

Министерство цифрового развития, связи и массовых коммуникаций
Ордена Трудового Красного Знамени федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Кафедра Математической кибернетики и информационных технологий

Проект
по дисциплине «Введение в информационные технологии»

Выполнил: студент группы БВТ1901

Лапин Виктор Андреевич

Проверила:

Мосева Марина Сергеевна

Москва

2021

СОДЕРЖАНИЕ

1. ЗАДАНИЕ НА РАЗРАБОТКУ	3
2. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ	4
3. ВЫПОЛНЕНИЕ РАБОТЫ	6
3.1. Структура проекта.....	6
3.2. Объект UserRegistry	8
3.3. Объект QuickstartApp	9
3.4. Класс UserRoutes	10
3.5. Объект JsonFormats	13
3.6. Запуск и тестирование	14
3.7. Модульное тестирование.....	16
4. ВЫВОД	19

1. ЗАДАНИЕ НА РАЗРАБОТКУ

Цель работы заключается в запуске и тестировании HTTP-приложения Akka, получении предварительного обзора того, как маршруты упрощают обмен данными по HTTP. Приложение должно быть реализовано в следующих четырёх исходных файлах:

- `QuickstartApp.scala` — содержит основной метод начальной загрузки приложения.
- `UserRoutes.scala` — HTTP-маршруты Akka, определяющие открытые эндпоинты.
- `UserRegistry.scala` — актор, обрабатывающий запросы на регистрацию.
- `JsonFormats.scala` — преобразует данные JSON из запросов в типы Scala и из типов Scala в ответы JSON.

2. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Akka — это бесплатный инструмент с открытым исходным кодом и среда выполнения, упрощающая создание параллельных и распределенных приложений на JVM на языке Scala. Akka поддерживает несколько моделей программирования для параллелизма, включая параллелизм на основе акторов.

Модель акторов — математическая модель параллельных вычислений, строящаяся вокруг понятия актора, считающегося универсальным примитивом параллельного исполнения. Актор в данной модели взаимодействует путём передачи сообщений с другими акторами, в ответ на получаемые сообщения может принимать локальные решения, создавать новые акторы, посылать свои сообщения, устанавливать, как следует реагировать на последующие сообщения.

Создана как теоретическая база для ряда практических реализаций параллельных систем.

В этом примере серверная часть использует только одного основного актора. Использование акторов в приложениях, подобных этому, увеличивает ценность, а не просто предоставляет функции, которые возвращают Futures. Фактически, если логика не имеет состояния и имеет очень простой стиль запроса / ответа, то может не потребоваться подкреплять ее с помощью актора. Акторы действительно полезны, когда нужно сохранить некоторую форму состояния и разрешить различным запросам получить доступ к чему-либо в (или через) актора. Другая полезная особенность акторов, с которой Futures не справится, — это очень простое масштабирование на кластер с помощью кластерного сегментирования или других методов. Однако основное внимание уделяется тому, как взаимодействовать с серверной частью актора изнутри Akka HTTP, а не самому актору.

Каждый HTTP-маршрут Akka содержит один или несколько `akka.http.scaladsl.server.Directives`, таких как: `path`, `get`, `post`, `complete` и т. д. Существует также низкоуровневый API, который позволяет проверять запросы и создавать 7 ответы вручную. Для службы реестра пользователей пример должен

поддерживать действия, перечисленные ниже. Для каждого можно определить путь, метод HTTP и возвращаемое значение:

Функциональность	HTTP метод	Путь	Возврат
Создать пользователя	POST	/users	Подтверждающее сообщение
Получить пользователя	GET	/users/\$ID	JSON
Удалить пользователя	DELETE	/users/\$ID	Подтверждающее сообщение
Получить всех пользователей	GET	/users	JSON

В приложении определение маршрута разделено на класс `UserRoutes` и доступно через поле `userRoutes`. В более крупных приложениях определяются отдельные подсистемы в разных местах, а затем объединяем различные маршруты приложения в большой, используя директиву `concat` следующим образом: `val route = concat (UserRoutes.userRoutes, healthCheckRoutes, ...)`.

3. ВЫПОЛНЕНИЕ РАБОТЫ

3.1. Структура проекта

Для создания проекта были созданы несколько файлов, которые были размещены в соответствии с определённой структурой. Структура файлов проекта показана на рисунке 1.

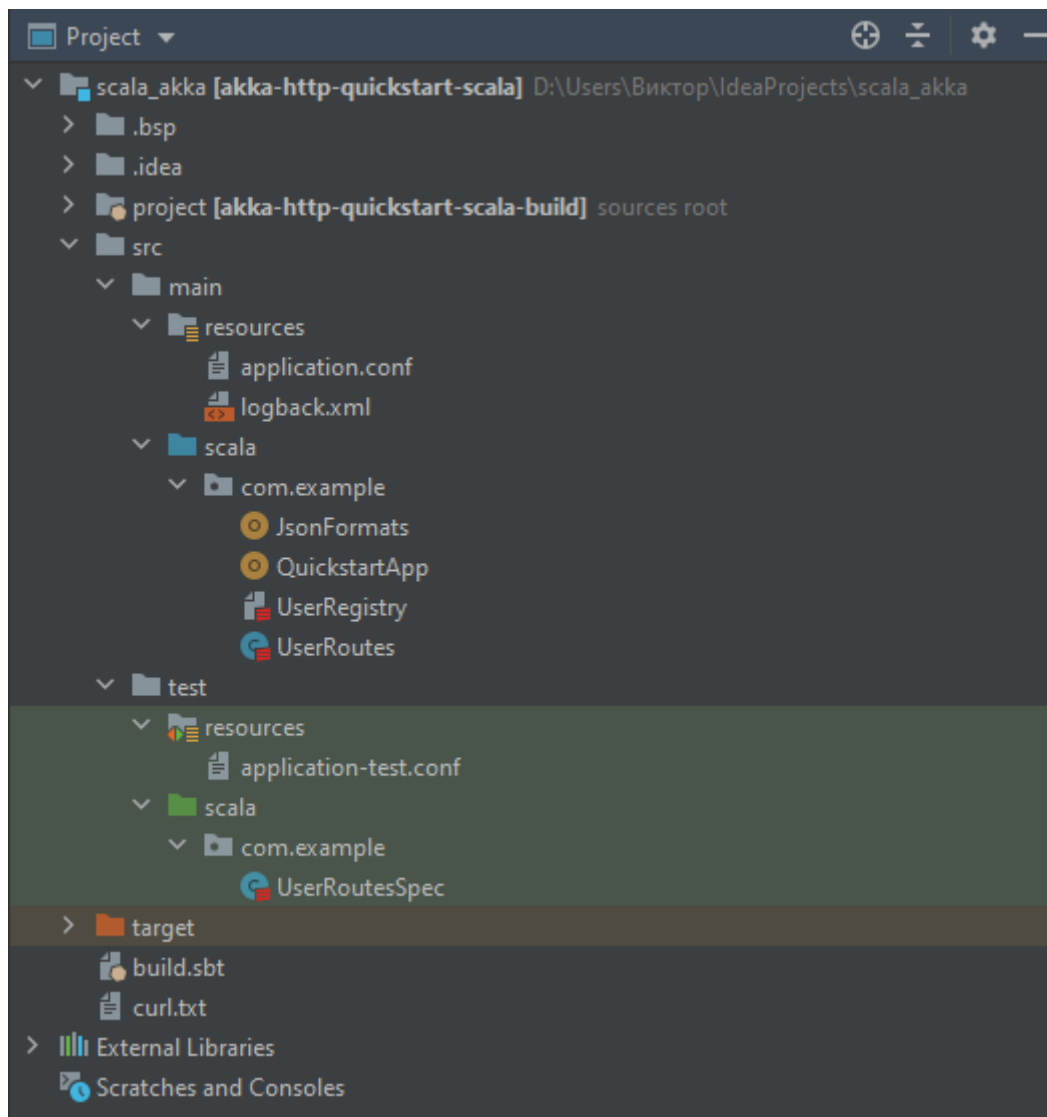


Рисунок 1 – Структура проекта

Содержимое файла build.sbt представлено в листинге 1.

Листинг 1 – Файл build.sbt

```
1 lazy val akkaHttpVersion = "10.2.7"
2 lazy val akkaVersion = "2.6.17"
3 lazy val root = (project in file(".")).
4 settings(
```

```

5    inThisBuild(List(
6      organization := "com.example",
7      scalaVersion := "2.13.4"
8    )),
9    name := "akka-http-quickstart-scala",
10   libraryDependencies ++= Seq(
11     "com.typesafe.akka" %% "akka-http" % akkaHttpVersion,
12     "com.typesafe.akka" %% "akka-http-spray-json" % akkaHttpVersion,
13     "com.typesafe.akka" %% "akka-actor-typed" % akkaVersion,
14     "com.typesafe.akka" %% "akka-stream" % akkaVersion,
15     "ch.qos.logback" % "logback-classic" % "1.2.3",
16     "com.typesafe.akka" %% "akka-http-testkit" % akkaHttpVersion %
17   Test,
18     "com.typesafe.akka" %% "akka-actor-testkit-typed" % akkaVersion %
19   Test,
20     "org.scalatest" %% "scalatest" % "3.1.4" %
21   Test
22   )

```

Содержимое файла application.conf представлено в листинге 2.

Листинг 2 – Файл application.conf

```

1  my-app {
2    routes {
3      # If ask takes more time than this to complete the request is failed
4      ask-timeout = 5s
5    }
6  }

```

Содержимое файла logback.xml представлено в листинге 3.

Листинг 3 – Файл logback.xml

```

1  <configuration>
2    <!-- This is a development logging configuration that logs to
3    standard
4    out, for an example of a production
5    logging config, see the Akka docs:
6    https://doc.akka.io/docs/akka/2.6/typed/logging.html#logback -->
7    <appender name="STDOUT" target="System.out"
8    class="ch.qos.logback.core.ConsoleAppender">
9      <encoder>
10     <pattern>[%date{ISO8601}] [%level] [%logger] [%thread]
11     [%X{akkaSource}] - %msg%n</pattern>
12   </encoder>
13 </appender>
14 <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
15   <queueSize>1024</queueSize>
16   <neverBlock>true</neverBlock>
17   <appender-ref ref="STDOUT" />
18 </appender>

```

```

18 <root level="INFO">
19 <appender-ref ref="ASYNC"/>
20 </root>
21 </configuration>

```

3.2. Объект UserRegistry

Код, который содержится в UserRegistry представлен в листинге 4. Он сохраняет зарегистрированных пользователей в наборе. После получения сообщений он сопоставляет их с определенными случаями, чтобы определить, какое действие предпринять.

Листинг 4 – Файл UserRegistry.scala

```

1  package com.example
2
3  import akka.actor.typed.ActorRef
4  import akka.actor.typed.Behavior
5  import akka.actor.typed.scaladsl.Behaviors
6  import scala.collection.immutable
7
8  final case class User(name: String, age: Int, countryOfResidence:
  String)
9  final case class Users(users: immutable.Seq[User])
10
11 object UserRegistry {
12
13   // actor protocol
14   sealed trait Command
15   final case class GetUsers(replyTo: ActorRef[Users]) extends Command
16   final case class CreateUser(user: User, replyTo:
  ActorRef[ActionPerformed]) extends Command
17   final case class GetUser(name: String, replyTo:
  ActorRef[GetUserResponse]) extends Command
18   final case class DeleteUser(name: String, replyTo:
  ActorRef[ActionPerformed]) extends Command
19
20   final case class GetUserResponse(maybeUser: Option[User])
21   final case class ActionPerformed(description: String)
22
23   def apply(): Behavior[Command] = registry(Set.empty)
24
25   private def registry(users: Set[User]): Behavior[Command] =
26     Behaviors.receiveMessage {
27       case GetUsers(replyTo) =>
28         replyTo ! Users(users.toSeq)
29         Behaviors.same
30       case CreateUser(user, replyTo) =>
31         replyTo ! ActionPerformed(s"User ${user.name} created.")

```



```

32     registry(users + user)
33   case GetUser(name, replyTo) =>
34     replyTo ! GetUserResponse(users.find(_.name == name))
35     Behaviors.same
36   case DeleteUser(name, replyTo) =>
37     replyTo ! ActionPerformed(s"User $name deleted.")
38     registry(users.filterNot(_.name == name))
39   }
40 }

```

3.3. Объект QuickstartApp

Основной класс QuickstartApp запускается, потому что у него есть основной метод, как показано в листинге 5. Этот класс предназначен для «объединения всего», это основной класс, который запускает систему акторов с корневым поведением, которое загружает всех акторов и другие зависимости (соединения с базой данных и т. д.).

Листинг 5 – Файл QuickstartApp.scala

```

1  package com.example
2
3  import akka.actor.typed.ActorSystem
4  import akka.actor.typed.scaladsl.Behaviors
5  import akka.http.scaladsl.Http
6  import akka.http.scaladsl.server.Route
7
8  import scala.util.{Failure, Success}
9
10 object QuickstartApp {
11   private def startHttpServer(routes: Route)(implicit system:
ActorSystem[_]):
Unit = {
12     // Akka HTTP still needs a classic ActorSystem to start
13     import system.executionContext
14     val futureBinding = Http().newServerAt("localhost",
8080).bind(routes)
15     futureBinding.onComplete {
16       case Success(binding) =>
17         val address = binding.localAddress
18         system.log.info("Server online at http://{}:{}/",
19           address.getHostString, address.getPort)
20       case Failure(ex) =>
21         system.log.error("Failed to bind HTTP endpoint, terminating
system", ex)
22         system.terminate()
23     }
24   }
25 }

```

```

26
27 def main(args: Array[String]): Unit = {
28     val rootBehavior = Behaviors.setup[Nothing] { context =>
29         val userRegistryActor = context.spawn(UserRegistry(),
30             "UserRegistryActor")
31         context.watch(userRegistryActor)
32         val routes = new UserRoutes(userRegistryActor)(context.system)
33         startHttpServer(routes.userRoutes)(context.system)
34         Behaviors.empty
35     }
36     val system = ActorSystem[Nothing](rootBehavior,
37         "HelloAkkaHttpServer")
38 }

```

Привязка Route к HTTP-серверу на TCP-порту выполняется корневым актором поведения при запуске с помощью отдельного метода `startHttpServer`, он был введен, чтобы избежать случайного доступа к внутреннему состоянию актора начальной загрузки. Метод `bindAndHandle`, выполняющий фактическую привязку, принимает три параметра; маршруты, имя хоста и порт. Привязка происходит асинхронно, и поэтому метод `bindAndHandle` возвращает `Future`, который завершается объектом, представляющим привязку, или терпит неудачу, если привязка HTTP-маршрута не удалась, например, если порт уже занят. Чтобы убедиться, что приложение останавливается, если оно не может выполнить привязку, завершается система акторов в случае сбоя.

В `QuickstartApp.scala` также содержится код, который связывает всё вместе, запуская различные акторы в корневом поведении. Наблюдая за актором реестра пользователей и не обрабатывая сообщение `Terminated`, гарантируется, что в случае его остановки или взлома корневого поведения выйдет из строя и остановит саму систему акторов.

3.4. Класс `UserRoutes`

Класс `UserRoutes` связывает эндпоинты, методы HTTP и сообщение или результат для каждого действия. Маршрут создается путем вложения различных директив, которые направляют входящий запрос в соответствующий блок обработчика. Содержимое класса `UserRoutes` представлено в листинге 6.

Листинг 6 – Файл UserRoutes.scala

```
1  package com.example
2
3  import akka.actor.typed.scaladsl.AskPattern.{Askable,
4    schedulerFromActorSystem}
5  import akka.actor.typed.{ActorRef, ActorSystem}
6  import akka.http.scaladsl.model.StatusCodes
7  import akka.http.scaladsl.server.Directives._
8  import akka.util.Timeout
9  import com.example.UserRegistry._
10
11 import scala.collection.immutable
12 import scala.concurrent.Future
13
14 class UserRoutes(userRegistry:
15   ActorRef[UserRegistry.Command])(implicit val
16   system: ActorSystem[_]) {
17   import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
18   import JsonFormats._
19
20   private implicit val timeout: Timeout =
21     Timeout.create(system.settings.config.getDuration("my-app.routes.ask-
22       timeout"))
23
24   def getUsers(): Future[Users] =
25     userRegistry.ask(GetUsers)
26   def getUser(name: String): Future[GetUserResponse] =
27     userRegistry.ask(GetUser(name, _))
28   def createUser(user: User): Future[ActionPerformed] =
29     userRegistry.ask(CreateUser(user, _))
30   def deleteUser(name: String): Future[ActionPerformed] =
31     userRegistry.ask>DeleteUser(name, _))
32
33   val userRoutes: Route =
34     pathPrefix("users") {
35       concat(
36         pathEnd {
37           concat(
38             get {
39               complete(getUsers())
40             },
41             post {
42               entity(as[User]) { user =>
43                 onSuccess(createUser(user)) { performed =>
44                   complete((StatusCodes.Created, performed))
45                 }
46             }
47           }
48         }
49       ),
50       path(Segment) { name =>
```

```

47         concat(
48             get {
49                 rejectEmptyResponse {
50                     onSuccess(getUser(name)) { response =>
51                         complete(response.maybeUser)
52                     }
53                 }
54             },
55             delete {
56                 onSuccess(deleteUser(name)) { performed =>
57                     complete((StatusCodes.OK, performed))
58                 }
59             })
60     })
61 }
62 }

```

В приведенном листинге используются следующие директивы:

- `pathPrefix ("users")`: путь, который используется для сопоставления входящего запроса.
- `pathEnd`: используется на внутреннем уровне, чтобы отличать «уже полностью согласованный путь» от других альтернатив. В этом случае будет соответствовать пути «users».
- `concat`: объединяет два или более альтернативных маршрута. Маршруты проверяются один за другим. Если маршрут отклоняет запрос, выполняется попытка следующего маршрута в цепочке. Это продолжается до тех пор, пока маршрут в цепочке не даст ответ. Если все альтернативы маршрута отклоняют запрос, объединенный маршрут также отклоняет маршрут. В этом случае предпринимаются попытки альтернативного маршрута на следующем более высоком уровне. Если маршрут корневого уровня также отклоняет запрос, возвращается ответ об ошибке, содержащий информацию о том, почему запрос был отклонен.

При получении пользователей «get» соответствует методу GET HTTP, «complete» завершает запрос, что означает создание и возврат ответа из аргументов.

При создании пользователя «post» соответствует методу POST HTTP, «entity (as [User])» преобразует тело HTTP-запроса в объект домена типа User. Неявно предполагается, что запрос содержит содержимое «application/json». Завершает запрос «complete», что означает создание и возврат ответа из аргументов.

Кортеж (StatusCodes.Created, "...") типа (StatusCode, String) неявно преобразуется в ответ с заданным кодом состояния и текстовым / простым телом с заданной строкой.

3.5. Объект JsonFormats

При выполнении приложения происходит взаимодействие с JSON. Приложение преобразует данные из формата JSON в данные, которые могут использоваться классами Scala с помощью объекта JsonFormats. Исходный код этого объекта представлен в листинге 7.

Листинг 7 – Файл JsonFormats.scala

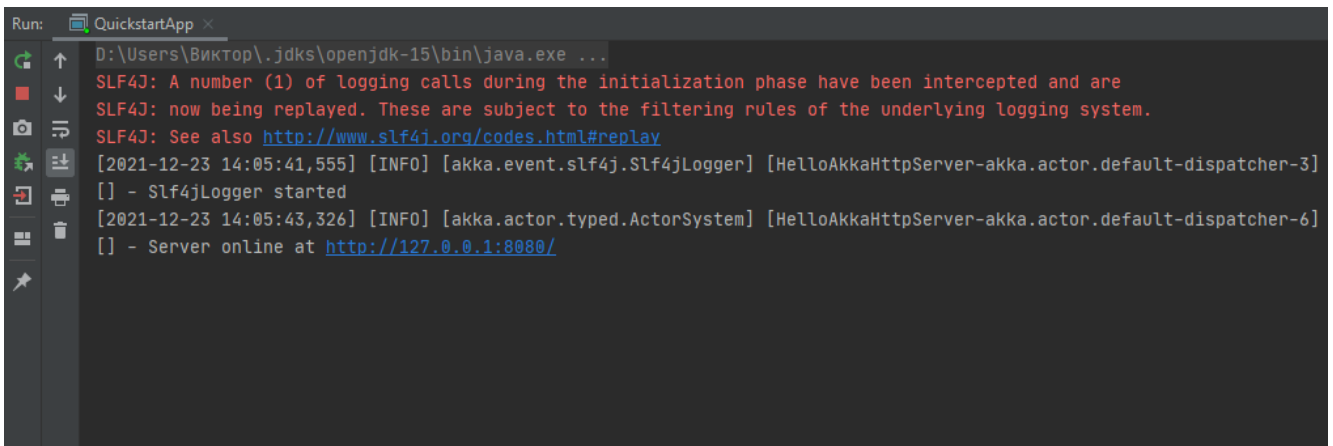
```
1 package com.example
2
3 import com.example.UserRegistry.ActionPerformed
4 import spray.json.DefaultJsonProtocol
5 object JsonFormats {
6   // import the default encoders for primitive types (Int, String,
   Lists etc)
7
8   import DefaultJsonProtocol._
9
10  implicit val userJsonFormat = jsonFormat3(User)
11  implicit val usersJsonFormat = jsonFormat1(Users)
12  implicit val actionPerformedJsonFormat =
    jsonFormat1(ActionPerformed)
13 }
```

Здесь используется библиотека Spray JSON, которая позволяет определять форматы json безопасным для типов способом. Другими словами, если не предоставляется экземпляр формата для типа, но возникает попытка вернуть его в маршруте, вызвав complete (someValue), код не будет компилироваться – заявив, что он не знает, как преобразовывать SomeValuetype. Это дает преимущество в том,

что полностью контролируется то, что необходимо раскрыть, и не раскрывается случайно какой-либо тип в HTTP API. Для обработки двух разных JSON трейт определяет два неявных значения; `userJsonFormat` и `usersJsonFormat`. Определение средств форматирования как `implicit` гарантирует, что компилятор может сопоставить функции форматирования с case классами для преобразования.

3.6. Запуск и тестирование

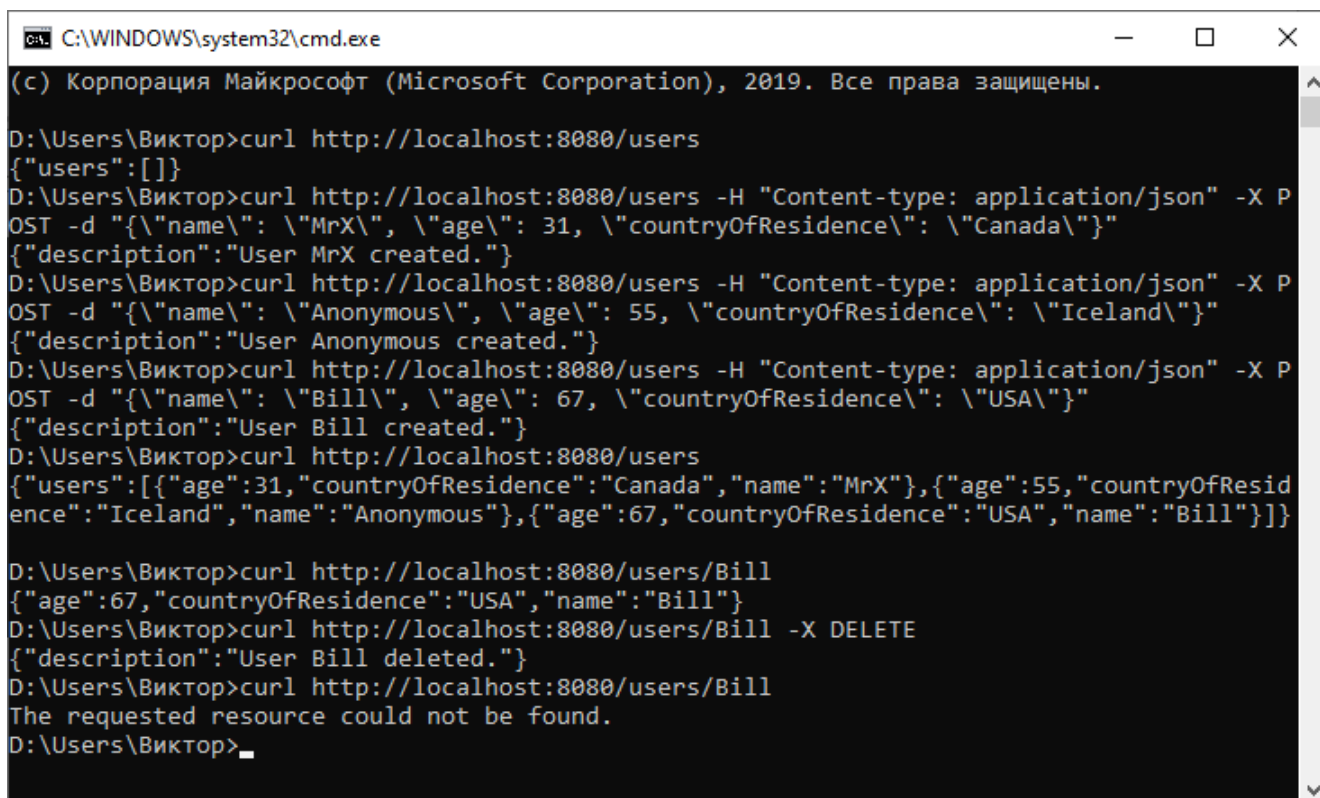
Сервер был запущен, сообщение о чём представлено на рисунке 2.



```
Run: QuickstartApp x
D:\Users\Виктор\.jdk\openjdk-15\bin\java.exe ...
SLF4J: A number (1) of logging calls during the initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay
[2021-12-23 14:05:41,555] [INFO] [akka.event.Slf4j.Slf4jLogger] [HelloAkkaHttpServer-akka.actor.default-dispatcher-3]
[] - Slf4jLogger started
[2021-12-23 14:05:43,326] [INFO] [akka.actor.typed.ActorSystem] [HelloAkkaHttpServer-akka.actor.default-dispatcher-6]
[] - Server online at http://127.0.0.1:8080/
```

Рисунок 2 – Запуск сервера

Для тестирования работоспособности были выполнены несколько HTTP-запросов с помощью cURL. Снимок экрана с запросами и ответами сервера представлен на рисунке 3.



```
C:\WINDOWS\system32\cmd.exe
(с) Корпорация Майкрософт (Microsoft Corporation), 2019. Все права защищены.

D:\Users\Виктор>curl http://localhost:8080/users
{"users":[]}
D:\Users\Виктор>curl http://localhost:8080/users -H "Content-type: application/json" -X POST -d "{\"name\": \"MrX\", \"age\": 31, \"countryOfResidence\": \"Canada\"}"
{"description":"User MrX created."}
D:\Users\Виктор>curl http://localhost:8080/users -H "Content-type: application/json" -X POST -d "{\"name\": \"Anonymous\", \"age\": 55, \"countryOfResidence\": \"Iceland\"}"
{"description":"User Anonymous created."}
D:\Users\Виктор>curl http://localhost:8080/users -H "Content-type: application/json" -X POST -d "{\"name\": \"Bill\", \"age\": 67, \"countryOfResidence\": \"USA\"}"
{"description":"User Bill created."}
D:\Users\Виктор>curl http://localhost:8080/users
{"users":[{"age":31,"countryOfResidence":"Canada","name":"MrX"}, {"age":55,"countryOfResidence":"Iceland","name":"Anonymous"}, {"age":67,"countryOfResidence":"USA","name":"Bill"}]}
D:\Users\Виктор>curl http://localhost:8080/users/Bill
{"age":67,"countryOfResidence":"USA","name":"Bill"}
D:\Users\Виктор>curl http://localhost:8080/users/Bill -X DELETE
{"description":"User Bill deleted."}
D:\Users\Виктор>curl http://localhost:8080/users/Bill
The requested resource could not be found.
D:\Users\Виктор>
```

Рисунок 3 – Выполнение запросов

Первый GET-запрос получает список пользователей, ответ сервера представляет пустой массив, так как при запуске сервера набор пользователей представляет собой пустой набор.

Далее выполнены три POST-запроса для создания пользователей. Содержимое запроса представлено в виде JSON, содержащем имя, возраст и страну. При успешном создании пользователя сервер возвращает сообщение.

После этого выполнен аналогичный первому запросу, результатом которого является массив, содержащий информацию о трёх добавленных пользователях в формате JSON.

Затем выполнен DELETE-запрос, который удаляет пользователя. При успешном выполнении сервер возвращает сообщение.

При выполнении GET-запроса для удалённого пользователя сервер возвращает сообщение о том, что пользователя не существует.

3.7. Модульное тестирование

Модульное тестирование в Akka HTTP – это просто «выполнение» маршрутов путем передачи `HttpResponse` в маршрут и последующей проверки того, к чему `HttpResponse` (или отклонение, если запрос не может быть обработан) он привел. Все это в памяти, без необходимости запускать настоящий HTTP-сервер, что дает максимальную скорость и время оборачиваемости при разработке приложения с использованием Akka. Исходный код файла `UserRoutesSpec.scala`, содержащего модульные тесты, представлен в листинге 8.

Листинг 8 – Файл `UserRoutesSpec.scala`

```
1  import akka.actor.testkit.typed.scaladsl.ActorTestKit
2  import akka.http.scaladsl.marshalling.Marshal
3  import akka.http.scaladsl.model._
4  import akka.http.scaladsl.testkit.ScalatestRouteTest
5  import com.example.{User, UserRegistry, UserRoutes}
6  import org.scalatest.concurrent.ScalaFutures
7  import org.scalatest.matchers.should.Matchers
8  import org.scalatest.wordspec.AnyWordSpec
9  class UserRoutesSpec extends AnyWordSpec with Matchers with
    ScalaFutures with
10     ScalatestRouteTest {
11     // the Akka HTTP route testkit does not yet support a typed actor
    system (https://github.com/akka/akka-http/issues/2036)
12     // so we have to adapt for now
13     lazy val testKit = ActorTestKit()
14     implicit def typedSystem = testKit.system
15     override def createActorSystem(): akka.actor.ActorSystem =
16         testKit.system.classicSystem
17     // Here we need to implement all the abstract members of UserRoutes.
18     // We use the real UserRegistryActor to test it while we hit the
    Routes,
19     // but we could "mock" it by implementing it in-place or by using a
    TestProbe
20     // created with testKit.createTestProbe()
21     val userRegistry = testKit.spawn(UserRegistry())
22     lazy val routes = new UserRoutes(userRegistry).userRoutes
23     // use the json formats to marshal and unmarshall objects in the
    test
24     import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
25     import com.example.JsonFormats._
26     "UserRoutes" should {
27         "return no users if no present (GET /users)" in {
28             // note that there's no need for the host part in the uri:
29             val request = HttpRequest(uri = "/users")
```



```

30     request ~> routes ~> check {
31         status should ===(StatusCodes.OK)
32         // we expect the response to be json:
33         contentType should ===(ContentTypes.`application/json`)
34         // and no entries should be in the list:
35         entityAs[String] should ===(""{\"users\":[]}\"")
36     }
37 }
38 "be able to add users (POST /users)" in {
39     val user = User("Kapi", 42, "jp")
40     val userEntity = Marshal(user).to[MessageEntity].futureValue //
futureValue is from ScalaFutures
41     // using the RequestBuilding DSL:
42     val request = Post("/users").withEntity(userEntity)
43     request ~> routes ~> check {
44         status should ===(StatusCodes.Created)
45         // we expect the response to be json:
46         contentType should ===(ContentTypes.`application/json`)
47         // and we know what message we're expecting back:
48         entityAs[String] should ===(""{\"description\":\"User Kapi
created.\"}\"")
49     }
50 }
51 "be able to remove users (DELETE /users)" in {
52     // user the RequestBuilding DSL provided by ScalatestRouteSpec:
53     val request = Delete(uri = "/users/Kapi")
54     request ~> routes ~> check {
55         status should ===(StatusCodes.OK)
56         // we expect the response to be json:
57         contentType should ===(ContentTypes.`application/json`)
58         // and no entries should be in the list:
59         entityAs[String] should ===(""{\"description\":\"User Kapi
deleted.\"}\"")
60     }
61 }
62 }
63 }

```

Здесь используется ScalaTest, который предоставляет стиль тестирования WordSpec и трейт Matchers, который предоставляет синтаксис `something should === (somethingElse)` и многое другое. Затем наследуется связующий трейт `ScalatestRouteTest`, предоставленный Akka HTTP, который предоставляет средства тестирования `Route` и связывается с методами жизненного цикла `ScalaTest`, так что система акторов запускается и останавливается автоматически.

Затем в тестовый класс вносятся маршруты, которые необходимо протестировать. Для этого создается экземпляр `UserRoutes` и импортируются

форматы из `JsonFormats` в область видимости для использования тестов, а также позволяет реализовать все абстрактные элементы этого трейта в самом тесте. Используется фактический актор, поскольку у него нет зависимостей, это делает тест больше похожим на интеграционный тест, чем на модульный тест. Затем производится тестирование каждого эндпоинта метода HTTP.

Результат модульного тестирования представлен на рисунке 4.

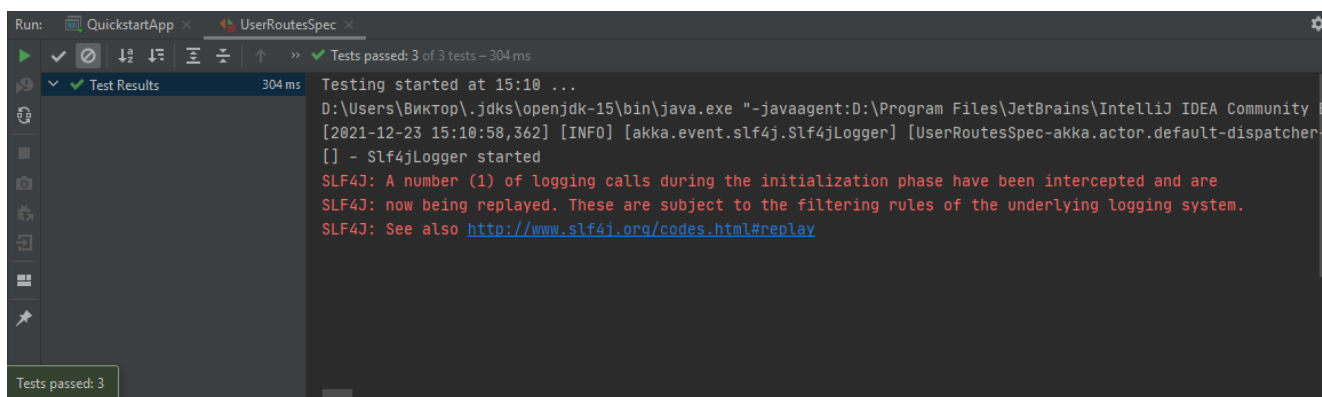


Рисунок 4 – Модульное тестирование

В результате тестирования сообщается о трёх пройденных тестах, следовательно приложение успешно прошло модульное тестирование.

4. ВЫВОД

В результате работы выполнены запуск и тестировании HTTP-приложения Akka, получен предварительный обзор того, как маршруты упрощают обмен данными по HTTP. Приложение реализовано в четырёх исходных файлах, а также выполнено модульное тестирование.