

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**"МИРЭА - Российский технологический университет"**

**РТУ МИРЭА**

Институт информационных технологий (ИТ)

Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

**ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №4**

**по дисциплине**

**«Разработка клиент-сервисных мобильных приложений»**

Выполнил студент группы ИКБО-11-22 Гришин А. В.

Приняла Исабекова О. А.

Москва 2025

Ktor (произносится как Кэй-тор) - это созданный с нуля фреймворк, основу которого составляет Kotlin и корутины. С помощью Ktor можно создавать клиентские и серверные приложения, которые будут работать на разных платформах. Фреймворк отлично подходит для приложений, требующих связь по HTTP и/или сокетам. Это могут быть, например, HTTPбэкенды и RESTful-системы, независимо от того, построены ли они по микросервисному принципу или нет.

Ktor родился под влиянием других фреймворков, таких как Wasabi и Kara, с целью максимально использовать некоторые возможности языка Kotlin, такие как DSL (внутренний предметно-ориентированный язык) и корутины. Если необходимо создать системы, которые будут связаны друг с другом, Ktor отлично подходит для этого и является производительным, асинхронным, многоплатформенным решением.

В настоящее время клиентская часть Ktor работает на всех платформах, на которые ориентирован Kotlin, то есть JVM, JavaScript и Native. Однако, серверная часть Ktor пока ограничена только JVM.

Рассмотрим пример простейшего серверного приложения, созданного с помощью Ktor:

**1 Создание проекта Ktor.**

Для создания нового проекта Ktor необходимо использовать веб-генератор проектов Ktor (рис. 1).

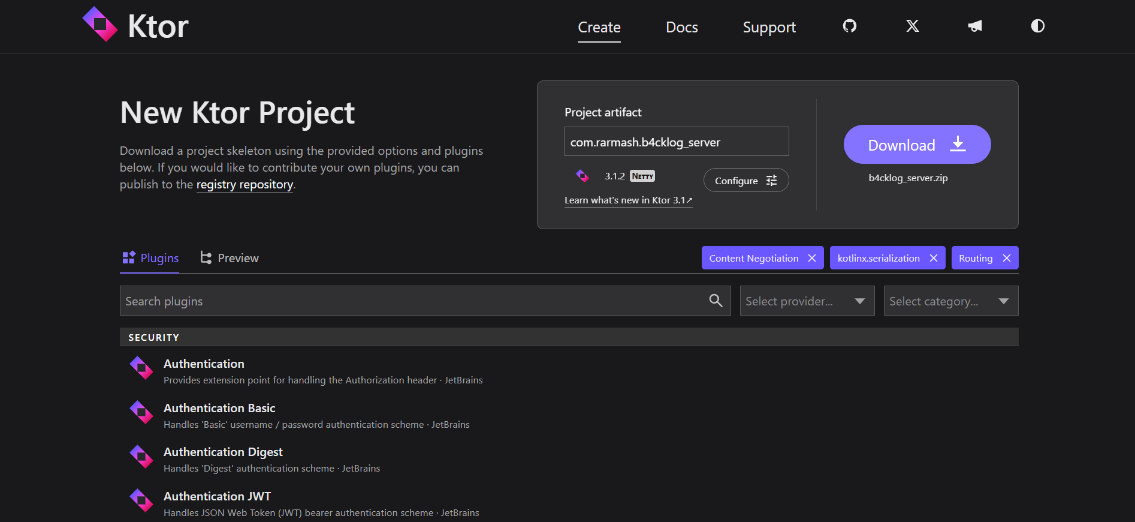


Рисунок 1 – Генератор проектов

**2. Запустите проект в любой IDE (рис 2).**

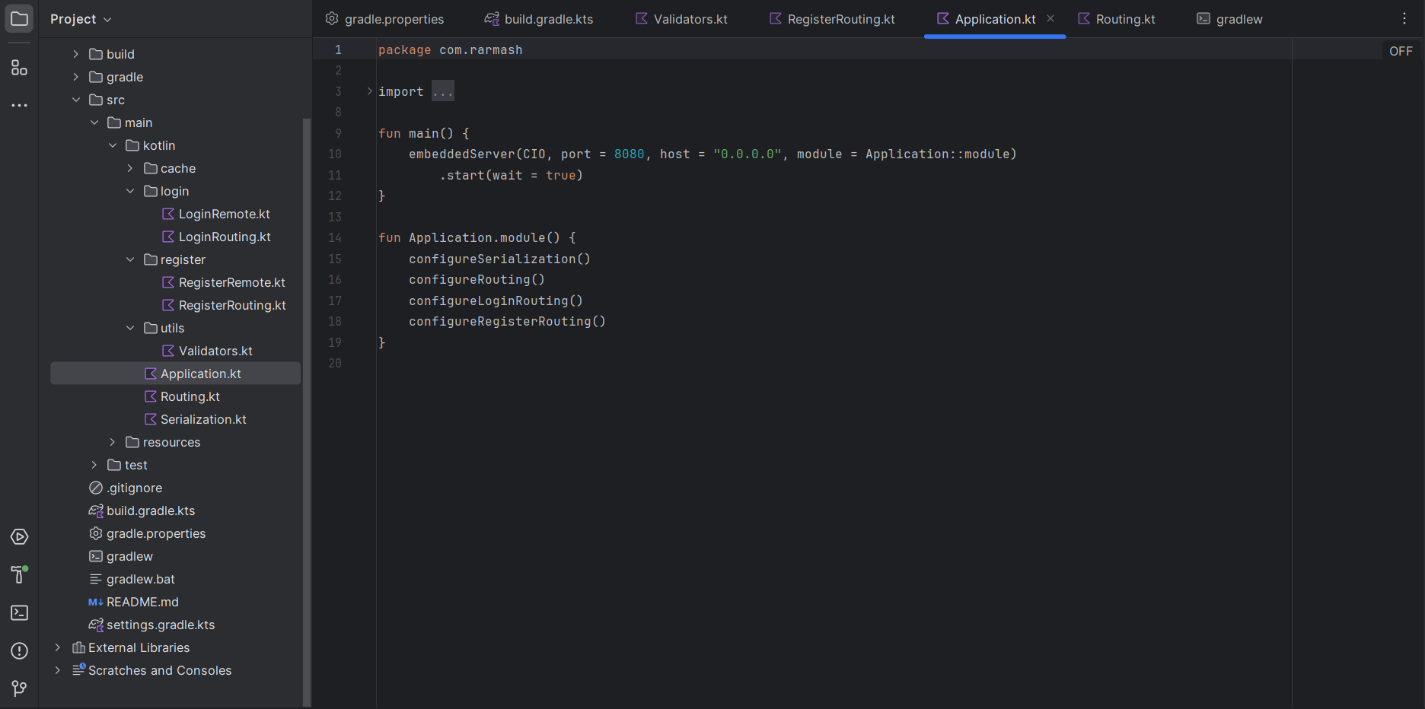
****

Рисунок 2 – Созданный проект

Запустим Postman, и отправим GET-запрос по адресу, указанному в файле Application.kt (host и port) с применением call.respondText. Укажите в отчете, какой ответ был получен (листинг 1, рис. 3).

*Листинг 1 – Routing.kt*

|  |
| --- |
| package com.rarmash  import io.ktor.serialization.kotlinx.json.\* import io.ktor.server.application.\* import io.ktor.server.plugins.contentnegotiation.\* import io.ktor.server.response.\* import io.ktor.server.routing.\* import kotlinx.serialization.Serializable  fun Application.configureRouting() {  routing {  get("/") {  call.respondText("Hello World!")  }  } } |

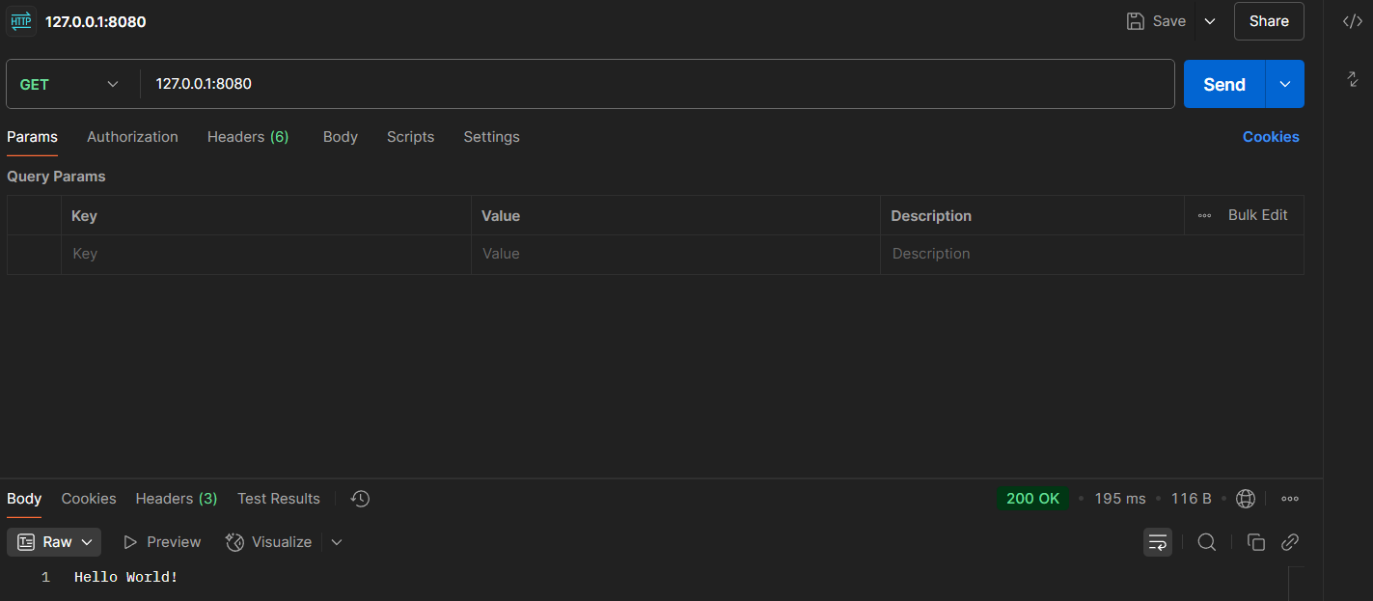


Рисунок 3 – Запрос в Postman

Запрос успешно обрабатывается на сервере. Добавим data class Test, который содержит поле text строкового формата. Функцию configureRouting() меняем таким образом, чтобы в качестве ответа на GET-запрос был получен объект класса Test (листинг 2).

*Листинг 2 – класс Test и метод обращения к этому классу*

|  |
| --- |
| @Serializable data class Test(  val text: String )  fun Application.configureRouting() {  routing {  get("/") {  call.respond(Test("Hello World!"))  }  } } |

Проверим работоспособность сервера после изменений в Postman (рис 4).

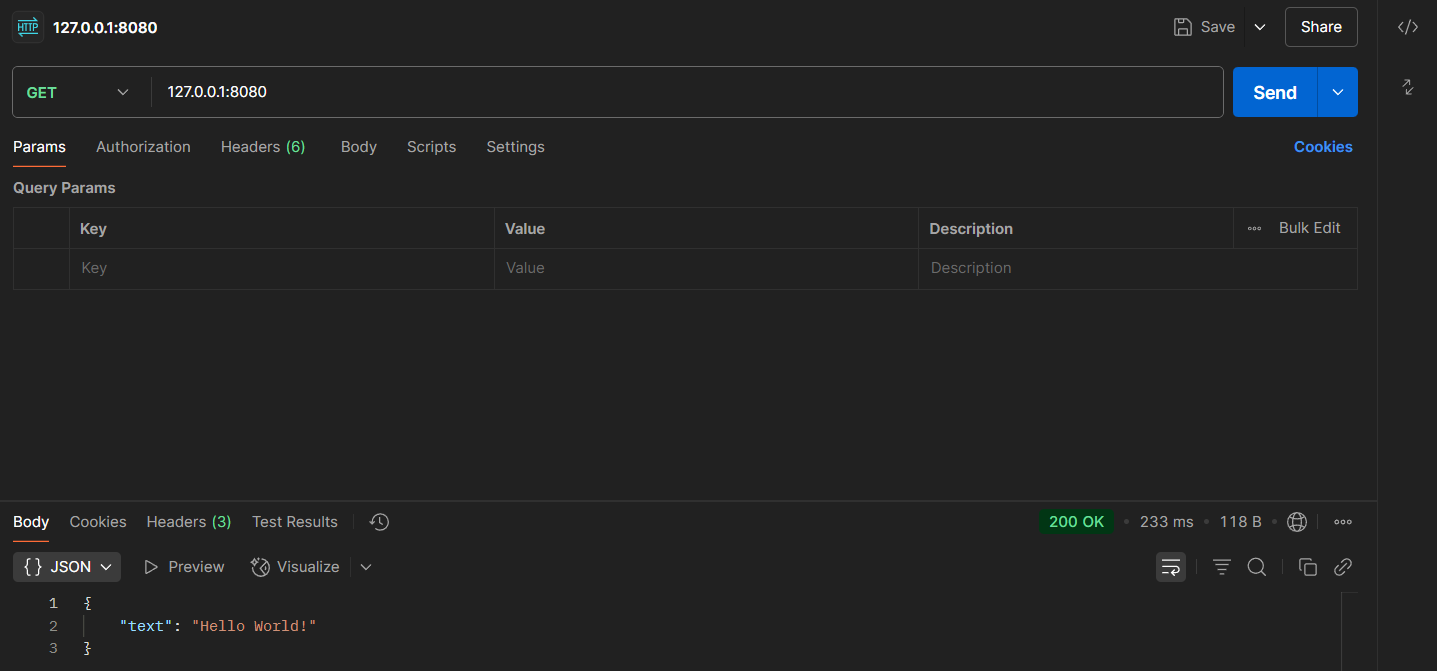


Рисунок 4 – Полученный ответ

**3. Регистрация и логин**

Создадим пакет login, а в нем класс LoginRemote (листинг 3).

*Листинг 3 – LoginRemote.kt*

|  |
| --- |
| package com.rarmash.login  import kotlinx.serialization.Serializable  @Serializable data class LoginReceiveRemote(  val login: String,  val password: String )  @Serializable data class LoginResponseRemote(  val token: String ) |

Создадим пакет register, а в нем класс RegisterRemote (листинг 4).

*Листинг 4 – RegisterRemote.kt*

|  |
| --- |
| package com.rarmash.register  import kotlinx.serialization.Serializable  @Serializable data class RegisterReceiveRemote(  val login: String,  val email: String,  val password: String )  @Serializable data class RegisterResponseRemote(  val token: String ) |

В данных классах мы указываем, какие данные мы будем отправлять при запросах и получать в ответ. Таким образом, при регистрации мы отправляем логин, пароль и почту, а в ответ получим токен, то есть уникальный номер пользователя, а при входе (логине) отправляем логин и пароль, в ответ, также, получаем токен.

Создадим пакет cache, а в нем класс InMemoryCache (листинг 5).

*Листинг 5 – InMemoryCache.kt*

|  |
| --- |
| package com.rarmash.cache  import com.rarmash.register.RegisterReceiveRemote  data class TokenCache(  val login: String,  val token: String )  object InMemoryCache {  val userList: MutableList<RegisterReceiveRemote> = mutableListOf()  val token: MutableList<TokenCache> = mutableListOf() } |

Рисунок 9 – Класс InMemoryCache

На данном этапе мы не подключаем базу данных, поэтому хранить все данные будем в этом классе.

В пакете login создайте класс LoginRouting (листинг 6).

*Листинг 6 – LoginRouting.kt*

|  |
| --- |
| package com.rarmash.login  import com.rarmash.cache.InMemoryCache import com.rarmash.cache.TokenCache import io.ktor.http.\* import io.ktor.serialization.kotlinx.json.\* import io.ktor.server.application.\* import io.ktor.server.plugins.contentnegotiation.\* import io.ktor.server.request.\* import io.ktor.server.response.\* import io.ktor.server.routing.\* import kotlinx.serialization.Serializable import java.util.\*  fun Application.configureLoginRouting() {  routing {  post("/login") {  val receive = call.receive<LoginReceiveRemote>()  if (InMemoryCache.userList.map { it.login }.contains(receive.login)) {  val token = UUID.randomUUID.toString()  InMemoryCache.token.add(TokenCache(login = receive.login, token = token))  call.respond(LoginResponseRemote(token = token))  return@post  }  call.respond(HttpStatusCode.BadRequest)  }  } } |

Переменная receive будет содержать тело запроса в формате LoginRemote. Если пользователь уже зарегистрирован, то ему будет присвоен токен.

В пакете register создайте класс RegisterRouting (листинг 7).

*Листинг 7 – RegisterRouting.kt*

|  |
| --- |
| package com.rarmash.register  import com.rarmash.cache.InMemoryCache import com.rarmash.cache.TokenCache import io.ktor.http.\* import io.ktor.serialization.kotlinx.json.\* import io.ktor.server.application.\* import io.ktor.server.plugins.contentnegotiation.\* import io.ktor.server.request.\* import io.ktor.server.response.\* import io.ktor.server.routing.\* import kotlinx.serialization.Serializable import com.rarmash.utils.isValidEmail import java.util.\*  fun Application.configureRegisterRouting() {  routing {  post("/register") {  val receive = call.receive<RegisterReceiveRemote>()  if (!receive.email.isValidEmail()) {  call.respond(HttpStatusCode.BadRequest, "Email is not valid")  } else if (InMemoryCache.userList.map {it.login}.contains(receive.login)) {  call.respond(HttpStatusCode.Conflict, "User already exists")  } else {  val token = UUID.randomUUID().toString()  InMemoryCache.userList.add(receive)  InMemoryCache.token.add(TokenCache(login = receive.login, token = token))   call.respond(RegisterResponseRemote(token = token))  }  call.respond(HttpStatusCode.BadRequest)  }  } } |

В пакете utils создайте класс Validators. Данный класс необходим для проверки на правильность введенной электронной почты (листинг 8).

*Листинг 8 – Validators.kt*

|  |
| --- |
| package com.rarmash.utils  fun String.isValidEmail(): Boolean {  return Regex("^[\\w.-]+@[\\w.-]+\\.\\w+$").matches(this) } |

**4. Проверка работоспособности сервера**

Проверьте функцию регистрации с помощью программы Postman (рисунок 5).

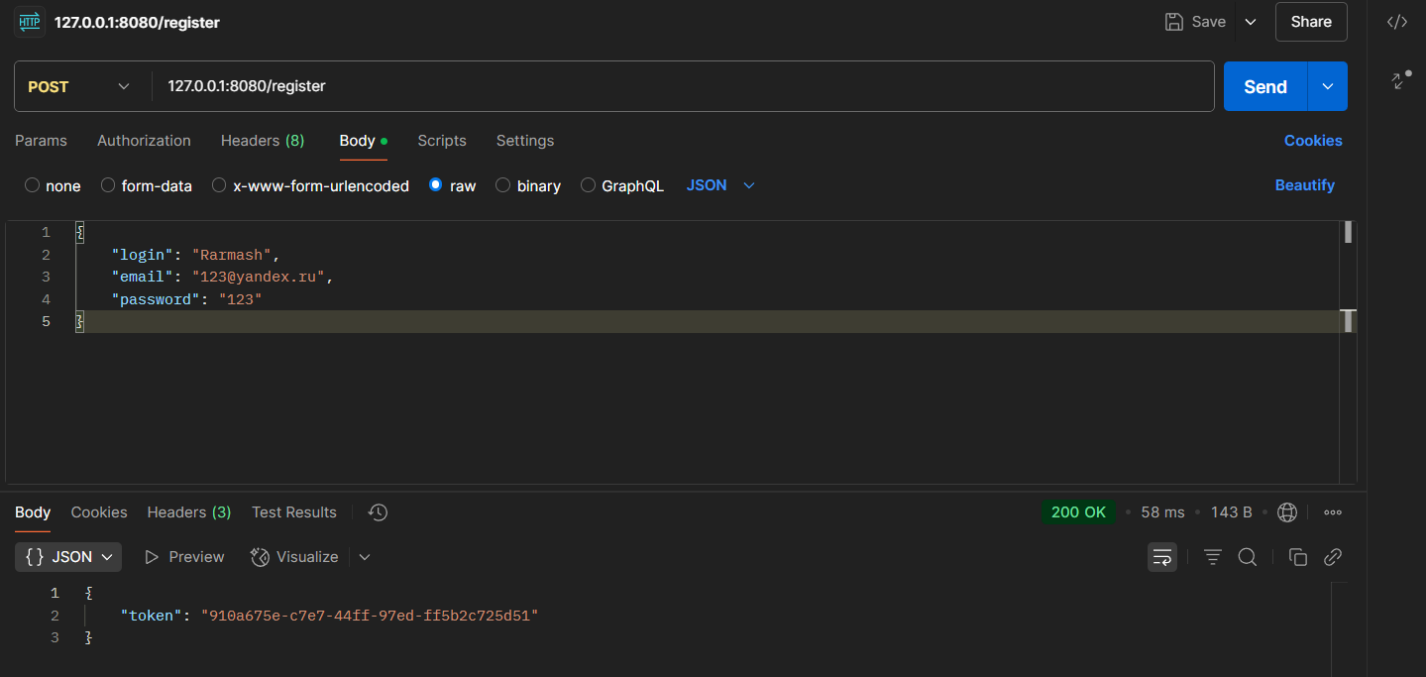


Рисунок 5 – Проверка работы функции регистрации

Перезапустите сервер. Данные зарегистрированного пользователя не должны сохраниться. Проверьте функцию входа (логина) с помощью программы Postman (рисунок 6).

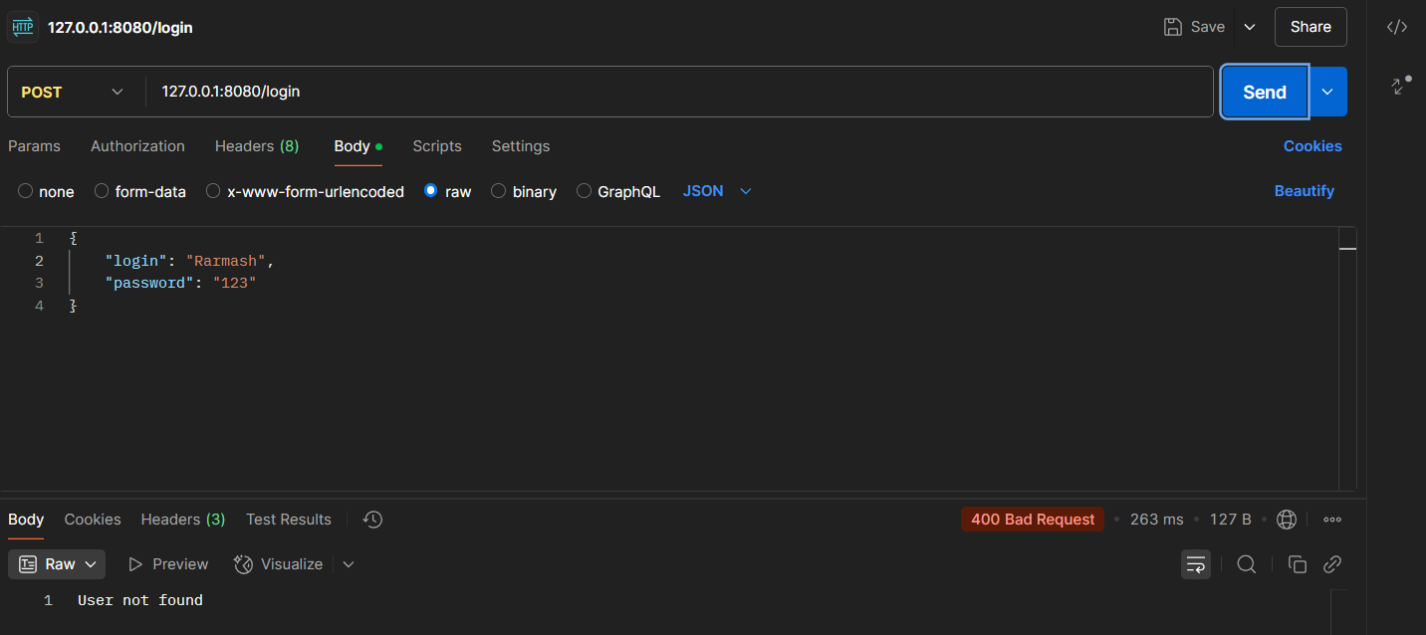


Рисунок 6 – Проверка работы функции входа

Так как, при перезапуске сервера данные зарегистрированного пользователя не сохранились, то необходимо провести регистрацию заново (рисунок 7).

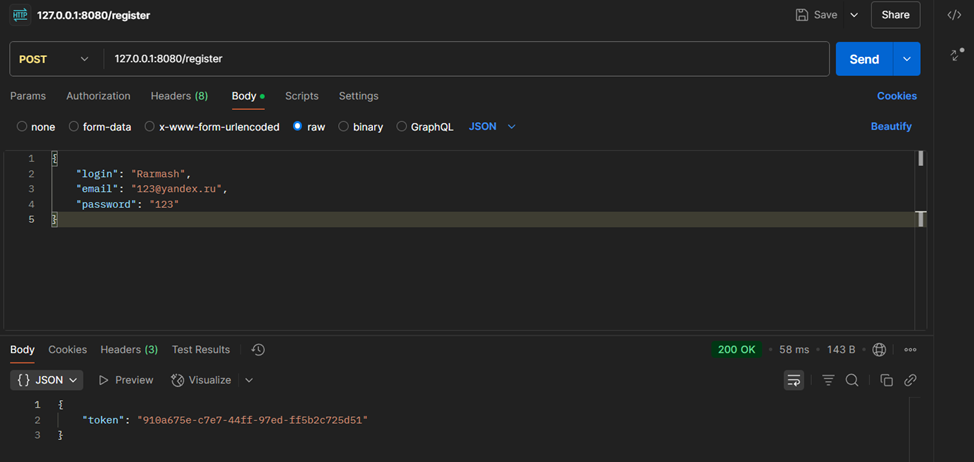


Рисунок 7 – Проверка работы функции регистрации

Повторно производим вход (логин) (рисунок 8).

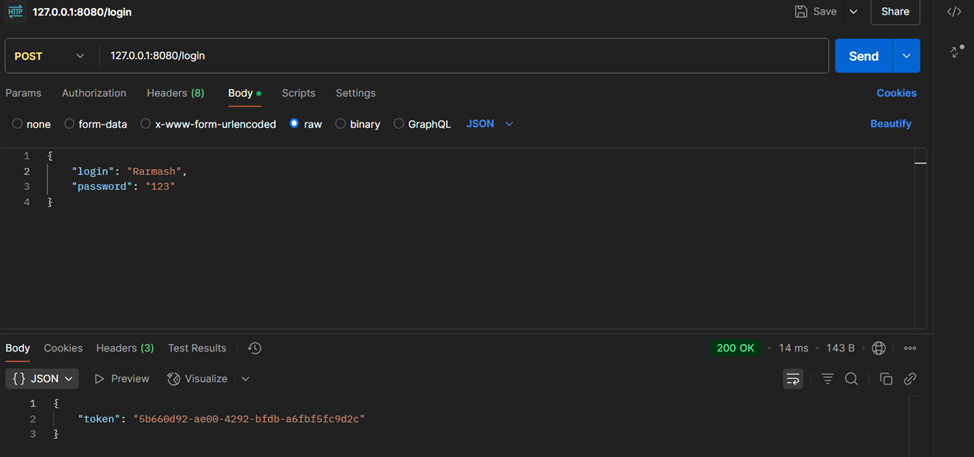


Рисунок 8 – Проверка работы функции входа

Если в обоих случаях в ответ пришел токен, значит сервер работает верно. Теперь созданный сервер можно подключать к клиенту.

**Вывод**

В ходе выполнения практической работы была успешно реализована серверная часть приложения с использованием фреймворка **Ktor** на языке **Kotlin**. В качестве первого этапа была настроена сериализация данных с использованием плагинов **Content Negotiation** и **kotlinx.serialization**, а также создана простейшая модель Test, возвращающая JSON-ответ на GET-запрос. Далее были разработаны механизмы регистрации и авторизации пользователей без подключения к базе данных — данные временно сохранялись в памяти с использованием класса InMemoryCache.

Были созданы модели запросов и ответов RegisterRemote и LoginRemote, а также реализованы маршруты /register и /login для POST-запросов. Регистрация включала проверку валидности email с помощью пользовательской функции валидации. Все маршруты корректно обрабатывали входящие данные, возвращая соответствующие ответы (например, токен пользователя, либо сообщение об ошибке — «Email is not valid», «User already exists», «User not found», «Invalid password» и др.).

Работоспособность сервера была проверена с помощью **Postman**: в результате тестирования сервер успешно обрабатывал запросы, возвращая ожидаемые JSON-ответы. При перезапуске сервера данные сбрасывались, что подтверждает использование хранения в оперативной памяти. Работа показала высокую отзывчивость, читаемость кода и удобство фреймворка **Ktor** для построения простых и эффективных REST-серверов.

**Список литературы**

1) Android Architecture Components | Android Developers [Электронный ресурс] — URL: https://developer.android.com/topic/libraries/architecture (дата обращения: 16.04.2025).

2) Android Studio | Android Developers [Электронный ресурс] — URL: https://developer.android.com/studio (дата обращения: 16.04.2025).

3) Официальная документация Kotlin [Электронный ресурс] // KotlinLang – Электрон. дан. – [Б. м., б. г.] – URL: https://kotlinlang.org/docs/home.html (дата обращения: 16.04.2025).

4) Официальная документация Android [Электронный ресурс] — URL: https://developer.android.com/docs (дата обращения: 16.04.2025).

5) Сомон., П. И. Волшебство Kotlin : руководство / П. И. Сомон. ; перевод с английского А. Н. Киселева.. — Москва : ДМК Пресс, 2020. — 536 с. — ISBN 978-5-97060-801-2. — Текст : электронный // Лань : электроннобиблиотечная система. — URL: https://e.lanbook.com/book/140599 (дата обращения: 15.02.2024). — Режим доступа: для авториз. пользователей.

6) Калгина, И. С. Разработка мобильных приложений : учебное пособие / И. С. Калгина. — Чита : ЗабГУ, 2022. — 163 с. — ISBN 978-5-9293-3137- 4. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/363323 (дата обращения: 15.02.2024). — Режим доступа: для авториз. пользователей.

7) Алпатов, А. Н. Архитектура, проектирование и разработка программных средств : учебное пособие / А. Н. Алпатов, И. Е. Рогов. — Москва : РТУ МИРЭА, 2023. — 120 с. — ISBN 978-5-7339-1972-0. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/386189 (дата обращения: 15.02.2024). — Режим доступа: для авториз. пользователей.