**2 Spring Web**

**2.1 Java Persistence API**

2.1.1 Основные понятия

Сущность в базе данных - некоторый объект рассматриваемой предметной области, информация о котором должна быть отражена в базе данных. Этот объект должен иметь экземпляры - конкретные представители данной сущности, отличающиеся друг от друга и допускающие однозначную идентификацию.

Entity (сущность в контексте Java) — легковесный хранимый объект бизнес логики (persistent domain object). Основная программная сущность это entity класс, который так же может использовать дополнительные классы, который могут использоваться как вспомогательные классы или для сохранения состояния еntity.

JPA (Java Persistence API) - это спецификация Java EE и Java SE, описывающая систему управления сохранением java объектов в таблицы реляционных баз данных в удобном виде. Сама Java не содержит реализации JPA, однако есть существует много реализаций данной спецификации от разных компаний (открытых и нет). Это не единственный способ сохранения java объектов в базы данных (ORM систем), но один из самых популярных в Java мире.

1) Entity класс должен быть отмечен аннотацией Entity или описан в XML файле конфигурации JPA;

2) Entity класс должен содержать public или protected конструктор без аргументов (он также может иметь конструкторы с аргументами);

3) Entity класс должен быть классом верхнего уровня (top-level class);

4) Entity класс не может быть enum или интерфейсом;

5) Entity класс не может быть финальным классом (final class);

6) Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables);

7) Если объект Entity класса будет передаваться по значению как отдельный объект (detached object), например через удаленный интерфейс (through a remote interface), он так же должен реализовывать Serializable интерфейс;

8) Поля Entity класс должны быть напрямую доступны только методам самого Entity класса и не должны быть напрямую доступны другим классам, использующим этот Entity. Такие классы должны обращаться только к методам (getter/setter методам или другим методам бизнес-логики в Entity классе);

9) Enity класс должен содержать первичный ключ, то есть атрибут или группу атрибутов которые уникально определяют запись этого Enity класса в базе данных.

2.1.2 Создание сущностей

Существует простой класс User, который представляет данные о пользователе в системе:

public class User {

private UUID id;

private String firstname;

private String lastname;

private Role role;

}

Объект Role - описывает роль пользователя. И выглядит следующим образом:

public class Role {

private String name;

}

Класс сущности должен реализовывать интерфейс Serializable:

public class User implements Serializable{

private UUID id;

private String firstname;

private String lastname;

private Role role;

}

Класс сущности должен иметь аннотацию @Entity:

@Entity

public class User implements Serializable{

private UUID id;

private String firstname;

private String lastname;

private Role role;

}

Также необходимо убедиться, что сущность имеет конструктор без аргументов и первичный ключ (о первичном ключе позже).

Таким образом предоставляется фреймворку информацию о том, что класс является сущностью в базе данных.

В большинстве случаев имя таблицы в базе данных и имя объекта не будут совпадать. В этих случаях можно указать имя таблицы с помощью аннотации @Table:

@Table(name = "users")

@Entity

public class User implements Serializable{

private UUID id;

private String firstname;

private String lastname;

private Role role;

}

Каждый объект JPA должен иметь первичный ключ, который однозначно идентифицирует его. Аннотация @Id определяет первичный ключ. Можно генерировать идентификаторы разными способами, которые указаны в аннотации @GeneratedValue*.*

Выбирается одна из четырех стратегий генерации идентификатора с элементом стратегии. Значение может быть AUTO, TABLE, SEQUENCE или IDENTITY:

@Table(name = "users")

@Entity

public class User implements Serializable{

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id", nullable = false)

private UUID id;

private String firstname;

private String lastname;

private Role role;

}

Как и в случае с аннотацией @Table, можно использовать аннотацию @Column для указания сведений о столбце в таблице:

@Table(name = "users")

@Entity

public class User implements Serializable{

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id", nullable = false)

private UUID id;

@Column(nullable = false, length = 25,name = "first\_name")

private String firstname;

@Column(nullable = false, length = 25, name = "last\_name")

private String lastname;

private Role role;

}

Аннотация javax.persistence.JoinColumn помечает столбец как столбец соединения для ассоциации сущностей или коллекции элементов.

Аннотация @ManyToOne подразумевает отношение объекта User к Role как многие к одному (Множество пользователей может обладать одной и той же ролью):

@Entity

@Table(name = "roles")

public class Role implements Serializable {

@Id

@Column(name = "name", nullable = false)

private String name;

}

Итоговые сущности:

* User;

@Getter

@Setter

@Entity

@Table(name = "users")

public class User implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id", nullable = false)

private UUID id;

@Column(nullable = false, length = 25,name = "first\_name")

private String firstname;

@Column(nullable = false, length = 25, name = "last\_name")

private String lastname;

@ManyToOne

@JoinColumn(name = "role")

private Role role;

}

* Role.

e

Можно аннотировать любое поле с помощью @Getter и/или @Setter, чтобы ломбок автоматически генерировал геттер/сеттер по умолчанию.

Геттер по умолчанию просто возвращает поле и получает имя getFoo, если вызывается поле foo (или isFoo если поле имеет тип boolean).

Сеттер по умолчанию называется, setFoo если поле вызывается foo, возвращает void и принимает 1 параметр того же типа, что и поле. Он просто устанавливает поле в это значение.

Сгенерированный метод получения/установки будет, public если явно не указать AccessLevel.

Допустимые уровни доступа: PUBLIC, PROTECTED, PACKAGE и PRIVATE.

Также можно добавить аннотацию @Getter и/или @Setter к классу. В этом случае аннотируется все нестатические поля в этом классе аннотацией.

Всегда можно вручную отключить генерацию геттера/сеттера для любого поля, используя AccessLevel.NONE уровень доступа.

2.1.3 Внедрение зависимостей

Spring Framework позволяет внедрять объекты без их инициализации клиентским кодом.

В итоговом классе контроллера отсутствует инициализация репозитория. Всё потому, что этим занимается сам фреймворк.

Достаточно указать аннотацию @Autowired на уровне конструктора, а в аргументе объект, который требуется внедрить.

**2.2 JPA Repository**

Spring Data JPA ориентирована на использование JPA для сохранения данных в реляционную БД. Наиболее полезной её возможностью является автоматическое создание реализаций репозитория во время выполнения из интерфейса.

Основное понятие в Spring Data — это репозиторий. Это несколько интерфейсов которые используют JPA Entity для взаимодействия с базой данных, реализующих CRUD операции.

Интерфейс JpaRepository предоставляет методы, связанные с JPA, такие как обновление постоянных данных и пакетное удаление.

JPA репозиторий является объектом доступа к данным — DAO.

Пример репозиториев:

public interface UserRepository extends JpaRepository<User, UUID> {

}

public interface RoleRepository extends JpaRepository<Role, String>

}

При наследовании от JpaRepository в обобщении (треугольные скобки) следует указать тип сущности и тип первичного ключа сущности.

**2.3 Миграция базы данных**

Flyway - это инструмент миграции (переноса) базы данных. Проще говоря, это инструмент, который помогает вам выполнять сценарии базы данных при развертывании приложений. Flyway поддерживает два типа сценариев, SQL и Java. Вы можете упаковать сценарий в приложение. Когда приложение запускается, Flyway управляет выполнением этих сценариев. Эти сценарии называются миграцией Flyway.

Чтобы создать скрипты миграции (в Intellij IDEA) кликнете правой кнопкой мыши по папке resources и создайте папку со скриптами (рис. 2.1):

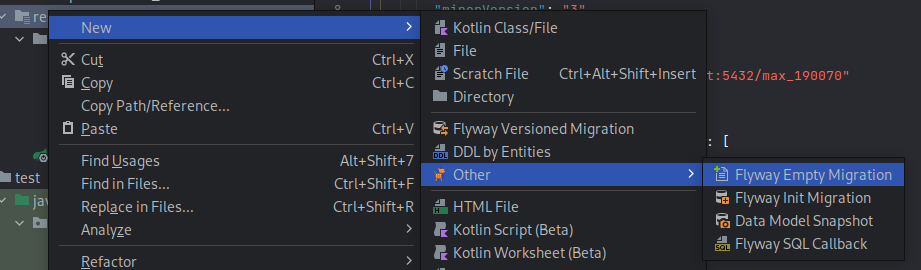


Рис. 2.1 - Создание пустой миграции

Далее нужно создать собственную миграцию, где будет происходить инициализация сущностей пользователя и роли.

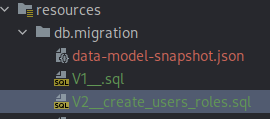


Рис. 2.2 - Создание нового скрипта инициализации

Внутри скрипта нужно описать создание таблиц на языке postgresql.

Название скрипта миграции должно соответствовать следующим требованиям:

1. Первые два символа описывают номер версии скрипта (V[n]).

2. Затем следуют два нижних подчеркивания (\_\_).

3. Затем следует произвольное название скрипта (его предназначение).

Скрипт создания таблиц пользователя и роли:

CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

create table if not exists roles

(

name varchar not null,

UNIQUE (name),

PRIMARY KEY (name)

);

create table if not exists users

(

id uuid not null,

first\_name varchar(100) not null,

last\_name varchar(100) not null,

role varchar,

UNIQUE (id),

PRIMARY KEY (id)

);

ALTER TABLE users ADD CONSTRAINT fk\_roles FOREIGN KEY (role) references roles(name)

**2.4 Rest Controller**

Контроллер - это класс, предназначенный для непосредственной обработки запросов от клиента и возвращения результатов.

Аннотация @RestController, эквивалентная комбинации двух аннотаций @ Controller + @ ResponseBody, для возврата данных json не требуется добавлять аннотацию @ResponseBody перед методом, но при использовании этой аннотации @RestController вы не можете вернуть jsp, html-страницу, парсер представления не может разобрать JSP, HTML-страница:

@RequestMapping("/users")

@RestController

public class UserRestController {

}

@RequestMapping — аннотация, в аргументах которой обязательно указывается путь к данному контроллеру.

@RestController — помечает класс как принимающий REST API запросы, ответы которого содержат тело, в котором находится JSON объект.

Контроллер обрабатывает все виды http методов, в нашем случае мы используем только 4:

1. @GetMapping — обрабатывает запросы GET метода;

@RequestMapping("/users")

@RestController

public class UserRestController {

@GetMapping

public List<User> getAll(){

return userRepository.findAll();

}

@GetMapping("/{id}")

public User getById(@PathVariable("id") UUID id){

return userRepository.findById(id).get();

}

}

1. @DeleteMapping — обрабатывает запросы DELETE метода;

@RequestMapping("/users")

@RestController

public class UserRestController {

@GetMapping

public List<User> getAll(){

return userRepository.findAll();

}

@GetMapping("/{id}")

public User getById(@PathVariable("id") UUID id){

return userRepository.findById(id).get();

}

@DeleteMapping("/{id}")

public void remove(@PathVariable("id") UUID id){

userRepository.deleteById(id);

}

}

1. @PutMapping — обрабатывает запросы PUT метода;

@RequestMapping("/users")

@RestController

public class UserRestController {

@GetMapping

public List<User> getAll(){

return userRepository.findAll();

}

@GetMapping("/{id}")

public User getById(@PathVariable("id") UUID id){

return userRepository.findById(id).get();

}

@DeleteMapping("/{id}")

public void remove(@PathVariable("id") UUID id){

userRepository.deleteById(id);

}

@PutMapping

public User update(@RequestBody User user){

if(userRepository.existsById(user.getId())){

return userRepository.save(user);

}

throw new EntityExistsException("User with id:'"+ user.getId() +"' doesn't exists");

}

}

1. @PostMapping — обрабатывает запросы POST метода.

@RequestMapping("/users")

@RestController

public class UserRestController {

@GetMapping

public List<User> getAll(){

return userRepository.findAll();

}

@GetMapping("/{id}")

public User getById(@PathVariable("id") UUID id){

return userRepository.findById(id).get();

}

@DeleteMapping("/{id}")

public void remove(@PathVariable("id") UUID id){

userRepository.deleteById(id);

}

@PutMapping

public User update(@RequestBody User user){

if(userRepository.existsById(user.getId())){

return userRepository.save(user);

}

throw new EntityExistsException("User with id:'"+ user.getId() +"' doesn't exists");

}

@PostMapping

public User create(@RequestBody User user){

UUID id = user.getId();

if(id !=null){

if(userRepository.existsById(user.getId())){

throw new EntityExistsException("User already exists");

}

}

return userRepository.save(user);

}

}

Если в значении mapping аннотации не указан дополнительный путь(/users/subpath), запрос поступает по адресу указанному в @RequestMapping контроллера (/users).

@PathVariable — аннотация для аргумента метода, указывающая на значение в пути адреса. В примере случае /{id}.

@RequestBody — указывает на передачу данных в теле запроса. В примере JSON с полями класса User и вложенным объектом Role:

{

"id": "cea9b6aa-1c3f-41cd-83d0-5858ed6ed1e8",

"firstname": "user",

"lastname": "user",

"role": {

"name": "USER"

}

}

Код итогового котроллера REST:

@RequestMapping("/users")

@RestController

public class UserRestController {

private final UserRepository userRepository;

@Autowired

public UserRestController(UserRepository userRepository){

this.userRepository = userRepository;

}

@GetMapping

public List<User> getAll(){

return userRepository.findAll();

}

@GetMapping("/{id}")

public User getById(@PathVariable("id") UUID id){

return userRepository.findById(id).get();

}

@PutMapping

public User update(@RequestBody User user){

if(userRepository.existsById(user.getId())){

return userRepository.save(user);

}

throw new EntityExistsException("User with id:'"+ user.getId() +"' doesn't exists");

}

@PostMapping

public User create(@RequestBody User user){

UUID id = user.getId();

if(id !=null){

if(userRepository.existsById(user.getId())){

throw new EntityExistsException("User already exists");

}

}

return userRepository.save(user);

}

@DeleteMapping("/{id}")

public void remove(@PathVariable("id") UUID id){

userRepository.deleteById(id);

}

}

**2.5 Postman**

2.5.1 Создание запроса

Postman — это HTTP-клиент для тестирования API. HTTP-клиенты тестируют отправку запросов с клиента на сервер и получение ответа от сервера.

Например, с помощью Postman можно протестировать, как API регистрирует нового пользователя приложения, как добавляет и удаляет данные о нем на сервере. Скачать можно по ссылке <https://www.postman.com/downloads/>

Collection (коллекция) — отправная точка для нового API. Можно рассматривать коллекцию, как файл проекта. Коллекция объединяет в себе все связанные запросы. Обычно API описывается в одной коллекции, но если вы желаете, то нет никаких ограничений сделать по-другому.

Folder (папка) — используется для объединения запросов в одну группу внутри коллекции. К примеру, вы можете создать папку для первой версии своего API — "v1", а внутри сгруппировать запросы по смыслу выполняемых действий — "Order & Checkout", "User profile" и т. п. Всё ограничивается лишь вашей фантазией и потребностями. Папка, как и коллекция может иметь свои скрипты, но не переменные.

Request (запрос) — основная составляющая коллекции, то ради чего все и затевалось. Запрос создается в конструкторе (рис. 2.3, рис. 2.4). Конструктор запросов это главное пространство, с которым вам придётся работать. Postman умеет выполнять запросы с помощью всех стандартных HTTP методов, все параметры запроса под вашим контролем. Вы с лёгкостью можете поменять или добавить необходимые вам заголовки, cookie, и тело запроса. У запроса есть свои скрипты. Обратите внимание на вкладки "Pre-request Script" и "Tests" среди параметров запроса. Они позволяют добавить скрипты перед выполнением запроса и после. Именно эти две возможности делают Postman мощным инструментом помогающим при разработке и тестировании.

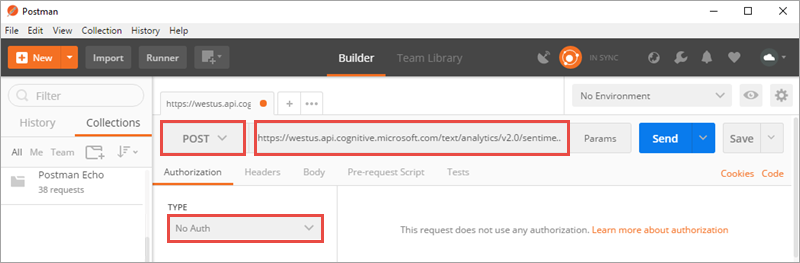


Рис. 2.3 — Создание запроса без тела

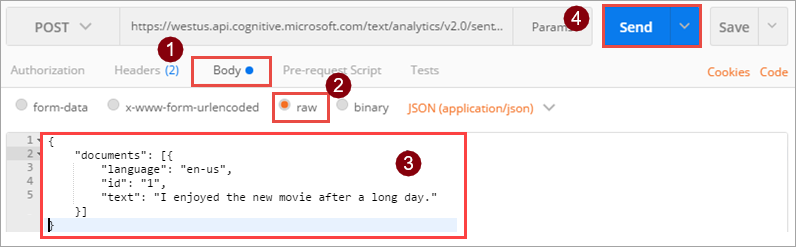


Рис. 2.4 — Создание запроса с JSON телом

2.5.2 Создание запроса в Postman

После регистрации следует создать workspace. Рабочие пространства Postman помогают организовать работу с API и сотрудничать в рамках вашей организации или по всему миру. Существует три различных типа рабочих пространств Postman для различных нужд: личные рабочие пространства, командные рабочие пространства и общедоступные рабочие пространства (рис. 2.5).

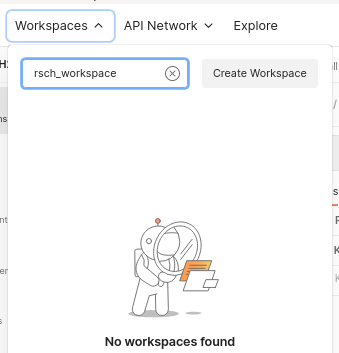


Рис. 2.5 — Создание рабочего пространства (workspace)

Создать коллекцию для запросов нажав + во вкладке коллекций (рис. 2.6).

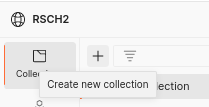


Рис. 2.6 — Создание коллекции запросов

Создать запросы в коллекции с именованием и методом GET/POST/PUT/DELETE (рис. 2.7).

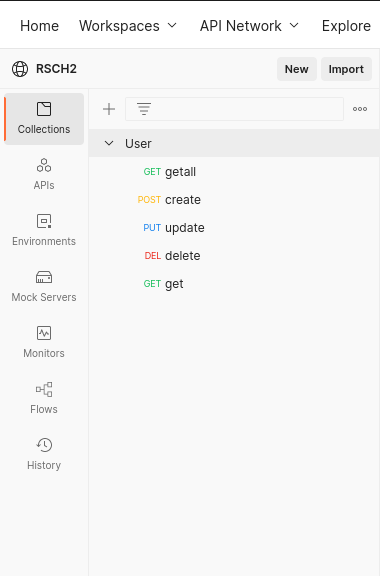


Рис. 2.8 — Создание коллекции запросов для маршрута /user

2.5.3 Тестирование адресов

Тестирование API получения данных всех пользователей (рис. 2.9)

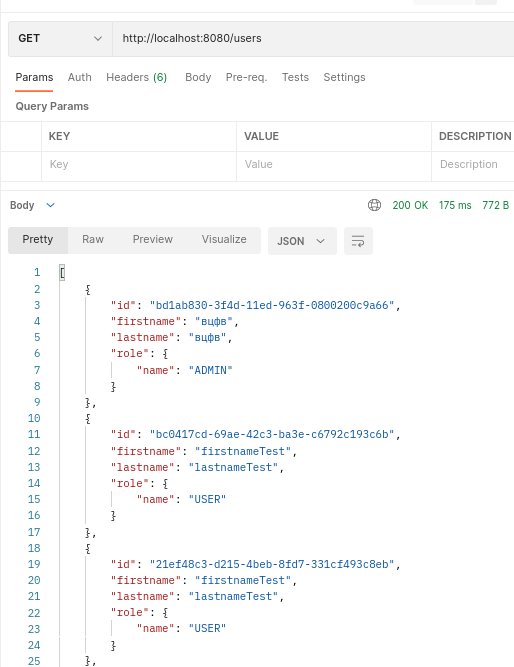


Рис. 2.9 — Запрос и ответ для получения данных всех пользователей

Тестирование API создания пользователя с отправкой данных в теле запроса (рис. 2.10).

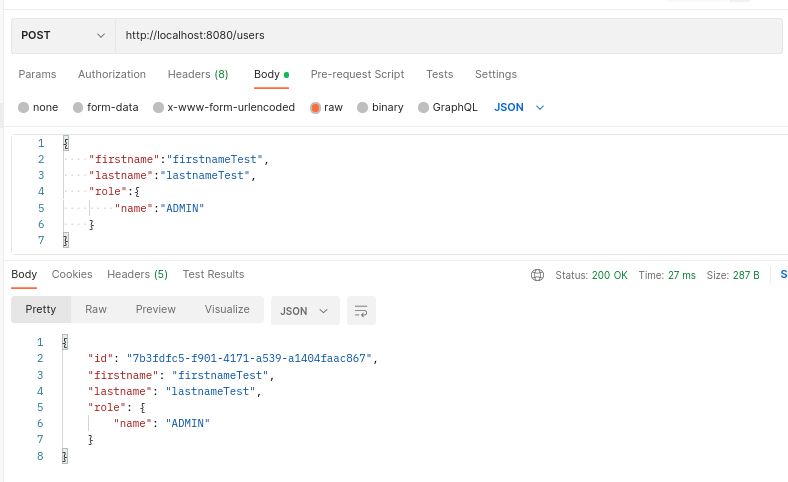


Рис. 2.10 — Запрос и ответ для создания пользователя

Тестирование изменения данных пользователя с указанием новых данных пользователя в теле запроса (рис. 2.11).

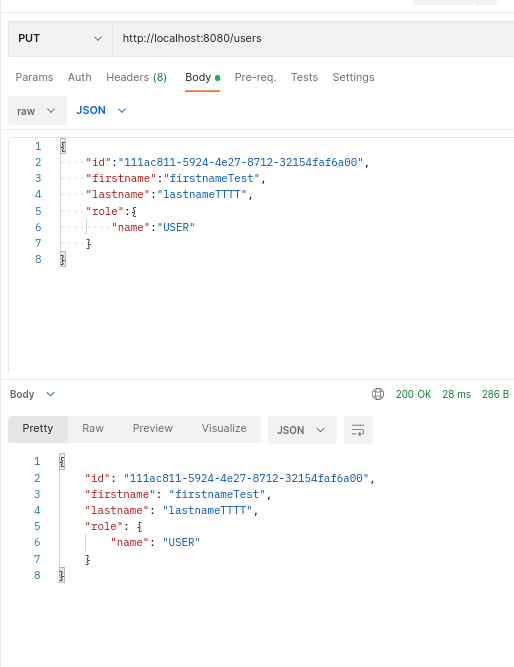


Рис. 2.11 — Запрос и ответ для изменение данных пользователя

Следует отметить, что при обновлении данных пользователя следует указать id в теле запроса ("id": "cea9b6aa-1c3f-41cd-83d0-5858ed6ed1e8"), иначе мы получил исключение, т. к. не удастся найти изменяемого пользователя.

Тестирование удаления пользователя, с передачей id в переменной пути (Path variable) (рис. 2.12).

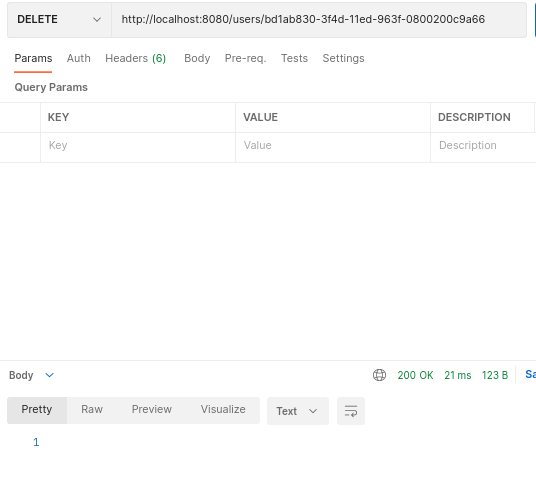


Рис. 2.12 — Запрос и ответ удаления пользователя по его id

Тестирование получения данных пользователя, с передачей id в переменной пути (Path variable) (рис. 2.13).

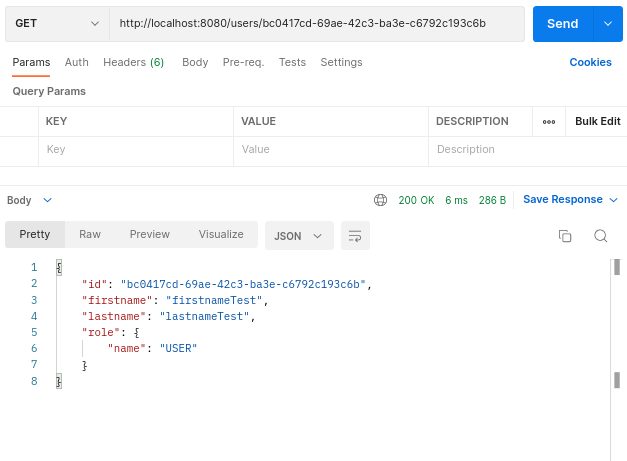


Рис. 2.13 — Запрос получения данных пользователя по его id

**2.6 Лабораторная работа - Spring Web. Создание REST API веб-приложений**

Цель лабораторной работы: изучение создания JPA-сущностей, Data Access Objects (DAO) — JPA Repository, REST контроллеров с помощью фреймворка Spring Framework.

Задание:

1. Создать сущности, репозитории, контроллеры и протестировать работу приложения через Postman.
2. Создать сущности в пакете entity.
3. Создать репозитории в пакете dao.
4. Создать контроллеры в пакете controller.

После выполнения задания запушить проект в gitflic.

Варианты задания:

1. Вариант

Добавить сущность «животное» с полями:

* вид (связь с сущностью «вид»);
* название.

2. Вариант

Добавить сущность «страна» с полем:

* столица (связь с сущностью «город»).

Добавить сущность «город» с полями:

* название;
* население;
* страна (связь с сущностью страна).

В дополнение к обычным запросам так же, при получении страны нужно подсчитывать население всех городов.

3. Вариант

Добавить сущность «машина» с полями:

* модель;
* владелец (связь с таблицей «пользователи»);
* цвет;
* номер.

При выводе списка машин сразу, владельца не выводить.

4. Вариант

Добавить сущность «книга» с полями:

* названием;
* количеством страниц;
* автором (связь с таблицей пользователи);
* год выпуска.

При выводе списка книг, количество страниц и год выпуска не выводить.

5. Вариант

Добавить сущность «игра» с полями:

* названием;
* студией разработки (связь с сущностью студии разработки).

Добавить отдельный путь, где можно получить список игр студии. В маршруте списка студий, игры не выводить, в маршруте получения студии игры не выводить.

6. Вариант

Добавить сущность «фильм» с полями:

* название;
* актерским составом (связь с пользователями, и соответственно кто кем является).

Ко всем вариантам, ко всем сущностям нужно сделать маршруты на удаление, изменение, создание, чтение. Так же добавить свою миграцию, не изменяя созданную.

7. Вариант

Добавить сущность «валюта» с полями:

* название (любые 2);
* количество;
* цена;
* стоимость;
* дата.

Курс валюты меняется на 5% каждый месяц. Вывести значения полей за 8 месяцев назад и за 20 месяцев вперед от текущей даты.

8. Вариант

Добавить сущность «магазин» с полями:

* название магазина;
* адрес.

Добавить сущность «товар» с полями:

* название магазина;
* название товара;
* количество;
* цена;
* стоимость.

Добавить сущность «склад» с полями:

* адрес;
* название товара;
* количество.

Вывести в каком магазине находится товар. Если товар отсутствует в магазине, то вывести на каком складе находится.

9. Вариант

Добавить сущность «регистратура» с полями:

* ФИО;
* дата приема;
* время приема;
* диагноз;
* экстренное (да/нет);
* врач.

Врач может принять 6 человек в день. Если значение поля «экстренное» стоит «да», то пациент перемещается в начало очереди. Вывести очередь на два ближайшие дня.

10. Вариант

Добавить сущность «деканат» с полями:

* предмет (любые 3);
* ФИО студента;
* дата;
* оценка.

Вывести отличников, хорошистов, троечников и кто не сдал, по всем предметам. Назначить стипендию, кто сдал на 4 и 5 в установленный срок.

Контрольные вопросы:

Вопросы к защите лабораторной работы включают процесс установки и настройки по каждому пункту задания, а также теоретический материал к данной работе.

**2.7 Дополнительный материал**

Работа с сервисом gitflic, необходимо предварительно зарегистрироваться на сайте <https://gitflic.ru/>

1. Создание (рис. 2.14) и настройка нового проекта (рис. 2.15).

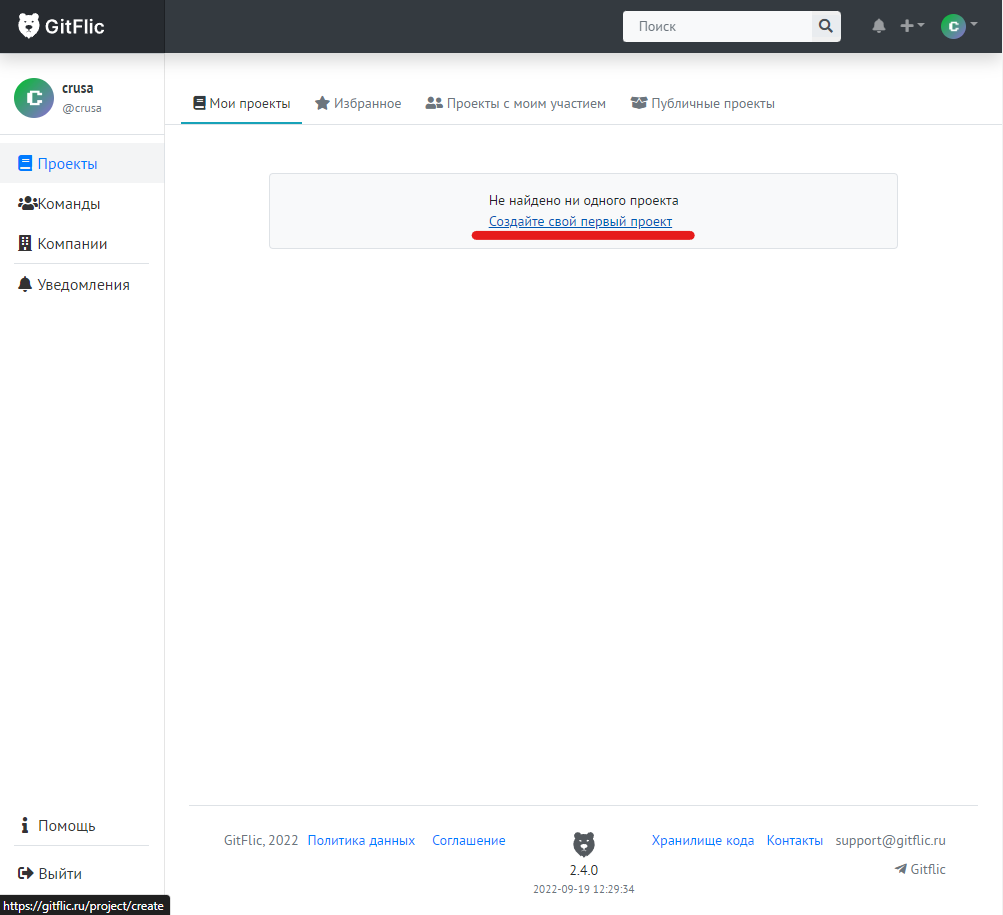


Рис. 2.14 - Создание проекта в gitflic

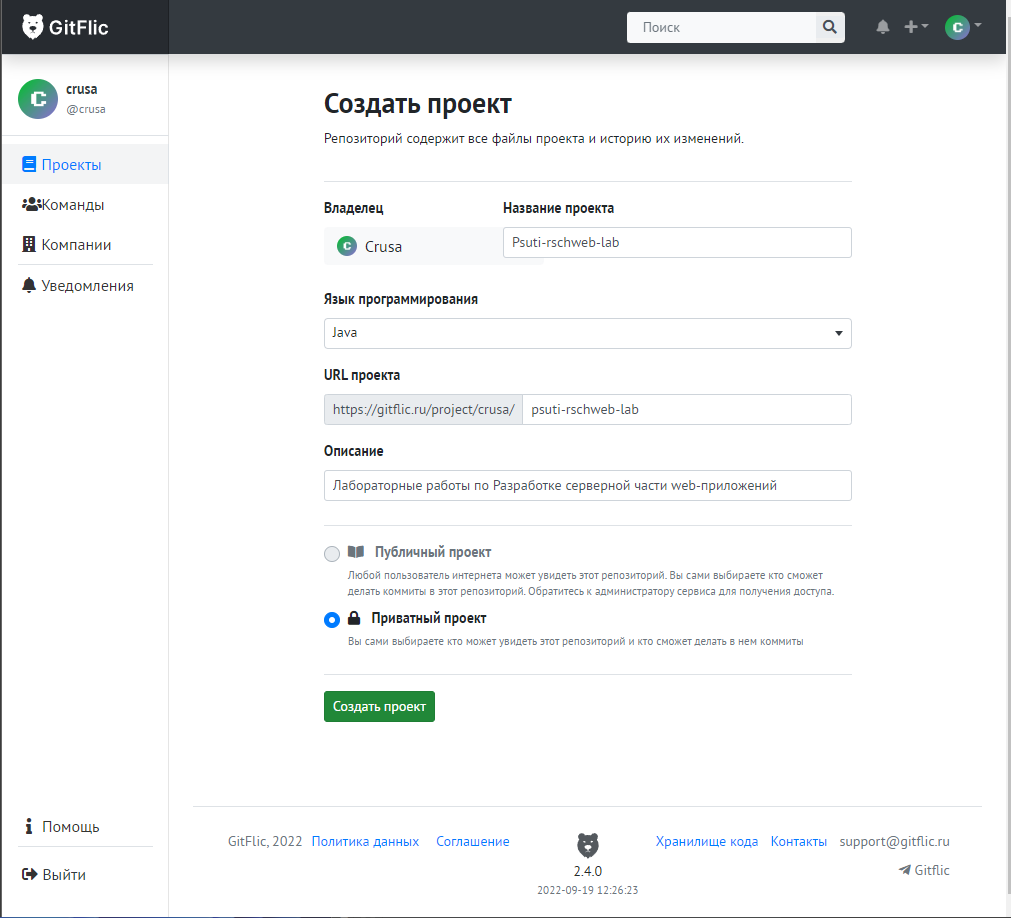


Рис. 2.15 - Именование, описание проекта. Указание url, языка программирования

1. Создание репозиторий.

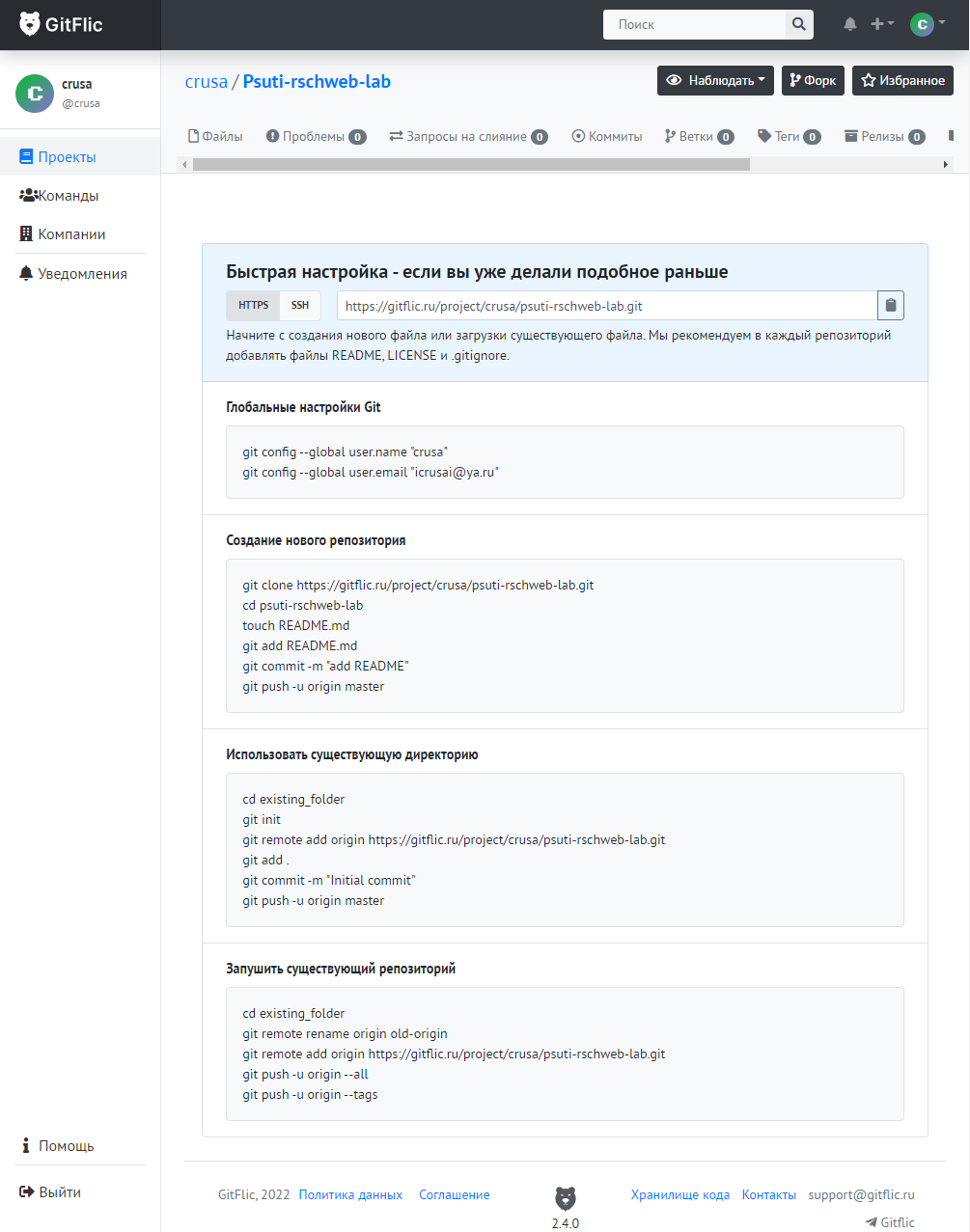


Рис. 2.15 - Созданный репозиторий. Помощь в использовании

1. После чего можно привязать существующий проект к репозиторию