Дополнительные советы ментора

Доработка модели данных

Новый класс `Booking` будет содержать следующие поля:

- `id` — уникальный идентификатор бронирования;
- `start` — дата и время начала бронирования;
- `end` — дата и время конца бронирования;
- `item` — вещь, которую пользователь бронирует;
- `booker` — пользователь, который осуществляет бронирование;
- `status` — статус бронирования. Может принимать одно из следующих значений: `WAITING` — новое бронирование, ожидает одобрения, `APPROVED` — бронирование подтверждено владельцем, `REJECTED` — бронирование отклонено владельцем, `CANCELED` — бронирование отменено создателем.

Создание базы данных

Основные поля, которые нужно создать в таблицах, в целом аналогичны полям в уже реализованных сущностях:

- **`users`**: `id`, `name`, `email`.
- **`items`**: `id`, `name`, `description`, `is_available`, `owner_id`, `request_id`.
  Обратите внимание: поля `owner_id` и `request_id` — это ключи записей в других таблицах (`users` и `requests` соответственно). Так будет реализована связь между сущностями на уровне базы данных.
- **`bookings`**: `id`, `start_date`, `end_date`, `item_id`, `booker_id`, `status`.
- **`requests`**: `id`, `description`, `requestor_id`.
- **`comments`**: `id`, `text`, `item_id`, `author_id`.

Обратите внимание: для хранения полей, содержащих дату, удобнее использовать тип `TIMESTAMP WITHOUT TIME ZONE`. Почитать подробнее о типах данных для хранения

даты и времени в PostgreSQL можно в документации.

Также продумайте ограничения (`CONSTRAINT`) для таблиц.

Добавьте параметры доступа к базе данных в файл `application.properties`. Именно их Spring будет использовать при выполнении скрипта, а также при дальнейшей работе приложения.

Запустите проект и убедитесь, что старт прошёл без ошибок. Подключитесь к базе данных через консоль и проверьте, что все таблицы созданы. Поздравляем — структура базы готова! А вы готовы сохранять данные.

Настройка JPA

Переходите к созданию JPA-репозиториев — кода, через который ваш проект будет работать с БД. Spring Data JPA позволяет автоматизировать создание DAO-классов в приложении — для этого достаточно реализовать интерфейс репозитория.

Начните с самого простого — с пользователя. В пакете `user` создайте интерфейс `UserRepository`, наследующий интерфейс `JpaRepository`. Все базовые методы для работы с данными, то есть чтение, сохранение, обновление, будут доступны пользователям этого интерфейса автоматически. Теперь вы можете внедрить репозиторий в класс `UserServiceImpl` и заменить реализацию хранения данных в памяти на работу с БД.

Повторите те же шаги для `item` — создайте интерфейс `ItemRepository` и теперь уже полноценно реализуйте в `ItemServiceImpl` все сценарии, которые вы наметили в предыдущем спринте.

Большинство этих сценариев покрывается базовыми методами CRUD, однако вам также нужно добавить функцию поиска вещи. Причём переданный пользователем текст нужно искать как в названии, так и в описании. Этот шаг не реализуется Spring автоматически, так что запрос для поиска придётся написать вручную.

<details>
<summary>Подсказка: как написать SQL-запрос в репозитории</summary>

Необходимо декларировать метод поиска в интерфейсе репозитория, а также указать текст нужного SQL-запроса с помощью аннотации `@Query`. Например, вот так.
</details>

```
@Query(" select i from Item i " +
        "where upper(i.name) like upper(concat('%', ?1, '%')) " +
        "   or upper(i.description) like upper(concat('%', ?1, '%'))")
List<Item> search(String text);
```

Этот код не «чистый» SQL, а SQL-подобный язык. Он работает с сущностями модели данных, а не с полями таблицы: обратите внимание, например, на конструкцию `select i from Item i`. Если имена полей модели и наименования полей в базе данных отличаются, в запросе необходимо указать именно поля модели данных.

Не забудьте добавить в код сервиса логические проверки: например, у пользователя не должно быть возможности создать вещь с пустым названием.

Запустите приложение и выполните несколько запросов. Сначала создайте пользователя. От его имени создайте новую вещь. Обновите её статус на «недоступна для аренды» и обратно. Попробуйте найти эту вещь с помощью поиска и посмотреть подробную информацию о ней.

Реализация функции бронирования

Создайте интерфейсы `BookingService` и `BookingRepository`, а также класс `BookingServiceImpl` — всё по аналогии с тем, что вы делали для работы с `User` и `Item`.

▼ Подсказка: как научить Spring генерировать нужный метод

Для работы с бронированием понадобятся более сложные запросы в репозитории. Для части запросов придётся написать SQL-код вручную, как это было с поиском вещи.

Также пригодится ещё одна возможность Spring Data JPA. Вы можете создать в интерфейсе `BookingRepository` метод, названный по конкретным правилам, а Spring автоматически сгенерирует его содержимое. Этот вариант полезен для запросов, которые не покрываются стандартными CRUD-операциями, но при этом обладают достаточно простой логикой — например, найти запись по значению определённого поля.

```
List<Booking> findByBooker_IdAndEndIsBefore(Long bookerId, LocalDateTime end, Sort sort);
```

> Если добавить в репозиторий метод с таким определением, Spring сгенерирует метод, который будет осуществлять поиск всех записей по переданному `bookerId` с датой окончания (поле `end`) раньше переданной.

Запустите приложение и попробуйте создать несколько бронирований для добавленных ранее вещей, а также подтвердить либо отклонить их другим пользователем. Проверьте, как работает просмотр списка бронирований и отдельного бронирования. Убедитесь, что функционал полностью работоспособен!

Добавление дат бронирования при просмотре вещей

Сначала доработайте DTO-класс, который приложение возвращает пользователю при просмотре списка его вещей. Добавьте в класс две записи с датами бронирования. Обратите внимание: теперь объект вещи, который возвращается пользователю по эндпоинту `GET /items` и по `GET /items/{itemId}`, отличается. Поэтому правильнее будет создать отдельный DTO-класс для каждого из этих запросов.

Далее доработайте слой сервиса и репозиториев. Здесь есть ещё одна сложность: объект `Item` не содержит ссылок на относящиеся к нему бронирования. Это связано с тем, что для реализации такого хранения пришлось бы использовать связь вида `@OneToMany`, у которой свои особенности. Вместо этого мы применяем более простую связь `@ManyToOne` и храним в бронировании ссылку на соответствующую `Item`.

Поэтому для реализации получения дат бронирования вам нужно выполнить дополнительный запрос в репозиторий бронирований. Добавьте в `BookingRepository` метод, получающий бронирования для определённой вещи. `Booking` содержит ссылку на `Item`, так что всё получится. Вызовите этот метод внутри сервиса `ItemServiceImpl`, а затем вручную добавьте в результирующий DTO-объект нужные данные. Для этого внедрите `BookingRepository` в класс `ItemServiceImpl`.

> 💡 Подобный метод часто используется при разработке крупных проектов, поскольку получить все нужные данные из базы одним запросом бывает либо невозможно, либо совсем неэффективно.

Убедитесь, что приложение корректно возвращает даты бронирований для каждой вещи, и двигайтесь дальше.

Добавление отзывов

Отзыв относится к одной вещи, а вот вещь может иметь много комментариев, поэтому вы будете хранить в `Comment` ссылку на `Item`, но в `Item` ссылок на `Comment` не будет. Этот пункт аналогичен тому, в котором вы реализовали связь `Item` — `Booking`.

Таким образом `Comment` будет содержать следующие поля:

- `id` — уникальный идентификатор комментария;
- `text` — содержимое комментария;
- `item` — вещь, к которой относится комментарий;
- `author` — автор комментария;
- `created` — дата создания комментария.

Теперь дайте пользователям возможность оставлять отзыв на вещь. Отзыв может оставить только тот пользователь, который брал эту вещь в аренду, и только после окончания срока аренды. Так комментарии будут честными. Добавление комментария будет происходить по эндпоинту `POST /items/{itemId}/comment`.

Это единственный эндпоинт для работы с отзывами — разместите его в `ItemController`. Создавать отдельный контроллер не потребуется: это было бы уместно в приложении с более продвинутым функционалом отзывов. А вот написать DTO-класс для комментария нужно.

Добавьте в нужные DTO-объекты список отзывов. Для получения списка из БД реализуйте обращение к новому репозиторию `CommentRepository` в соответствующих методах в `ItemServiceImpl`. Вероятно, вам также захочется добавить в `CommentRepository` новые методы для поиска комментариев к вещи.

Убедитесь, что отзывы успешно добавляются и отображаются.