МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт кибербезопасности и цифровых технологий

Кафедра КБ-14 «Цифровые технологии обработки данных»

Кроссплатформенная среда исполнения программного обеспечения

Практическая работа №2
Разработка бэкенд-приложения на Spring Framework

Вводная

Представим, что есть некий банк, назовём его “Сенебанк”. Банк современный, модный, а значит клиентского офиса у него нет и его заменяет веб-продукт. Для него нужно разработать бэкенд на Spring, который бы обеспечивал выполнение основных бизнес-процессов компании: обслуживание клиентов и их счётов, переводы между счётами, расчёт кредитов и тому подобное.

Для учебной задачи освоения разработки на Spring, достаточно будет написать лишь часть функционала, которое вполне укладывается в базовые навыки работы с фреймворком.

Рассмотрим основные функциональные требования к приложению:

- Для пользователя должна быть предусмотрена регистрация и авторизация в сервисе;
- Для пользователя должна быть возможность выполнять следующие операции с банковским счётом:
    - Открытие и закрытие счёта;
    - Перевод денег между своими счетами;
    - Перевод денег на счета других пользователей банка;
    - Просмотр истории операций всех счетов или одного.

В текущей практической работе написание фронтенда не нужно - все операции можно осуществлять через запросы к соответствующим энд-поинтам.

Задание 1.

Создать Spring Framework проект со следующими зависимостями:

- Lombok;
- Spring Web;
- PostgreSQL Driver;
- Spring Data JPA;
- Spring Security.

В качестве системы сборки можно использовать Maven или Gradle.

База данных - PostgreSQL, подробнее о её установке и использовании в разделе рекомендаций по выполнению.

В проекте описать основные сущности (Entity), которые будут храниться в базе данных: пользователь, счёт, транзакция (и другие, при

необходимости). На основе сущностей описать модели данных, которые будут использоваться в бизнес-логике сервиса.

Создать классы-репозитории, взаимодействующие с таблицами в БД. Использование репозиториев обернуть через DAO-классы (по одному на каждую сущность).

В DAO внести следующие методы для взаимодействия с данными:

- Сохранение сущности в базу;
- Поиск сущности по полю (email пользователя, id счёта, id транзакции);
- Поиск всех сущностей по условию (все счета пользователя, все транзакции пользователя);
- Удаление сущности по условию.

Задание 2.

Создать классы-контроллеры, которые будут содержать в себе следующие энд-поинты:

UserController

- Авторизация пользователя;
- Регистрация пользователя;
- Удаление пользователя;

AccountController

- Открытие счёта;
- Закрытие счёта;
- Просмотр данных о счёте;

TransactionController

- Перевод денег между счётами;

AdminController

- Вывод всех транзакций;

- Вывод транзакций пользователя\счёта.

Для всех энд-поинтов предусмотреть модели данных JSON-запроса (Request) и модель данных JSON-ответа (Response).

Тестировать доступность и передачу данных можно через Postman или любую другую программу для отправки запросов на API.

Для реализации бизнес-логики создать соответствующие классы-сервисы, реализующие указанные во введении функциональные требования.

Задание 3.

Сконфигурировать Spring Security таким образом, чтобы доступ к работе с основными функциями могли иметь только авторизованные пользователи. Для энд-поинтов администратора использовать проверку на роль ADMIN. Добавить конфигурацию stateless сессий.

Для регистрации и авторизации пользователя предусмотреть создание и возвращение в качестве ответа JWT-токена, в котором содержатся следующие данные:

- Email-пользователя;
- Роль пользователя (USER или ADMIN);
- Срок действия токена.

Каждый запрос на энд-поинты должен проходить проверку на наличие JWT токена, для этого использовать класс, расширяющий базовый OncePerRequestFilter.

В итоге процесс каждого запроса должен выглядеть следующим образом:

1. Запрос идёт на end-point
2. Перехватывается фильтром

3. Проверяется на наличие JWT:
    a. Если JWT есть, то происходит парсинг токена и проверка на срок действия.
    b. Далее, если пользователь существует в базе, а токен подписан тем же ключом, то создаём токен аутентификации и добавляем его в контекст безопасности (см. рекомендации по выполнению).
    c. Если JWT нет или на любом из этапов происходит ошибка, то, запрос проходит дальше без проверки из создания аутентификации.

Что должно быть в отчёте

В отчёте должно быть следующее:

1. Текст задания;
2. Листинг кода по каждому заданию;
3. Список энд-поинтов;
4. Для каждого энд-поинта не менее одного результата тестирования.

Оформление отчёта

Обратите внимание на то, что отчёт перед сдачей на проверку должен быть оформлен согласно следующим требованиям:

- Основной шрифт для текста - Times New Roman, 14 пт;
- Межстрочный интервал - 1,5;
- Выравнивание - по ширине;
- Шрифт для листинга кода, названия классов и т.д - Courier New, Consolas или JetBrains Mono, размер не более 12 пт;
- Листинг кода должен быть на белом фоне (никаких скриншотов); форматирование можно осуществлять через

плагин Easy Code Formatter (для MS Word 2016 и выше), либо через сервис https://pastebin.com/

- Схема для Pastebin такая:
    1. Вставляете код класса из IDE;
    2. В настройках снизу указываете:
        a. Syntax Highlighting: **Java**,
        b. Paste Expiration: **Burn after read**,
        c. Paste Exposure: **Unlisted**.
    3. Нажимаете Create new paste;
    4. Копируете полученный код и вставляете в документ.

- Оформление титульного листа должно быть по единой утверждённой форме (брать у старост).
    - На данный момент название института: “Институт кибербезопасности и цифровых технологий”.

Не оформленный должным образом отчёт приниматься на проверку **не будет**.

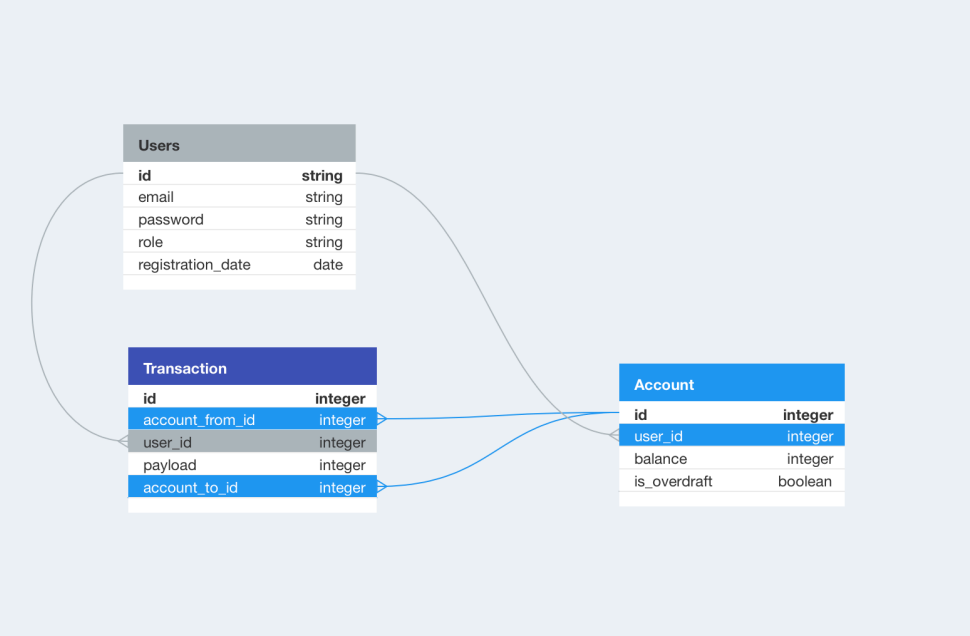
Рекомендации по выполнению практической работы

Модель базы данных

Для реализации классов-сущностей в бэкэнде можно воспользоваться готовой моделью базы данных банка. В ней указаны основные поля и связи между таблицами. Связи от внешних ключей представлены в виде “многие к одному”.

То есть: у одного пользователя может быть много счётов, у одного счёта и пользователя может быть много транзакций.

При этом, у каждого счёта может быть только один владелец. И у каждой транзакции может быть только один пользователь её совершающий.



Установка PostgreSQL

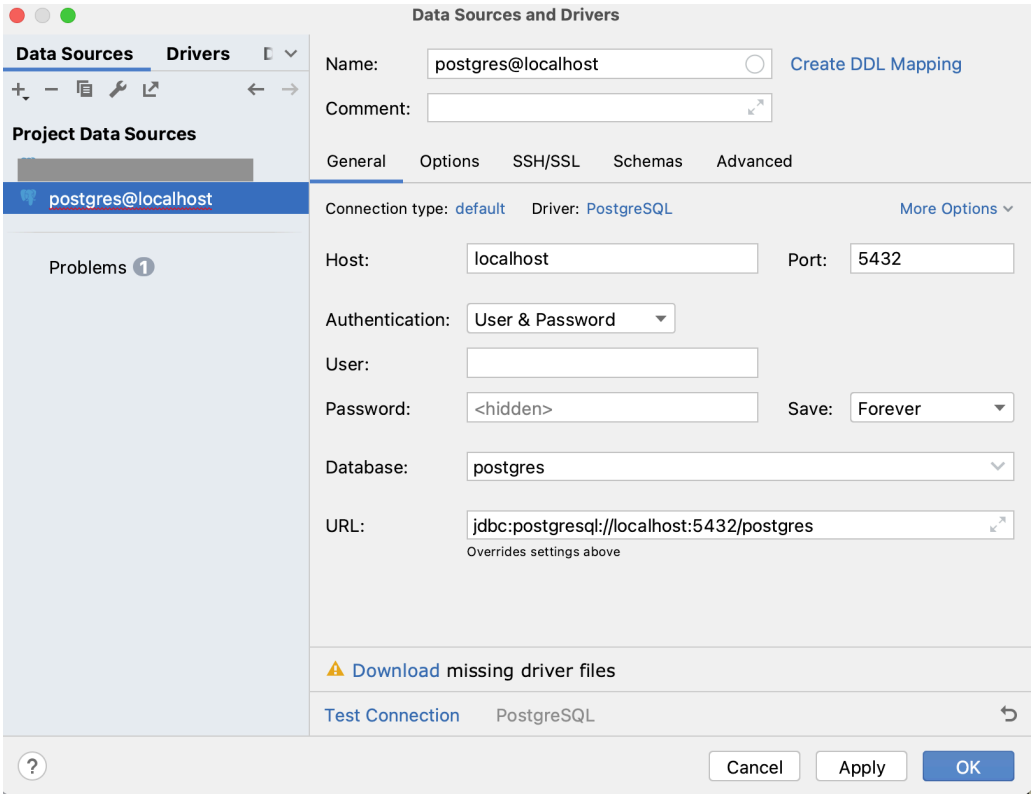
Для установки базы данных и программ для удобного администрирования можно воспользоваться одним из гайдов по этой ссылке.

Подключение к БД внутри IntelliJ IDEA

В Ultimate версии (а может и в Community) IDEA существует возможность удалённого доступа к базе данных с возможностью взаимодействия с её структурой через GUI или SQL-запросы.

Для этого в интерфейсе IDE сбоку присутствует вкладка Database, где нужно нажать на + и во вкладке Data Source выбрать PostgreSQL.

Откроется такое окно:



Если ваша база находится на локальной машине, то достаточно ввести стандартное имя и пароль администратора БД, которое вы указывали ранее при установке.

База данных по умолчанию всегда будет “postgres”.

При необходимости можно скачать драйвер для использования PostgreSQL (Download missing driver files).

После подключения можно будет создавать как таблицы в текущей базе данных, так и создать новую БД для использования в конкретном

проекте. Для взаимодействия с БД при помощи SQL запросов, нужно нажать на + и выбрать Query Console. Откроется новая вкладка для ввода запроса.

Справка по основным SQL командам

Отличный базовый гайд по работе с SQL - тык, более подробный гайд - тык.

Подключение Spring-приложения к базе данных

Конфигурация подключения к базе данных может выглядеть следующим образом. В файле application.properties нужно добавить такие строки:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/senebank
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.hibernate.ddl-auto=update
```

Создавать самостоятельно какие-то таблицы не следует - Spring сам осуществит добавление соответствующих сущностей при первом запуске. Этому способствует указанный выше параметр:

```
spring.jpa.hibernate.ddl-auto=update
```

Справка по аннотациям Lombok

Библиотека Lombok достаточно удобный инструмент для избавления от рутинных действий по типу создания гетеров\сетеров для закрытых полей, модификаторов доступа, конструктов и т.п. Благодаря механизму метапрограммирования с помощью аннотаций, библиотека самостоятельно встраивает нужные элементы кода на этапе компиляции в байт-код.

Ниже приведены основные аннотации и способы их применения:

Аннотация	**Область применения**	**Описание**
@Getter	Класс, поле	Создаёт геттер для полей всего класса или же для конкретных полей
@Setter	Класс, поле	Создаёт сеттер для полей всего класса или же для конкретных полей
@NoArgsConstructor	Класс	Создаёт пустой конструктор для класса. Не будет работать, если есть хотя бы одно поле с модификатором final
@AllArgsConstructor	Класс	Создаёт конструктор для класса со всеми полями в качестве аргументов
@RequiredArgsConstructor	Класс	Создаёт конструктор для класса только для полей с модификатором final
@ToString	Класс	Создаёт реализацию метода toString() с выводом значений всех полей класса
@Data	Класс	Объединяет в себе аннотации Getter,Setter,ToString, RequiredArgsConstructor
@Builder	Класс	Создаёт для класса шаблон проектирования “строитель”
@UtilityClass	Класс	Превращает весь класс в статический: все поля, методы и внутренние классы имеют модификатор static, а сам класс становится final.
@FieldDefaults	Класс	У данной аннотации есть два параметра, указываемых в скобках, например: `@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)`

		означает, что все поля будут иметь приватный доступ, а также являться final. Класс AccessLevel содержит перечисление всех модификаторов доступов для полей.
@Slf4j	Класс	Создаёт внутри класса экземпляр Logger с названием log. Через него можно осуществлять логирование. Например, log.info("Text");

JSON Web Token (JWT)

Для начала **крайне рекомендуется** прочитать данную статью, где доступным языком даются основы для понимания JWT.

Далее, чтобы внедрить такой токен в любое Java приложения, в том числе и Spring, нужно добавить некоторые Maven зависимости:

```
<dependency>
 <groupId>io.jsonwebtoken</groupId>
 <artifactId>jjwt-api</artifactId>
 <version>0.11.5</version>
</dependency>
<dependency>
 <groupId>io.jsonwebtoken</groupId>
 <artifactId>jjwt-impl</artifactId>
 <version>0.11.5</version>
</dependency>
<dependency>
 <groupId>io.jsonwebtoken</groupId>
 <artifactId>jjwt-jackson</artifactId>
 <version>0.11.5</version>
</dependency>
```

Все они относятся к библиотеке JJWT предназначенной для лёгкого внедрения JWT в проект (в том числе и для Android).

Сам JWT объект представляет собой простую строку и для её создания нужно воспользоваться вызовом метода .builder() из класса

JwtBuilder, представляющий собой реализацию шаблона проектирования “Строитель” для класса Jwts.

Иными словами, простейший способ создания:

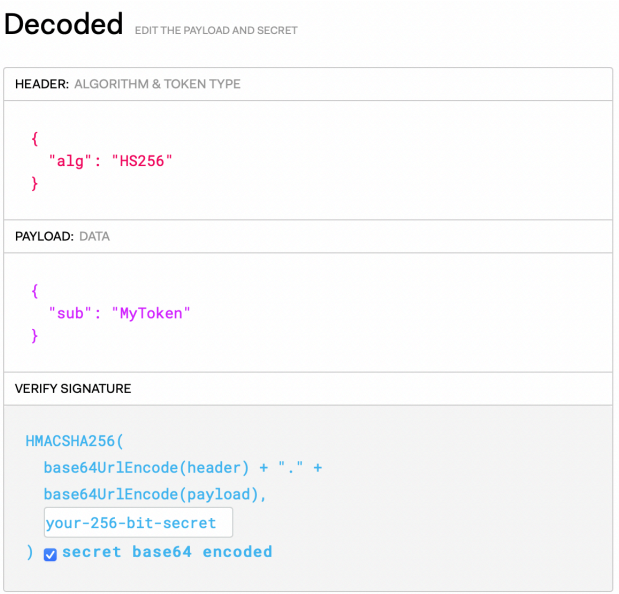
```
Key myKey = Keys.secretKeyFor(SignatureAlgorithm.HS256);

String jwt = Jwts
        .builder()
        .setSubject("MyToken")
        .signWith(myKey)
        .compact();
```

Для начала нам необходим секретный ключ для верификации нашего токена. Затем мы создаём сам токен. С помощью .setSubject() мы заполняем поле “sub” в разделе полезных данных. .signWith() создаёт цифровую подпись нашего токена с ранее сгенерированным ключом.

На выходе получается строка такого вида:

eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJNeVRva2VuIn0.0kio2r48ZIo8AtZlKq5K4HVdCLZLlOBJZgh4_mqoKjI

Если проверить её через декодер на специальном сервисе jwt.io, то можно увидеть содержимое токена:



Чтобы такое же провернуть в Java - спарсить JWT - нужно сделать следующее:

```
Jwts
        .parserBuilder()
        .setSigningKey(myKey)
        .build()
        .parseClaimsJws(jwt)
```

В результате, мы получаем JSON объект, который в строковом представлении будет выглядеть таким образом:

header={alg=HS256},body={sub=MyToken},signature=N7m-JVBbZYoX_QnyEjHB6u-7HVoKIcWaIq2tW4S3JpU

Важно понимать, что проверяющая система должна обладать **тем же ключом**, которым и был подписан исходный токен, иначе валидация будет провалена.

Фильтрация запросов

Любой клиентский запрос в Spring приложение всегда проходит определённый путь через, так называемые, фильтры. Более подробно об этом написано в этой статье. Здесь же мы рассмотрим создание собственной реализации фильтра для третьего задания.

Поскольку наше приложение является “stateless” - то есть, **оно не хранит в себе сессию**, нам необходимо как-то отделять запросы от авторизованного пользователя от всех остальных. На помощь нам приходят JWT токены из предыдущего раздела.

Основная идея такая: если в заголовках запроса присутствует нужный нам токен, то отправляем на проверку и при валидации авторизуем сессию и даём доступ к нужному ресурсу.

Сама настройка фильтра выглядит следующим образом.

Создадим класс, расширяющий базовый фильтр OncePerRequestFilter:

```
@Component
public class JwtTokenFilter extends
OncePerRequestFilter
```

В нём необходимо переопределить метод `doFilterInternal:`

```
@Override
protected void doFilterInternal(
HttpServletRequest request,
HttpServletResponse response,
FilterChain chain)
            throws ServletException, IOException { }
```

Этот метод в последующем будет вызываться в цепочке основных фильтров для запроса. Если разобрать входящие аргументы метода, то становится очевидно, что он принимает некоторый клиентский запрос, ответ сервера и объект, содержащий цепочку фильтров.

Каким образом разработчик может влиять на эти аргументы:

1. Получать из объекта request различную информацию о запросе: заголовки, содержимое, параметры и так далее;
2. Настраивать объект ответа - response, например, встраивать в него содержимое для куки;
3. Объект chain необходим для обязательного вызова следующих фильтров по цепочке в конце выполнения метода.

Пример простейшей проверки токена:

```
final String token = getToken(request);
if (token == null) {
    log.info("token is null");
    chain.doFilter(request, response);
    return;
}
```

Если токена нет, то просто передаём request и response дальше по цепочке фильтров.

А если всё таки токен есть в запросе, тогда неплохо бы его преобразовать в модель пользователя, например так (очевидно, что у нас в классе уже должно быть определено поле JwtTokenService):

```
UserModel user = jwtTokenService.parseToken(token);
```

В теле метода parseToken(), может быть следующий функционал:

```
Claims body = Jwts.parserBuilder()
        .setSigningKey(getSigninKey())
        .build()
        .parse(token)
        .getBody();

String userRole = ((String)
body.get("role")).replace("[","").replace("]","");

return UserModel.builder()
        .username(body.getSubject())
        .role(Collections.singleton(new
SimpleGrantedAuthority(userRole)))
        .build();
```

Claims это по сути интерфейс для получения значений по именам полей в JSON объекте. Например, чтобы получить роль пользователя, нужно вызвать метод get() и передать название поля "role", определённое в структуре токена.

И метод в итоге строит и возвращает модель пользователя на основе данных из токена: sub, role.

Получив модель пользователя, можно проверить - а есть ли вообще такой в базе, не пытается ли злоумышленник подделать токен?

```
if(!userDao.existsByEmail(user.getUsername())){
    log.info("user not exists");
    chain.doFilter(request, response);

    return;
}
```

Если такой пользователь действительно есть, время выдать ему внутренний объект аутентификации - `Authentication`

По своей сути - это интерфейс для реализации специального токена, который может хранить в себе основные данные о пользователе: username, password, authorities (права доступа). Его конкретная реализация, которая используется для простой аутентификации - `UsernamePasswordAuthenticationToken`

Основной способ создания токена:

```
UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(
        user, null,
        user.getAuthorities()
);
```

Класс UserModel должен реализовывать интерфейс UserDetails, с которым взаимодействует класс `UsernamePasswordAuthenticationToken`.

К токену можно добавить дополнительные *детали* запроса: IP-адрес и прочее.

```
authentication.setDetails(
       new WebAuthenticationDetailsSource().buildDetails(request)
);
```

Осталось лишь сохранить этот токен в текущий **контекст безопасности.**

```
SecurityContextHolder.getContext().setAuthentication(aut
hentication);
```

Полное описание механизма работы не имеет смысла приводить в виду его сложности. Для простоты понимания достаточно знать, что его смысл в хранении аутентификации пользователя в текущем потоке выполнения (который, как известно, создаётся на каждый запрос от клиента).

Наличие аутентификации позволяет запросу пройти необходимые фильтры и получить доступ к запрашиваемому ресурсу (URL).