ФЕДЕРАЛЬНОЕ АГЕНСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ

ОРДЕНА ТРУДОВОГО КРАСНОГО ЗНАМЕНИ

БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО

ОБРАЗОВАНИЯ

МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ СВЯЗИ И ИНФОРМАТИКИ

(МТУСИ)

Кафедра Информационной безопасности

Курсовая работа

По дисциплине

«Разработка безопасного программного обеспечения»

На тему

«Мобильное приложение»

Выполнили студенты группы БАС2101 Герейакаев Гаджи

Казинский Кирилл

Сердюк Данил

Проверил доц. кафедры \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва 2023

Оглавление

[Анализ выбранной темы 3](#_Toc145757782)

[Проектирование серверной части 5](#_Toc145757783)

[Проектирование структуры базы данных 6](#_Toc145757784)

[Разработка 7](#_Toc145757785)

[Тестирование 8](#_Toc145757786)

[Описание клиентской части 9](#_Toc145757787)

[Демонстрация функционала 10](#_Toc145757788)

# Анализ выбранной темы

Тема: Мобильное приложение агрегатор с новостной сводкой и погодой в городе пользователя на платформе iOS с использованием серверной части на Java и фреймворка Spring Bootы

1. Введение:

Мобильные приложения-агрегаторы пользуются большой популярностью из-за своей функциональности. В данной курсовой работе рассматривается разработка мобильного приложения-агрегатора на платформе iOS с учетом серверной части, реализованной на Java с использованием фреймворка Spring Boot.

2. Цели и задачи:

- Целью данной работы является разработка полнофункционального мобильного приложения-агрегатора на устройствах iOS.

- Задачи включают в себя проектирование пользовательского интерфейса приложения, реализацию функциональности для загрузки, хранения и просмотра данных, а также создание серверной части для обеспечения аутентификации и авторизации пользователей.

3. Технологический стек:

- Мобильное приложение будет разработано для платформы iOS, что предполагает использование языка программирования Swift и инструментария Apple Xcode.

- Серверная часть будет создана с использованием Java и фреймворка Spring Boot, что обеспечит высокую производительность и масштабируемость приложения.

- Для хранения данных будет использоваться реляционная база данных (например, PostgreSQL или MySQL).

- Для методов, реализующих логику, будут написаны модульные тесты, которые будут выполняться во время сборки проекта.

- Взаимодействие с базой данных будет происходить при помощи фреймворка Hibernate и языка запросов HQL

4. Функциональные требования:

- Регистрация и аутентификация пользователей: Пользователи должны иметь возможность создавать аккаунты, а также входить в приложение с помощью своих учетных данных.

- Просмотр данных о погоде на ближайшие 7 дней

- Чтение последних новостей из города пользователя.

5. Ожидаемые результаты:

- Разработка мобильного приложения для iOS с функциональностью приложения-агрегатора

- Разработка серверной части на Java с использованием Spring Boot для обеспечения аутентификации, авторизации и управления данными пользователей и книг.

- Работающее приложение с возможностью регистрации, входа, просмотра погода и новостей.

# Проектирование серверной части

1. Модель *OpenWeatherMapResponse*:

- Создан класс *OpenWeatherMapResponse*, представляющий структуру данных, получаемых от OpenWeatherMap API.

- Для каждого поля в API-ответе созданы соответствующие переменные класса с аннотациями `@Getter` и `@JsonIgnoreProperties`.

- Разбиты поля на несколько внутренних классов: Weather, Main, Wind, Rain, Clouds, и Sys. Каждый из этих классов представляет подсекцию данных в ответе API и также содержит геттеры для полей.

2. Контроллер *WeatherController*:

- Создан класс *WeatherController*, который обрабатывает HTTP-запросы, связанные с погодой.

- Использует аннотацию @RestController и задает базовый URL пути "/weather".

- В контроллере определен метод *getWeatherData*, который принимает параметр "cityName" из URL-пути и возвращает объект *WeatherData*.

3. Модель *WeatherData*:

- Создан класс *WeatherData*, представляющий структуру данных о погоде, которые будут возвращены клиенту.

- В классе определены переменные для хранения информации о погоде и аннотация `@Getter` для создания геттеров.

- Также определены сеттеры для установки значений полей класса.

4. Сервис *WeatherService*:

- Создан класс *WeatherService*, который выполняет запросы к OpenWeatherMap API и обрабатывает полученные данные.

- В сервисе используется ключ API для доступа к данным OpenWeatherMap.

- Метод *getWeatherData* выполняет HTTP GET-запрос к API, получает ответ в виде объекта *OpenWeatherMapResponse*, и конвертирует его данные в объект *WeatherData*.

5. Конфигурация безопасности:

- Создан класс *SecurityConfig*, который использует аннотации `@Configuration` и `@EnableWebSecurity`.

- В классе определен бин *SecurityFilterChain*, который конфигурирует безопасность приложения.

6. Модель *AuthenticationRequest*:

-Создан класс *AuthenticationRequest* для представления структуры данных запроса аутентификации.

7. Модель *AuthenticationResponse*:

-Создан класс *AuthenticationResponse* для представления структуры данных ответа на запрос аутентификации.

8. Модель *RegisterRequest*:

-Создан класс *RegisterRequest* для представления структуры данных запроса регистрации.

9. Модель *PreAuthenticationRequest*:

-Создан класс *PreAuthenticationRequest* для представления структуры данных запроса предварительной аутентификации.

10. Сервис *JwtService*:

-Создан класс *JwtService* для работы с JWT-токенами.

-Метод *extractUsername*: Извлекает имя пользователя из JWT-токена.

-Метод *extractClaim*: Обобщенный метод для извлечения утверждений из JWT-токена.

-Метод *generateToken*: Генерирует JWT-токен на основе *UserDetails* с возможностью добавления дополнительных утверждений.

-Метод *isTokenValid*: Проверяет валидность JWT-токена по имени пользователя и сроку действия.

-Приватные методы: Включают в себя обработку срока действия токена, извлечение утверждений и создание ключа для подписи.

11. Класс *JwtAuthenticationFilter*:

Создан класс *JwtAuthenticationFilter*, отвечающий за обработку JWT-токенов в процессе аутентификации.

- В классе определен метод *doFilterInternal*, который обрабатывает JWT-токены, извлекает имя пользователя, проверяет валидность токена, и, при успешной аутентификации, устанавливает контекст безопасности для текущего запроса.

12. Контроллер *AuthenticationController*:

-Создан класс *AuthenticationController* обрабатывающий запросы, связанные с аутентификацией и регистрацией пользователей.

-Базовый URL пути для контроллера установлен как "/api/v1/auth". В конструкторе класса используется аннотация @RequiredArgsConstructor для внедрения зависимостей.

-Метод *register*: Обрабатывает запрос по пути "/register". Принимает объект *RegisterRequest* в теле запроса, представляющий данные для регистрации нового пользователя и возвращает объект *AuthenticationResponse*, который содержит токен.

-Метод *preAuthenticate*: Обрабатывает запрос по пути "/preAuthenticate". Принимает объект *PreAuthenticationRequest* в теле запроса, содержащий данные для предварительной аутентификации пользователя и возвращает токен в виде строки.

-Метод *authenticate*: Обрабатывает запрос по пути "/authenticate". Принимает объект *AuthenticationRequest* в теле запроса, содержащий данные для аутентификации пользователя и возвращает объект *AuthenticationResponse*.

-Метод *refreshToken*: Обрабатывает запрос по пути "/refresh". Принимает заголовок Authorization, содержащий текущий токен пользователя. Использует сервис JwtService для извлечения имени пользователя из токена и проверки его валидности, генерирует новый токен с обновленным временем истечения и возвращает его в ответе.

13. Сервис *AuthenticationService*:

-Создан класс *AuthenticationService* для обработки запросов, связанных с аутентификацией и регистрацией пользователей.

-Метод *register*: Создает и сохраняет нового пользователя в репозитории, генерирует токен с использованием JwtService, и возвращает объект *AuthenticationResponse*.

-Метод *preAuthenticate*: Предварительно аутентифицирует пользователя, генерирует и сохраняет случайный код для двухфакторной аутентификации и отправляет код по электронной почте.

-Метод *authenticate*: Проверяет соответствие кода двухфакторной аутентификации, генерирует токен с использованием JwtService, и возвращает объект *AuthenticationResponse*.

14. Класс *ApplicationConfig*:

-Создан класс *ApplicationConfig* для конфигурации приложения.

-Метод *userDetailsService*: Возвращает объект *UserDetailsService*, который использует репозиторий для поиска пользователя по электронной почте. В случае отсутствия пользователя выбрасывает исключение.

-Метод *authenticationProvider*: Устанавливает сервис пользователей и кодировщик паролей для аутентификации.

-Метод *authenticationManager*: Возвращает объект, полученный из *AuthenticationConfiguration*.

-Метод *passwordEncoder*: Возвращает объект, представляющий кодировщик паролей *BCryptPasswordEncoder*.

15. Класс *MailConfig*:

-Создан класс *MailConfig* для конфигурации отправки электронных писем.

-Метод *javaMailSender*: Возвращает объект, представляющий настройки для отправки почты через *JavaMailSenderImpl*. Устанавливает хост, порт, учетные данные для авторизации, и свойства *JavaMail*. Включает протокол SMTP, аутентификацию, возможность старта TLS, отладку и настройки безопасности сокетов для порта 587.

16. Сервис *EmailService*:

-Создан класс *EmailService* для отправки электронных сообщений.

-Метод *sendSimpleMessage*: Принимает адрес получателя и код для отправки. Создает объект *SimpleMailMessage*, устанавливает адрес, тему и текст сообщения. Отправляет сообщение через *JavaMailSender*, используя настройки, предоставленные в *JavaMailSender*.

17. Сервис *OtpCodeService*:

-Создан класс *OtpCodeService* для управления одноразовыми кодами аутентификации.

-Метод *saveOtpCode*: Сохраняет в базе данных объект *OtpCode*, связанный с пользователем, кодом и временем истечения срока действия. Время действия кода устанавливается на 60 минут.

-Метод *generateRandomCode*: Генерирует случайный четырехзначный код.

-Метод *verifyOtpCode*: Проверяет, существует ли в базе данных код, связанный с пользователем и совпадающий с предоставленным кодом. Проверяет, не истекло ли время действия кода, и удаляет его из базы данных при успешной проверке.

-Метод *cleanUpExpiredCodes*: Удаляет из базы данных все истекшие коды, время действия которых истекло.

18. Модель *OtpCode*:

- Создан класс *OtpCode*, представляющий одноразовые коды аутентификации в базе данных.

- В классе конструкторы позволяют создавать объекты *OtpCode* и аннотация `@Getter` для создания геттеров.

- Также определены сеттеры для установки значений полей класса.

19. Модель *NasaResponse*:

- Создан класс *NasaResponse*, представляющий объект, используемый для маппинга данных, получаемых от API.

- Для каждого поля в API-ответе созданы соответствующие переменные класса с аннотациями `@Getter` и `@JsonIgnoreProperties`.

- Поля класса аннотированы @JsonProperty, что обеспечивает сопоставление JSON-полей с соответствующими полями класса, которые представляют информацию о фотографии или видео: copyright, date, explanation, hdurl, media\_type, service\_version, title и url.

20. Контроллер *NasaController*:

- Создан класс *NasaController*, который обрабатывает HTTP-запросы, связанные с данными от NASA.

- Использует аннотацию @RestController и задает базовый URL пути "/api/v1/nasa ".

- В контроллере определен метод *getNasaData*, который принимает параметр "date" из URL-пути и возвращает объект NasaResponse, содержащий данные от NASA для указанной даты.

21. Модель *NasaData*:

- Создан класс *NasaData*, представляющий структуру данных о NASA, которые будут возвращены клиенту.

- В классе определены поля для хранения информации о копирайте, дате, объяснении, URL изображения высокого разрешения (hdurl), типе медиа, версии сервиса, заголовке и URL-адресе данных `@Getter` для создания геттеров.

- Также определены сеттеры для установки значений полей класса.

22. Сервис *NASAService*:

- Создан класс *NASAService*, который представляет сервис для взаимодействия с API NASA.

- В сервисе используется поле *nasaApiKey* для доступа к данным API NASA.

- Метод *getData* выполняет HTTP GET-запрос к API NASA, используя ключ API и переданный параметр *date*. получает ответ в виде объекта *NasaResponse*, и конвертирует его данные в объект *NasaData*.

23. Контроллер *CityController*:

-Класс *CityController* представляет собой REST-контроллер, обрабатывающий HTTP-запросы, связанные с сущностью City.

-Аннотация @RestController указывает, что все методы контроллера возвращают данные, которые непосредственно записываются в тело ответа

-URL-путь к контроллеру задан аннотацией @RequestMapping("/api/v1/city").

-Метод *getAllCities* обрабатывает HTTP GET-запрос и возвращает список всех городов, полученных из репозитория *CityRepository*.

25. Репозиторий *CityRepository*:

-Интерфейс *CityRepository* предоставляет методы для взаимодействия с базой данных для сущности City.

-Расширяет интерфейс *JpaRepository*, что обеспечивает основные операции CRUD.

-Позволяет выполнять различные операции, такие как сохранение, поиск и удаление объектов типа City в базе данных.

26. Репозиторий *OtpCodeRepository*:

-Интерфейс *OtpCodeRepository* предоставляет методы для взаимодействия с базой данных для сущности *OtpCode*.

-Унаследован от *JpaRepository*, что позволяет использовать стандартные CRUD-операции для сущности *OtpCode*.

-Метод *findByUserAndCode* позволяет искать объект *OtpCode* по пользователю и коду.

-Метод *findByExpirationTimeBefore* возвращает список объектов *OtpCode*, у которых время истечения срока действия находится до указанного момента времени *now*.

27. Репозиторий *RoleRepository*:

-Интерфейс *RoleRepository* предоставляет методы для взаимодействия с базой данных для сущности Role.

-Расширяет интерфейс *JpaRepository*, что обеспечивает основные операции CRUD

-Содержит дополнительный метод *findByRoleName* для поиска роли по наименованию

28. Репозиторий *SubscriptionRepository*:

-Интерфейс *SubscriptionRepository* предоставляет методы для взаимодействия с базой данных для сущности *Subscription*.

-Расширяет интерфейс *JpaRepository*, что обеспечивает основные операции CRUD для работы с подписками.

29. Репозиторий *UserRepository*:

-Интерфейс *UserRepository* предоставляет методы для взаимодействия с базой данных для сущности *User*.

-Расширяет интерфейс *JpaRepository*, обеспечивая основные операции CRUD для работы с пользователями.

-Содержит дополнительный метод *findByEmail*, который позволяет находить пользователя по адресу электронной почты.

# Проектирование структуры базы данных

Мы создали базу данных PostgreSQL для нашего приложения, которая содержит следующие таблицы:

1. *Users*: В этой таблице хранятся данные о пользователях нашего приложения. Каждый пользователь имеет уникальный идентификатор (user\_id), имя пользователя (username), хешированный пароль (password) и ссылку (role\_id) на роль пользователя, которая хранится в таблице *Roles*. Роль пользователя связана с таблицей *Roles* по полю role\_id.

2. *Roles*: В этой таблице хранятся роли пользователей. Каждая роль имеет уникальный идентификатор (role\_id) и имя роли (role\_name). Роли определяют различные уровни доступа и привилегии пользователей в вашем приложении.

3. *Cities*: Эта таблица содержит информацию о городах. Каждый город имеет уникальный идентификатор (city\_id), название города (city\_name) и страну, в которой находится город (country).

4. *Subscriptions*: Эта таблица связывает пользователей с городами, на которые они подписались. Каждая подписка имеет уникальный идентификатор (subscription\_id) и ссылки (user\_id и city\_id) на соответствующие записи в таблицах *Users* и *Cities*. Это позволяет отслеживать, какие пользователи подписаны на какие города.

Эта структура базы данных предоставляет необходимую основу для нашего приложения, позволяя хранить информацию о пользователях, их ролях, городах и подписках. Таблица *Subscriptions* может быть полезной для реализации функциональности отслеживания погоды в выбранных городах пользователями.

Создание схемы *weatherapp* помогает ограничить доступ к таблицам только из этой схемы, обеспечивая лучшую организацию и безопасность базы данных.

# Разработка

Наше iOS приложение представляет собой простое приложение для получения и отображения данных о погоде. Ниже приведено краткое описание ключевых компонентов приложения:

1. *ContentView*: Это основное представление приложения. Он содержит текстовое поле для ввода названия города, кнопку для запроса данных о погоде и отображение результатов запроса. Некоторые ключевые элементы:

- Используется переменная `@State` для отслеживания введенного пользователем названия города и для управления состоянием приложения.

- При вводе города и нажатии кнопки "Поиск", вызывается метод *fetchWeatherData(cityName:)* из объекта *weatherDataVM*, который обрабатывает запрос данных о погоде.

- Отображение состояний приложения: "Загрузка..." при выполнении запроса, сообщение об ошибке, если запрос завершился с ошибкой, и результаты запроса о погоде.

2. *WeatherDataViewModel*: Это класс, представляющий модель данных для приложения. Класс *WeatherDataViewModel* отвечает за обработку запросов к серверу и управление данными о погоде. Ключевые аспекты:

- Использует аннотацию `@Published` для публикации изменений данных, таким образом, обновления могут автоматически отражаться на пользовательском интерфейсе.

- Метод *fetchWeatherData(cityName:)* выполняет запрос данных о погоде на сервер с использованием URLSession, обрабатывает ошибки и декодирует данные с сервера в объект *WeatherData*.

3. *WeatherData*: Это структура данных, которая соответствует структуре данных, возвращаемой сервером. Она реализует протокол Codable, что позволяет декодировать данные JSON из серверного ответа в эту структуру.

4. *WeatherDataFieldView*: Это представление для отображения данных о погоде. Каждый *WeatherDataFieldView* содержит информацию о конкретном аспекте погоды, таком как температура, условия, давление, влажность и т. д. Он содержит значок, заголовок и значение.

Наше IOS приложение позволяет пользователям вводить название города и получать информацию о текущей погоде в этом городе с использованием нашего сервера, который в свою очередь обращается к OpenWeatherMap API. Приложение отображает полученные данные о погоде в удобном и понятном формате, предоставляя пользователю актуальную информацию о погоде.

# Тестирование

# Описание клиентской части

После запуска приложения появляется окно авторизации с двумя полями для логина и пароля от учетной записи и кнопками авторизации и регистрации:

-При нажатии на кнопку "Регистрация" открывается окно с четырьмя полями: имя, фамилия, почта и пароль. После успешного ввода данных открывается главный экран приложения.

-В случае, если учетная запись уже существует, после окна авторизации происходит подтверждение входа с использованием двухфакторной аутентификации через почту.

На главном экране приложения есть 3 вкладки:  
 -Погода: Пользователь вводит название города и отправляется соответствующий запрос на сервер, где обрабатывается всё и клиент получает готовые данные: температуру, погодные условия, давление, влажность и времени заката, рассвета.

-Nasa: Пользователь выбирает дату, отправляет запрос на сервер, и клиент получает данные о фотографии/видео дня и статье, связанной с этим изображением/видео.

-Личный кабинет: Отображаются имя, фамилия пользователя, почта. Пользователь может указать свой город для автоматического отображения погоды. Также присутствует кнопка выхода из аккаунта.

# Демонстрация функционала

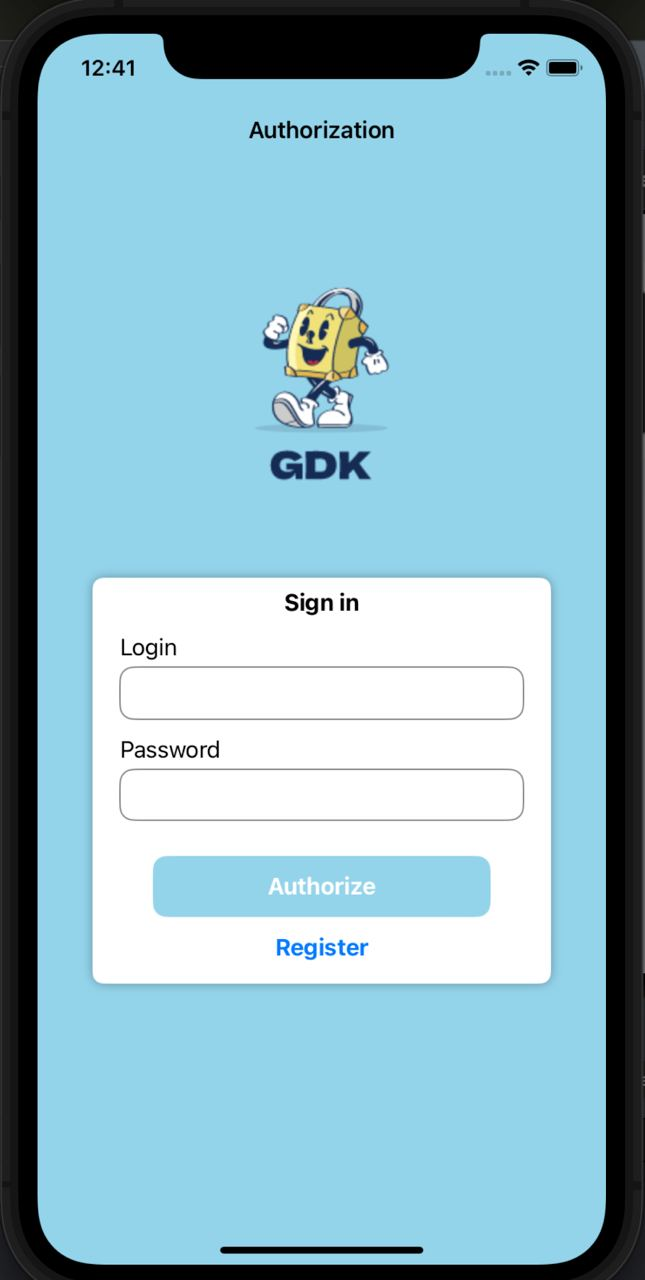


Рис.1 Окно авторизации

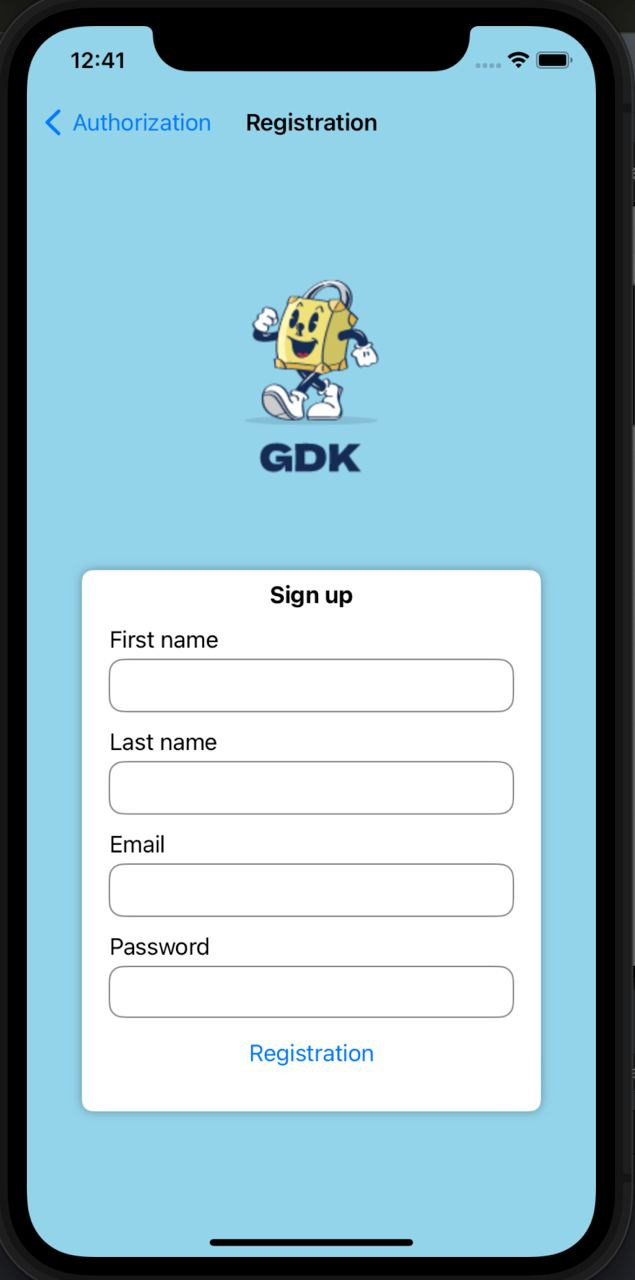


Рис. 2 Окно регистрации

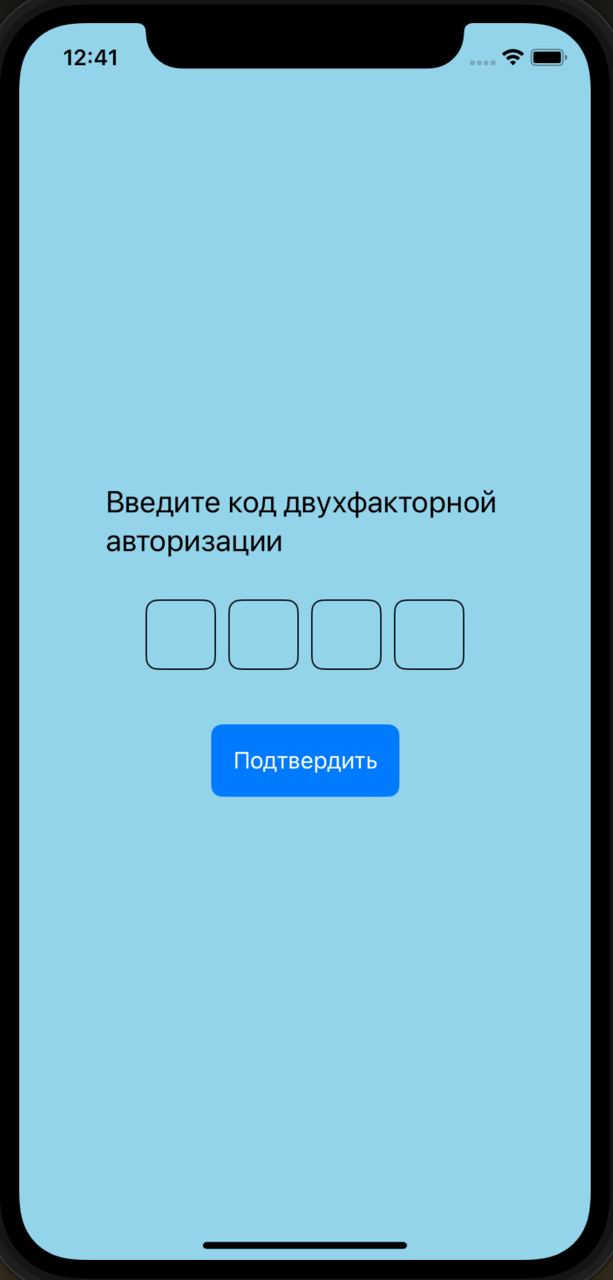


Рис. 3 Код для авторизации.

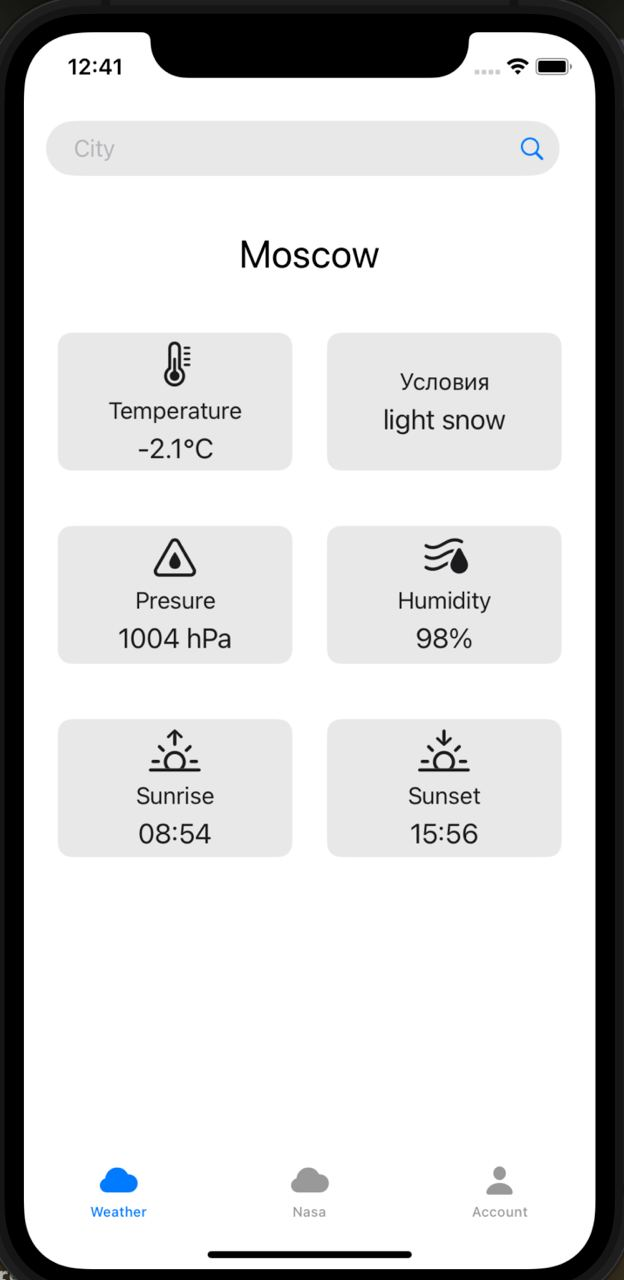


Рис.4 Погода



Рис.5 NASA

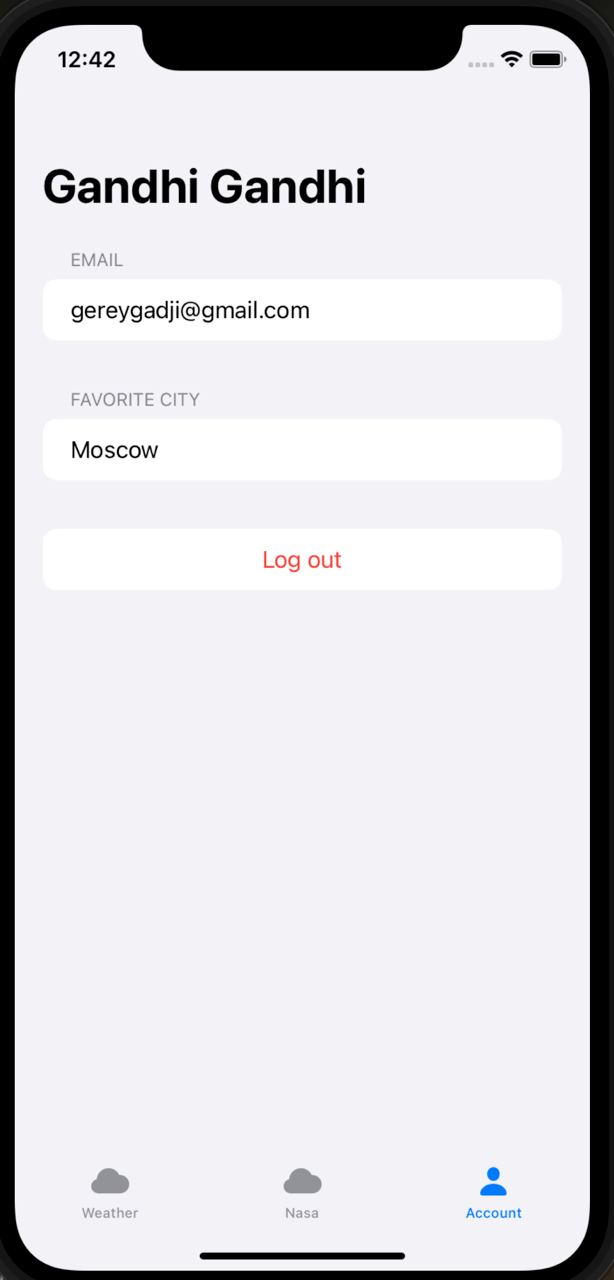


Рис.6 Личный кабинет.

Список используемой литературы

1. Документация Spring Boot [https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#documentation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/%23documentation)
2. Документация Apple Swift [http://developer.apple.com/documentation/swift](http://developer.apple.com/documentation/swift%20)