Урок 4

Серверный Swift для mock-сервера

Проект на стороне сервера. Perfect. Обработка запроса на сервере.

Серверный Swift

Большинству iOS-разработчиков рано или поздно приходится выходить за рамки клиентского мира. А необходимость эта кроется в клиент-серверной архитектуре, которую поддерживает большинство мобильных приложений. И серверная часть преимущественно закрыта для мобильных разработчиков. И даже если она открыта, что-нибудь в ней поменять можно только очень медленно, через согласование с серверными разработчиками, которые ревностно относятся к требованиям что-то изменить в API. Даже если попробовать самим реализовать серверную часть для проверки функционала, нужно будет окунуться в другой язык (PHP, Python, Ruby, C#) и фреймворки. Но все не так плохо — есть набор web-серверов, которые написаны на Swift и используют уже известные нам фреймворки.

Что даст подобный подход с использованием web-серверов, написанных на Swift?

- Независимость от серверной части — но не забываем про функциональное и UI-тестирование;
- Знакомый язык — Swift. Не нужно переучиваться, чтобы использовать другие языки и фреймворки.
- Знакомые фреймворки (Foundation...) — нужно отметить, что не все фреймворки из iOS SDK будут доступны (например, UIKit), но можно использовать другие, направленные на серверную разработку.

Доступные фреймворки web-серверов

Рассмотрим, какие фреймворки web-серверов нам доступны. На сегодняшний день самые распространенные — **Perfect**, **Vapor**, **Kitura**, **Zewo**.

Perfect

Из документации: «Perfect — это идеальное решение для многих популярных веб-приложений и приложений, доступных в iTunes». Громко сказано, но все же Perfect — действительно мощный фреймворк, который покроет все требования для разработки облегченных, поддерживаемых и масштабируемых приложений. Perfect позволяет создавать **REST API** сервисы, используя исключительно Swift. Perfect — opensource-проект.

По Perfect есть довольно обширная и подробная документация — www.perfect.org. Также много информации, сравнений и туториалов можно найти на многочисленных форумах, посвященных фреймворку.

Perfect — наиболее динамичный и популярный серверный инструментарий, на котором мы остановим внимание и ниже покажем пример его использования.

Vapor

Vapor — web-фреймворк для Swift. Он позволит писать web-приложения, сайты, API, используя **HTTP** либо **WebSockets**. По заявлению разработчиков, этот фреймворк до 100 раз быстрее других популярных альтернатив, написанных на Ruby и PHP. По Vapor есть много официальной документации, можно подробнее узнать о настройке сервера, использовании **WebSockets**. Ресурс от разработчиков — vapor.university, на котором можно найти много информации, учебников и статей.

Если чему и стоит уделить свое внимание после Perfect, это Vapor, который тоже является opensource-проектом.

Kitura

Kitura — бесплатный opensource web-фреймворк на Swift. Разработан компанией IBM и предоставляется под лицензией Apache 2.0. Kitura предоставляет простое развертывание на облачных платформах, таких как IBM Cloud, с уже установленными пакетами Docker. Документация — kitura.io. Но Kitura «медленнее» своих оппонентов Vapor и Perfect — по количеству ответов, отданных в секунду времени.

Zewo

В Zewo используется философия Go (это компилируемый многопоточный язык программирования): «Don't communicate by sharing memory. Share memory by communicating». В Zewo используются **coroutines** (основанные на **libdill**), являющиеся по своей сути **single-threaded** (однопоточными). Это означает, что больше не нужно беспокоиться о взаимных блокировках или гонках (**race conditions**). Код становится по умолчанию безопаснее. Zewo — это open source проект, предоставляемый под MIT-лицензией. Документация представлена на ресурсе zewo.io.

Perfect

Вернемся к web-фреймворку Perfect, который будем использовать далее. Еще раз о причинах выбора в его пользу:

- Быстрый — большая скорость отдачи пакетов в секунду;
- Простой — чтобы писать и разворачивать;
- Большое комьюнити — можно быстро найти ответ на интересующий вопрос;
- Хорошая документация — прозрачность и ясность API.

MVC

Perfect использует в своей основе паттерн **MVC** — стандартное разбиение на Model/View/Controller.

- **Model** — это данные, сервисы для их обработки и хранилища;
- **View** — это html-шаблон, куда передаются переменные из контроллера. Он может и не понадобиться, так как мы будем сразу обрабатывать входящие запросы и отдавать по ним ответы;
- **Controller** — сущность, которая обрабатывает запрос пользователя и подготавливает ответ. Именно она перехватит и обработает запрос от мобильного приложения.

Обработка запросов

Рассмотрим, как Perfect производит разбор запроса. Ведь HTTP-запрос — это, по сути, большой набор текстовой информации.

```
GET /data/2.5/forecast?q=Moscow,DE HTTP/1.1
Host: samples.openweathermap.org
```

```
Connection: close
User-Agent: Paw/3.1.5 (Macintosh; OS X/10.13.4) GCDHTTPRequest
```

Разбором этой информации занимается **Middleware**, преобразовывая все данные и предоставляя удобное api для доступа к ним. **Middleware** — это фильтры и преобразователи запросов на пути к обработчику. Далее преобразованные данные попадают в **Route**, который переадресует запрос к методу-обработчику определенного класса.

Route — это маршрут, определяющий список запросов, которые будут обрабатываться, а также инструкция, что и как их будет обрабатывать.

Установка Perfect

Согласно документации perfect.org, в терминале выполним небольшой список команд. Для начала нужно скачать проект Perfect с помощью Git. Для этого перейдем в каталог установки и выполним команду:

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

В результате мы скачаем проект Perfect. Далее перейдем в каталог **PerfectTemplate** и произведем сборку проекта.

```
cd PerfectTemplate
swift build
```

Проект будет собран, а web-сервер — готов к использованию. В сборке участвует много C-библиотек, что добавляет скорости в работе Perfect.

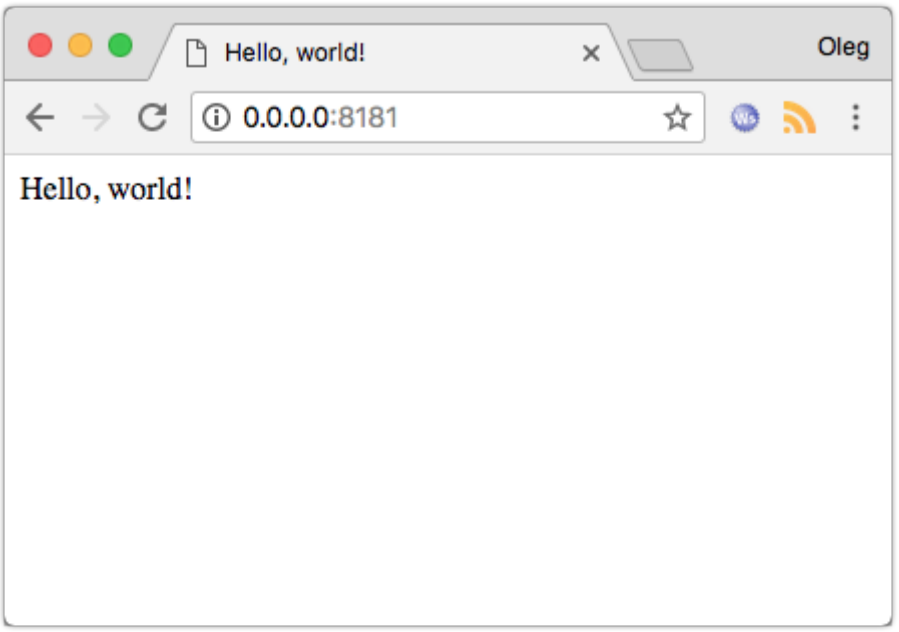
Запускаем сервер.

```
.build/debug/PerfectTemplate
```

При успешном запуске сервера в терминале увидим запись:

```
[INFO] Starting HTTP server localhost on 0.0.0.0:8181
```

Значит, сервер успешно запущен и доступен по адресу **0.0.0.0:8181**, который можем запустить в браузере. Сделав это, увидим приветственную строку от Perfect.

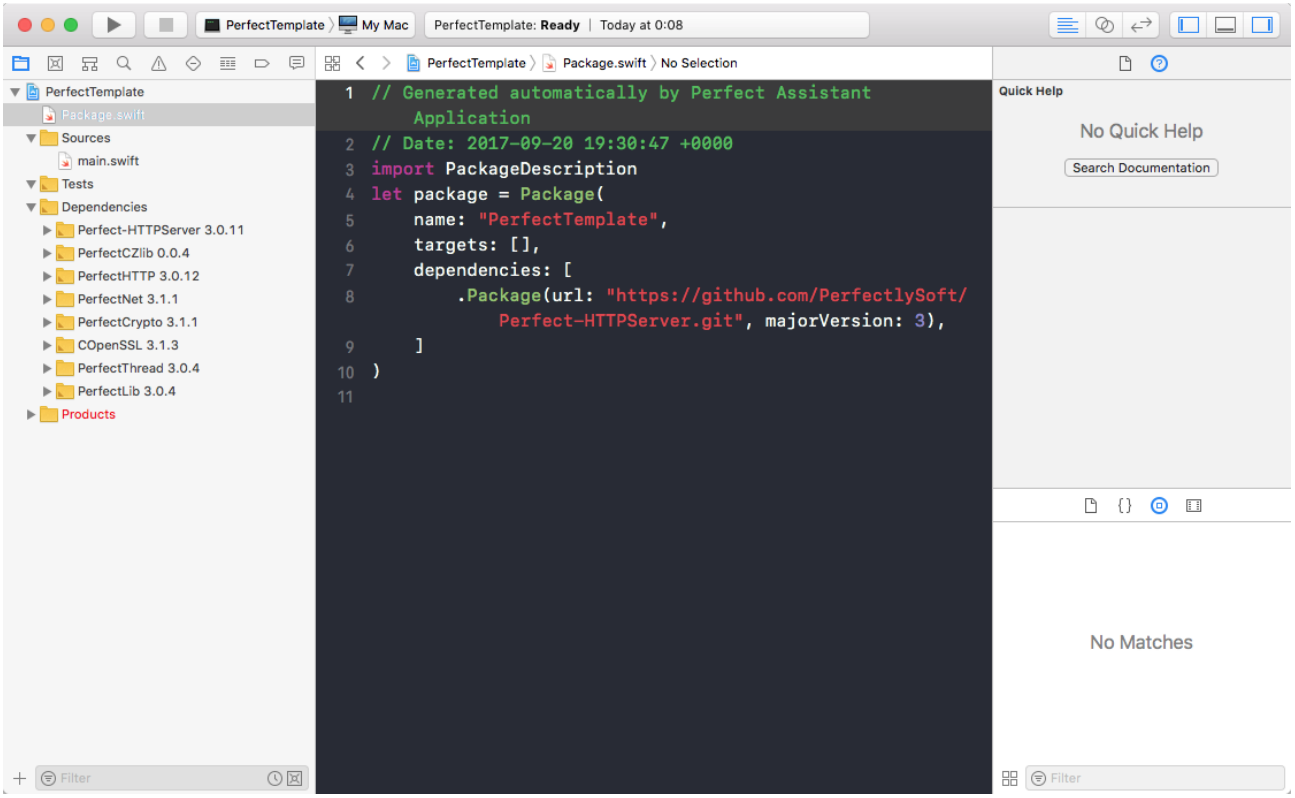


Завершить работу можно, нажав в терминале с запущенным сервером комбинацию клавиш **Ctrl+C** или просто закрыв терминал.

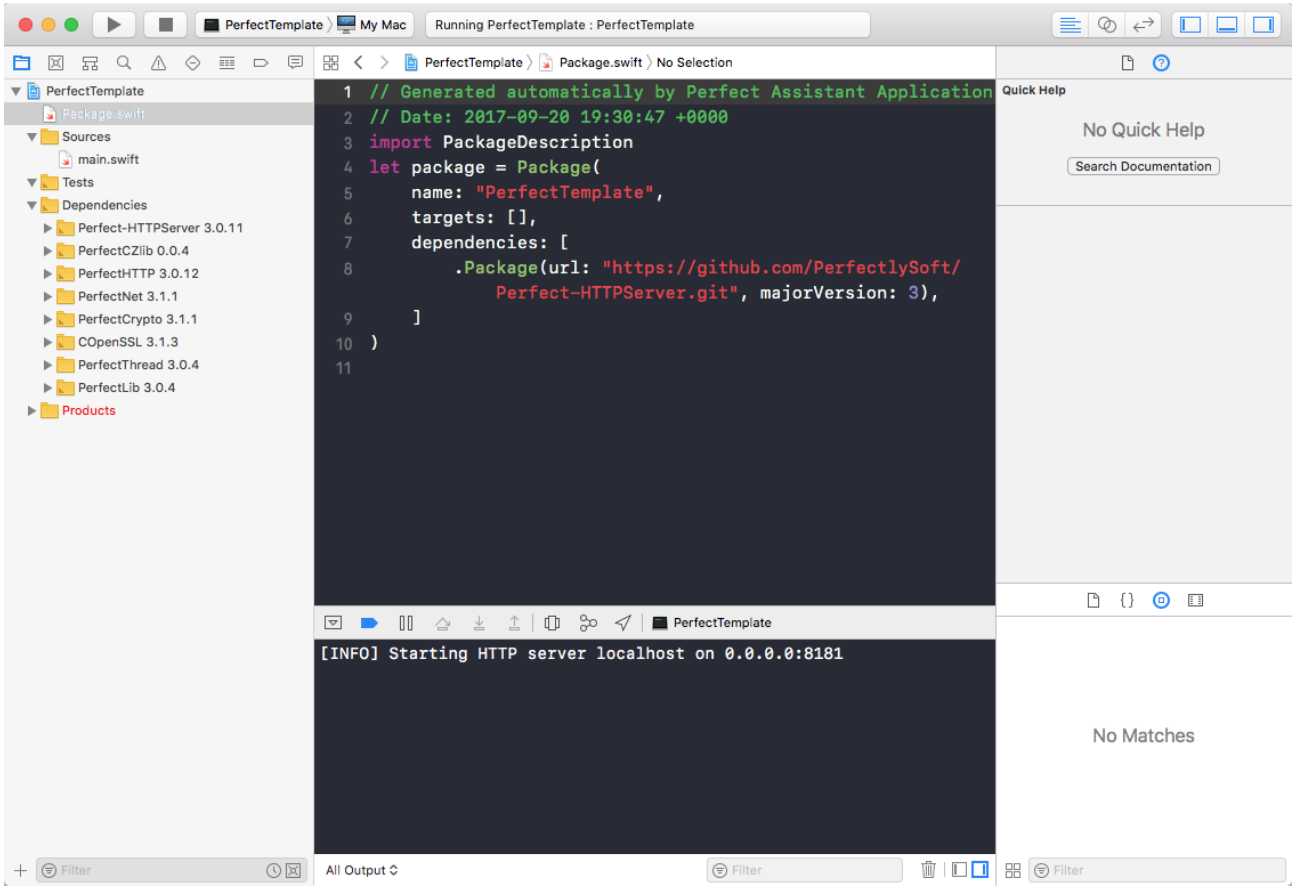
Чтобы разрабатывать сервер в знакомой среде Xcode, производя манипуляции с кодом и запуском сервера непосредственно из нее, воспользуемся командой:

```
swift package generate-xcodeproj
```

В результате получим **PerfectTemplate.xcodeproj**. Откроем его:

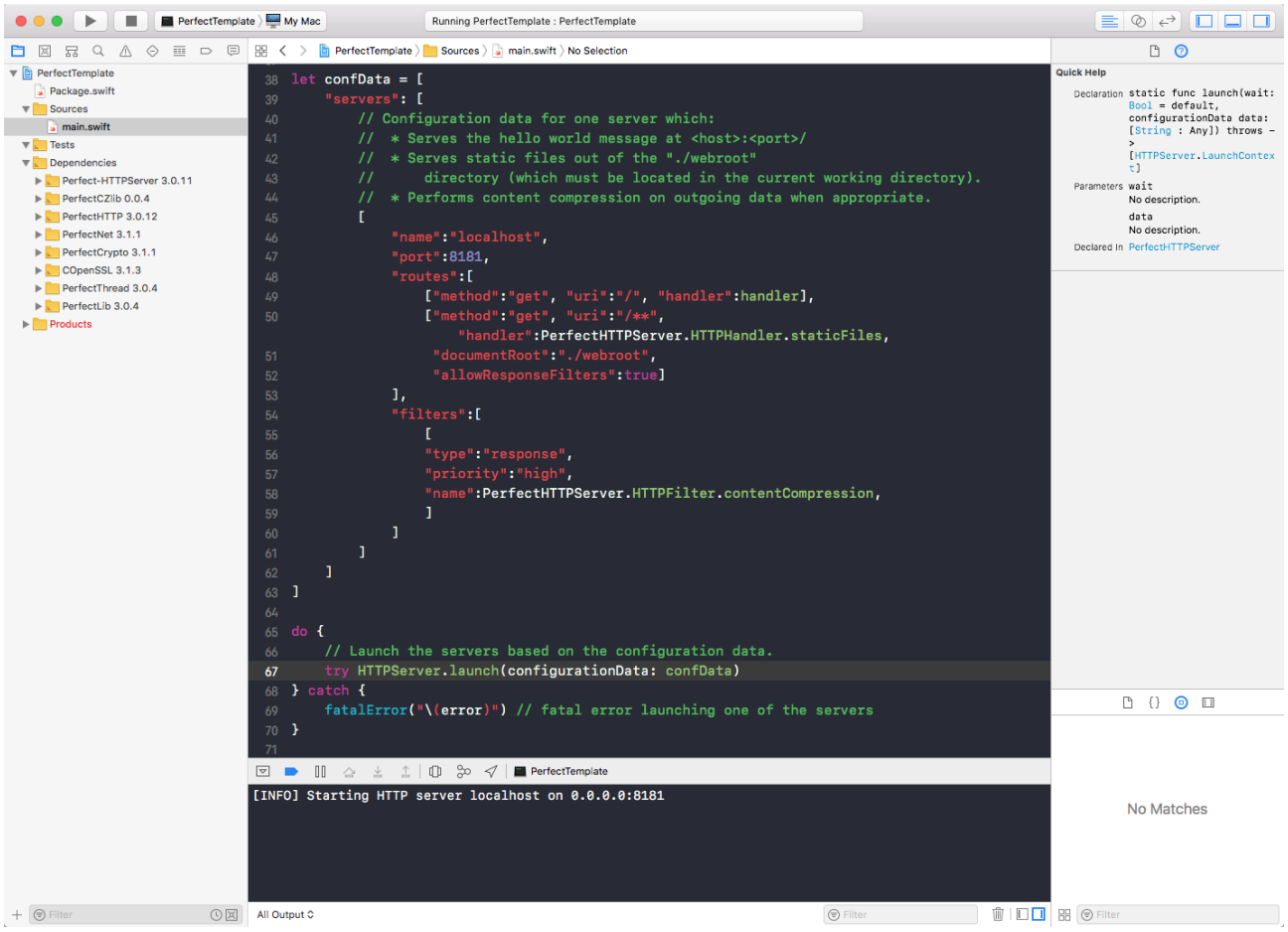


Получили знакомый Xcode-проект. При выборе схемы **PerfectTemplate** и запуске ее на **My Mac** увидим тот же самый запуск сервера, что и при работе через терминал. И в логах Xcode будет выведена уже известная нам строка **‘[INFO] Starting HTTP server localhost on 0.0.0.0:8181’**.



Настройка Perfect

Изменить настройки Perfect можно из словаря **confData** файла **./Sources/main.swift**.



Разберем основные параметры настройки web-сервера Perfect.

- **name** — обязательный параметр со строковым значением. В основном используется для идентификации сервера и, как правило, совпадает с именем домена сервера. Perfect будем использовать преимущественно как локальный сервер, поэтому **name** будет принимать значения **‘0.0.0.0’**, **‘127.0.0.1’** или **‘localhost’**;

- **port** — обязательный параметр с целочисленным значением, указывающий порт, который «слушает» наш сервер. TCP-порты варьируются от 0 до 65535. Для портов в диапазоне от 0 до 1024 требуются права **root**. По умолчанию порт **Perfect** использует 8181. Но данный порт может быть занят другим сервисом, в этом случае значение **port** нужно изменить;

- **filters** — фильтры могут отображать или обрабатывать данные входящих запросов. Например, фильтр проверки подлинности может удостовериться, имеет ли запрос определенные разрешения, а если нет — вернуть ошибку клиенту. Фильтры ответов выполняют то же для исходящих данных, имея возможность изменять заголовки ответов или данные тела. Значение для ключа '**filters**' представляет собой массив словарей, содержащих ключи, которые описывают каждый фильтр. Необходимыми ключами для этих словарей являются '**type**' и '**name**'. Возможные значения для ключа '**type**' — '**request**' или '**response**', чтобы указать запрос либо фильтр ответов. Ключ '**priority**' может принимать значения '**high**' (высокий), '**medium**' (средний) или '**low**' (низкий). Если приоритет не указан, значение по умолчанию будет '**high**'. По умолчанию представлен фильтр сжатия ответа, что является базовой настройкой для многих серверов и экономит трафик «дешевым» сжатием:

```
"filters":[
    [
    "type":"response",
```

```
    "priority":"high",
    "name":PerfectHTTPServer.HTTPFilter.contentCompression,
    ]
]
```

- **routes** — этот необязательный элемент, массив словарей [**String: Any**]. Каждый элемент массива указывает маршрут **URI** или группу **URI**, которые отображают входящий HTTP-запрос и передают его обработчику **handler**.

GBShop

По умолчанию нам предоставлен обработчик для URI "/", который предоставляет в качестве ответа html-текст (**<html><title>Hello, world!</title><body>Hello, world!</body></html>**), который мы видели при вводе '**0.0.0.0:8181**' в браузер. Реализуем обработчик нашего первого запроса — о регистрации нового клиента.

Обработка запроса регистрации GBShop

Вспомним об API (API Description.xlsx). Данные запроса — это данные в json-формате, пример:

```
{
"id_user" : 123
"username" : "Somebody",
"password" : "mypassword",
"email" : "some@some.ru",
"gender": "m",
"credit_card" : "9872389-2424-234224-234",
"bio" : "This is good! I think I will switch to another language"
}
```

Ответ должен прийти тоже в формате **json**, с кодом **1** — при успешной регистрации и с кодом **0** — при ошибке. Пример:

```
{ result: 1, userMessage: "Регистрация прошла успешно!" }
{ result: 0, errorMessage : "Сообщение об ошибке" }
```

Приступаем к кодированию. Выделим класс модели запроса **RegisterRequest**. Для разбора json-тела запроса будем использовать JSONSerialization, поэтому предусмотрим в нашем классе инициализатор:

```
import Foundation

struct RegisterRequest {
    var id_user: Int = 0
    var username: String = ""
    var password: String = ""
    var email: String = ""
    var gender: String = ""
    var credit_card: String = ""
    var bio: String = ""

    init(_ json: [String: AnyObject]) {
        if let id_user = json["id_user"] as? Int {
```

```
            self.id_user = id_user
        }
        if let username = json["username"] as? String {
            self.username = username
        }
        if let password = json["password"] as? String {
            self.password = password
        }
        if let email = json["email"] as? String {
            self.email = email
        }
        if let gender = json["gender"] as? String {
            self.gender = gender
        }
        if let credit_card = json["credit_card"] as? String {
            self.credit_card = credit_card
        }
        if let bio = json["bio"] as? String {
            self.bio = bio
        }
    }
}
```

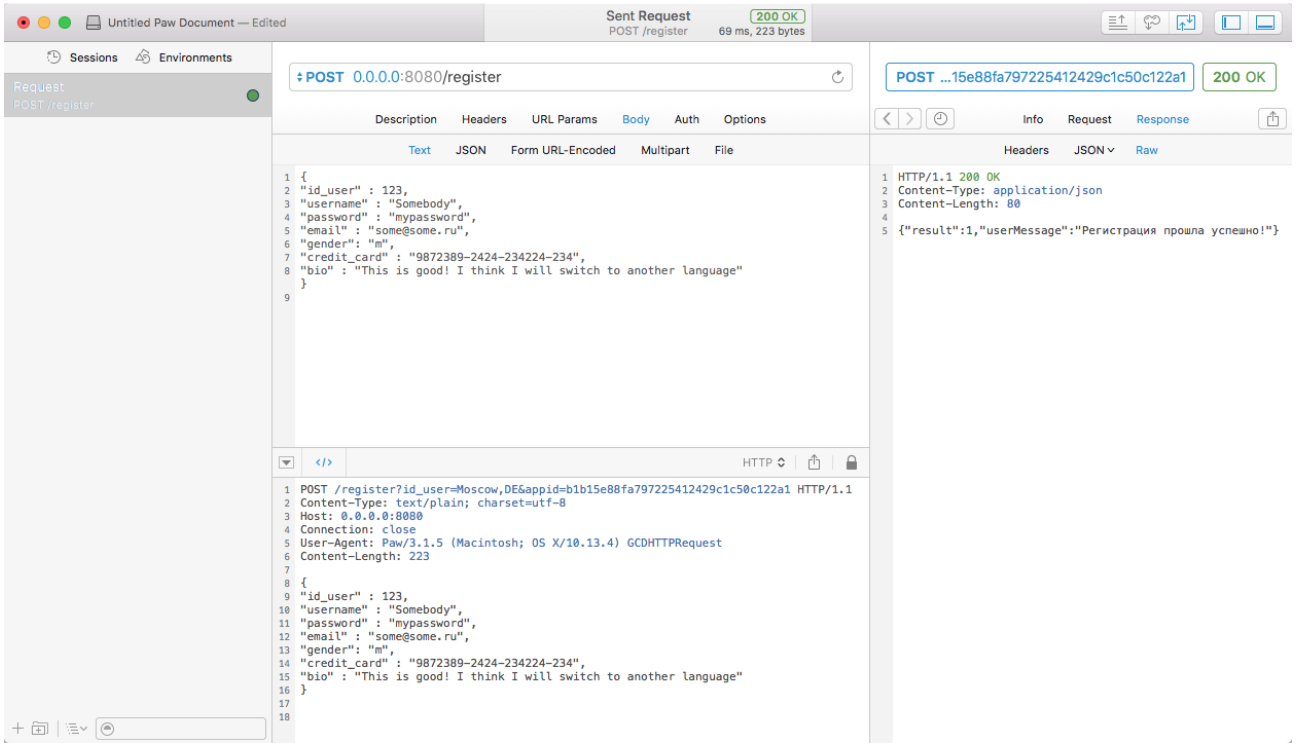
Выделим отдельный класс обработки HTTP-запроса (HTTPRequest) на регистрацию **AuthController**, используя **API Perfect**:

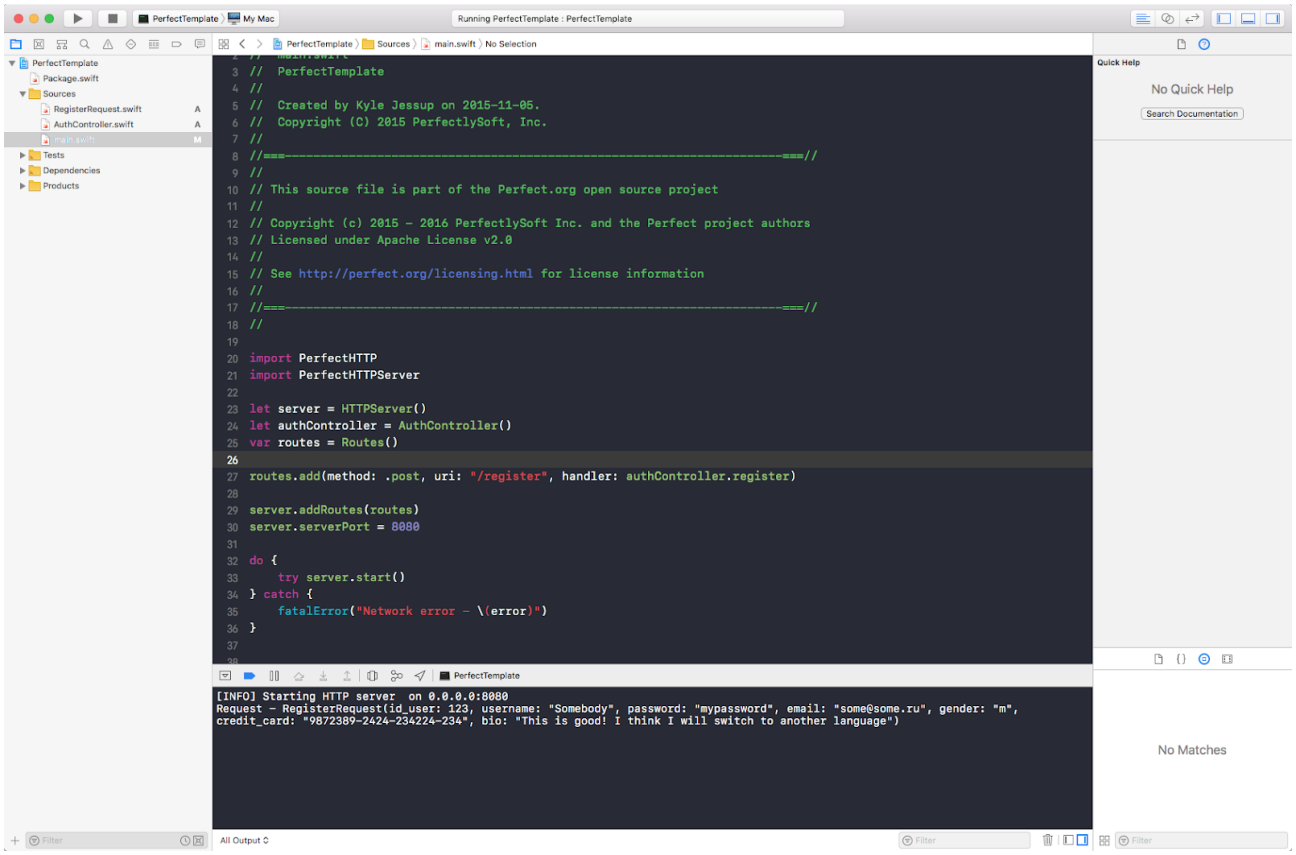
```
import Foundation
import PerfectHTTP

class AuthController {

    let register: (HTTPRequest, HTTPResponse) -> () =  { request, response in
        guard let str = request.postBodyString, let data = str.data(using:
.utf8) else {
            response.completed(status: HTTPResponseStatus.custom(code: 500,
message: "Wrong user data"))
            return
        }

        do {
            let json = try JSONSerialization.jsonObject(with: data, options: [])
as! [String: AnyObject]
            let registerRequest = RegisterRequest(json)
            print("Request - \(registerRequest)")

            try response.setBody(json: ["result": 1, "userMessage": "Регистрация
прошла успешно!"])
            response.completed()
        } catch {
            response.completed(status: HTTPResponseStatus.custom(code: 500,
message: "Parse data error - \(error)"))
        }
    }
}
```

Убираем все лишнее из файла **main.swift**:

```
import PerfectHTTP
import PerfectHTTPServer

let server = HTTPServer()
let authController = AuthController()
var routes = Routes()

routes.add(method: .post, uri: "/register", handler: authController.register)

server.addRoutes(routes)
server.serverPort = 8080

do {
      try server.start()
} catch {
      fatalError("Network error - \(error)")
}
```

Запускаем сервер и отправляем запрос на него (например, через приложение **Paw**):





Видим, что запрос успешно разобрался и в ответ мы получили нужные данные.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 80

{"result":1,"userMessage":"Регистрация прошла успешно!"}
```

Практическое задание

1. Реализовать mock-сервер для тестирования приложения.
2. Отныне и до конца курса ваше приложение должно работать подключаясь к вашему мок серверу.
3. Моки можете брать из файлов с API.
4. Недостающие методы добавляете сами.

Дополнительные материалы

1. GeekBrainsTutorial/online-store-api.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. www.perfect.org.
2. vapor.university.
3. www.kitura.io.
4. zewo.io.