ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук

Департамент программной инженерии

|  |  |
| --- | --- |
|  | УТВЕРЖДАЮ  Академический руководитель образовательной программы «Программная инженерия»,  профессор департамента программной инженерии, канд. техн. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Шилов  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. |

Выпускная квалификационная работа

на тему «Сервер для мокирования асинхронных протоколов»

по направлению подготовки 09.03.04 «Программная инженерия»

|  |  |
| --- | --- |
| Научный руководитель  профессор  департамента программной инженерии ФКН НИУ ВШЭ  кандидат технических наук  Гринкруг Е.М.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Подпись, Дата | Выполнил  студент группы БПИ152  4 курса бакалавриата  образовательной программы  «Программная инженерия»  Данилин П.И.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Подпись, Дата |

Москва 2019

# Реферат

Использование мок-серверов при разработке приложений дает разработчику определенные возможности. Среди них: разработка прототип программы без настоящего сервера и воспроизведение поведения клиентского приложения в зависимости от заданного поведения мок-сервера. На данный момент существуют решения для мокирования HTTP сервисов. Такие решения получают от пользователя конфигурацию поведения сервера, обычно это JSON, и на заданном порту отдают ответы на HTTP запросы согласно описанной конфигурации. Однако текущие решения для мокирования асинхронных протоколов WebSocket и Server-Sent Events не поддерживают их асинхронную природу. В результате данной работы был разработан сервер для мокирования протоколов WebSocket и Server-Sent Events.

Работа содержит \* страниц, 3 главы, \* рисунков, \* источников, 4 приложения.

**Ключевые слова:** сервер, мокирование, асинхронный протокол, WebSocket, Server-Sent Events.

# Abstract

Using a mock server in development gives a developer a certain amount of possibilities. Among them are prototyping without a real server or testing a client application depending on a given mock server behavior. There are implemented solutions for HTTP services mocking currently. Such solutions receive a behavior configuration from a user, which is usually a JSON configuration, and give out responses to HTTP requests according to the given configuration. But contemporary solutions for mocking asynchronous protocols like WebSocket and Server-Sent Events do not support their asynchronous nature. In the course of this work we developed a server for mocking protocols WebSocket and Server-Sent Events.

The paper contains \* pages, 3 chapters, \* images, \* references, \* appendices.

**Keywords:** server, mocking, stubbing, asynchronous protocol, WebSocket, Server-Sent Events.

**Содержание**

[Реферат 2](#_Toc8596437)

[Abstract 3](#_Toc8596438)

[Определения 6](#_Toc8596439)

[Введение 7](#_Toc8596440)

[Актуальность работы 7](#_Toc8596441)

[Цель и задачи работы 8](#_Toc8596442)

[Структура выпускной квалификационной работы 8](#_Toc8596443)

[Глава 1. Описание асинхронных протоколов и существующих аналогов решения 9](#_Toc8596444)

[1. Описание асинхронных протоколов 9](#_Toc8596445)

[1.1. WebSocket 9](#_Toc8596446)

[1.2. Server-Sent Events 11](#_Toc8596447)

[2. Описание существующих аналогов решения 11](#_Toc8596448)

[Глава 2. Архитектура решения 14](#_Toc8596449)

[1. API сервера 14](#_Toc8596450)

[2. Особенности используемых библиотек 16](#_Toc8596451)

[3. Пример работы с решением 19](#_Toc8596452)

[Глава 3. Технические детали реализации 21](#_Toc8596453)

[1. Компоненты реализации 21](#_Toc8596454)

[2. Основные абстракции реализации 22](#_Toc8596455)

[2.1. AdminMockController 22](#_Toc8596456)

[2.2. MockEndpointManager 23](#_Toc8596457)

[2.3. MockConfiguration 24](#_Toc8596458)

[2.4. StubConfiguration 24](#_Toc8596459)

[2.5. ResponseStub 24](#_Toc8596460)

[2.6. HttpPredicate 25](#_Toc8596461)

[2.7. Response 26](#_Toc8596462)

[2.8. SseEventsResponse 26](#_Toc8596463)

[2.9. WebSocketEventsResponse 26](#_Toc8596464)

[2.10. WsReaction 26](#_Toc8596465)

[2.11. WsEventPredicate 26](#_Toc8596466)

[2.12. Routable 26](#_Toc8596467)

[2.13. DelayedSource 28](#_Toc8596468)

[Заключение 29](#_Toc8596469)

[Список источников 30](#_Toc8596470)

# Определения

**Мокирование** – задание содержания и формата ответа на сетевой запрос к серверу. В объектно-ориентированном программировании мокирование – создание объектов-заглушек вместо настоящих объектов путем имитации их актуальных свойств [1]. Объект-заглушка на вызов метода с определенными аргументами возвращает заданный пользователем ответ.

**Сервер** – программа, обрабатывающая запросы от других программ по сети.

**WebSocket** – дуплексный протокол для взаимодействия двух программ по сети.

**Server-Sent Events** – симплексный протокол взаимодействия двух программ по сети, в котором только одна сторона посылает сообщения.

# Введение

## Актуальность работы

На текущий момент при разработке программных систем часто используются клиент-серверный и микросервисный подходы к архитектуре таких систем. В упомянутых подходах есть сервисы-потребители и сервисы-провайдеры. Сервис-потребитель обращается к сервису-провайдеру по сети для получения данных или выполнения какого-то действия. В подобных условиях разработчикам сервиса-потребителя может быть полезным использовать мок-сервер в качестве замены сервиса-провайдера. Особенно уместным использование сервера для мокирования может быть в следующих сценариях. Например, сервис-провайдер еще не разработан или не поддерживает работу с данным сервисом-потребителем. Тогда разработчики могут договориться и сконфигурировать мок-сервер, чтобы он отвечал на запросы сервиса-клиента, как будто он и есть сервис-провайдер. Разработчики смогут дорабатывать свои приложения без ожидания доработок других сервисов. Также с помощью сервера для мокирования можно сильно ускорить создание прототипа приложения, поскольку часть доработок будет идти параллельно, а часть можно отбросить и заменить на конфигурацию мок-сервера. Помимо уже упомянутых ситуаций, мок-сервер может использоваться при тестировании сервиса-потребителя. Тогда не потребуется создание условий (таких, как создание пользователя с нужными ролями или заполнение базы данных) для сервиса-провайдера, чтобы получить от него нужный ответ. Достаточно сконфигурировать мок-сервер, что сделать потенциально проще и быстрее.

Так как одним из наиболее распространенных протоколов взаимодействия сервисов является HTTP, есть несколько популярных реализаций мок-серверов для данного протокола. Среди них Mountebank [8], MockServer [9], WireMock [10]. Данные реализации дают возможность отвечать на заданных портах согласно описанной конфигурации. Однако упомянутые решения работают только с протоколами HTTP(S) и TCP. Существует еще набор асинхронных протоколов, для которых есть потребность в подобном мок-сервере. В результате поиска подобного решения для протоколов WebSocket и Server-Sent Events был сделан вывод, что текущие решения обладают сильными недостатками, которые необходимо преодолеть для удовлетворения потребностей пользователей.

## Цель и задачи работы

Цель моей работы – разработать сервер для мокирования протоколов WebSocket и Server-Sent Events. Для ее достижения необходимо выполнить следующие задачи:

1. Изучить имеющиеся аналоги серверов для мокирования;
2. Изучить особенности протоколов WebSocket и Server-Sent Events;
3. Реализовать сервер для мокирования WebSocket и Server-Sent Events;
4. Подготовить техническую документацию;
5. Выложить исходный код приложения и техническую документацию на GitHub под открытой лицензией.

## Структура выпускной квалификационной работы

В первой главе описаны особенности протоколов WebSocket, Server-Sent Events, а также недостатки существующих решений. Во второй главе можно найти общую архитектуру решения и описание особенностей используемых библиотек. В третьей главе находится подробное описание технической стороны решения. В заключении описаны полученные результаты и возможные пути развития проекта.

# Глава 1. Описание асинхронных протоколов и существующих аналогов решения

В данной главе описаны основные особенности асинхронных протоколов WebSocket и Server-Sent Events, а также решения, позволяющие получить сервер для мокирования данных протоколов, и их недостатки.

## Описание асинхронных протоколов

### WebSocket

WebSocket – дуплексный протокол для двух взаимодействующих сервисов, основанный на TCP. Данный протокол задумывался в качестве замены существующих вариантов двусторонней коммуникации, которые использовали HTTP в качестве транспортного протокола. Среди них, например, long polling [11]. Long polling работает следующим образом. Клиент в цикле посылает запросы на сервер. Сервер отдает ответ только по истечении определенного интервала времени или когда происходит определенное событие. Клиент при получении ответа сразу же посылает повторный запрос на сервер. Такой подход имеет следующие недостатки: лишняя передача заголовков в запросах, необходимость установления HTTP соединения, необходимость правильного выбора таймаута для запроса [11]. Протокол WebSocket не имеет перечисленных проблем.

Далее будут описаны базовые концепции протокола WebSocket. В качестве источника используется RFC 6455 [12]. Взаимодействие участников коммуникации осуществляется в 3 этапа: установление соединения, передача сообщений, завершение коммуникации.

Установление соединения осуществляется инициатором путем отправки HTTP запроса определенного формата. Наглядно пример такого запроса можно увидеть на рис. 1. Данный запрос должен иметь метод GET, а также определенные заголовки, про состав и назначение которых можно подробнее прочитать в стандарте [12].



Рисунок 1. Пример запроса для установления WebSocket соединения [12]

При успешном установлении соединения сервер выдает ответ, формат которого можно увидеть на рис. 2.

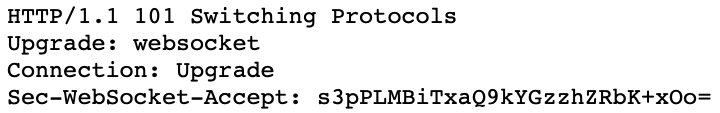


Рисунок 2. Пример ответа для установления WebSocket соединения [12]

После успешного установления соединения стороны могут начинать обмениваться сообщениями. Сообщение состоит из так называемых фреймов, которые используют TCP фреймы для передачи по сети. За счет того, что полезная информация передается по TCP, при передаче информации не происходит передачи лишних заголовков, Cookie, как было бы в случае использования HTTP в качестве транспорта для двустороннего взаимодействия. В протоколе также описан механизм проверки «жизнеспособности» второй стороны с помощью механизма Ping-Pong.

Для завершения соединения отсылается особый Close фрейм, получив который сторона прекращает отправку и получение любых других фреймов.

### Server-Sent Events

Server-Sent Events – симплексный протокол, отдающий фрагментированный HTTP ответ. Работает он следующим образом. Клиент посылает обычный HTTP запрос на сервер. Сервер возвращает ответ, в заголовках Transfer-Encoding: chunked, Content-Type: text/event-stream и не закрывает соединение сразу, передавая тело ответа по сети частями. Сообщения от сервера приходят в теле ответа текстовом формате в кодировке UTF-8. Разделитель между сообщениями – два символа переноса строки, идущие подряд. Сообщение обязательно имеет данные и опционально – id, event. Свойства сообщения разделяются символом перевода строки. Имя свойства от значения отделяется двоеточием. Пример ответа можно увидеть на рис. 3.

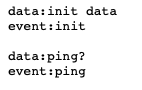


Рисунок 3. Пример Server-Sent Event ответа сервера

Сообщения могут быть обработаны до закрытия соединения сервера, поэтому клиент может реактивно реагировать на приходящие события, не дожидаясь завершения HTTP ответа. В основном данный протокол используется для получения уведомлений от сервера.

## Описание существующих аналогов решения

На текущий момент существует несколько распространенных серверов для мокирования. Наиболее популярные – Mountebank [8], MockServer [9], WireMock [10], работают с мокированием HTTP взаимодействия. В данных приложениях отсутствует явная поддержка WebSocket [13; 14]. Однако в обсуждении продукта Mountebank есть предложение, как поддержку можно реализовать самому [14]. Для этого требуется использовать утилиту Websockify, которая занимается установлением WebSocket соединения и его прерыванием и проксирует фреймы передачи сообщений в Mountebank. Это возможно, поскольку Mountebank умеет работать с мокированием TCP. Таким образом, для решения потребуется иметь работающий Mountebank сервер и утилита Websockify. В Mountebank потребуется задать поведение сервера - указать правила для ответа на фреймы, которые приходят от второй стороны WebSocket соединения. У приведенного решения есть ряд проблем. Во-первых, WebSocket сервер может инициировать отправку сообщений в отсутствие сообщений от клиента, что в данной схеме реализовать невозможно (сообщения будут отправляться только в ответ на сообщения клиента). Причиной данной проблемы является то, что Mountebank умеет мокировать только ответы, совершенные в ответ на запрос клиента. Во-вторых, такой вариант тяжело поддерживать, поскольку требуется работать с протоколом на низком уровне TCP фреймов. Для реализации сценария, когда надо по-разному отвечать на разный вид сообщений в WebSocket, описать imposter для Mountebank будет нетривиально. Вариантов реализации сервера для мокирования WebSocket с помощью других существующих решений не было выявлено.

Так как протокол Server-Sent Events является частью спецификации HTTP с определенным набором заголовков и форматом тела ответа, упомянутые мок-сервера для HTTP, способны работать с данным протоколом. Однако есть одна проблема. Все решения отдают тело ответа на запрос полностью, а не частями. То есть, пользователь может сконфигурировать мок-сервер так, чтобы тот на определенный запрос отвечал в формате Server-Sent Events. Но ответ на реальный запрос к мок-серверу придет сразу (демонстрация на рис. 4), что может быть недопустимо в некоторых случаях. Например, если браузер отображает уведомления, которые получил в ответе на запрос, то в случае с таким поведением все уведомления будут отображены в браузере сразу (рис. 4), а не по одному с определенным временным интервалом (рис. 5), как, возможно, хотелось бы разработчику, чтобы протестировать функциональность отображения уведомлений.

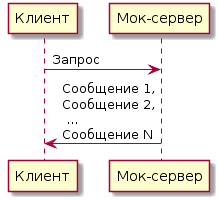


Рисунок 4. Схема фактического Server-Sent Events взаимодействия с мок-сервером

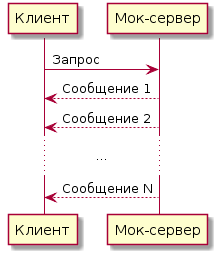


Рисунок 5. Схема желаемого Server-Sent Events взаимодействия с мок-сервером

Таким образом, протоколы WebSocket и Server-Sent Events позволяют асинхронно обмениваться сообщениями между сторонами коммуникации. Современные реализованные утилиты позволяют мокировать данные протоколы лишь со значительными неудобствами и оговорками.

# Глава 2. Архитектура решения

## API сервера

Для управления мок-сервером требуется предоставить интерфейс (API, программный интерфейс приложения). В качестве основного архитектурного стиля для API был выбран REST. Данный выбор был сделан, поскольку REST широко распространен, прост в понимании и не привязан к определенной технологии или языку программирования. REST (Representational State Transfer) – архитектурный стиль для создания распределенных систем [15]. Согласно ему нужно придерживаться определенного подхода к сетевому интерфейсу систем. В качестве транспортного протокола используется HTTP. Основной единицей является ресурс. У ресурса существует идентификатор – URL, унифицированное имя ресурса, например, http://example.com/user/128. Для осуществления операций с ресурсами нужно осмысленно использовать HTTP глаголы. Для операций чтения – GET, создания новых ресурсов – POST, удаления – DELETE. REST был выбран в качестве основы Согласно упомянутым идеям был разработан следующий интерфейс для разрабатываемого сервера для мокирования асинхронных протоколов. Для создания мока на определенном порту используется запрос POST /mock, в теле которого передается JSON объект-конфигурация поведения мока. Для получения текущей конфигурации используется запрос GET /mock/{port}, где вместо {port} требуется указать числовой идентификатор порта запрашиваемого мока. В теле ответа сервер вернет JSON конфигурацию мока на запрошенном порту, если таковой имеется. Чтобы удалить мок на определенном порту используется запрос DELETE /mock/{port}. В результате запроса мок на заданном порту будет удален, а прежняя конфигурация будет в теле ответа сервера на запрос.

Для облегчения работы с сетевым интерфейсом приложения была описана OpenAPI схема приложения [2]. Ее можно найти в файле swagger.yaml в исходных кодах проекта. OpenAPI схема позволяет описать интерфейс приложения и использовать ее как документацию. Также в сервер был встроен дистрибутив Swagger UI, который упрощает демонстрацию интерфейса приложения с OpenAPI схемой и позволяет наглядно предоставить описанную OpenAPI схему в браузере с возможностью взаимодействия с интерфейсом [3]. Пример отображения Swagger UI интерфейса можно увидеть на рис. 6.

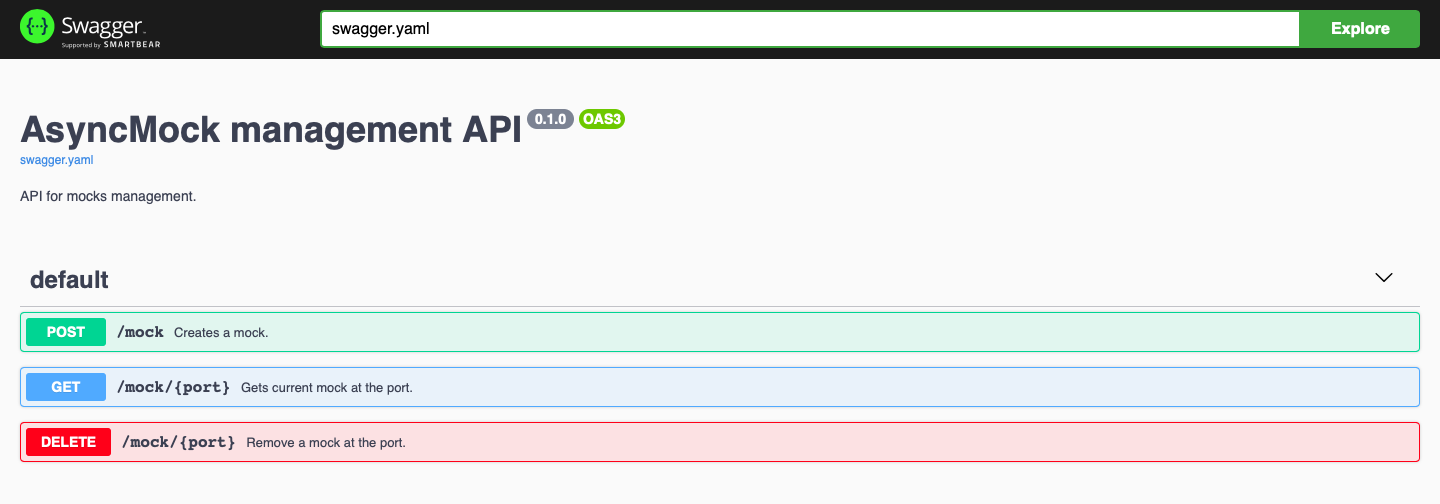


Рисунок 6. Интерфейс Swagger UI, демонстрирующий REST интерфейс приложения

С данным интерфейсом складывается примерно следующая схема работы с сервером. Пользователь описывает мок и создает его на сервере запросом POST /mock. Далее пользователь может проверить мок на заданном порту с помощью запроса GET /mock/{port}. После этого пользователь взаимодействует с созданным моком и при необходимости удаляет свежесозданный мок. Пример схем взаимодействия можно увидеть на рис. 7 и 8.



Рисунок 7. Схема взаимодействия пользователя с сервером для мокирования SSE



Рисунок 8. Схема взаимодействия пользователя с сервером для мокирования WebSocket

## Особенности используемых библиотек

Для работы с запросами к серверу была выбрана библиотека Akka HTTP [4]. Это библиотека для создания HTTP интерфейсов на Scala, которая использует Akka и, соответственно интегрирована для работы с Akka Streams [5]. Данная библиотека позволяет создавать и интерфейсы на основе протоколов WebSocket и Server-Sent Events, что и необходимо для проекта. Библиотека Akka Streams предоставляет абстракции для работы с графами вычислений. В ней существует три основных типа строительных блоков графа. Первый называется Source – блок, у которого есть только один выход, и которые служат источником событий в графе. Еще один тип блоков – Flow. У данного блога есть один вход и один выход. Flow является основным и предназначен для преобразования данных в графе и выполнения действий над сообщениями. Последний основной тип элементов графа – Sink. У него есть только один вход, и служит он для потребления результата вычислений графа. Для примера рассмотрим граф из документации к Akka Streams на рис. 9. Он состоит из четырех блоков. Первый – Source, запускает события в граф из массива чисел 1, 2, 3. Каждое число по очереди попадает во Flow под названием map, в котором производится изменение числа – прибавляется единица. Далее числа проходят через еще один map, в котором умножаются на 2. После этого каждый элемент попадает в Sink ignore, который просто игнорирует все приходящие в него события.

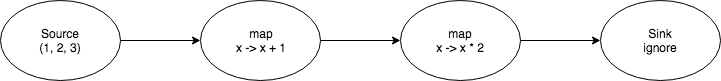


Рисунок 9. Пример графа вычислений Akka Stream <https://doc.akka.io/docs/akka/current/stream/stream-flows-and-basics.html?language=scala>

Абстракция графов вычислений, предоставляемая библиотекой удобна для работы с асинхронными потоковыми протоколами, поскольку у данных видов одинаковое представление. Например, WebSocket взаимодействие естественно ложится на граф, похожий на иллюстрацию на рис. 10.



Рисунок 10. Изображение графа взаимодействия WebSocket

В WebSocket есть события от клиента соединения, которые надо обрабатывать (Flow), возможно, производя события в ответ, при этом также могут быть свои Source-источники событий. В итоге получается граф с событиями между двумя пользователями.

Для протокола Server-Sent Events граф получается гораздо проще, поскольку в нем взаимодействие однонаправленное. Демонстрацию можно увидеть на рис. 11.



Рисунок 11. Изображение графа взаимодействия Server-Sent Events

Для протокола Server-Sent Events требуется предоставить только Source источник событий, которые будут отправлены клиенту по сети.

Одной из особенностей использованных фреймворков для формирования API решения является подход библиотеки Akka HTTP к описанию REST интерфейса приложения. Для одинаковых функций библиотеки вместо переопределения методов используется паттерн магнит. Основная идея данного паттерна заключается в определении метода с параметром, типизированным интерфейсом, в свою очередь параметризованным типом результата метода. Для демонстрации возьмем пример из статьи про данный паттерн одного из авторов библиотеки Akka HTTP [6]. Подобная конструкция работает как привычная перегрузка метода, однако лишена некоторых ее таких недостатков, как коллизия переопределенных методов в результате стирания типов в обобщенных типах, дублирования кода в похожих переопределениях. Объект, передаваемый в метод будет неявно преобразован в реализацию интерфейса параметра метода из неявных методов, определенных в объекте-компаньоне, и передан непосредственно в метод.

**def** complete(magnet: CompletionMagnet): magnet.Result = magnet()  
  
**sealed trait** CompletionMagnet {  
 **type** Result  
 **def** apply(): Result  
}  
  
**object** CompletionMagnet {  
 **implicit def** fromStatusObject[T :Marshaller](tuple: (StatusCode, T)) =  
 **new** CompletionMagnet {  
 **type** Result = Unit  
 **def** apply(): Result = *??? // implementation using (StatusCode, T) tuple* }  
 **implicit def** fromHttpResponseFuture(future: Future[HttpResponse]) =  
 **new** CompletionMagnet {  
 **type** Result = Int  
 **def** apply(): Result = *??? // implementation using future* }  
 **implicit def** fromStatusCodeFuture(future: Future[StatusCode]) =  
 **new** CompletionMagnet {  
 **type** Result = Int  
 **def** apply(): Result = *??? // implementation using future* }  
}

Данный подход к методам библиотеки позволяет красиво и емко описывать сетевой интерфейс приложения. В примере далее можно увидеть описание REST интерфейса управления моками сервера.

**val** *adminRoute*: Route = pathPrefix(**"mock"**) {  
 (post & entity(as[MockConfiguration])) { stub =>  
 onSuccess(endpointManager.addMock(stub)) {  
 complete(StatusCodes.*Created*, **s"Created a mock on port $**{stub.port}**"**)  
 }  
 } ~  
 pathPrefix(IntNumber) { port =>  
 get {  
 complete(endpointManager.getMock(port))  
 } ~  
 delete {  
 complete(endpointManager.deleteMock(port))  
 }  
 }  
}

## Пример работы с решением

Далее будет описан пример работы с сервером для мокирования асинхронных протоколов для более наглядного понимания его сущности.

Запустим сервер. Для мока будем использовать следующую конфигурацию поведения:

{  
 **"port"**: 5000,  
 **"stubs"**: [  
 {  
 **"predicates"**: [],  
 **"responses"**: [  
 {  
 **"events"**: [  
 {  
 **"data"**: **"Hello World!"**,  
 **"delay"**: **"1 second"** }  
 ],  
 **"type"**: **"sse"** }  
 ]  
 }  
 ]  
}

В данной конфигурации сервер посылает одно Server-Sent Events событие с текстом “Hello World!” с задержкой в одну секунду в ответ на запрос на порт сервера 5000. Для того, чтобы добавить данный мок, надо отправить следующий запрос к серверу, например, с помощью консольной утилиты curl [7], для этого в консоли введем:

curl -X POST **"http://localhost:2525/mock"** -H **"accept: \*/\*"** -H **"Content-Type: application/json"** -d **"{\"port\":5000,\"stubs\":[{\"predicates\":[],\"responses\":[{\"events\":[{\"data\":\"Hello World!\",\"delay\":\"1 second\"}],\"type\":\"sse\"}]}]}"**

Убедимся, что сервер корректно добавил мок, и отсылает сообщение “Hello World!” с задержкой в одну секунду также с помощью утилиты curl:

curl localhost:5000

Утилита примерно через секунду печатает в консоль следующий вывод:

data:**"Hello World!"**

Данный текст в теле ответа является корректным Server-Sent Events сообщением с текстом “Hello World!”, что и было задано конфигурацией мока.

# Глава 3. Технические детали реализации

## Компоненты реализации

Реализация содержит четыре основных компоненты. Компонента controller содержит классы и утилиты, работающие с сетевым интерфейсом сервера, логику обработки сетевых запросов на основании абстракций других компонент. В данной компоненте также присутствует логика предоставления интерфейса Swagger UI и Open API схемы.

Компонента manager содержит абстракции для управления поведения сервера на портах – добавление мока, удаление, получение мока по порту.

Компонента configuration содержит классы и утилиты для работы с конфигурациями поведения сервера. Центральная абстракция – MockConfiguration является базовым интерфейсом, реализации которого будут непосредственно являться описанием для задания поведения сервера. Компонента json содержит реализации классов для сериализации и десериализации JSON объектов в объекты MockConfiguration. Компонента route содержит реализации абстракций, которые по данному инстансу MockConfiguration создают route объект, являющийся объектом фреймворка Akka Http для обработки сетевых запросов. То есть, в Routable содержатся реализации объектов, преобразующих описание поведения сервера непосредственно в имплементацию данного поведения. Диаграмму компонентов можно увидеть на рис. 12.

Представленное разделение компоненты configuration необходимо для разделения независимой логики парсинга JSON описания поведения сервера и непосредственной реализации данного поведения. В первом случае используются абстракции библиотеки Circe. В singleton объекте JsonUtils объявлены implicit реализации соответствующих абстракций Encoder и Decoder для всех типов конфигураций поведения сервера. При необходимости их использования достаточно импортировать внутреннее содержимое объекта JsonUtils, и implicit реализации будут использованы для сериализации и десериализации объектов поведения сервера. Механизм Routable не является частью использованных фреймворков. Routable является интерфейсом, реализация которого должна уметь преобразовывать тип в абстракцию Route. Также singletone объект Routable содержит реализации данного интерфейса для всех реализаций MockConfiguration, что позволяет работать с любой конфигурацией поведения сервера.



Рисунок 12. Диаграмма компонентов реализации

## Основные абстракции реализации

Далее будут описаны основные классы, используемые для реализации сервера для мокирования асинхронных протоколов.

### AdminMockController

Данный класс отвечает за обработку HTTP запросов, управляющих работой сервера – добавление моков, удаление, получение мока по порту.

В параметрах конструктора принимает реализацию MockEndpointManager и порт, который приложение будет слушать для обработки входящих запросов по управлению сервером.

Данный класс имеет методы start и stop для соответственно старта сервера и его завершения. При завершении сервера происходит также остановка всех моков с их портов.

При получении метода POST на путь /mock происходит десериализация объекта MockConfiguration из тела запроса и вызов метода addMock реализации MockEndpointManager. При GET и DELETE запросах на путь /mock/{port}, где {port} – числовой идентификатор порта сервера, происходит вызов методов getMock и deleteMock реализации MockEndpointManager соответственно.

### MockEndpointManager

Данный интерфейс нужен для управления моками серверов на портах приложения. Он имеет следующие методы: addMock для добавления мока, deleteMock для удаления мока, getMock для получения мока по номеру порта, deleteMocks для удаления всех моков, которые были добавлены данным менеджером моков.

У данного интерфейса имеется одна реализация MockEndpointManagerImpl, которая реализует данный интерфейс. Данный класс использует реализацию конкурентной мутируемой хеш-таблицы TrieMap для хранения соответствия порта и мока. Изначально данная хеш-таблица пустая. Таким образом, при любом взаимодействии с объектом класса проверяется, занят или не занят ли данный порт в зависимости от семантики вызываемого метода.

При вызове метода addMock полученный mock преобразуется в Route с помощью реализации Routable[MockConfiguration], которая представлена в Routable объекте. Так как Route является абстракцией фреймворка Akka Http для обработки HTTP запросов, он и используется для установки мока на порту. Также в хранилище портов соответствия мокам добавляется новая запись с соответствующим моком и результата так называемого связывания порта и мока для последующей возможности удаления мока. Далее можно увидеть реализацию добавления мока:

**val** route = mock.toRoute(Routable.*mockRoutable*)  
**for** {  
 binding <- *Http*().bindAndHandle(handler = *loggerDirective*(route), port = mock.port, interface = **"0.0.0.0"**)  
 \_ = *portBindings* += mock.port -> (binding, mock)  
 \_ = *logger*.info(**s"Successfully bound on port $**{mock.port}**"**)  
} **yield** ()

При вызове метода deleteMock из хеш-таблицы моков достается мок по порту, и происходит отвязывание сервера от порта:

**for** {  
 bindingOpt <- Future.*successful*(*portBindings*.remove(port))  
 (binding, mock) = bindingOpt.getOrElse(**throw** *NoEndpointStartedOnPort*(port))  
 \_ <- binding.unbind()  
 \_ = *logger*.info(**s"Endpoint on $**port **successfully stopped"**)  
} **yield** mock

При вызове метода getMock, мок берется из хранилища моков, и возвращается вызывающему коду. Если мок на данном порту отсутствует, выбрасывается соответствубщая ошибка.

**override def** getMock(port: Int): Future[MockConfiguration] = **for** {  
 bindingOpt <- Future.*successful*(*portBindings*.get(port))  
 (\_, mock) = bindingOpt.getOrElse(**throw** *NoEndpointStartedOnPort*(port))  
} **yield** mock

Метод deleteMocks извлекает все моки, хранящиеся в данном объекте, и производит их отвязывание от портов:

**override def** deleteMocks: Future[Seq[MockConfiguration]] = {  
 **val** bindings = *portBindings*.values.toSeq  
 **val** unbindingsFuture = bindings.map { **case** (binding, configuration) =>  
 binding.unbind()  
 .map(\_ => configuration)  
 }  
  
 Future.*sequence*(unbindingsFuture)  
 .andThen { **case** *Success*(\_) => *portBindings*.clear() }  
}

### MockConfiguration

Данный интерфейс является базовым для всех конфигураций моков, которые реализованы в решении. В данном интерфейсе имеется только одно поле – целочисленное поле port, которое означает порт, на котором мок будет существовать.

### StubConfiguration

Данный класс является реализацией интерфейса MockConfiguration и состоит из множества ответов типа Stub, которые описывают поведение ответов сервера. Данная конфигурация возвращает первый ответ, всем предикатам которого удовлетворяет входящий запрос.

### ResponseStub

Данный класс является реализацией интерфейса Stub, содержащий множество ответов и предикатов. Ответы – описания ответов, которые сервер отправляет на входящие запросы. Предикаты - описания правил, определяющих применимость данных ответов на запрос. Если множество ответов имеет размер больше одного, то при последовательных запросах, удовлетворяющих предикатам, ответы будут возвращаться циклично по порядку их описания в конфигурации. Так, если конфигурация содержит два ответа, то первый пришедший запрос получит первый ответ, второй пришедший запрос – второй, третий пришедший запрос – опять первый.

### HttpPredicate

Данный интерфейс позволяет необходим для определения применимости ответа входящему запросу в ResponseStub. Диаграмма классов предикатов для Http запросов можно увидеть на рис. 13.



Рисунок 13. Диаграмма классов предикатов HttpPredicate

От базового HttpPredicate наследуются два интерфейса LeafHttpPredicate и CompoundHttpPredicate. LeafHttpPredicate является элементарным предикатом, который содержит некоторое ожидание RequestExpectation и проверяет соответствие входящего запроса этому ожиданию. Реализации данного интерфейса: Equals, StartsWith, Contains – соответственно проверяют, что содержимое входящего равняется, начинается или содержит ожидание.

CompoundHttpPredicate содержит множество других HttpPredicate и либо применяет логическую операцию И (And) и ИЛИ (Or).

Ожидания к входящему запросу реализованы интерфейсом RequestExpectation. Данный интерфейс реализуют классы BodyExpectation, PathExpectation, MethodExpectation, QueryExpectation. BodyExpectation содержит ожидание к содержимому тела входящего запроса. PathExpectation содержит одидание к пути входящего запроса. MethodExpectation – к методу, а Query – к параметрам запроса.

### Response

Данный интерфейс является базовым для определения ответа сервера на запрос. Интерфейс EventsSequenceLikeResponse наследуется от Response и представляет ответ, состоящий из сообщений, отправляемых асинхронно, с возможной задержкой. Диаграмму классов, связанных с Response можно увидеть на рис. 14.



Рисунок 14. Диаграмма классов конфигурации ответов

### SseEventsResponse

Данный класс реализует интерфейс EventsSequenceLikeResponse и является конфигурацией ответов по протоколу Server-Sent Events. Он содержит последовательность событий протокола Server-Sent Events и опциональный таймаут – время, по истечении которого соединение закрывается автоматически.

### WebSocketEventsResponse

Данный класс реализует интерфейс EventsSequenceLikeResponse и является конфигурацией ответов по протоколу Websocket. Он содержит последовательность событий протокола Websocket, последовательность реакций на приходящие запросы и опциональный таймаут – время, по истечении которого соединение закрывается автоматически.

### WsReaction

### WsEventPredicate

### Routable

Данный интерфейс параметризуется типом, преобразование которого к типу Route будет производиться. То есть, объект с типом Routable[Any] будет иметь метод toRoute, преобразующий объект типа Any в объект Route. Объект RoutableSyntax добавляет любому типу метод toRoute, если в скоупе вызывающего кода будет существовать соответствующая реализация интерфейса Routable. Например, метод toRoute можно будет использовать у объекта типа Any, если в той же области видимости будет определен implicit val routableAnyImpl: Routable[Any].

В объекте Routable содержатся реализации Routable интерфейса для всех конфигураций моков, с которыми работает сервер для мокирования асинхронных протоколов. Для StubConfiguration все элементы конфигураций преобразуются к Route и конкатенируются между собой с помощью метода fold, определенного над Scala коллекциями, осуществляя операцию конкатенации попарно над всеми элементами коллекции:

configuration.stubs.map(\_.toRoute).fold(reject)(\_ ~ \_)

Для Routable[Response] требуется создать Route, который будет отдавать асинхронно последовательность событий и обрабатывать входящие события в случае WebSocket. Разберем случай с Websocket, поскольку он требует обработки входящих сообщений от клиента. Сначала происходит преобразование событий от сервера в Source.

**val** source = DelayedSource.*createMessageLike*(response.events)  
 .map(\_.toWs)

Далее происходит преобразование описания обработчика входящих событий от клиента в Flow:

**val** incomingMessagesHandler = WsReaction.*toFlow*(response.reactions)

И осуществляется преобразование в граф потока событий для WebSocket соединения:

handleWebSocketMessages(incomingMessagesHandler.merge(source))

Схему получившегося графа можно увидеть на рис. :



Рисунок . Схема реализации графа для обработки WebSocket соединения

### DelayedSource

# Заключение

# Список источников

1. Spadini D. и др. To Mock or Not to Mock? An Empirical Study on Mocking Practices // 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). , 2017. С. 402–412.

2. About Swagger Specification | Documentation | Swagger | Swagger [Электронный ресурс]. URL: https://swagger.io/docs/specification/about/ (дата обращения: 16.04.2019).

3. Swagger UI | API Development Tools | Swagger [Электронный ресурс]. URL: https://swagger.io/tools/swagger-ui/ (дата обращения: 16.04.2019).

4. Akka HTTP [Электронный ресурс]. URL: https://doc.akka.io/docs/akka-http/current/ (дата обращения: 20.11.2018).

5. Streams • Akka Documentation [Электронный ресурс]. URL: https://doc.akka.io/docs/akka/current/stream/index.html (дата обращения: 20.11.2018).

6. The Magnet Pattern [Электронный ресурс]. URL: http://spray.io/blog/2012-12-13-the-magnet-pattern/ (дата обращения: 18.04.2019).

7. curl [Электронный ресурс]. URL: https://curl.haxx.se/ (дата обращения: 07.05.2019).

8. Mountebank - over the wire test doubles [Электронный ресурс]. URL: http://www.mbtest.org/ (дата обращения: 19.11.2018).

9. MockServer [Электронный ресурс]. URL: http://www.mock-server.com/ (дата обращения: 19.11.2018).

10. WireMock [Электронный ресурс]. URL: http://wiremock.org/ (дата обращения: 27.01.2019).

11. Loreto S. и др. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP [Электронный ресурс]. URL: https://tools.ietf.org/rfc/rfc6202.txt (дата обращения: 12.04.2019).

12. Fette I., Melnikov A. The WebSocket Protocol [Электронный ресурс]. URL: https://tools.ietf.org/html/rfc6455 (дата обращения: 20.11.2018).

13. Support WebSockets mocking [Электронный ресурс]. URL: https://github.com/jamesdbloom/mockserver/issues/360 (дата обращения: 12.04.2019).

14. WebSocket Support – Группы Google [Электронный ресурс]. URL: https://groups.google.com/forum/#!topic/mountebank-discuss/t19u-sc8b6E (дата обращения: 15.01.2019).

15. Patni S. Pro RESTful APIs. Berkeley, CA: Apress, 2017.