Для послідовності розробки back-end проекту я поділив весь процес на наступні етапи:

* Створення моделі, репозиторіїв
* Налаштування Spring security
* Створення сервісів для роботи з репозиторіями
* Створення Rest-контролерів

На даний момент я реалізував перші два пункти.

**Модель**

Для взаємодії з базою даних я використовую ORM фреймворк Hibernate. Hibernate - це об’єктно-реляційний інструмент відображення для мови програмування Java. Він забезпечує основу для відображення об'єктно-орієнтованої моделі в реляційну базу даних.

Класи моделі або їх поля проанотовані наступними анотаціями:

* @Entity – інтерфейс, що вказує на те, що даний клас може бути відображений у таблицю реляційної бази даних. Класи анотовані цим інтерфейсом, мають містити обов’язкове поле помічене анотацією @Id або @EmbeddedId
* @Table – інтерфейс, що дає можливість вказати певні деталі таблиці, в яку буде відображено клас, наприклад назву таблиці чи її схему.
* @Column – інтерфейс, що вказує, що певному анотованому полю відповідає певне значення з конкретної колонки та дає можливість вказати деталі колонки, наприклад назву та обмеження на значення колонки (size, nullable, updatable, unique)
* @Id – інтерфейс вказує, що це поле ідентифікує об’єкт
* @EmbeddedId – інтерфейс вказує, що анотоване поле є ідентифікатором об’єкту, але первинний ключ є складеним. Клас, що представляє ідентифікатор повинен бути проанотованим анотацією @Embeddable та імплементувати інтерфейс Serializable
* @ManyToOne – визначає тип зв'язку сутностей багато до одного. Це означає, що поле анотоване цією анотацією є зовнішнім ключем іншої сутності, а також те, що багато об'єктів першої сутності можуть бути пов'язані з одним об'єктом іншої сутності. Тип поля це клас, анотований анотацією @Entity.
* @ManyToMany – визначає тип зв'язку сутностей багато до багатьох. Це означає, що об’єкти першої сутності можуть бути пов’язані з багатьма об’єктами іншої сутності і навпаки. Таким чином анотоване поле є списком, де тип другої сутності вказується дженеріком.
* @JoinColumn – визначає, що ця сутність є власником зв’язку, тобто таблиця містить колонку з зовнішнім ключем іншої сутності. Також інтерфейс надає змогу вказати певні деталі колонки(insertable, nullable, updatable, unique та ін)
* @GeneratedValue – інтерфейс вказує, що значення поля генерується автоматично. Може бути застосованим лише до поля анотованого анотацією @Id. Параметри, які можна змінювати це strategy та generator. Strategy – стратегія генерації, яку потрібно використати, generator - назва генератора первинного ключа.
* @GenericGenerator – анотація генератора, що описує будь-який тип генератора Hibernate. Для визначення генератора потрібно вказати атрибути name (унікальна назва) та strategy (cтратегія генератора, заздалегідь визначена Hibernate стратегія або повне ім'я класу.)

Для того, щоб мати доступ до бази даних потрібно вказати певні властивості у файлі application.properties, а саме:

* spring.datasource.url – лінк до бази даних, що містить IP-адресу хосту, номер порту з’єднання та назву бази даних
* spring.datasource.username – ім’я користувача-власника бази даних
* spring.datasource.password – пароль, необхідний для доступу до даних

Також потрібно додати властивість з Spring Data JPA, а саме: spring.jpa.hibernate.dll-auto, яка визначає яке значення із create, create-drop, validate та update буде передано для hibernate.hbm2ddl.auto, що визначає спосіб взаємодії Hibernate з базою даних при зміні моделі.

Всі класи моделі знаходяться в директорії model, а саме:

* UserEntity– клас, що відображає користувача додатку. Містить поля: id, login, hashedPassword, firstName, lastName, phone1, phone2, status, role. Поле id генерується автоматично як UUID. Пароль записується одразу зашифрованим, тобто він ніде не зберігається у відкритому вигляді. Поле login є унікальним та використовується для авторизації.
* Beacon– клас, що відображає нову мітку. Містить поля: id, xCoordinate, yCoordinate, description, range, userEntityId, tags. Поле id генерується автоматично як UUID. Поле userEntityId це користувач-власник мітки. Поле tags це список міток, які будуть використовуватись для фільтрації маячків.
* Social – клас, що відображає соцмережі, дані яких може надавати користувач. Містить поле id та socialname.
* Tag – відображає мітку для маячка. Містить поле id та name.
* UserHistory – відображає історію відгуків користувача на маячки інших користувачів. Містить поля id, beaconId, userEntity1, userEntity2, date, labelDescription. Поле id генерується як UUID. userEntity1, userEntity2 – зовнішні ключі, які визначають зв’язки користувачів з маячком beaconId.
* UserSocial – відображає дані користувачів зв’язаних з соцмережею. Містить поле usid та shareableData. Поле usId є складеним первинним ключем.
* UserSocialId – відображає складений первинний ключ сутності UserSocial. Містить поля userId, socialId.

**Репозиторії**

Репозиторії для всіх сутностей були створені однаково за допомогою інтерфейсу JpaRepository.

JpaRepository - це розширення Repository, специфічне для JPA. Він містить повний API CrudRepository та PagingAndSortingRepository. Отже, він містить API для основних CRUD-операцій, а також API для пагінації та сортування.

**Налаштування Spring security**

Щоб налаштували доступ до додатку за ролями, спочатку потрібно створити самі ролі та дозволи. Роль - це набір дозволів. Для цього я створив enum-и Roles та Permissions. Permissions мають лише один обов’язковий параметр це стрічка-дозвіл. Ролі групують дозволи у сети та мають метод getGrantedAuthorities(), який із сету звичайних дозволів створює та повертає Set<SimpleGrantedAuthority>.

Далі я створив конфігураційний клас AuthConfig який наслідує клас WebsSecurityConfigurerAdapter та перевизначає метод configure(HttpSecurity http). В цьому методі будуть встановлюватись всі фільтри, які забезпечують безпеку, а також встановлюватимуться відкриті ендпоінти. Я написав два власні фільтри JwtAuthenticationFilter та JwtTokenVerifier.

JwtAuthenticationFilter – це фільтр, що перевіряє чи є користувач з певним логіном та паролем в базі даних да повертає Jwt токен у хедері Authorization. JwtAuthenticationFilter наслідує UsernamePasswordAuthenticationFilter з Spring Security та перевизначає методи attemptAuthentication, successfulAuthentication. Задача методу attemptAuthentication перевірити чи існує користувач за переданими логіном та паролем. Для цього також був створений клас ApplicationUserService, що наслідує клас UserDetailsService та перевизначає метод loadUserByUsername, що повертає об’єкт класу org.springframework.security.core.userdetails.User. Також клас ApplicationUserService містить метод, який власне і генерує Jwt токен за переданими значеннями з класу JwtConfig, який є існує лише для того, щоб підтягувати конфігурації потрібні для генерування та валідації токена з файлу application.properties. Якщо метод successfulAuthentication виконається успішно, викличеться метод successfulAuthentication, в якому до відповіді буде додано хедер Authrization з токеном.

JwtTokenVerifier – це, як можна зрозуміти з назви, фільтр, що перевіряє чи є токен у користувача та чи правильний цей токен. Спочатку ми перевіряємо префікс токена, потім розшифровуємо його за допомогою секретного ключа з JwtConfig та дістаємо authorities, які одразу конвертуємо у список SimpleGrantedAutthority. Якщо все пройшло без збоїв, то аутентифіковуємо користувача, інакше отримаємо помилку IllegalStateException, а отже відовідь на запит 403 Forbidden.

Завдяки налаштуванню .antMatchers("/sign-up").permitAll() у методі configure(HttpSecurity http) ендпоінт для реєстрації є відкритим, тож до нього можна доступитись без будь-якого токену. Важливим також є і налаштування класу DaoAuthenticationProvider адже, в ньому вказується PasswordEncoder паролів та ApplicationUserService, написаний для аутентифікації користувачів, які зберігаються у базі даних.