**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И**

**МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский технический университет связи и информатики»**

Курсовая работа

«Akka HTTP сервер»

по дисциплине:

«Введение в информационные технологии»

Выполнил:

Студент группы БФИ-1901

Бардюк Д.В.

Москва, 2021

**Содержание**

[**Цель: 3**](#_Toc89615153)

[**Задания: 3**](#_Toc89615154)

[**Ход работы 3**](#_Toc89615155)

[**1. Создание проекта 3**](#_Toc89615156)

[**1.1 Предварительная настройка 3**](#_Toc89615157)

[**1.2 Backend Логика actora 5**](#_Toc89615158)

[**1.3 HTTP сервер 6**](#_Toc89615159)

[**1.4 Определение маршрутизации 8**](#_Toc89615160)

[**1.5 Формирование JSON 10**](#_Toc89615161)

[**2. Тестирование 10**](#_Toc89615162)

[**3. Запуск приложения 13**](#_Toc89615163)

[**3.1 Команды cURL 13**](#_Toc89615164)

[**3.2 Браузерные клиенты 14**](#_Toc89615165)

[**Вывод 18**](#_Toc89615166)

# **Цель:**

Запуск и тестирование HTTP-приложения Akka, получение предварительного обзора того, как маршруты упрощают обмен данными по HTTP.

# **Задания:**

1. Приложение должно быть реализовано в следующих четырех 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.

# **Ход работы**

## **1. Создание проекта**

* 1. **Предварительная настройка**

Создаём новый проект и используем следующую структуру, как на рисунке 1. Содержание конфигурационных файлов представлено на рисунках 2-4.

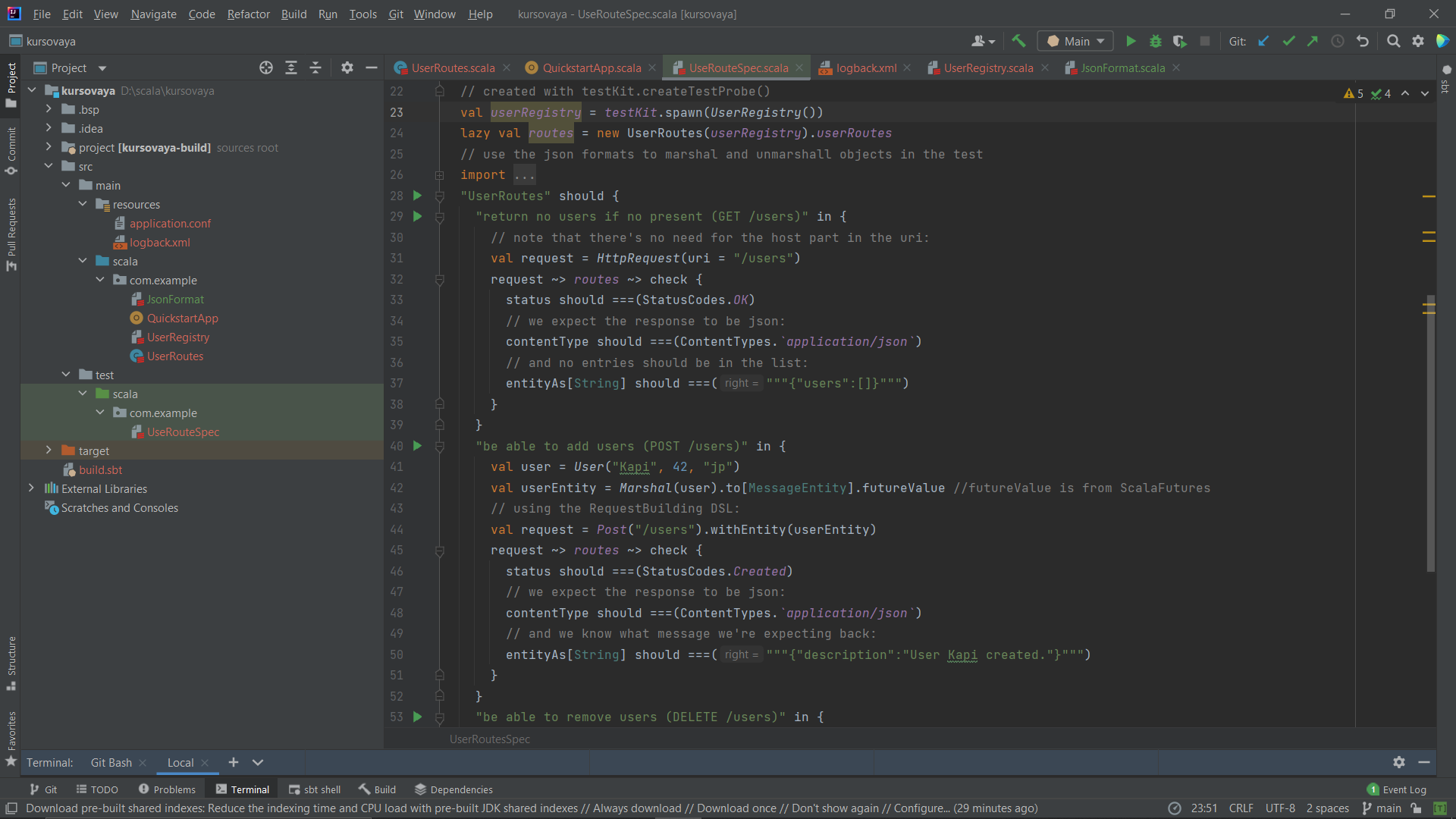


Рисунок 1 – Создание нового проекта

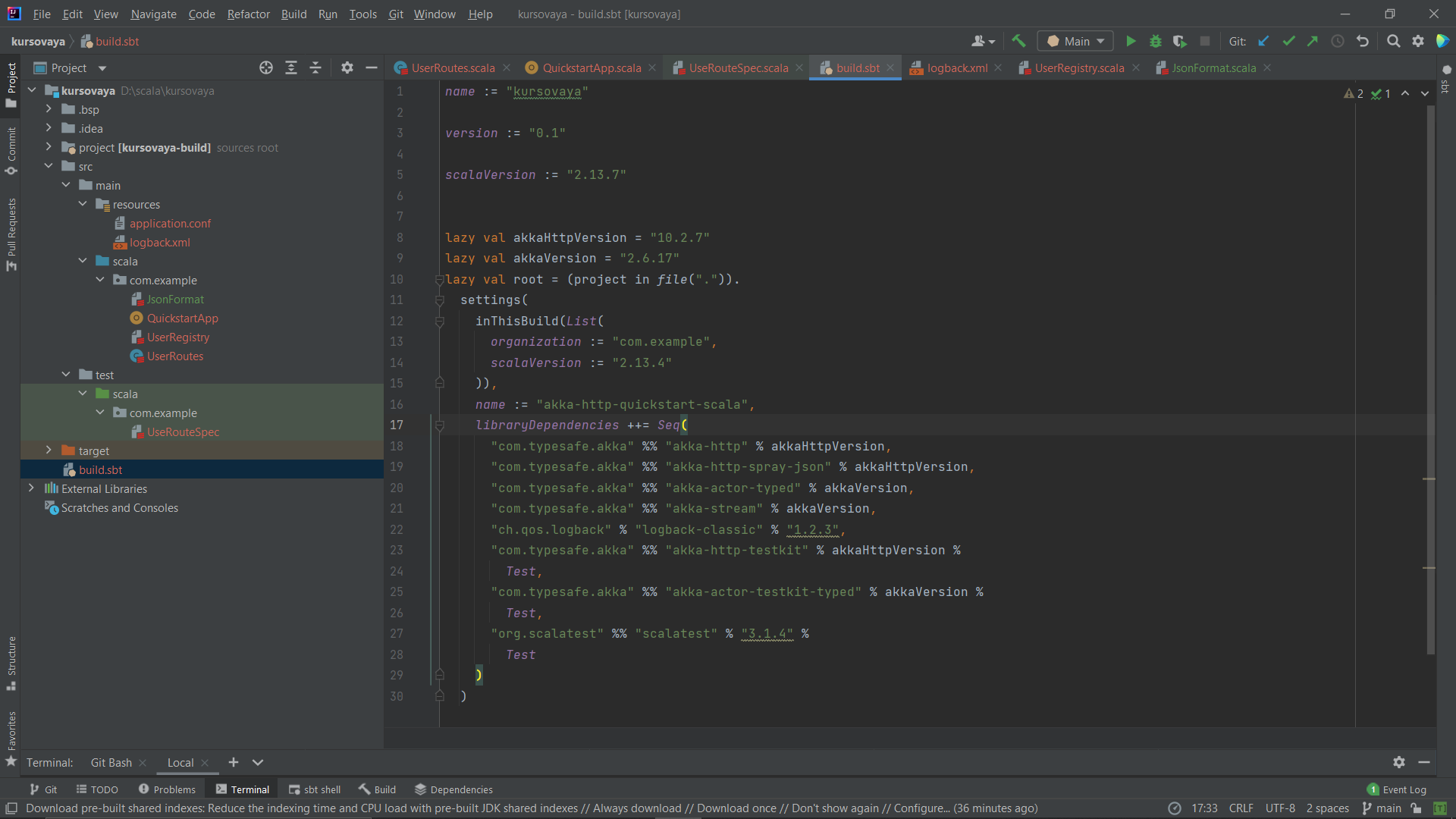


Рисунок 2 – Файл build.sbt

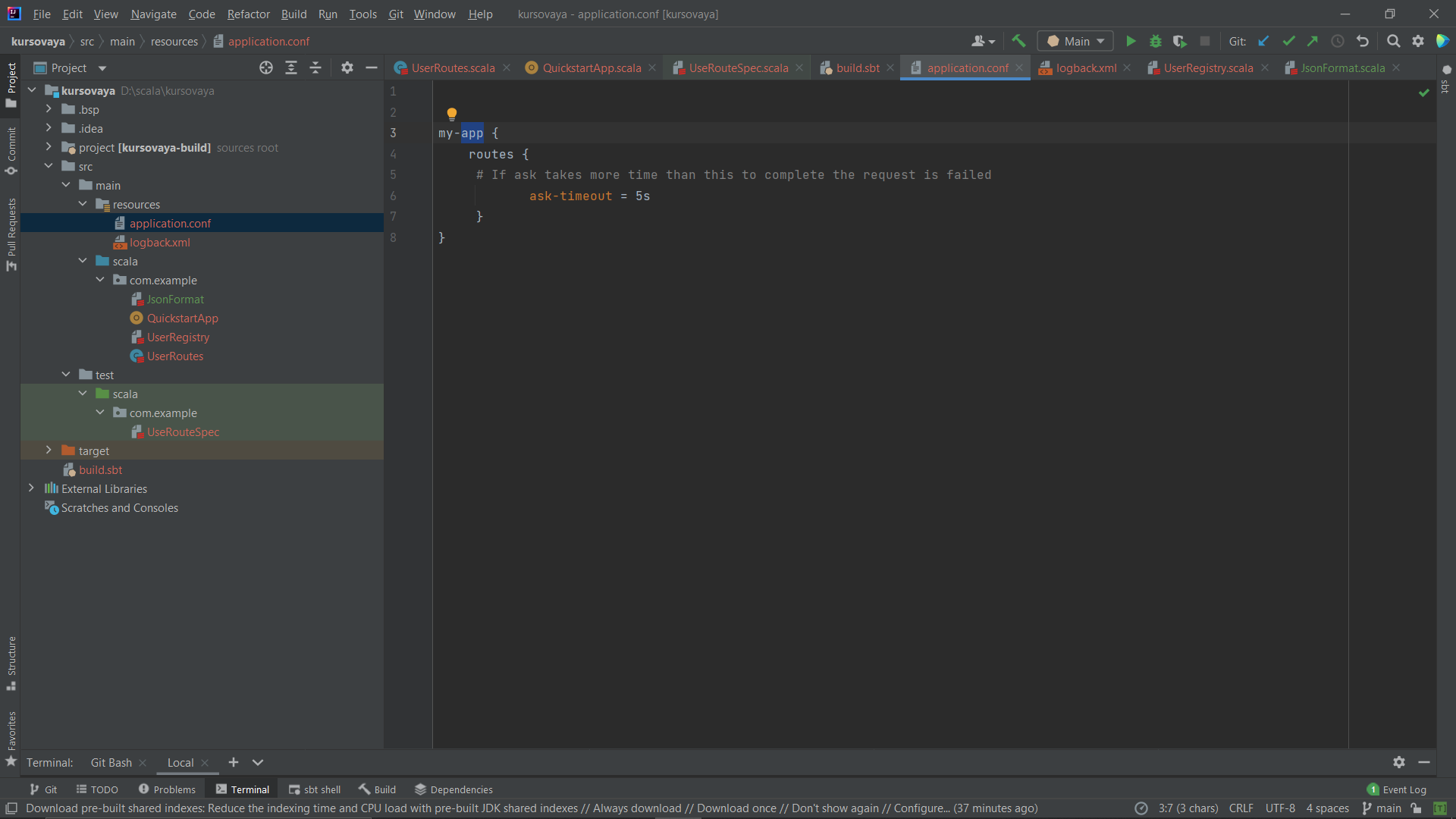


Рисунок 3 – Файл application.conf

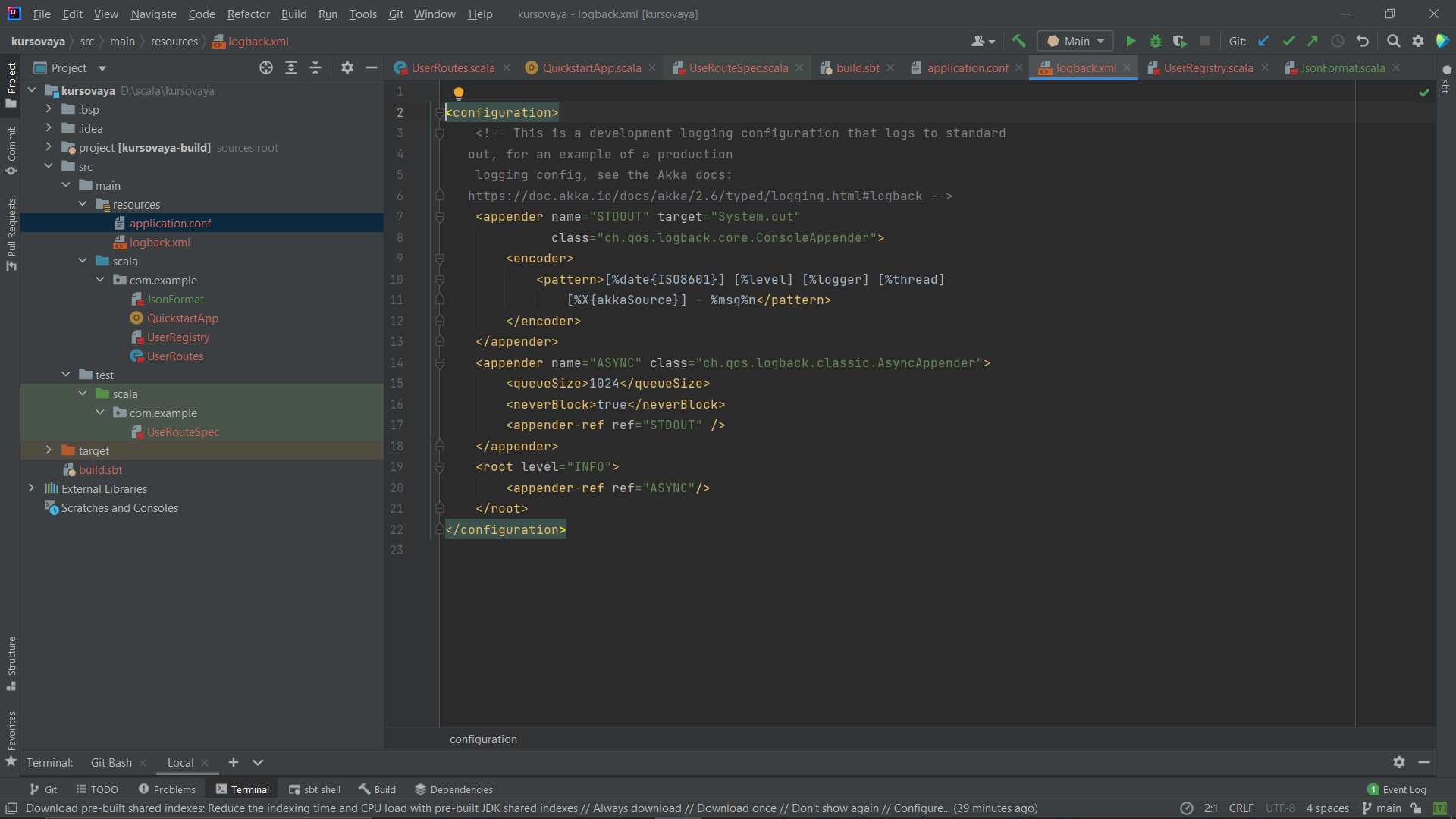


Рисунок 4 – Файл logback.xml

* 1. **Backend Логика actora**

Код, который содержится в UserRegistry представлен в листинге 1. В нём представлена серверная часть актора, который сохраняет зарегистрированных пользователей в наборе. После получения сообщений он сопоставляет их с определенными случаями, чтобы определить, какое действие предпринять:

**Листинг 1**

package com.example  
  
import akka.actor.typed.ActorRef  
import akka.actor.typed.Behavior  
import akka.actor.typed.scaladsl.Behaviors  
import scala.collection.immutable  
  
 final case class User(name: String, age: Int, countryOfResidence: String)  
 final case class Users(users: immutable.Seq[User])  
  
object UserRegistry {  
 // actor protocol  
 sealed trait Command  
  
 final case class GetUsers(replyTo: ActorRef[Users]) extends Command  
  
 final case class CreateUser(user: User, replyTo: ActorRef[ActionPerformed])  
 extends Command  
  
 final case class GetUser(name: String, replyTo: ActorRef[GetUserResponse])  
 extends Command  
  
 final case class DeleteUser(name: String, replyTo: ActorRef[ActionPerformed])  
 extends Command  
  
 final case class GetUserResponse(maybeUser: Option[User])  
  
 final case class ActionPerformed(description: String)  
  
 def apply(): Behavior[Command] = registry(Set.empty)  
  
 private def registry(users: Set[User]): Behavior[Command] =  
 Behaviors.receiveMessage {  
 case GetUsers(replyTo) =>  
 replyTo ! Users(users.toSeq)  
 Behaviors.same  
 case CreateUser(user, replyTo) =>  
 replyTo ! ActionPerformed(s"User ${user.name} created.")  
 registry(users + user)  
 case GetUser(name, replyTo) =>  
 replyTo ! GetUserResponse(users.find(\_.name == name))  
 Behaviors.same  
 case DeleteUser(name, replyTo) =>  
 replyTo ! ActionPerformed(s"User ${name} deleted.")  
 registry(users.filterNot(\_.name == name))  
 }  
}

* 1. **HTTP сервер**

Основной класс QuickstartServer запускается, потому что у него есть основной метод, как показано в Листинге 2. 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.). Выделенный класс UserRoutes, в который помещены все фактические определения маршрутов, является хорошим шаблоном для подражания, особенно когда приложение начинает расти, и вам может понадобится какая-то форма разделения их на группы маршрутов, обрабатывающих определенные части открытого API.

**Листинг 2**

package com.example  
  
import akka.actor.typed.ActorSystem  
import akka.actor.typed.scaladsl.Behaviors  
import akka.http.scaladsl.Http  
import akka.http.scaladsl.server.Route  
import scala.util.{Failure, Success}  
  
object QuickstartApp {  
 private def startHttpServer(routes: Route)(implicit system: ActorSystem[\_]): Unit = {  
 // Akka HTTP still needs a classic ActorSystem to start  
 import system.executionContext  
  
 val futureBinding = Http().newServerAt("localhost", 8880).bind(routes)  
 futureBinding.onComplete {  
 case Success(binding) =>  
 val address = binding.localAddress  
 system.log.info("Server online at http://{}:{}/", address.getHostString, address.getPort)  
 case Failure(ex) =>  
 system.log.error("Failed to bind HTTP endpoint, terminating system", ex)  
 system.terminate()  
 }  
 }  
 def main(args: Array[String]): Unit = {  
 val rootBehavior = Behaviors.setup[Nothing] { context =>  
 val userRegistryActor = context.spawn(UserRegistry(), "UserRegistryActor")  
 context.watch(userRegistryActor)  
  
 val routes = new UserRoutes(userRegistryActor)(context.system)  
 startHttpServer(routes.userRoutes)(context.system)  
  
 Behaviors.empty  
 }  
 val system = ActorSystem[Nothing](rootBehavior, "HelloAkkaHttpServer")  
 }  
}

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. Обратите внимание, что 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 находится код, который связывает все 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.

* 1. **Определение маршрутизации**

В листинге 3 представлено полное определение маршрута для приложения.

**Листинг 3**

package com.example  
  
import akka.actor.typed.scaladsl.AskPattern.{Askable, schedulerFromActorSystem}  
import akka.actor.typed.{ActorRef, ActorSystem}  
import akka.event.Logging  
import akka.http.scaladsl.model.StatusCodes  
import akka.http.scaladsl.server.Directives.\_  
import akka.http.scaladsl.server.Route  
import akka.util.Timeout  
import com.example.UserRegistry.{ActionPerformed, CreateUser, DeleteUser, GetUser, GetUserResponse, GetUsers}  
  
import scala.concurrent.Future  
import scala.concurrent.duration.DurationInt  
  
class UserRoutes(userRegistry: ActorRef[UserRegistry.Command])(implicit val  
system: ActorSystem[\_]) {  
  
 implicit lazy val timeout = Timeout(5.seconds)  
// lazy val log = Logging(system, classOf[UserRoutes])  
  
 import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport.\_  
 import JsonFormats.\_  
  
 val userRoutes: Route =  
 pathPrefix("users") {  
 concat(  
 pathEnd {  
 concat(  
 get {  
 def getUsers():Future[Users] = userRegistry.ask(GetUsers(\_))  
 complete(getUsers() )  
 },  
 post {  
  
 entity(as[User]) { user =>  
 def createUser(user: User): Future[ActionPerformed] =  
 userRegistry.ask(CreateUser(user,\_))  
 onSuccess(createUser(user)) { performed =>  
 complete((StatusCodes.Created, performed))  
 }  
 }  
 })  
 },  
 path(Segment) { name =>  
 concat(  
 get {  
 rejectEmptyResponse {  
 def getUser(name:String): Future[GetUserResponse] =  
 userRegistry.ask(GetUser(name,\_))  
 onSuccess(getUser(name)) { response =>  
 complete(response.maybeUser)  
 }  
 }  
 },  
 delete {  
 def deleteUser(name: String): Future[ActionPerformed] =  
 userRegistry.ask(DeleteUser(name,\_))  
 onSuccess(deleteUser(name)) { performed =>  
 complete((StatusCodes.OK, performed))  
 }  
 })  
 })  
 }  
  
}

* 1. **Формирование JSON**

В Листинге 4 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.

**Листинг 4**

package com.example  
  
import com.example.UserRegistry.ActionPerformed  
import spray.json.DefaultJsonProtocol  
  
import scala.collection.immutable  
  
object JsonFormats {  
 // import the default encoders for primitive types (Int, String, Lists etc)  
 import DefaultJsonProtocol.\_  
 implicit val userJsonFormat = jsonFormat3(User)  
 implicit val usersJsonFormat = jsonFormat1(Users)  
 implicit val actionPerformedJsonFormat = jsonFormat1(ActionPerformed)  
}

1. **Тестирование**

Так как все маршруты выделены в отдельный класс, тестирование сильно упрощено. Для нашего проекта мы используем модульное тестирование. В этом стиле тестирования даже не нужно запускать реальный сервер - все тесты будут выполняться непосредственно на маршрутах - без необходимости попадания в реальную сеть. Это связано с чистым дизайном Akka HTTP и разделением между сетевым уровнем (представленным как двунаправленный поток байтовых строк к объектам домена Http).

Другими словами, 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. Код нашего тестирования представлен в листинге 5. Результат тестирования представлен на рисунке 5.

**Листинг 5**

package com.example  
  
import akka.actor.testkit.typed.scaladsl.ActorTestKit  
import akka.http.scaladsl.marshalling.Marshal  
import akka.http.scaladsl.model.\_  
import akka.http.scaladsl.testkit.ScalatestRouteTest  
import org.scalatest.concurrent.ScalaFutures  
import org.scalatest.matchers.should.Matchers  
import org.scalatest.wordspec.AnyWordSpec  
  
class UserRoutesSpec extends AnyWordSpec with Matchers with ScalaFutures with  
 ScalatestRouteTest {  
 // the Akka HTTP route testkit does not yet support a typed actor system (https://github.com/akka/akka-http/issues/2036)  
 // so we have to adapt for now  
 lazy val testKit = ActorTestKit()  
 implicit def typedSystem = testKit.system  
 override def createActorSystem(): akka.actor.ActorSystem =  
 testKit.system.classicSystem  
 // Here we need to implement all the abstract members of UserRoutes.  
 // We use the real UserRegistryActor to test it while we hit the Routes,  
 // but we could "mock" it by implementing it in-place or by using a TestProbe  
 // created with testKit.createTestProbe()  
 val userRegistry = testKit.spawn(UserRegistry())  
 lazy val routes = new UserRoutes(userRegistry).userRoutes  
 // use the json formats to marshal and unmarshall objects in the test  
 import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport.\_  
 import JsonFormats.\_  
 "UserRoutes" should {  
 "return no users if no present (GET /users)" in {  
 // note that there's no need for the host part in the uri:  
 val request = HttpRequest(uri = "/users")  
 request ~> routes ~> check {  
 status should ===(StatusCodes.OK)  
 // we expect the response to be json:  
 contentType should ===(ContentTypes.`application/json`)  
 // and no entries should be in the list:  
 entityAs[String] should ===("""{"users":[]}""")  
 }  
 }  
 "be able to add users (POST /users)" in {  
 val user = User("Kapi", 42, "jp")  
 val userEntity = Marshal(user).to[MessageEntity].futureValue //futureValue is from ScalaFutures  
 // using the RequestBuilding DSL:  
 val request = Post("/users").withEntity(userEntity)  
 request ~> routes ~> check {  
 status should ===(StatusCodes.Created)  
 // we expect the response to be json:  
 contentType should ===(ContentTypes.`application/json`)  
 // and we know what message we're expecting back:  
 entityAs[String] should ===("""{"description":"User Kapi created."}""")  
 }  
 }  
 "be able to remove users (DELETE /users)" in {  
 // user the RequestBuilding DSL provided by ScalatestRouteSpec:  
 val request = Delete(uri = "/users/Kapi")  
 request ~> routes ~> check {  
 status should ===(StatusCodes.OK)  
 // we expect the response to be json:  
 contentType should ===(ContentTypes.`application/json`)  
 // and no entries should be in the list:  
 entityAs[String] should ===("""{"description":"User Kapi deleted."}""")  
 }  
 }  
 }  
}

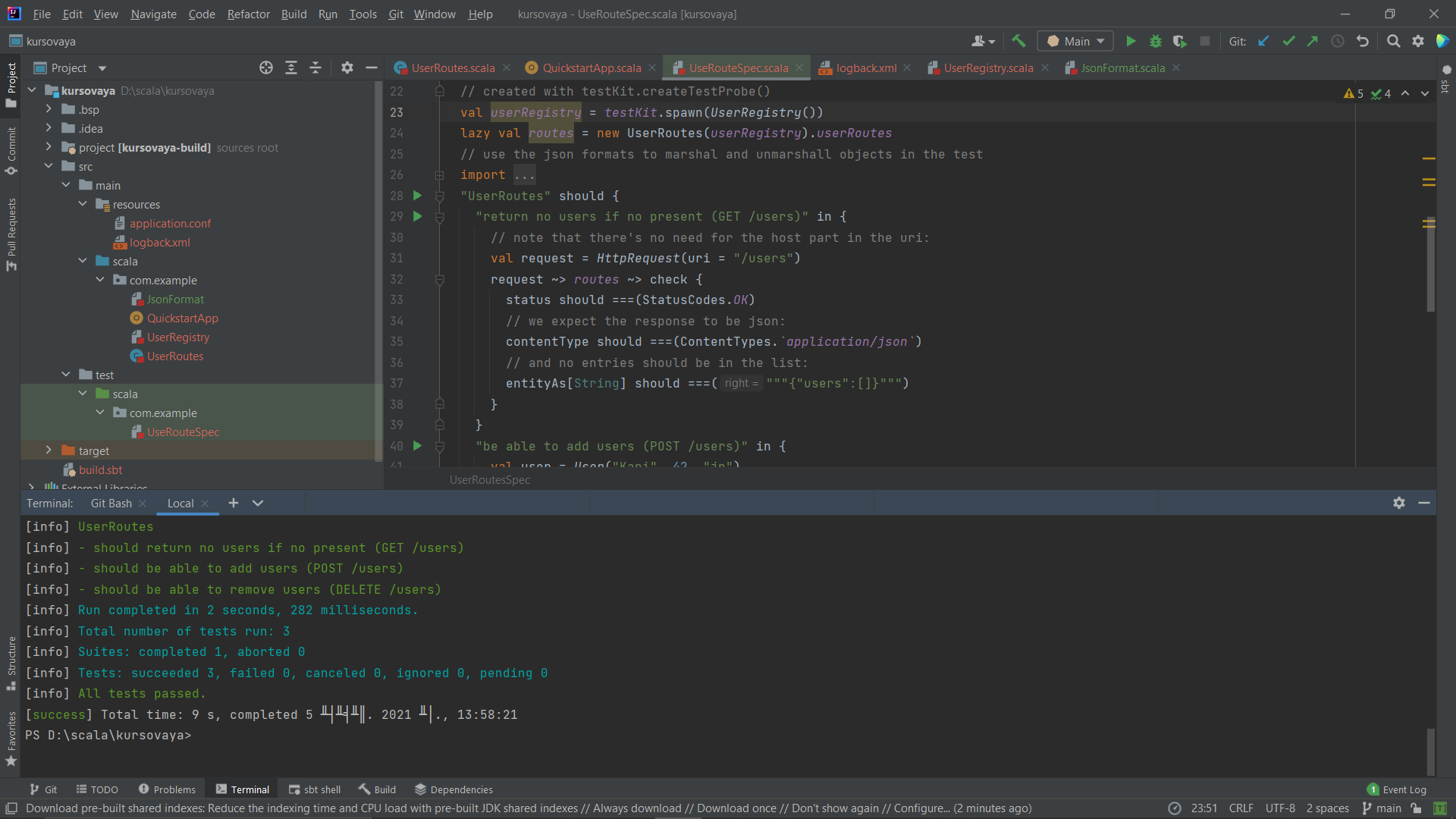


Рисунок 5 – Результат тестирования

Как мы видим, тестирование прошло успешно.

1. **Запуск приложения**

Нам необходимо запустить наше приложение и попробовать зарегистрировать несколько пользователей, получить список всех пользователей, а так же получить и удалить пользователя. У каждого пользователя есть имя, возраст, и страна проживания.

Передавать запросы можно с помощью:

* Программы командной строки cURL.
* Надстройки браузера, такие как RESTClient для FireFox или Postman для Chrome.
  1. **Команды cURL**

Зарегистрируем нескольких пользователей, результат представлен на рисунке 6.

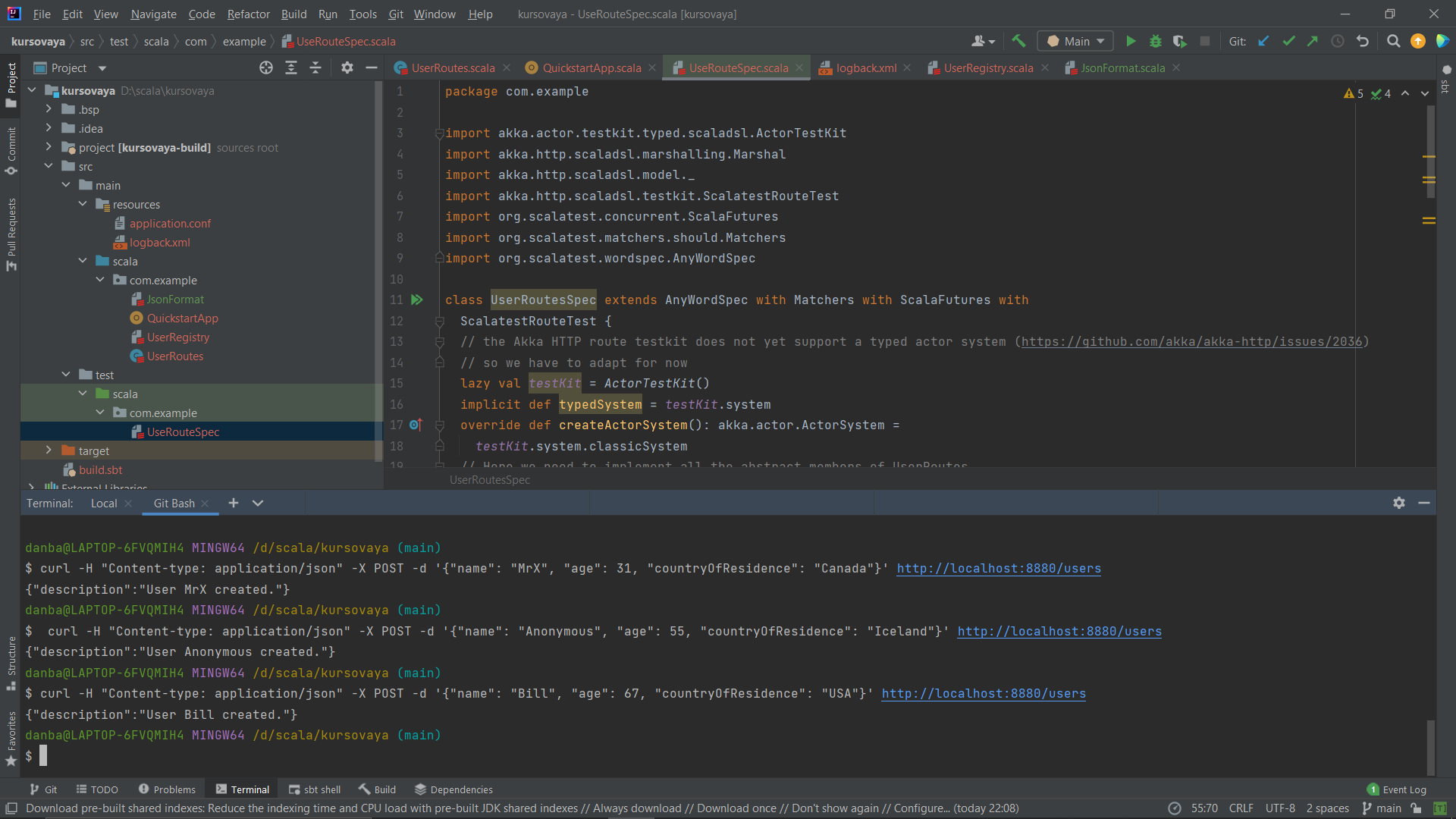


Рисунок 6 – Регистрация пользователей.

Теперь получим список всех пользователей, как на рисунке 7.

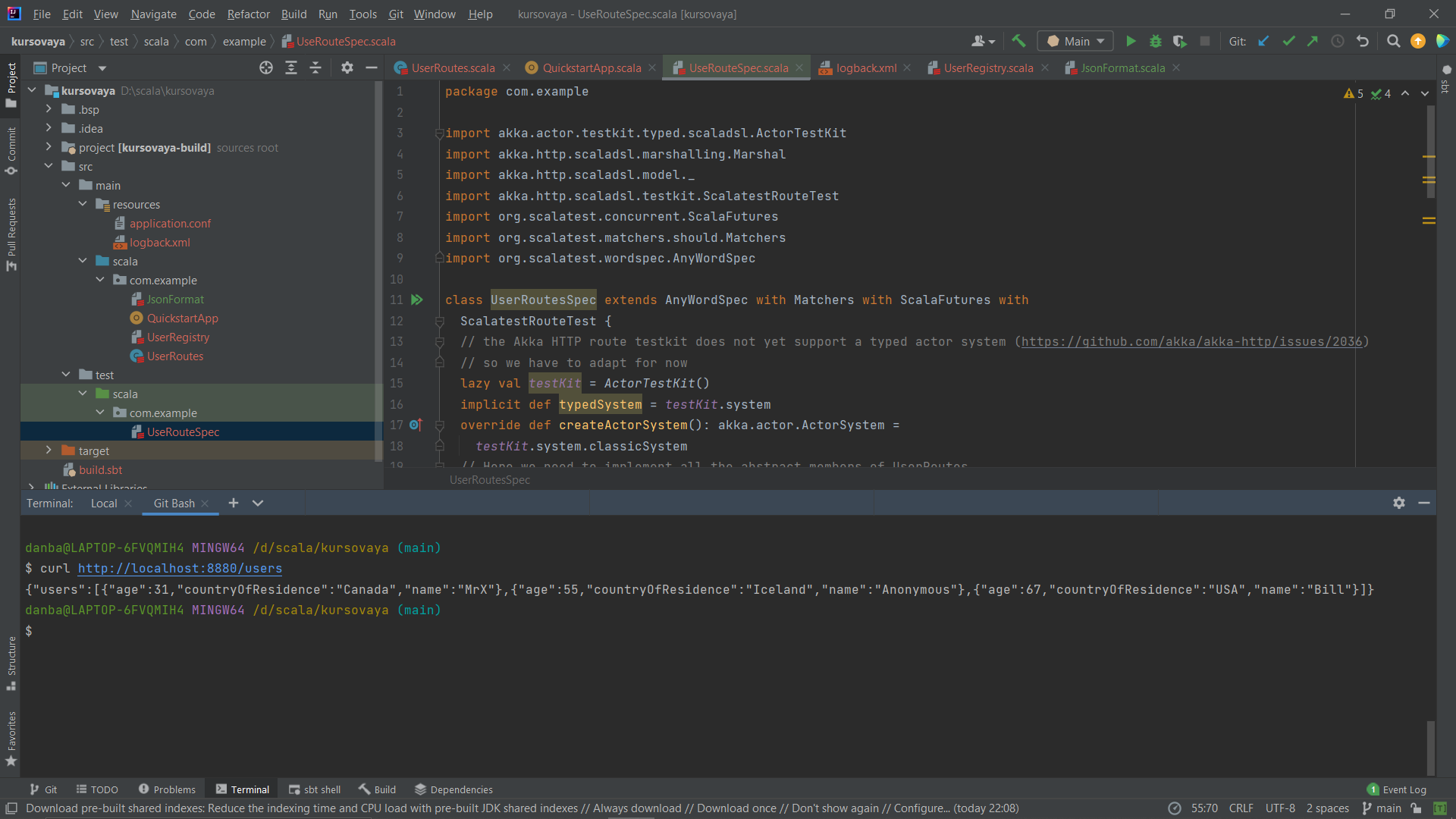


Рисунок 7 – Получение пользователей

Далее получим конкретного пользователя, как показано на рисунке 8.

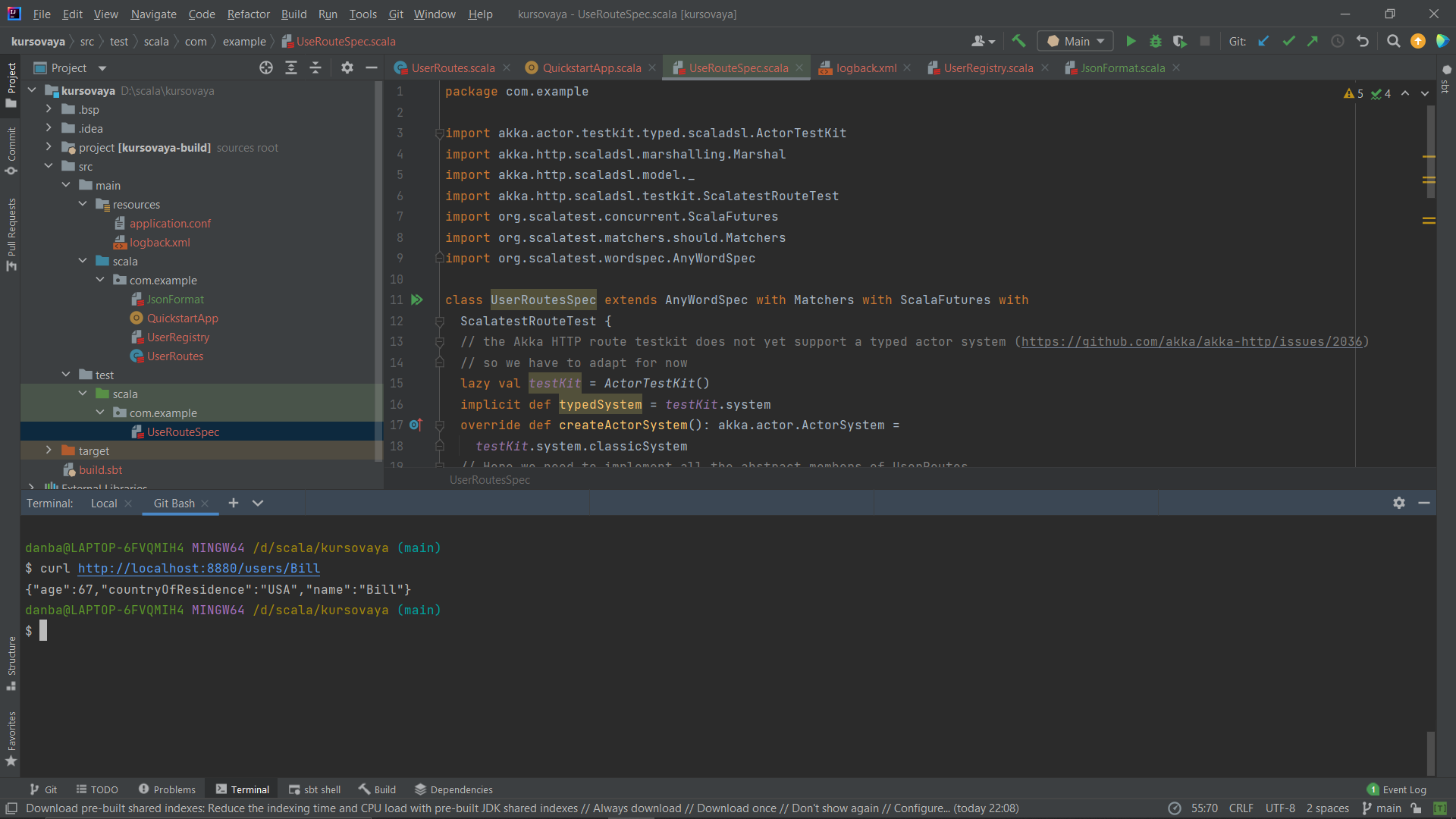


Рисунок 8 – Получение конкретного пользователя

А сейчас удалим пользователя, как на рисунке 9.

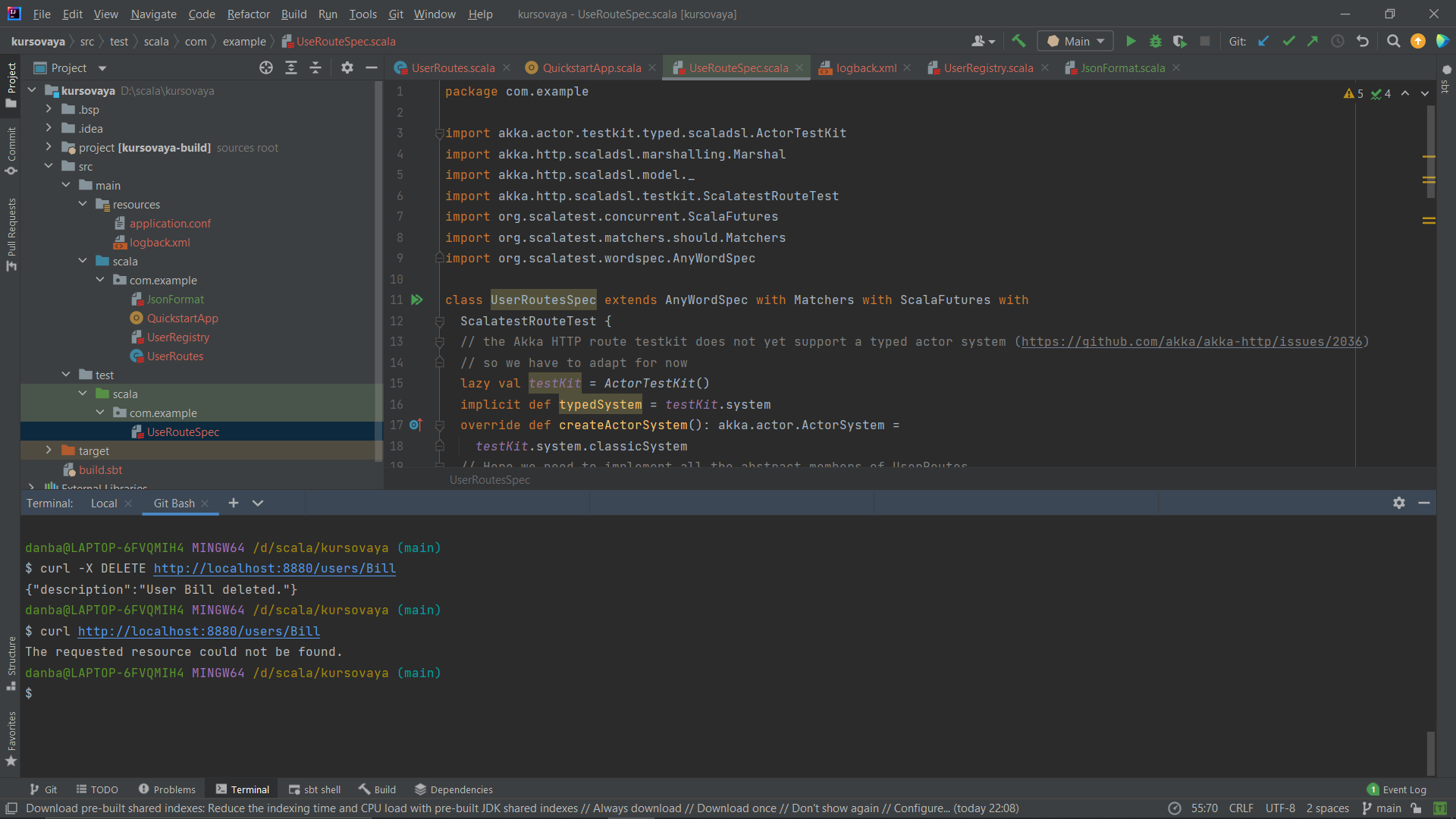


Рисунок 9 – Удаление пользователя

* 1. **Браузерные клиенты**

Откроем инструмент Postman и зарегистрируем несколько пользователей. Результат представлен на рисунках 10 – 12.

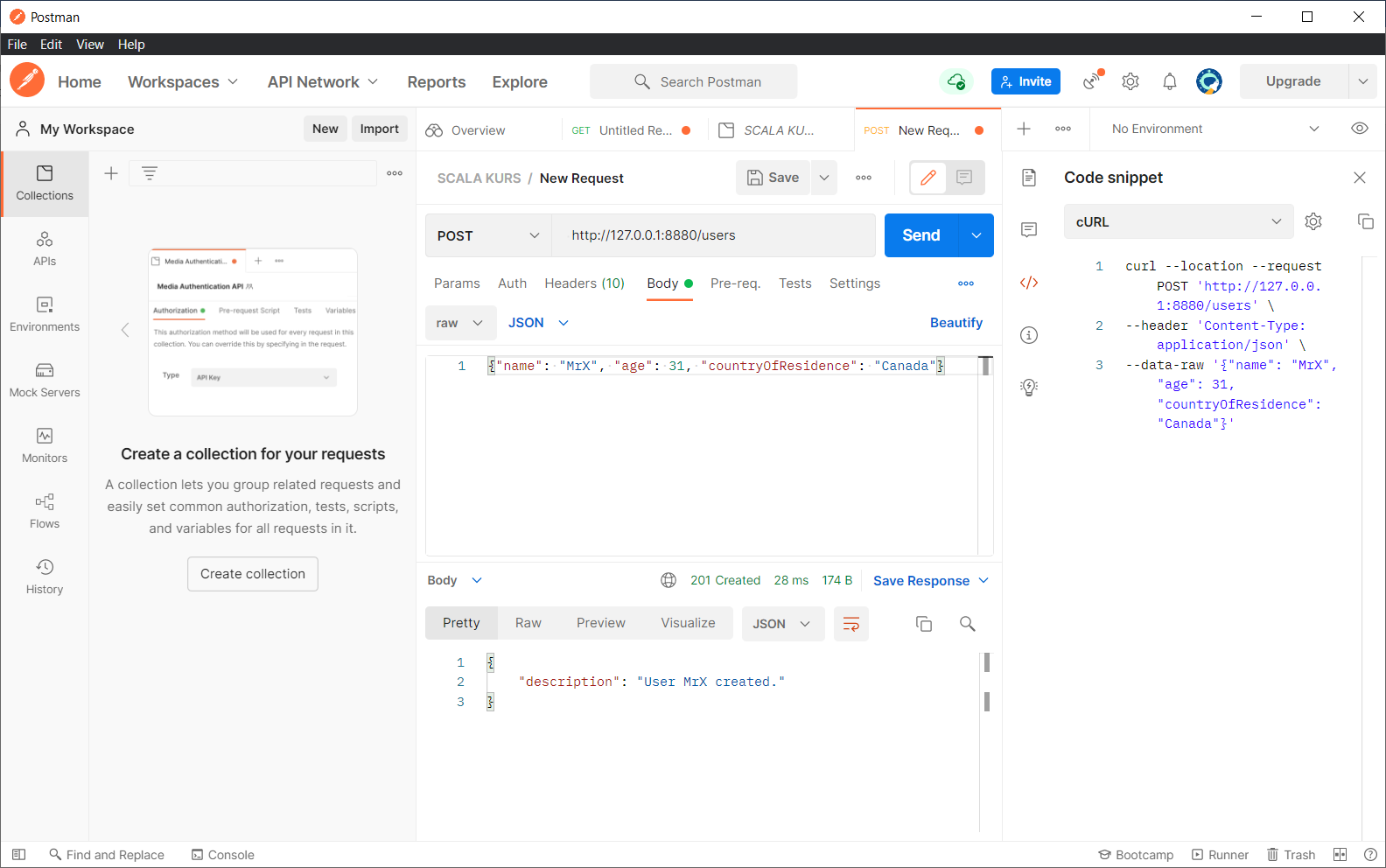


Рисунок 10 – Регистрация пользователя MrX

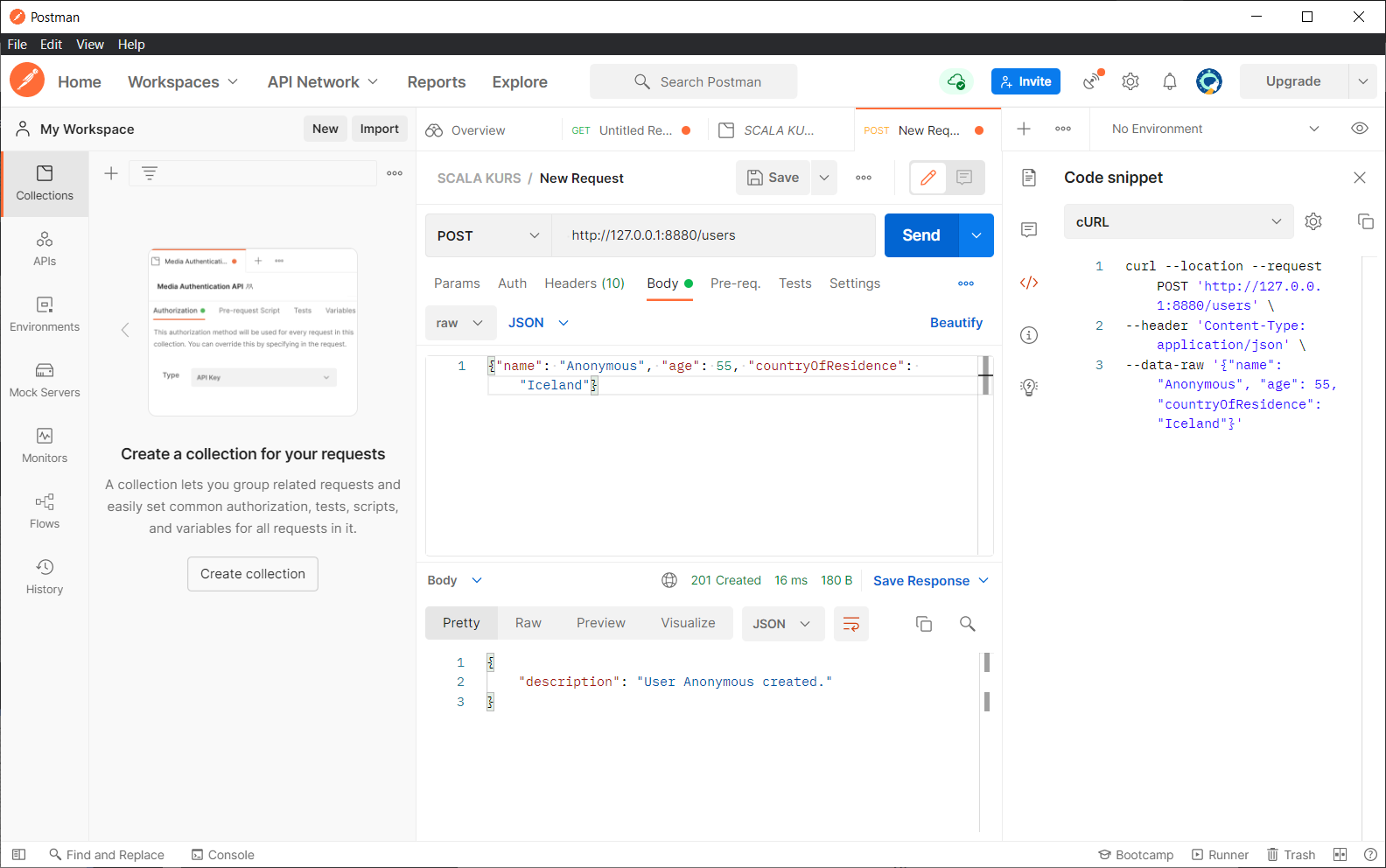


Рисунок 11 – Регистрация пользователя Anonymous

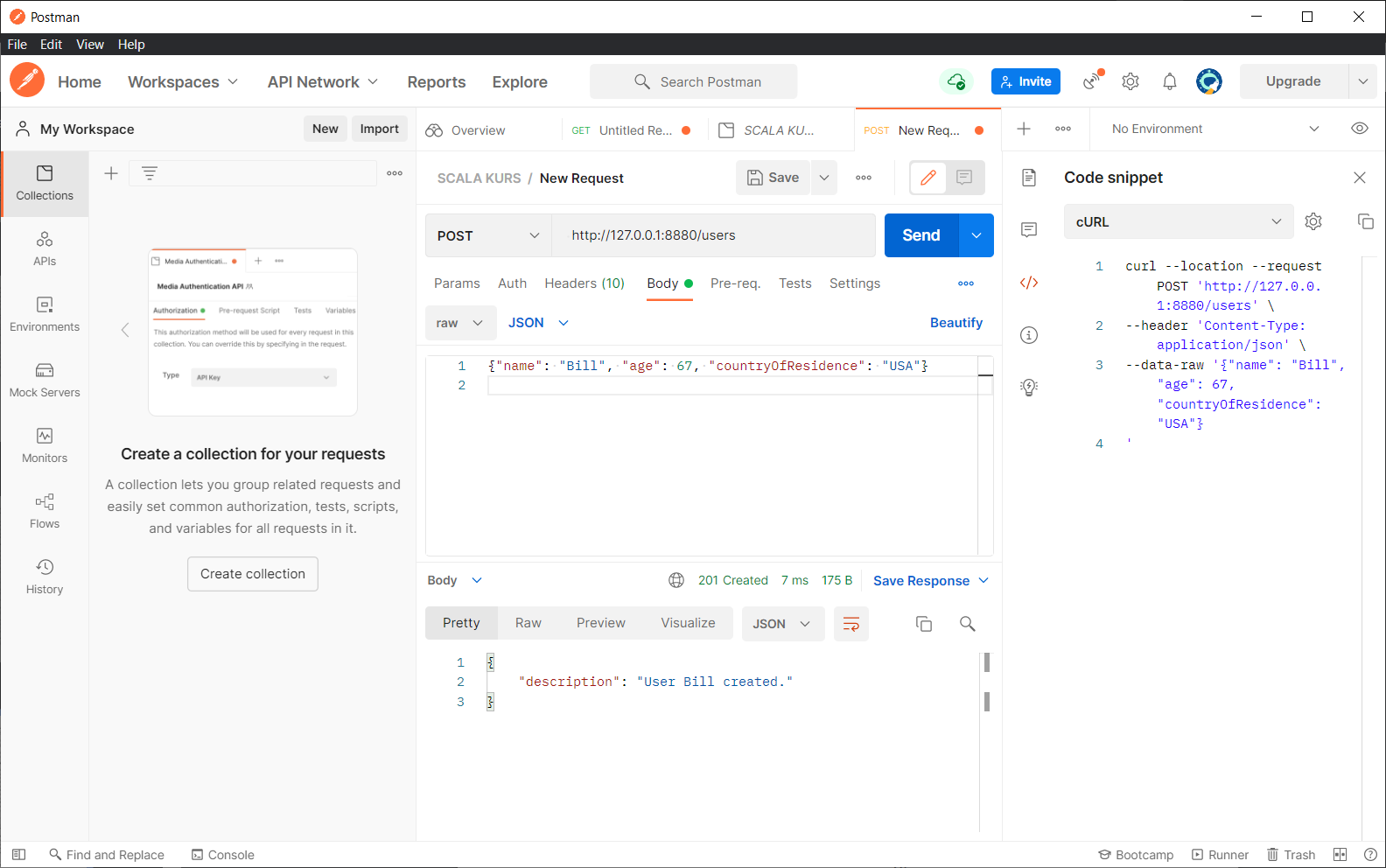


Рисунок 12 – Регистрация пользователя Bill

Теперь получим всех пользователей, как показано на рисунке 13.

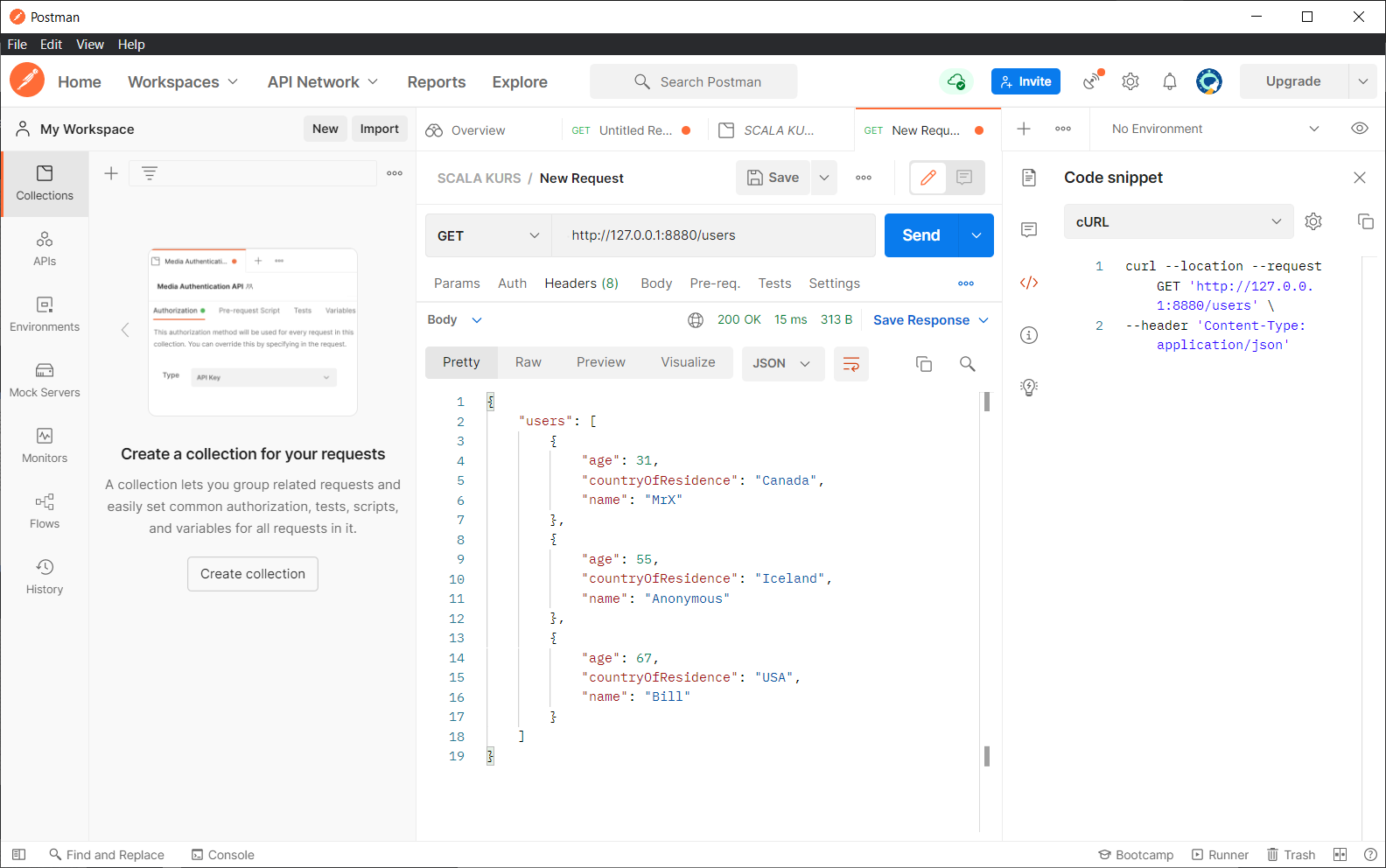


Рисунок 13 – Получение всех пользователей

Теперь перезапустим наш HTTP-сервер. Теперь без зарегистрированных пользователей, попробуем получить список всех пользователей, который должен быть пустым, как показано на рисунке 14.

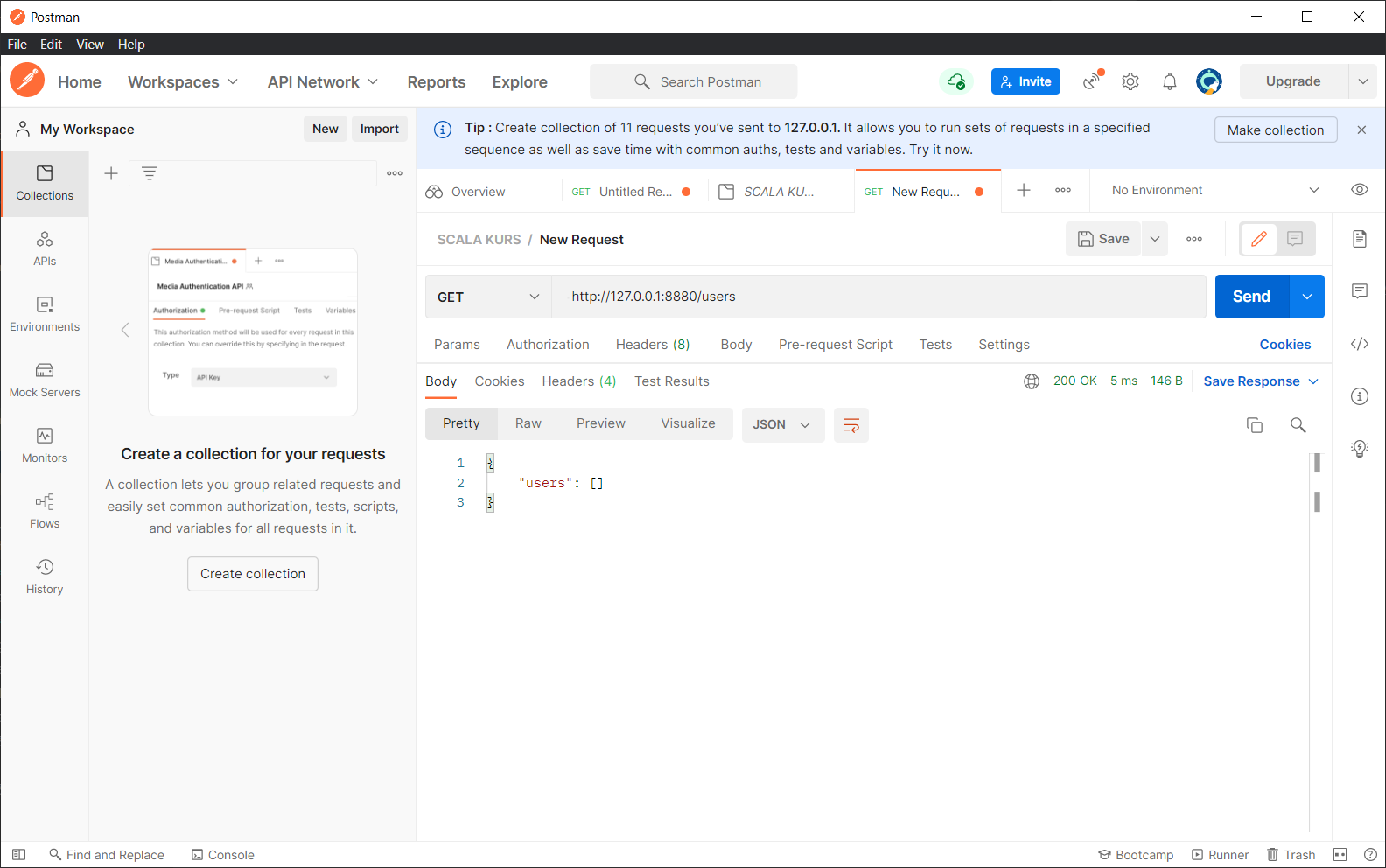


Рисунок 14 – Получение списка пользователя

Далее попробуем получить пользователя с именем MrX, как показано на рисунке 15.

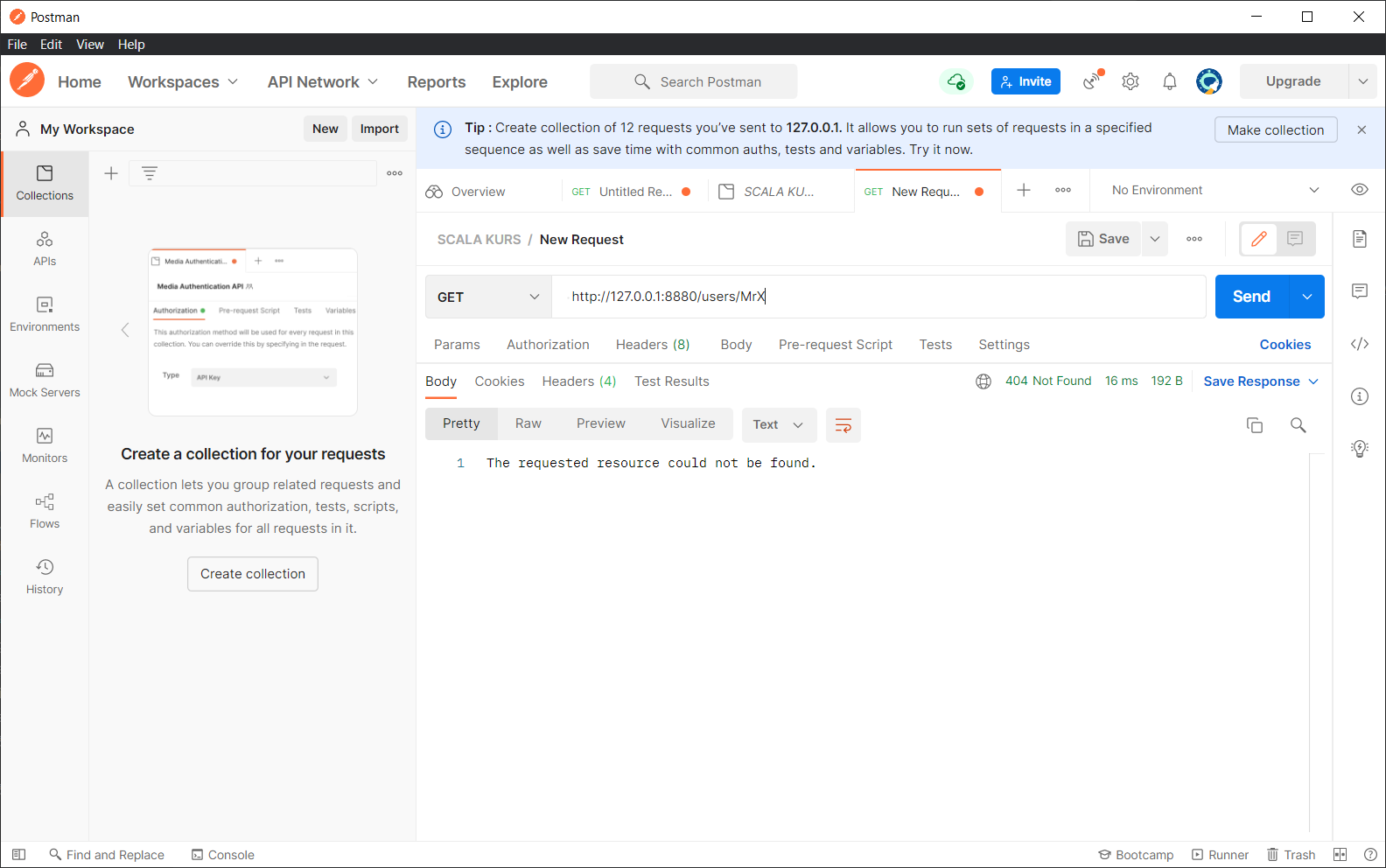


Рисунок 15 – Получение пользователя с именем MrX

Теперь добавим нового пользователя, как на рисунке 16.

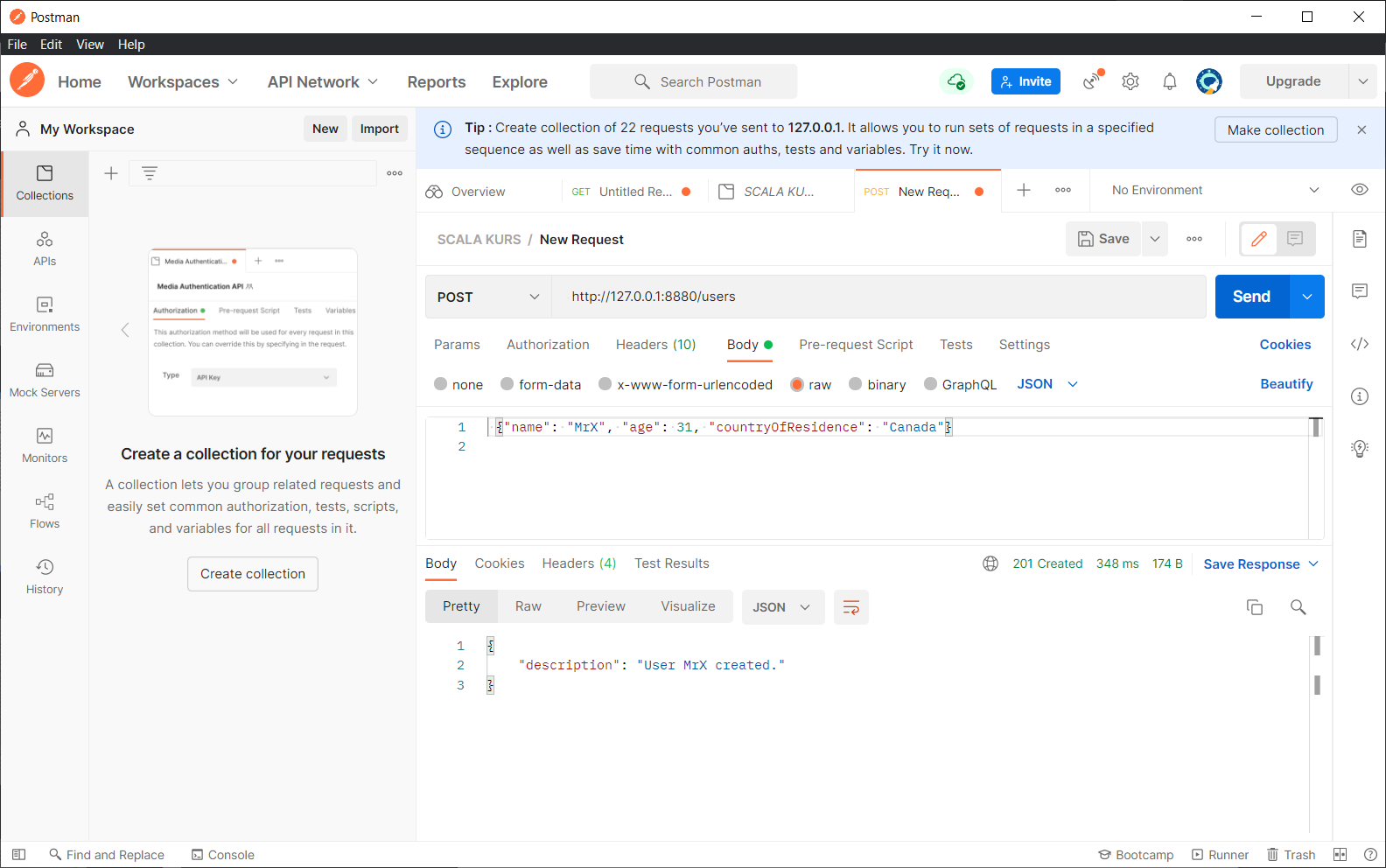


Рисунок 16 – Регистрация пользователя

Теперь удалим Пользователя, как показано на рисунке 17.

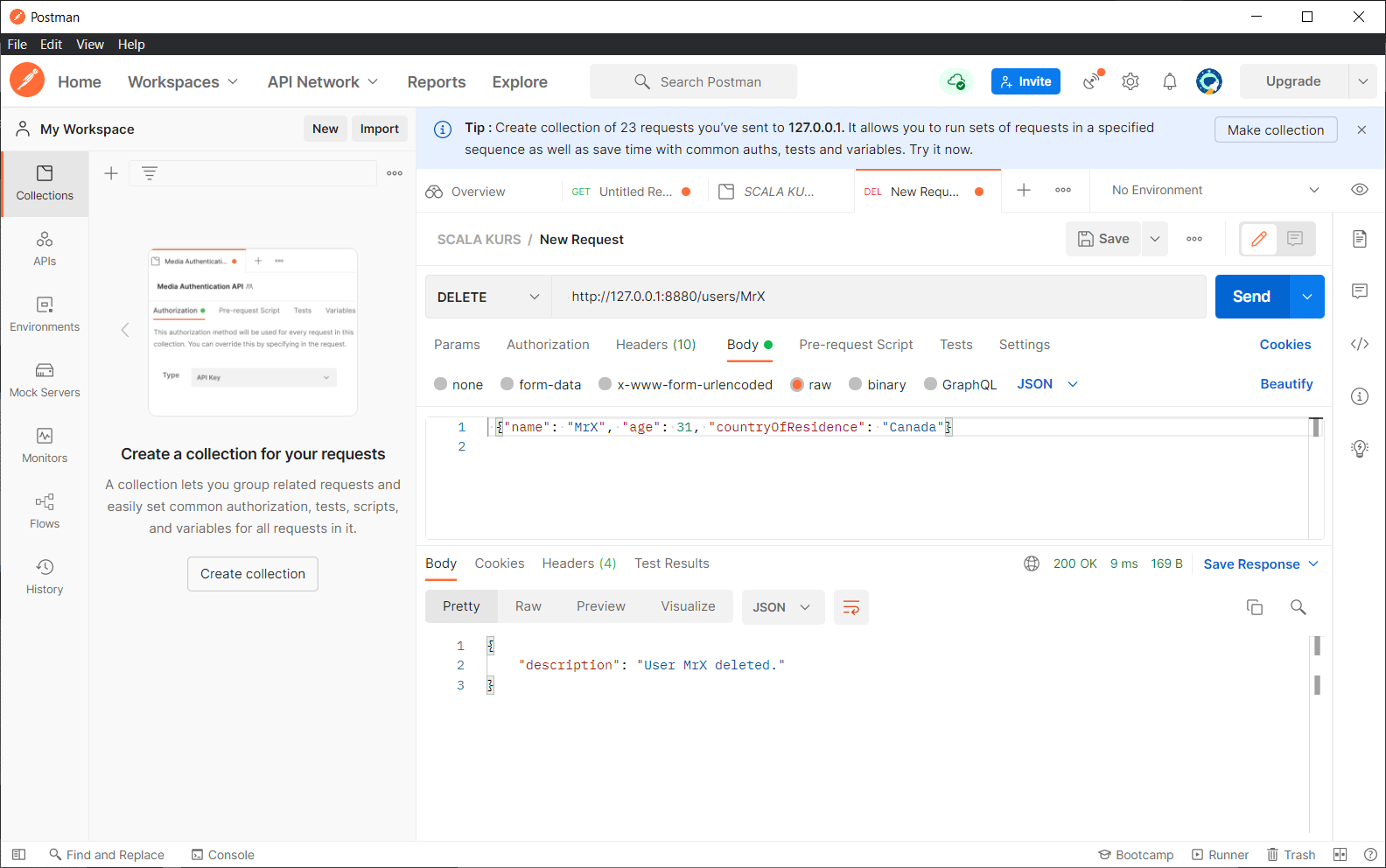


Рисунок 17 – удаление пользователя

# **Вывод**

Мы изучили библиотеку Akka HTTP, запустили свое первое приложение, получили предварительный обзор того, как маршруты упрощают обмен данными HTTP, а так же провели тестирование.