states is not provided, the API will enumerate all application states and return the counts of them.                                                                                                                                                                                          |
| applicationTypes | [String]    | types of the applications. If applicationTypes is not provided, the API will count the applications of any application type. In this case, the response shows * to indicate any application type. Note that we only support at most one applicationType temporarily, such as<br><pre>applicationTypes: [ "MapReduce" ]</pre> |

## Ubuntu 16.04: Starting Services at System Boot

A compiled Swift binary can be run on demand on Ubuntu, however for common server side applications such as API servers, it is best to have the startup and monitoring of state performed by the system automatically.

Ubuntu 16.04 leverages `systemd` to manage services, and this guide explains how to set up and enable a binary to run under `systemd`.

Once your Swift binary is placed on the server, you will need to create a `.service` file in the `/etc/systemd/system/` directory.

The contents of the file will be, at minimum, the following (substituting the obvious values):

```
[Unit]
Description=XXX API Server

[Service]
Type=simple
ExecStart= /path/to/binary/APIServer
Restart=always
PIDFile=/var/run/apiserver.pid

[Install]
WantedBy=multi-user.target
```

Once this file has been saved, set the permissions:

```
chmod +x /etc/systemd/system/apiserver.service
chmod 755 /etc/systemd/system/apiserver.service
```

Then enable, and start the service:

```
sudo systemctl enable apiserver.service
sudo systemctl start apiserver.service
```

Further detail about `systemd` can be found at [Managing Services with systemd](#) # Perfect - Python

This project provides an expressway to import Python 2.7 module as a Server Side Swift Library.

This package builds with Swift Package Manager and is part of the [Perfect](#) project, but can also run independently.

Ensure you have installed and activated the latest Swift 3.1 / 4.0 tool chain.

## Linux Build Note

Please make sure `libpython2.7-dev` was installed on Ubuntu 16.04:

```
$ sudo apt-get install libpython2.7-dev
```

## MacOS Build Note

Please make sure Xcode 8.3.3 / 9.0 or later version was installed.

## Quick Start

Add PerfectPython dependency to your Package.swift

```
.Package(url: "https://github.com/PerfectlySoft/Perfect-Python.git", majorVersion: 1)
```

Then import two different libraries into the swift source code:

```
import PythonAPI
import PerfectPython
```

Before any python api calls, make sure to initialize the library by calling `Py_Initialize()` function:

```
Py_Initialize()
```

## Import Python Modules

Use `PyObj` class to import python modules. In the following example, a python script `/tmp/clstest.py` has been imported into the current Swift context:

```
let pymod = try PyObj(path: "/tmp", import: "clstest")
```

## Access Python Variables

Once imported modules, you can use `PyObj.load()` function to access a variable value, or using `PyObj.save()` to store a new value to the current python variable.

For example, if there is a variable called `stringVar` in a python script:

```
stringVar = 'Hello, world'
```

Then you can read its value in such a form:

```
if let str = pymod.load("stringVar")?.value as? String {
    print(str)
    // will print it out as "Hello, world!"
}
```

You can also directly overwrite the value of the same variable:

```
try pymod.save("stringVar", newValue: "Hola, 🇨🇦! ")
```

**NOTE** Currently, Perfect-Python supports the following data types between Swift and Python:

| Python Type | Swift Type   | Remark      |
|-------------|--------------|-------------|
| int         | Int          |             |
| float       | Double       |             |
| str         | String       |             |
| list        | [Any]        | Recursively |
| dict        | [String:Any] | Recursively |

For example, you can convert a Swift `String` to `PyObj` by: `let pystr = "Hello".python()` or `let pystr = try PyObj(value:"Hello")` .

To convert a `PyObj` to a Swift data type, e.g., a `String` , there are also two available approaches: `let str = pystr.value as? String` and `let str = String(python: pystr)` .

## Call A Python Function

Method `PyObj.call()` is available to execute function call with arguments. Consider the python code below:

```
def mymul(num1, num2):
    return num1 * num2
```

Perfect-Python can wrap this call by its name as a string and the arguments as an array:

```
if let res = pymod.call("mymul", args: [2,3])?.value as? Int {
    print(res)
    // the result will be 6
}
```

## Python Object Classes

The same `PyObj.load()` function helps to access the python class type, however, a following method `PyObj.construct()` should be called for object instance initialization. This method also supports parameters as an array for python object class construction.

Assume that there is a typical python class called `Person`, which has two properties `name` and `age`, and an object method called `intro()`:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def intro(self):
        return 'Name: ' + self.name + ', Age: ' + str(self.age)
```

To initialize such a class object in Swift, the first two steps look like:

```
if let personClass = pymod.load("Person"),
    let person = personClass.construct(["rocky", 24]) {
    // person is now the object instance
}
```

Then you can access the properties and class methods as common variables and functions do:

```
if let name = person.load("name")?.value as? String,
    let age = person.load("age")?.value as? Int,
    let intro = person.call("intro", args: [])?.value as? String {
    print(name, age, intro)
}
```

## Callbacks

Consider the following python code as you can execute a function as a parameter like `x = caller('Hello', callback)`:

```
def callback(msg):
    return 'callback: ' + msg

def caller(info, func):
    return func(info)
```

The equivalent Swift code is nothing special but using the objective callback function as an argument before calling:

```
if let fun = pymod.load("callback"),
    let result = pymod.call("caller", args: ["Hello", fun]),
    let v = result.value as? String {
    print(v)
    // it will be "callback: Hello"
}
```# Swift ORM: "StORM"
```

StORM is a modular ORM for Swift, layered on top of [Perfect](https://github.com/PerfectlySoft/Perfect).

It aims to be easy to use, but flexible, and maintain consistency between datasource implementations for the user: you, the developer. It tries to allow you to write great code without worrying about the details of how to interact with the database.

## Relevant Examples

PostgreSQL

```
* [PostgreStORM-Demo](https://github.com/PerfectExamples/PostgreStORM-Demo)
* [Perfect-Session-PostgreSQL-Demo](https://github.com/PerfectExamples/Perfect-Session-PostgreSQL-Demo)
* [Perfect-Turnstile-PostgreSQL-Demo](https://github.com/PerfectExamples/Perfect-Turnstile-PostgreSQL-Demo)
```

MySQL:

```
* [MySQLStorm-Demo](https://github.com/PerfectExamples/MySQLStorm-Demo)
* [Perfect-Turnstile-MySQL-Demo](https://github.com/PerfectExamples/Perfect-Turnstile-MySQL-Demo)
* [Perfect-Session-MySQL-Demo](https://github.com/PerfectExamples/Perfect-Session-MySQL-Demo)
```

#### CouchDB:

```
* [CouchDBStORM-Demo](https://github.com/PerfectExamples/CouchDBStORM-Demo)
* [Perfect-Session-CouchDB-Demo](https://github.com/PerfectExamples/Perfect-Session-CouchDB-Demo)
* [Perfect-Turnstile-CouchDB-Demo](https://github.com/PerfectExamples/Perfect-Turnstile-CouchDB-Demo)
```

#### MongoDB:

```
* [MongoDBStORM-Demo](https://github.com/PerfectExamples/MongoDBStORM-Demo)
* [Perfect-Session-MongoDB-Demo](https://github.com/PerfectExamples/Perfect-Session-MongoDB-Demo)
* [Perfect-Turnstile-MongoDB-Demo](https://github.com/PerfectExamples/Perfect-Turnstile-MongoDB-Demo)
```

#### SQLite

```
* [Perfect-Turnstile-SQLite-Demo](https://github.com/PerfectExamples/Perfect-Turnstile-SQLite-Demo)
* [Perfect-Session-SQLite-Demo](https://github.com/PerfectExamples/Perfect-Session-SQLite-Demo)
```

#### ## StORM Documentation topics

[Setting up a class:](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-Setting-up-a-class.md) How to create a class that inherits all the StORM functionality.

[Saving, Retrieving and Deleting Rows:](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-Saving-Retrieving-and-Deleting-Rows.md) Basic database operations

[StORMCursor:](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-Cursor.md) Managing found sets for result set pagination.

[Inserting rows:](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-Insert.md) More detailed access to inserting rows.

[Updating rows:](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-Update.md) More detailed access to the update process.

[Lifecycle events:](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORMLifecycleEvents.md) Details and examples of the global StORM lifecycle events.

#### ## Datasource specific documentation for StORM

```
* [SQLite3](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-SQLite.md)
* [PostgreSQL](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-PostgreSQL.md)
* [MySQL](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-MySQL.md)
* [Apache CouchDB](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-CouchDB.md)
* [MongoDB](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-MongoDB.md)
```

#### ### Including in your project

When including the dependency in your project's Package.swift dependencies, you will have access to all nested dependencies including the database connector.

To include PostgresStORM:

```
``` swift
.Package(url: "https://github.com/SwiftORM/Postgres-StORM.git", majorVersion: 1)
```

To include MySQLStORM:

```
.Package(url: "https://github.com/SwiftORM/MySQL-StORM.git", majorVersion: 1)
```

To include SQLiteStORM:

```
.Package(url: "https://github.com/SwiftORM/SQLite-StORM.git", majorVersion: 1)
```

To include CouchDBStORM:

```
.Package(url: "https://github.com/SwiftORM/CouchDB-StORM.git", majorVersion: 1)
```

To include MongoDBStORM:

```
.Package(url: "https://github.com/SwiftORM/MongoDB-StORM.git", majorVersion: 1)
```

Remember: after you change your `Package.swift` file, you need to regenerate your Xcode Project file using:

```
swift package generate-xcodeproj
```

## Swift ORM: "StORM"

StORM is a modular ORM for Swift, layered on top of [Perfect](#).

It aims to be easy to use, but flexible, and maintain consistency between datasource implementations for the user: you, the developer. It tries to allow you to write great code without worrying about the details of how to interact with the database.

## Relevant Examples

---

PostgresSQL

- [PostgreStORM-Demo](#)
- [Perfect-Session-PostgreSQL-Demo](#)
- [Perfect-Turnstile-PostgreSQL-Demo](#)

MySQL:

- [MySQLStorm-Demo](#)
- [Perfect-Turnstile-MySQL-Demo](#)
- [Perfect-Session-MySQL-Demo](#)

CouchDB:

- [CouchDBStORM-Demo](#)
- [Perfect-Session-CouchDB-Demo](#)
- [Perfect-Turnstile-CouchDB-Demo](#)

MongoDB:

- [MongoDBStORM-Demo](#)
- [Perfect-Session-MongoDB-Demo](#)
- [Perfect-Turnstile-MongoDB-Demo](#)

SQLite

- [Perfect-Turnstile-SQLite-Demo](#)
- [Perfect-Session-SQLite-Demo](#)

## StORM Documentation topics

---

[Setting up a class](#): How to create a class that inherits all the StORM functionality.

[Saving, Retrieving and Deleting Rows](#): Basic database operations

[StORMCursor](#): Managing found sets for result set pagination.

[Inserting rows](#): More detailed access to inserting rows.

[Updating rows](#): More detailed access to the update process.

[Lifecycle events](#): Details and examples of the global StORM lifecycle events.

## Datasource specific documentation for StORM

---

- [SQLite3](#)
- [PostgreSQL](#)
- [MySQL](#)
- [Apache CouchDB](#)
- [MongoDB](#)

### Including in your project

When including the dependency in your project's Package.swift dependencies, you will have access to all nested dependencies including the database connector.

To include PostgresStORM:

```
.Package(url: "https://github.com/SwiftORM/Postgres-StORM.git", majorVersion: 1)
```

To include MySQLStORM:

```
.Package(url: "https://github.com/SwiftORM/MySQL-StORM.git", majorVersion: 1)
```

To include SQLiteStORM:

```
.Package(url: "https://github.com/SwiftORM/SQLite-StORM.git", majorVersion: 1)
```

To include CouchDBStORM:

```
.Package(url: "https://github.com/SwiftORM/CouchDB-StORM.git", majorVersion: 1)
```

To include MongoDBStORM:

```
.Package(url: "https://github.com/SwiftORM/MongoDB-StORM.git", majorVersion: 1)
```

Remember: after you change your `Package.swift` file, you need to regenerate your Xcode Project file using:

```
swift package generate-xcodeproj
```

## Setting up a class to use StORM

The first thing you need to do is import the Library:

```
import PostgresStORM
// or
import SQLiteStORM
```

When you create a new class that will model a database table, it should inherit from a parent StORM class. This will provide most of the goodness you will need.

If you are using PostgreSQL:

```
class User: PostgresStORM {
}
```

If you are using SQLite:

```
class User: SQLiteStORM {
}
```

Next, you will need to add properties to the class. They will mirror closely your columns in the associated table. It's strongly recommended that you keep non-standard characters out of the column names and therefore property names too.

The first property should be your table's primary key. Convention is that this column is called `id` but in reality it can be any valid column name. Common data types for primary keys in SQL datasources are integers, strings, and UUIDs. If your primary key is not an auto-incrementing integer sequence, take care with setting your id and maintaining integrity.

```
// NOTE: First param in class should be the ID.
var id          : Int = 0
var firstname   : String = ""
var lastname    : String = ""
var email       : String = ""
```

You will see above that the default values are set for each property rather than set via an `init()`. This is simply to make this explanation easier. If you choose to write your own `init()` function, include `super.init()` and beware of the need to add the connection property.

### Specifying the table

The table name is to be manually specified in a function. This is to avoid any possible collision with the introspection that is performed on the class for database operations.

Include a function to specify the table as follows:

```
override open fun table() -> String {
    return "users"
}
```

## Class-specific assignment functions

In order for the class to do certain things, such as taking a result set from the database and converting the rows to an array of classed objects, add two functions to the class:

```
override fun to(_ this: StORMRow) {
    id          = this.data["id"] as? Int          ?? 0
    firstname    = this.data["firstname"] as? String ?? ""
    lastname     = this.data["lastname"] as? String ?? ""
    email        = this.data["email"] as? String   ?? ""
}

fun rows() -> [User] {
    var rows = [User]()
    for i in 0..
```

Note that in the `to` function, you assign each property. This has the added benefit that at this stage validation or any other special processing can be added.

The `rows` function is simply an array processor. Copy-paste this function and change the 3 occurrences of the class name.

## Putting it all together

The following is an example "complete" PostgresStORM implementation of a class:

```
import PostgresStORM

class User: PostgresStORM {
    // NOTE: First param in class should be the ID.
    var id          : Int = 0
    var firstname   : String = ""
    var lastname    : String = ""
    var email       : String = ""

    override open func table() -> String {
        return "users"
    }

    override func to(_ this: StORMRow) {
        id          = this.data["id"] as? Int          ?? 0
        firstname   = this.data["firstname"] as? String ?? ""
        lastname    = this.data["lastname"] as? String ?? ""
        email       = this.data["email"] as? String    ?? ""
    }

    func rows() -> [User] {
        var rows = [User]()
        for i in 0..

```

For the purposes of this documentation the class "User" from "[Setting up a class](https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/StORM-Setting-up-a-class.md)" will be used for most examples.

### ## Creating and Updating Rows

The act of creating and saving a record is remarkably similar: The `save` method will create a new row if the primary key property is 0 or empty, otherwise it will attempt to execute an update.

To create a new row:

```
``` swift
let obj          = User()
obj.firstname    = "Joe"
obj.lastname     = "Smith"

try obj.save {
    id in obj.id = id as! Int
}
```

Note that the `.email` property was omitted in the initial save. Let's fix that:

```
obj.email = "joe.smith@example.com"
try obj.save()
```

Because the `.id` property was set, an update is performed.

## Creating a new record that already has the Primary Key set

There are some situations where the you will wish to set the primary key, and the `.save` method above will not work in this case.

Here, you will need to invoke the `.create()` method instead.

This forces an insert instead, using your value for the primary key.

```
let obj          = User()
obj.id           = 10001
obj.firstname    = "Mister"
obj.lastname     = "PotatoHead"
obj.email        = "potato@example.com"
try obj.create()
```

## Capturing Errors

The simple form of the `.save` uses `try`, but the result *can* be ignored. However it's good practice to trap for these errors and act accordingly.

The initial `.save` could be handled instead like this:

```
do {
  try obj.save {id in obj.id = id as! Int }
} catch {
  // Inform the developer using console logging:
  print("There was an error: \(error)")
  // Do something about it in code.
}
```

## Retrieving Rows

There are three ways to retrieve one or more rows: `.get`, `.find`, and `.select`.

### Get Methods

The `.get` methods will attempt to retrieve rows via an exact primary key column match.

```
let obj = User()
try obj.get(1)
print("User's name: \(obj.firstname) \(obj.lastname)")
```

The user id can also be set before the `.get` is performed:

```
let obj      = User()
obj.id       = 2
try obj.get()
print("User's name: \(obj.firstname) \(obj.lastname)")
```

### The Find Method

`.find` will perform an exact match on the name/value pairs supplied.

```
let obj = User()
try obj.find([( "firstname", "Joe" )])
print("Find Record:  \(obj.id), \(obj.firstname), \(obj.lastname)")
```

### Using Select

By far the most powerful of the row retrieval methods is `.select`.

The select method will take several input forms; the simplest is:

```
try obj.select(
  whereclause: "firstname = $1",
  params: [ "Joe" ],
  orderby: [ "id" ]
)
```

The `.select` method takes advantage of parameter binding to protect from SQL injection. Actually all methods do, but only a few core methods expose this and require knowledge of how parameter binding works.

In the previous example, the parameter `$1` is used to signal that the first item in the array fed to `.params` should be used. The substitution will protect for data type and potential nasty input.

The `.select` method also will allow more specific search options such as LIKE and other comparison operators. Knowledge of SQL becomes necessary beyond the basics.

The additional options available in the `.select` method are:

```
columns:      [String],
cursor:       StORMCursor
```

- `columns` allows an array of column names to be supplied which will target specific data that will be returned from the query.
- `cursor` is a `StORMCursor` object that specifies the number of rows to return and the offset from the 1st record. This is often used for pagination.

## Deleting Rows

Deleting a row is done in a similar way to the `.get` method: it can be done by supplying a primary key, or by deleting the currently held row.

```
let obj = User()
try obj.delete(1)
```

```
let obj = User()
obj.id = 1
try obj.delete()
```

## StORM Cursor

In database terms, a cursor defines the size and location of the returned rows within the context of the complete found set.

For a request using StORM, the cursor governs the number of rows returned and the position of the cursor in the total possible result set.

For the response, this information is echoed but the total number of possible rows in the found set is also populated.

Practically, this means that for a request you set the number of rows and the offset from the first record that you wish to retrieve:

```
let thisCursor = StORMCursor(
  limit: 50,
  offset: 100
)
```

And in response the number of rows, the offset position and the total count is echoed back. This allows you to calculate pagination through the found set as required.

```
print(thisCursor.limit)
// 50
print(thisCursor.offset)
// 100
print(thisCursor.totalRecords)
// 1045
```

This cursor object is passed to a `.select` method, and returned as part of all objects using the database-specific StORM class inheritance. # Inserting a Row with StORM

In addition to the `.save()` method, StORM provides access to a more granular set of `.insert` functionality.

This can be used to rapidly insert larger number of rows without having to populate each property in advance.

It is also worth noting that StORM's `.save` functions are effectively convenience functions for `.insert`.

The three "forms" of `.insert` are:

```
insert([(String, Any)])
insert(cols: [String], params: [Any])
insert(cols: [String], params: [Any], idcolumn: String)
```

`insert([(String, Any)])` will take the name/value pairs and insert them into the table. It will infer the id column and return its value.

`insert(cols: [String], params: [Any])` performs the same operation, but with different input supplied.

`insert(cols: [String], params: [Any], idcolumn: String)` allows you to specify the primary key column. The resulting primary key is returned.

In each case, if the primary key value is specified, a successful insert will echo the supplied value.

## Using Insert

To insert a new row and return the auto-generated primary key:

```
var obj = User()
obj.id = try obj.insert(
    cols: ["firstname", "lastname", "email"],
    params: ["Donkey", "Kong", "donkey.kong@mailinator.com"]
) as! Int
```

To insert a new row and return the supplied primary key:

```
var obj = User()
obj.id = try obj.insert(
    cols: ["id", "firstname", "lastname", "email"],
    params: ["10001", "Donkey", "Kong", "donkey.kong@mailinator.com"]
) as! Int
```# Updating Rows with StORM
```

In a way similar to StORM's ``.insert`` methods, you can use ``.update`` methods to specify updates directly.

The two forms of ``.update`` available are:

```
``` swift
update(
    cols: [String],
    params: [Any],
    idName: String,
    idValue: Any
)

update(
    data: [(String, Any)],
    idName: String = "id",
    idValue: Any
)
```

The only difference between these two is the format of the data supplied.

## Using Update

Let's look at an example of how to create, then update a row:

```
let obj          = User()
obj.firstname    = "Joe"
obj.lastname     = "Smith"

// First, lets create a new row
try obj.save {
    id in obj.id = id as! Int
}

// Now, we change the values
obj.firstname = "Mickey"
obj.lastname = "Mouse"
obj.email = "Mickey.Mouse@mailinator.com"

try obj.update(
    cols: ["firstname", "lastname", "email"],
    params: [obj.firstname, obj.lastname, obj.email],
    idName: "id",
    idValue: obj.id
)
```

While the above illustrates the direct update process, in this case calling `try obj.save()` would be leaner and just as effective as the ``.update``.

Where an ``.update`` would be more effective, is a situation where you know all the information, including the primary key value, before any fetch from the datasource.

This way, an update would be called directly without any initial round trip connection to the database.

```

let obj          = User()
try obj.update(
    cols: ["firstname", "lastname", "email"],
    params: ["Mickey", "Mouse", "Mickey.Mouse@mailinator.com"],
    idName: "id",
    idValue: 100001
)

```# PostgresStORM

## Relevant Examples

* [PostgresStORM-Demo](https://github.com/PerfectExamples/PostgresStORM-Demo)
* [Perfect-Session-PostgreSQL-Demo](https://github.com/PerfectExamples/Perfect-Session-PostgreSQL-Demo)
* [Perfect-Turnstile-PostgreSQL-Demo](https://github.com/PerfectExamples/Perfect-Turnstile-PostgreSQL-Demo)

## Including in your project

When including the dependency in your project's Package.swift dependencies, you will have access to all nested dependencies including the database connector.

``` swift
.Package(url: "https://github.com/SwiftORM/Postgres-Storm.git", majorVersion: 1)

```

Please note that you may need to include the XML libraries for your platform as documented in <https://www.perfect.org/docs/xml.html>

## A note about protecting from SQL Injection attacks

StORM does its best to protect your data from SQL injection attacks by parameterizing values.

However in some methods it is possible to specify raw data or SQL query components, so it is important to take care to validate *all* incoming data prior to use in a SQL context.

## Creating a connection to your database

In order to connect to your database you will need to specify certain information.

```

PostgresConnector.host      = "localhost"
PostgresConnector.username  = "username"
PostgresConnector.password  = "secret"
PostgresConnector.database  = "yourdatabase"
PostgresConnector.port      = 5432

```

Once your connection information is specified it can be used in the object class to create the connection on demand.

```
let obj = User()
```

## PostgresStORM supported methods

### Connecting

`PostgresConnector` - Sets the connection parameters for the PostgreSQL server access. Host, port, username and password information, as well as the default database need to be specified.

### Creating tables

`setup()` - Creates the table by inspecting the object. Columns will be created that relate to the assigned type of the property. Properties beginning with an underscore or "internal\_" will be ignored.

**NOTE:** The **primary key** is first property defined in the class.

### Specifying a custom table creation statement

`setup(String)` - Specify a SQL statement to manually define the table creation process. This will override any StORM setup statement.

### Saving objects

`save()` - Saves object. Creates a new row if no primary key is assigned, otherwise performs an update.

`save {id in ... }` - Saves object. Creates a new row if no primary key is assigned, otherwise performs an update. The closure returns the new id if created.

`create()` - Forces the creation of a new row, even if a primary key has been supplied.

## Retrieving data

`findAll()` - Retrieves all rows in the table, only limited by the cursor (9,999,999 rows)

`get()` - Retrieves the row. Assumes primary key has been set in the object.

`get(Any)` - Retrieves the row with primary key supplied.

`find([(String, Any)])` - Performs a find, where the pairs supplied are name/value pairs corresponding to column names and find criteria.

`select(whereclause: String, params: [Any], orderby: [String])` - Performs a select with a specified where clause, having the values parameterized to protect from SQL injection. The orderby array is an array of column names that can optionally include the "ASC" or "DESC" keywords to indicate sort direction.

Additionally the `select` can include:

- `cursor: StORMCursor` - The optional `cursor` object is a [StORMCursor](#)
- `columns: [String]` - The optional `columns` is an array of column names to return in the result set. If not included this will default to all columns in the table.

## Deleting objects

`delete()` - Deletes the current object. Assumes an assigned primary key.

`delete(Any)` - Delete the object with primary key supplied.

`delete(Int, idname)` - Delete, specifying a key value, with the name of the key column.

`delete(String, idname)` - Delete, specifying a key value, with the name of the key column.

`delete(UUID, idname)` - Delete, specifying a key value, with the name of the key column.

## Inserts

Each `insert` method will attempt to return the new primary key value.

`insert([(String, Any)])` - Insert a new row specifying data as `[(String, Any)]` where the first in the pair is the column name, and the second is the value.

`insert(cols: [String], params: [Any])` - Insert a new row specifying data as matching arrays of column names, and the associated value.

`insert(cols: [String], params: [Any], idcolumn: String)` - As above, but specifying the id column name to return.

## Updates

`update(data: [(String, Any)], idName: String = "id", idValue: Any)` - Updates the row with the specified data, with a primary key value, and an optional idName column name.

`update(cols: [String], params: [Any], idName: String, idValue: Any)` - Updates the row by specifying data as matching arrays of column names, and the associated value. A primary key value, and an idName column name are also required.

## Upserts

`upsert(cols: [String], params: [Any], conflictkeys: [String])` - Inserts the row with the specified data; on conflict (conflictkeys columns) it will perform an update.

## SQL Statements

`sql(String, params: [String])` - Execute Raw SQL statement with parameter binding from the params array. Returns raw `PGResult` object.

`sqlRows(String, params: [String])` - Execute Raw SQL statement with parameter binding from the params array. Returns an array of `[StORMRow]`.

# SQLiteStORM

## Relevant Examples

- [Perfect-Turnstile-SQLite-Demo](#)
- [Perfect-Session-SQLite-Demo](#)

## Including in your project

When including the dependency in your project's Package.swift dependencies, you will have access to all nested dependencies including the database connector.

```
.Package(url: "https://github.com/SwiftORM/SQLite-StORM.git", majorVersion: 1)
```

### A note about protecting from SQL Injection attacks

StORM does its best to protect your data from SQL injection attacks by parameterizing values.

However in some methods it is possible to specify raw data or SQL query components, so it is important to take care to validate *all* incoming data prior to use in a SQL context.

## Creating a connection to your database

In order to connect to your database you will need to specify the path to the database file.

```
SQLiteConnector.db = "./mydb"
```

Once your connection information is specified it can be used in the object class to create the connection on demand.

```
let obj = User()
```

## SQLiteStORM supported methods

### Connecting

`SQLiteConnector.db` - Sets the connection parameters for the SQLite3 database file access.

### Creating tables

`setup()` - Creates the table by inspecting the object. Columns will be created that relate to the assigned type of the property. Properties beginning with an underscore or "internal\_" will be ignored.

**NOTE:** The **primary key** is first property defined in the class.

### Saving objects

`save()` - Saves object. Creates a new row if no primary key is assigned, otherwise performs an update.

`save {id in ... }` - Saves object. Creates a new row if no primary key is assigned, otherwise performs an update. The closure returns the new id if created.

`create()` - Forces the creation of a new row, even if a primary key has been supplied.

### Retrieving data

`findAll()` - Retrieves all rows in the table, only limited by the cursor (9,999,999 rows)

`get()` - Retrieves the row. Assumes primary key has been set in the object.

`get(Any)` - Retrieves the row with primary key supplied.

`find([(String, Any)])` - Performs a find, where the pairs supplied are name/value pairs corresponding to column names and find criteria.

`select(whereclause: String, params: [Any], orderby: [String])` - Performs a select with a specified where clause, having the values parameterized to protect from SQL injection. The orderby array is an array of column names that can optionally include the "ASC" or "DESC" keywords to indicate sort direction.

Additionally the `select` can include:

- `cursor: StORMCursor` - The optional `cursor` object is a [StORMCursor](#)

- `columns: [String]` - The optional `columns` is an array of column names to return in the result set. If not included this will default to all columns in the table.

## Deleting objects

`delete()` - Deletes the current object. Assumes an assigned primary key.

`delete(Any)` - Delete the object with primary key supplied.

`delete(Int, idname)` - Delete, specifying a key value, with the name of the key column.

`delete(String, idname)` - Delete, specifying a key value, with the name of the key column.

`delete(UUID, idname)` - Delete, specifying a key value, with the name of the key column.

## Inserts

`insert([(String, Any)])` - Insert a new row specifying data as `[(String, Any)]` where the first in the pair is the column name, and the second is the value.

`insert(cols: [String], params: [Any])` - Insert a new row specifying data as matching arrays of column names, and the associated value.

## Updates

`update(data: [(String, Any)], idName: String = "id", idValue: Any)` - Updates the row with the specified data, with a primary key value, and an optional idName column name.

`update(cols: [String], params: [Any], idName: String, idValue: Any)` - Updates the row by specifying data as matching arrays of column names, and the associated value. A primary key value, and an idName column name are also required.

## SQL Statements

`sqlExec(String)` - Execute Raw SQL statement.

`sql(String, params: [String])` - Execute Raw SQL statement with parameter binding from the params array. Returns an array of [SQLiteStmt].

`sqlAny(String, params: [String])` - Execute Raw SQL statement with parameter binding from the params array. Returns an ID column.

`sqlRows(String, params: [String])` - Execute Raw SQL statement with parameter binding from the params array. Returns an array of [StORMRow].

# MySQLStORM

## Relevant Examples

- [MySQLStorm-Demo](#)
- [Perfect-Turnstile-MySQL-Demo](#)
- [Perfect-Session-MySQL-Demo](#)

## Including in your project

When including the dependency in your project's Package.swift dependencies, you will have access to all nested dependencies including the database connector.

```
.Package(url: "https://github.com/SwiftORM/MySQL-StORM", majorVersion: 1)
```

## A note about protecting from SQL Injection attacks

StORM does its best to protect your data from SQL injection attacks by parameterizing values.

However in some methods it is possible to specify raw data or SQL query components, so it is important to take care to validate *all* incoming data prior to use in a SQL context.

## Creating a connection to your database

In order to connect to your database you will need to specify certain information.

```
MySQLConnector.host      = "localhost"
MySQLConnector.username  = "username"
MySQLConnector.password  = "secret"
MySQLConnector.database  = "yourdatabase"
MySQLConnector.port      = 3306
```

Once your connection information is specified it can be used in the object class to create the connection on demand.

```
let obj = User()
```

## MySQLStORM supported methods

### Connecting

`MySQLConnector` - Sets the connection parameters for the MySQL server access. Host, port, username and password information, as well as the default database need to be specified.

### Creating tables

`setup()` - Creates the table by inspecting the object. Columns will be created that relate to the assigned type of the property. Properties beginning with an underscore or "internal\_" will be ignored.

**NOTE:** The **primary key** is first property defined in the class.

### Specifying a custom table creation statement

`setup(String)` - Specify a SQL statement to manually define the table creation process. This will override any StORM setup statement.

### Saving objects

`save()` - Saves object. Creates a new row if no primary key is assigned, otherwise performs an update.

`save {id in ... }` - Saves object. Creates a new row if no primary key is assigned, otherwise performs an update. The closure returns the new id if created.

`create()` - Forces the creation of a new row, even if a primary key has been supplied.

### Retrieving data

`findAll()` - Retrieves all rows in the table, only limited by the cursor (9,999,999 rows)

`get()` - Retrieves the row. Assumes primary key has been set in the object.

`get(Any)` - Retrieves the row with primary key supplied.

`find([ (String, Any) ])` - Performs a find, where the pairs supplied are name/value pairs corresponding to column names and find criteria.

`select(whereclause: String, params: [Any], orderby: [String])` - Performs a select with a specified where clause, having the values parameterized to protect from SQL injection. The orderby array is an array of column names that can optionally include the "ASC" or "DESC" keywords to indicate sort direction.

Additionally the `select` can include:

- `cursor: StORMCursor` - The optional `cursor` object is a [StORMCursor](#)
- `columns: [String]` - The optional `columns` is an array of column names to return in the result set. If not included this will default to all columns in the table.

### Deleting objects

`delete()` - Deletes the current object. Assumes an assigned primary key.

`delete(Any)` - Delete the object with primary key supplied.

`delete(Int, idname)` - Delete, specifying a key value, with the name of the key column.

`delete(String, idname)` - Delete, specifying a key value, with the name of the key column.

`delete(UUID, idname)` - Delete, specifying a key value, with the name of the key column.

## Inserts

Each `insert` method will attempt to return the new primary key value.

`insert([(String, Any)])` - Insert a new row specifying data as `[(String, Any)]` where the first in the pair is the column name, and the second is the value.

`insert(cols: [String], params: [Any])` - Insert a new row specifying data as matching arrays of column names, and the associated value.

`insert(cols: [String], params: [Any], idcolumn: String)` - As above, but specifying the id column name to return.

## Updates

`update(data: [(String, Any)], idName: String = "id", idValue: Any)` - Updates the row with the specified data, with a primary key value, and an optional idName column name.

`update(cols: [String], params: [Any], idName: String, idValue: Any)` - Updates the row by specifying data as matching arrays of column names, and the associated value. A primary key value, and an idName column name are also required.

## Upserts

`upsert(cols: [String], params: [Any], conflictkeys: [String])` - Inserts the row with the specified data; on conflict (conflictkeys columns) it will perform an update.

## SQL Statements

`sql(String, params: [String])` - Execute Raw SQL statement with parameter binding from the params array. Returns raw `PGResult` object.

`sqlRows(String, params: [String])` - Execute Raw SQL statement with parameter binding from the params array. Returns an array of `[StORMRow]`.

# CouchDBStORM

## Relevant Examples

- [CouchDBStORM-Demo](#)
- [Perfect-Session-CouchDB-Demo](#)
- [Perfect-Turnstile-CouchDB-Demo](#)

## Including in your project

When including the dependency in your project's Package.swift dependencies, you will have access to all nested dependencies including the database connector.

```
.Package(url: "https://github.com/SwiftORM/CouchDB-Storm.git", majorVersion: 1)
```

## Creating a connection to your database

In order to connect to your database you will need to specify certain information.

```
CouchDBConnection.host = "localhost"
CouchDBConnection.username = "username"
CouchDBConnection.password = "secret"
CouchDBConnection.port = 5984
CouchDBConnection.ssl = true
```

Once your connection information is specified it can be used in the object class to create the connection on demand.

```
let obj = User()
```

## CouchDBStORM supported methods

### Connecting

`CouchDBConnection` - Sets the connection parameters for the CouchDB server access. Host, username and password information need to be specified. Port and SSL status need only be specified if different from the default (SSL false, and port 5984).

## Creating databases

The working database for the object is set by embedding this function in the object, and changing the name to the desired database name:

```
override open func database() -> String {
    return "mydbname"
}
```

`setup()` - Creates the database.

**NOTE:** The **primary key** is first property defined in the class.

## Saving objects

`save(rev: String = "")` - Saves object. If an ID has been defined, `save()` will perform an update, otherwise a new document is created. Takes an optional "rev" parameter which is the document revision to be used. If empty the object's stored `_rev` property is used.

`save(rev: String = "") {id in ... }` - Saves object. If an ID has been defined, `save()` will perform an update, otherwise a new document is created. Takes an optional "rev" parameter which is the document revision to be used. If empty the object's stored `_rev` property is used. The closure returns the new id if created.

`create()` - Forces the creation of a new database object. The revision property is also set once the save has been completed.

## Retrieving data

`get()` - Retrieves the document. Assumes id has been set in the object.

`get(String)` - Retrieves a document with a specified id.

`find([String: Any])` - Performs a find using the selector syntax. For example, `try find(["username": "joe"])` will find all documents that have a username equal to "joe". Full selector syntax can be found at <http://docs.couchdb.org/en/2.0.0/api/database/find.html#find-selectors>

Additionally the `find` can include:

- `cursor: StORMCursor` - The optional `cursor` object is a [StORMCursor](#)

## Deleting objects

`delete()` - Deletes the current object. Assumes the id and `_rev` properties are set.

# MongoDBStORM

## Relevant Examples

- [MongoDBStORM-Demo](#)
- [Perfect-Session-MongoDB-Demo](#)
- [Perfect-Turnstile-MongoDB-Demo](#)

## Including in your project

When including the dependency in your project's Package.swift dependencies, you will have access to all nested dependencies including the database connector.

```
.Package(url: "https://github.com/SwiftORM/MongoDB-Storm.git", majorVersion: 1)
```

## Creating a connection to your database

In order to connect to your database you will need to specify certain information.

```

MongoDBConnection.host = "localhost"
MongoDBConnection.port = 27017
MongoDBConnection.ssl = true
MongoDBConnection.database = "mydb"

// Authentication credentials.
// Only required if authModeType = .standard
MongoDBConnection.username = "username"
MongoDBConnection.password = "secret"
MongoDBConnection.authdb = "authenticationSource"

// Authentication mode:
// .none (default), or .standard
MongoDBConnection.authModeType = .none

```

Once your connection information is specified it can be used in the object class to create the connection on demand.

```
let obj = User()
```

## MongoDBStORM supported methods

### Connecting

`MongoDBConnection` - Sets the connection parameters for the MongoDB server access. Host, and database information need to be specified. Port and SSL status need only be specified if different from the default (SSL false, and port 27017).

### Creating collections

The working collection for the object is set by embedding this function in the object, and changing the name to the desired collection name:

```

override init() {
    super.init()
    _collection = "users_demo"
}

```

Note that unlike other StORM implementations, it is not required to invoke a `setup()` function call to ensure the collection is created. MongoDB will create the collection by default if it does not exist at the first data insertion attempt.

**NOTE:** The **primary key** is first property defined in the class.

### Saving objects

`save()` - Saves object. If an ID has been defined, `save()` will perform an update, otherwise a new document is created.

### Retrieving data

`get()` - Retrieves the document. Assumes id has been set in the object.

`get(String)` - Retrieves a document with a specified id.

`find([String: Any])` - Performs a find. For example, `try find([ "username": "joe" ])` will find all documents that have a username equal to "joe".

Additionally the `find` can include:

- `cursor: StORMCursor` - The optional `cursor` object is a [StORMCursor](#)

### Deleting objects

`delete()` - Deletes the current object. Assumes the id property is set.