GeekBrains

Основы веб-разработки на Spring Framework

Spring REST

Spring REST. HTTP 1.1. CRUD-операции.

Оглавление

Введение

В REST-сервисах запросы и ответы строятся вокруг передачи представлений ресурсов, которые определяются глобальными идентификаторами — обычно унифицированными идентификаторами ресурсов (Uniform Resource Identifier — URI). Клиентские же приложения используют для управления ресурсами HTTP-методы (такие как GET, POST, PUT, DELETE).

Например, ответом на GET-запрос http://localhost:8080/app/students/15 будет представление студента с ID = 15, которое содержит информацию о студенте в формате JSON/XML. Вид представления зависит от реализации на стороне сервера и MIME-типа запроса клиента.

Веб-сервисы RESTful реализуются с использованием HTTP и принципов REST. До появления поддержки в Spring для создания REST-сервисов в Java использовались библиотеки Restlet, RestEasy и Jersey.

Формат JSON

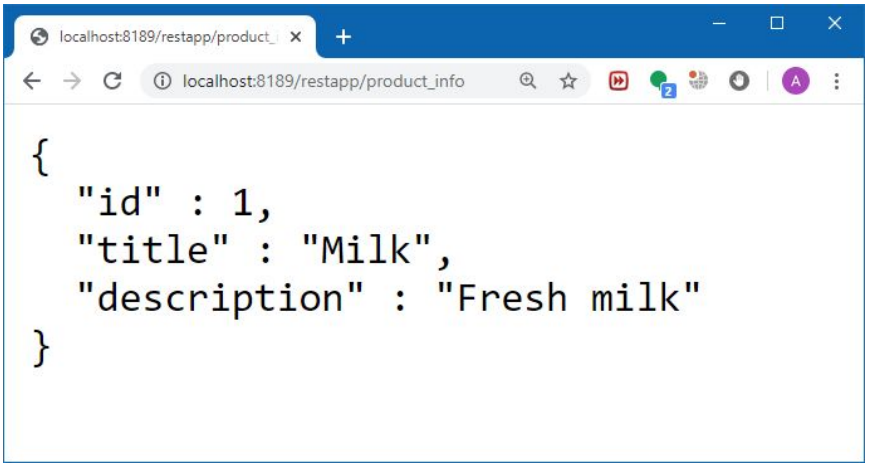
Давайте посмотрим как в обычном Spring Boot приложении работать с JSON представлением объектов. Допустим у нас есть простой класс Product, который не является сущностью, то есть нам для работы с продуктом не придется обращаться к базе данных.

```
public class Product {
    private Long id;
    private String title;
    private String description;

    // Геттеры и сеттеры

    public Product(Long id, String title, String description) {
        this.id = id;
        this.title = title;
        this.description = description;
    }
}
```

Мы хотим чтобы в ответ на GET запрос, веб-сервис прислал бы объект типа Product в формате JSON. Для этого создадим ProductController и пропишем в нем следующий код.

```
@Controller
public class ProductController {
    @GetMapping("/product_info")
    @ResponseBody
    public Product getOneProduct() {
        return new Product(1L, "Milk", "Fresh milk");
    }
}
```

Если раньше в ответ на наш запрос контроллер возвращал имя представления (view), и потом пользователю приходила html-страница, собранная по этому представлению. То в данном случае аннотация @ResponseBody говорит о том, что в ответ на запрос, сервер отправит клиенту указанный объект, и зашьет его в тело ответа. Вот так результат будет выглядеть в браузере.



В чем преимущество формата JSON - это текстовый формат представления объектов, который поддерживается многими языками программирования “из коробки”. А если вдруг подходящей библиотеки найти не удалось, то можно написать и свой парсер. Поскольку структура и состояние объектов описывается в текстовом виде, вы можете передавать объекты между сервисами, написанными на разных языках программирования.

Каким образом Spring преобразовал наш объект в JSON? В Spring Boot интегрирована библиотека Jackson, которая берет на себя задачу сериализации и десериализации в/из формата JSON.

Давайте разберемся в том, по каким правилам объекты преобразуются в JSON. Если у нас есть класс Product, имеющий числовое поле id, и два строковых поля title и description, то получим:

```
{
  "id": 1,
  "title": "Milk",
  "description": "Fresh milk"
}
```

Фигурные скобки означают сам объект, класс при этом не указывается. Внутри скобок прописываются пары ключ-значение, обозначающие поле объекта и его значение, каждая такая пара разделяется двоеточием с пробелом. Имя ключа заключено в двойные кавычки. Если значение числовое, то записывается без кавычек, если значение текстовое, то оно берётся в двойные кавычки. Если в объекте есть числовой массив, то его содержимое берется в квадратные скобки.

```
{
  "values": [ 1, 2, 3, 4, 5 ]
}
```

Если один объект ссылается на другой объект, например, экземпляр класса Пользователь содержит поле типа Адрес, то получим запись вида:

```
{
  "firstName": "Alexander",
  "age": 30,
  "address": {
    "city": Rostov-on-Don,
    "street": Central st. 1
  }
}
```

Любые объекты строятся по правилам, описанным выше, как видите формат достаточно простой. Разобравшись с тем, как можно передавать любые объекты в формате JSON,

Архитектура REST

REST (**Re**presentational **S**tate **T**ransfer - “передача состояния представления”) - архитектурный стиль построения распределенных веб-приложений, предложенный Роем Филдингом в его диссертации “*Architectural Styles and the Design of Network-based Software Architectures*”.

REST использует клиент-серверную архитектуру. Клиент и сервер решают разные задачи. Сервер хранит и/или манипулирует информацией и делает ее доступной для клиента. Клиент берет эту информацию и отображает ее пользователю и/или использует ее для выполнения последующих запросов. Такое разделение задач позволяет как клиенту, так и серверу развиваться независимо, так как требуется только то, чтобы интерфейс оставался прежним.

REST не хранит состояние (**stateless**). Это означает, что “общение” между клиентом и сервером всегда содержит всю информацию, необходимую для выполнения запросов. На сервере не сессия работы с клиентом, она полностью хранится на стороне клиента. Если для доступа к ресурсу требуется аутентификация, клиент должен аутентифицировать себя при каждом запросе.

REST кешируется. Клиент, сервер и любые промежуточные компоненты могут кэшировать ресурсы для повышения производительности.

REST обеспечивает единый интерфейс между компонентами. Это упрощает архитектуру, так как все компоненты используют одни и те же правила для общения друг с другом. Это также облегчает понимание взаимодействия между различными компонентами системы. Для этого требуется ряд ограничений. Они рассматриваются в остальной части главы.

REST имеет многослойную структуру. Отдельные компоненты не могут видеть за пределами непосредственного слоя, с которым они взаимодействуют. Это означает, что клиент, подключающийся к промежуточному компоненту, например прокси, не знает, что находится за его пределами. Это позволяет компонентам быть независимыми и, следовательно, легко заменяемыми или расширяемыми.

Подготовка к работе

Для разбора принципов работы с REST-сервисами на Spring необходимо подготовить базовый проект: подключить зависимости (**spring-boot-web-starter**, **spring-boot-starter-data-jpa**, **postgresql**). Проект работает со списком студентов и книг, для которых необходимо подготовить классы-сущности и скрипты для генерации таблиц в базе данных.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
```

```
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

SQL-скрипты для тестовой базы данных:

```
DROP TABLE IF EXISTS students;

CREATE TABLE students (
   id bigserial NOT NULL,
   name varchar(100) NOT NULL,
   PRIMARY KEY(id)
);

INSERT INTO students (name) VALUES ('Bob'), ('John'), ('Michael');

DROP TABLE IF EXISTS books;
CREATE TABLE books (
   id bigserial NOT NULL,
   title varchar(100) NOT NULL,
   PRIMARY KEY(id)
);

DROP TABLE IF EXISTS students_books;
CREATE TABLE students_books (
  student_id bigint NOT NULL,
   book_id bigint NOT NULL,

   PRIMARY KEY (student_id, book_id),

   FOREIGN KEY (student_id) REFERENCES students (id),
   FOREIGN KEY (book_id) REFERENCES books (id)
);

INSERT INTO books (title) VALUES ('Harry Potter'), ('Lord Of The Ring');
```

Классы сущности:

```
@Entity
@Table(name = "students")
public class Student {
  @Id
  @Column(name = "id")
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(name = "name")
  private String name;

  @ManyToMany
  @JoinTable(
    name = "students_books",
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "book_id")
  )
  private List<Book> books;

  // Геттеры и сеттеры

  public Student() {
  }
}

@Entity
@Table(name = "books")
public class Book {
  @Id
  @Column(name = "id")
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(name = "title")
  private String title;

  @ManyToMany
  @JoinTable(
    name = "students_books",
    joinColumns=@JoinColumn(name = "book_id"),
    inverseJoinColumns=@JoinColumn(name = "student_id")
  )
  @JsonBackReference
  private List<Student> students;

  // Геттеры и сеттеры

  public Book() {
  }
}
```

CRUD в REST

Рассмотрим методы, часто применяемые при построении REST-сервисов, определенные в спецификации HTTP 1.1:

- **GET** получает информацию о ресурсе по его URI. В контексте REST-сервисов этот метод позволяет запрашивать/получать ресурсы с сервера. Представляет операцию READ (C**R**UD). Возможные статусы ответов: 200(OK), 404(NOT FOUND), 400(BAD REQUEST).
- **POST** используется для передачи серверу объекта сущности, заключенной в запрос, для сохранения его на сервере как нового ресурса. Используется в качестве операции CREATE (**C**RUD). Возможные статусы ответов: 200(OK), 404(NOT FOUND), 400(BAD REQUEST), 201(CREATED).
- **PUT** служит для передачи серверу объекта-сущности и сохранения на стороне сервера под определенным URI. Используется в качестве операции UPDATE (CR**U**D). По спецификации данный метод может создать новый ресурс, если его не существует. С другой стороны, есть возможность оставить задачу создания новых ресурсов на сервере только методу POST. Выбор правила работы остается за разработчиком. Возможные статусы ответов: 200(OK), 400(BAD REQUEST).
- **DELETE** применяется для удаления ресурса. Используется в качестве операции DELETE (CRU**D**).

Реализация контроллера

Создадим контроллер для обработки запросов к REST-сервису. Чтобы отделить от основной логики работы веб-сервиса, можно сделать специальный **end-point**, например, **/api/v1**.

```
@RequestMapping("/api/v1")
@RestController
public class StudentsRestController {
    private StudentsService studentsService;

    @Autowired
    public void setStudentsService(StudentsService studentsService) {
        this.studentsService = studentsService;
    }

    @GetMapping("/students/{id}")
    public Student getStudentById(@PathVariable Long id) {
        return studentsService.getStudentById(id);
    }

    @GetMapping("/students")
    public List<Student> getAllStudents() {
        return studentsService.getAllStudentsList();
    }
```

```
    @PostMapping("/students")
    public Student addStudent(@RequestBody Student student) {
        student.setId(0L);
        return studentsService.saveOrUpdate(student);
    }

    @PutMapping(path = "/students", consumes =
{MediaType.APPLICATION_JSON_VALUE})
    public Student updateStudent(@RequestBody Student student) {
        return studentsService.saveOrUpdate(student);
    }

    @DeleteMapping("/students/{id}")
    public int deleteStudent(@PathVariable Long id) {
        studentsService.delete(id);
        return HttpStatus.OK.value();
    }

    @ExceptionHandler
    public ResponseEntity<StudentsErrorResponse>
handleException(StudentNotFoundException exc) {
        StudentsErrorResponse studentsErrorResponse = new
StudentsErrorResponse();
        studentsErrorResponse.setStatus(HttpStatus.NOT_FOUND.value());
        studentsErrorResponse.setMessage(exc.getMessage());
        studentsErrorResponse.setTimestamp(System.currentTimeMillis());
        return new ResponseEntity<>(studentsErrorResponse,
HttpStatus.NOT_FOUND);
    }
}
```

Из приведенного выше кода видно, что в url не включается информация о выполняемом над ресурсами действии. В качестве **end-point** используется один и тот же адрес **/students**, а в зависимости от того, какой именно запрос пришел, выполняется одна из CRUD-операций.

@RestController совмещает две аннотации: **@Controller** и **@ResponseBody.** @ResponseBody, добавленный к классу, означает, что над всеми методами этого контроллера будут автоматически проставлены @ResponseBody.

Рассмотрим каждый метод.

- **getStudentById()** — выполняется при получении от клиента GET-запроса **/students/{id}**, в котором в качестве **PathVariable** указан id запрашиваемого студента. Метод через **studentsService** находит в базе данных студента и возвращает его клиенту в виде JSON-объекта.
- **getAllStudents()** — выполняется при получении от клиента GET-запроса **/students** и в качестве ответа возвращает список всех студентов в виде **JSONArray**.
- **addStudent()** — реагирует на POST-запрос **/students**, в который вшит объект типа **Student**. Поскольку **POST** должен отвечать за добавление нового ресурса, id полученного объекта обнуляется.

- **updateStudent()** — получает PUT-запрос **/student**, из которого извлекает объект типа **Student** и использует его для обновления записи в базе данных.
- **deleteStudent()** — выполняется при получении от клиента DELETE-запроса **/students/{id}** и служит для удаления студента из базы данных по его id.

Для обработки возможных исключений реализован метод **handleException()**, который в случае возникновения исключения отправляет ответ клиенту в виде объекта типа **ResponseEntity<StudentsErrorResponse>**. Классы обработки исключений:

```
public class StudentNotFoundException extends RuntimeException {
    public StudentNotFoundException(String message) {
        super(message);
    }
}

public class StudentsErrorResponse {
    private int status;
    private String message;
    private long timestamp;

    // Геттеры и сеттеры

    public StudentsErrorResponse() {
    }
}
```

MapStruct

При разработке веб-приложений зачастую приходится делать преобразования вида Entity => DTO или DTO => Entity. Код для подобного рода операций можно прописывать вручную, но это приведет к появлению большого количества однотипного кода, и следовательно к возможным мелким ошибкам при написании кода. Вместо этого можно взять готовое решение - библиотеку MapStruct, которая на этапе компиляции генерирует код для преобразователей.

Для подключения MapStruct к Maven проекту необходимо:

1. Подключить зависимости

```
<dependency>
   <groupId>org.mapstruct</groupId>
   <artifactId>mapstruct</artifactId>
   <version>1.3.1.Final</version>
</dependency>
<dependency>
   <groupId>org.mapstruct</groupId>
   <artifactId>mapstruct-processor</artifactId>
   <version>1.3.1.Final</version>
</dependency>
```

2. И в build > plugins добавить

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.5.1</version>
   <configuration>
       <source>1.8</source>
       <target>1.8</target>
       <annotationProcessorPaths>
           <path>
               <groupId>org.mapstruct</groupId>
               <artifactId>mapstruct-processor</artifactId>
               <version>1.3.1.Final</version>
           </path>
       </annotationProcessorPaths>
   </configuration>
</plugin>
```

3. Готово, можно описывать правила преобразования.

Допустим у нас есть две сущности Category и Item. Item это товар, имеющий id, название, цену и категорию, к которой он относится (смартфоны, ноутбуки, процессоры и пр.). Категория соответственно имеет id, название и список товаров, которые к ней относятся.

```
@Entity
@Table(name = "categories")
public class Category {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Long id;

   @Column(name = "title")
   private String title;

   @OneToMany(mappedBy = "category")
   private List<Item> items;

   // … геттеры/сеттеры/конструкторы
}

@Entity
@Table(name = "items")
public class Item {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Long id;

   @Column(name = "title")
   private String title;

   @Column(name = "price")
   private int price;
```

```
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    // … геттеры/сеттеры/конструкторы
}
```

Давайте начнем с преобразования категории в Data Transfer Object. Для этого создадим класс CategoryDto. В DTO нас интересует только id и название категории и абсолютно не интересует список товаров.

```
public class CategoryDto {
    private Long id;
    private String title;

    // … геттеры/сеттеры/конструкторы
}
```

Теперь необходимо объяснить библиотеке MapStruct каким образом должно происходить преобразование Category > CategoryDto и CategorDto > Category. Для этого создается интерфейс, помеченный аннотацией **@Mapper**.

```
@Mapper
public interface CategoryMapper {
    CategoryMapper MAPPER = Mappers.getMapper(CategoryMapper.class); // (1)

    Category toCategory(CategoryDto categoryDto); // (2)

    @InheritInverseConfiguration // (3)
    CategoryDto fromCategory(Category category); // (4)

    List<Category> toCategoryList(List<CategoryDto> categoryDtos); // (5)

    List<CategoryDto> fromCategoryList(List<Category> categories); // (6)
}
```

В (1) мы создаем ссылку на mapper для получения к нему доступа в коде проекта (*CategoryMapper.MAPPER.toCategory(...)*). Метод toCategory будет заниматься преобразованием объекта типа CategoryDto в объект типа Category, название метода не имеет значения, mapper будет смотреть на тип аргумента и тип возвращаемого значения. То же самое можно сказать про (3-4), только преобразование производится в обратную сторону. Про @InheritInverseConfiguration будет рассказано на примере Item. Чтобы вручную не заниматься преобразованием листов объектов, добавлены соответствующие методы.

Каким образом MapStruct понимает как передавать данные из одного типа в другой? Маппер проводит соответствие через названия полей. То есть значение поля title из Category будет “перенесено” в поле title CategoryDto. И так для всех полей.

Может возникнуть ситуация, при которой названия полей в сущности и дто отличаются. В этом случае поможет аннотация @Mapping. Вот как выглядит маппер для Item.

```
@Mapper(uses = { CategoryMapper.class })
public interface ItemMapper {
```

```
    ItemMapper MAPPER = Mappers.getMapper(ItemMapper.class);

    @Mapping(source = "categoryDto", target = "category")
    Item toItem(ItemDto itemDto);

    List<Item> toItemList(List<ItemDto> itemDtos);

    @InheritInverseConfiguration
    ItemDto fromItem(Item item);

    List<ItemDto> fromItemList(List<Item> items);
}
```

Обратите внимание на

```
@Mapping(source = "categoryDto", target = "category")
Item toItem(ItemDto itemDto);
```

Этот метод занимается преобразованием **ItemDto > Item**, **source = “categoryDto”** означает что в классе источнике (**ItemDto**) есть поле **categoryDto**, которое необходимо преобразовать в поле **category** целевого класса **Item** (**target = “category”**).

Теперь должно стать понятным назначение аннотации **@InheritInverseConfiguration**, мы указываем мапперу, что ему необходимо брать “обратные” правила преобразования из уже описанных выше (условно говоря для **fromItem**: `@Mapping(source = "category", target = "categoryDto")`). Набор правил может получиться достаточно большим (10+ полей), так что проще сделать вот такую обратную ссылку.

Поскольку ItemDto внутри себя содержит ссылку на CategoryDto, то ему мапперу для Item необходимо уметь работать и с CategoryDto, поэтому мы добавляем ссылку на CategoryMapper.class.

```
@Mapper(uses = { CategoryMapper.class })
```

После описания всех необходимых мапперов, можем применить их в коде нашего сервера.

```
@Service
public class ItemService {
    // ...

    public ItemDto save(ItemDto itemDto) {
        Item item = ItemMapper.MAPPER.toItem(itemDto);
        item = itemRepository.save(item);
        return ItemMapper.MAPPER.fromItem(item);
    }

    public List<ItemDto> findAll() {
        return
ItemMapper.MAPPER.fromItemList(itemRepository.findAllItemsWithCategories());
    }

    public ItemDto findOne(Long id) {
        return ItemMapper.MAPPER.fromItem(itemRepository.findById(id).get());
    }
```

```
// ...
```

Практическое задание

1. Добавить к проекту REST API для одной из сущностей.

Дополнительные материалы

1. Spring REST Docs.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf