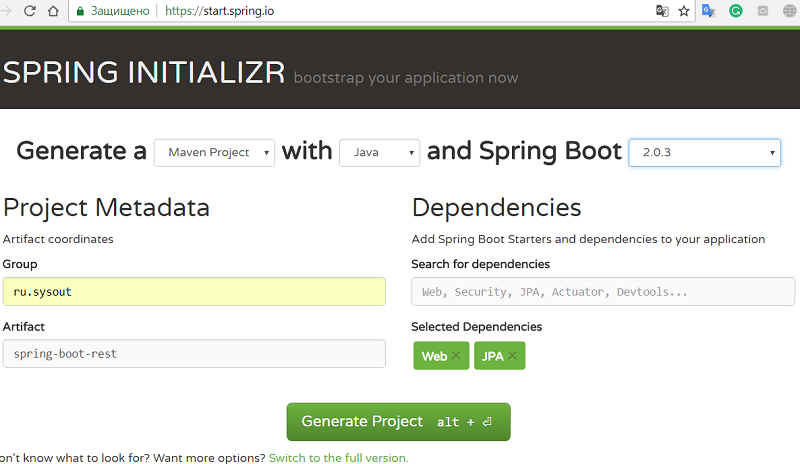
I. Spring Boot REST API

В этой статье мы напишем маленькое приложение на Spring Boot, которое предоставляет REST-сервисы.  
Архитектура приложения будет стандартна и включать несколько слоев: dao, (service отсутствует) и controller. Все шаги по построению приложения просты. Код приложения можно скачать на [GitHub](https://github.com/myluckagain/sysout/tree/master/spring-boot-rest).

**Spring Initializr**

Заготовку любого проекта на Spring Boot удобно взять на <https://start.spring.io/>. Здесь мы придумываем имя группы и имя артифакта будущего проекта на Maven, выбираем dependency, которые нам точно понадобятся и генерируем проект. А потом импортируем его в Eclipse как Maven-проект.

Инициализация Spring Boot проекта

Нам понадобятся зависимости WEB, JPA и H2.

Встроенную базу данных H2 прикрепляем потому, что ее проще использовать для демонстрационных целей: не придется устанавливать настоящую базу вроде MySQL, а также прописывать ее настройки.

**Maven-зависимости**

В результате получаем сгенерированный POM с такими зависимостями:

<**dependencies**>

<**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**>com.h2database</**groupId**>

<**artifactId**>h2</**artifactId**>

<**scope**>runtime</**scope**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-test</**artifactId**>

<**scope**>test</**scope**>

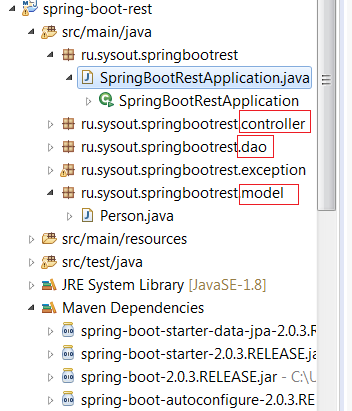
</**dependency**>

</**dependencies**>

Хоть файл и сгенерирован, это не мешает нам добавлять в него новые зависимости при необходимости.

**Слои (multi-layer architecture)**

Импортированный проект выглядит так, плюс мы создали пакеты для моделей, dao и контроллеров (выделены красным):



Service-layer отсутствует потому, что приложение слишком простое, бизнес-логики тут нет.

**Модель**

Модель будет состоять из одного класса *Person*:

@Entity

**public** **class** Person {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** **Long** id;

@NotNull

**private** **String** name;

**public** Person() {

}

**public** Person(**String** name) {

this.name = name;

}

// ...getters and setters

}

*Person* аннотирован как JPA-сущность, то есть при запуске приложения в базе данных  будет создана таблица с таким именем и полями.

**DAO**

DAO-layer предназначен для работы с данными. У нас он состоит из одного бина *PersonRepository*.

Благодаря аннотации *@Repository* и интерфейсу *JpaRepository* DAO-layer предельно прост:

@Repository

**public** **interface** PersonRepository **extends** JpaRepository<Person, **Long**> {

}

Мы создаем бин *PersonRepository*, аннотируя его с помощью *@Repository*. Полученный бин реализует все написанные в интерфейсе “JpaRepository” методы *(стандартные операции поиска, добавления и удаления*), можно ничего не писать самостоятельно, если не нужны какие-то особые запросы к базе.

Service-layer опускаем, поскольку приложение простое. В контроллере будем использовать бин *PersonRepository.*

**Контроллер**

Здесь реализованы запросы поиска, добавления, редактирования и удаления *Person*.

* Класс аннотирован *@RestController* и указан основной путь к запросам  этого контроллера- “/persons”.
* С помощью аннотации *@Autowired* бин *personRepository* инжектирован в поле контроллера – теперь его можно использовать.

@RestController

@RequestMapping("/persons")

**public** **class** PersonController {

@Autowired

**private** PersonRepository personRepository;

@**Post**Mapping

**public** ResponseEntity<Person> **create**Person(@RequestBody @Valid Person person) {

Person p = personRepository.save(person);

**return** ResponseEntity.status(201).body(p); //возможно метод “.status()“ автоматически формирует в ответе стартовую

//строку с переданным статусом (в данном случаи это “201”) + заголовки. А потом

//нужно применять метод “body()” для добавления тела к объекту ответа (*тут* “id”, созданной в БД записи)

//Аннотация @Valid (из пакета “javax.validation”) говорит Spring передать

// объект валидатору, прежде чем делать что-либо еще.

}

@**Get**Mapping

**public** ResponseEntity<List<Person>> listAllPersons() {

personRepository.save(**new** Person("Kate"));

List<Person> persons = personRepository.findAll();

**return** ResponseEntity.ok().body(persons); //возможно метод “.ok()“ автоматически формирует в ответе стартовую строку

//со статусом “ok” + заголовки. А потом нужно применять метод “body()” для

//добавления тела к объекту ответа (*как правило это html-страница*)

}

@**Get**Mapping(value = "/{personId}")

**public** ResponseEntity<Person> **get**Person(@PathVariable("personId") **Long** personId)

**throws** EntityNotFoundException {

Optional<Person> person = personRepository.findById(personId);

**if** (!person.isPresent())**throw** **new** EntityNotFoundException("id-" + personId);

**return** ResponseEntity.ok().body(person.get());

}

??? Не понимаю, какой строкой, в следующем методе, вносится изменение в БД вместо записи с запрашиваемым “id ” ???

@**Put**Mapping(value = "/{personId}")

**public** ResponseEntity<Person> **update**Person(@RequestBody @Valid Person person, @PathVariable("personId") **Long** personId) **throws** EntityNotFoundException {

Optional<Person> p = personRepository.findById(personId);

**if** (!p.isPresent()) **throw** **new** EntityNotFoundException("id-" + personId);

**return** ResponseEntity.ok().body(personRepository.save(person)); //тут в теле вернется “id”, измененной в БД записи

}

@**Delete**Mapping(value = "/{personId}")

**public** ResponseEntity<Person> **delete**Person(@PathVariable("personId") **Long** personId)

**throws** EntityNotFoundException {

Optional<Person> p = personRepository.findById(personId);

**if** (!p.isPresent()) **throw** **new** EntityNotFoundException("id-" + personId);

personRepository.deleteById(personId);

**return** ResponseEntity.ok().body(p.get()); //зачем кроме статуса возвращать еще и удаленную сущность мне непонятно??

}

}

Тут два метода для получения данных (аннотации *@GetMapping*) и три – для редактирования. Все методы аннотированны:

* *@GetMapping* – для GET-запросов, получения *Person*
* *@PostMapping* – для POST-запросов, т.е. добавления *Person*
* *@PutMapping* – для PUT-запросов, редактирования *Person*
* *@DeleteMapping* – для DELETE-запросов, удаления *Person*

Возвращаем обычно *ResponseEntity<Person>*, это более гибкий вариант, чем вернуть просто Person, поскольку для *ResponseEntity* можно установить Http-статус ответа – *ResponseEntity.ok()* – это 200 или *ResponseEntity.statu*s(201).

В методе*body()* передается возвращаемая сущность – в вышеприведенных методах это *Person*(либо список *Person)*. Под капотом она конвертируется в JSON благодаря тому, что у нас стоит аннотация *@RestController*. Для конвертации под капотом Spring Boot использует библиотеку Jackson – она включена благодаря Maven-зависимости spring-boot-starter-web.

Если надо возвратить JSON с описанием ошибки, выбрасываем исключение. Например, если запрос на редактирование содержит id несуществующего Person, то выбрасываем *EntityNotFoundException*. Как обрабатывать исключения и кастомизировать JSON с ошибкой, описано в следующей статье.

**Запуск**

Для запуска Spring Boot приложения запускаем *main()* этого класса:

@SpringBootApplication

**public** **class** SpringBootRestApplication {

**public** **static** **void** main(**String**[] args) {

SpringApplication.run(SpringBootRestApplication.class, args);

}

}

При этом будет запущен веб-сервер, отдельно его устанавливать и запускать не надо – это одно из преимуществ Spring Boot приложения.

Также не надо задавать пути для поиска бинов, они найдутся автоматически. Единственное, класс *SpringBootRestApplication* не надо перекладывать в подпакет, он должен быть на верхнем уровне иерархии, иначе с поиском бинов возникнут проблемы. Когда мы сгенерировали заготовку приложения, этот файл уже был именно там, где надо – перекладывать его не следует.

**Тестирование**

Осталось проверить, что методы контроллера работают. Составлять запросы будем с помощью бесплатного графического приложения [Postman](https://www.getpostman.com/" \t "_blank) (но учтите, оно ресурсоемкое). Как писать тесты, рассмотрим в другой статье.

**Добавление Person**

POST http://localhost:8080/persons/

{

"name": "Joe"

}

Ответ, возвращается вновь добавленный *Person* с id=1:

{

**"id":** 1,

**"name":** "Joe"

}

**Редактирование Person**

PUT http://localhost:8080/persons/1

{

"name": "Jane"

}

Ответ, возвращается отредактированный *Person* с id=1 и name=’Jane’:

{

**"id":** 1,

**"name":** "Jane"

}

**Получение Person**

GET http://localhost:8080/persons/1

Ответ, возвращается *Person*:

{

**"id":** 1,

**"name":** "Jane"

}

**Получение списка Person**

GET http://localhost:8080/persons/

Ответ, возвращается список, состоящий из одного элемента *Person*:

[

{

**"id":** 1,

**"name":** "Jane"

}

]

**Удаление Person**

DELETE http://localhost:8080/persons/1

Ответ, возвращается удаленный *Person* с id=1 и name=’Jane’:

{

**"id":** 1,

**"name":** "Jane"

}

II. Spring Boot REST API – обработка исключений

При создании REST API всегда возникают требования к формату ответа, выдаваемого при ошибке (например, при отсутствии данных или неправильном запросе). И конечно, этот формат не совпадает с тем, что выдается Spring Boot по умолчанию.  
  
Давайте рассмотрим:

• Как исправить формат ответа JSON, который в Spring Boot выдается по умолчанию при ошибке, на свой другой  
• Как обрабатывать исключения, выбрасываемые со всех контроллеров в едином классе, не распыляясь по всему приложению

**Предварительные условия**

Обрабатывать исключения будем в простом Spring Boot приложении, о его создании рассказано [тут](https://sysout.ru/spring-boot-rest-api/). Оно предоставляет REST API для сущности *Person:*

@Entity

**public** **class** Person {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

**private** **Long** id;

@NotNull

**private** **String** name;

**public** Person() {

}

**public** Person(**String** name) {

this.name = name;

}

// getters and setters

}

Исходники первоначальной заготовки можно скачать на [GitHub](https://github.com/myluckagain/sysout/tree/master/spring-boot-rest).

**Какой JSON выдается по умолчанию при ошибке**

Попробуем ошибиться – сделать запрос с ошибочным JSON. Я делаю запрос с помощью программы [POSTMAN](https://www.getpostman.com/), но можно тестировать запросы любым другим способом:

POST http://localhost:8080/persons

{

**"name":** "Jane"l

}

Spring выдает нам такой ответ:

{

**"timestamp":** "2018-07-09T09:58:25.455+0000",

**"status":** 400,

**"error":** "Bad Request",

**"message":** "JSON parse error: Unexpected character ('l' (code 108)): was expecting comma to separate Object entries; nested exception is com.fasterxml.jackson.core.JsonParseException: Unexpected character ('l' (code 108)): was expecting comma to separate Object entries\n at [Source: (PushbackInputStream); line: 3, column: 17]",

**"path":** "/persons"

}

**Какой JSON-ответ нам нужен**

Нас стандартный ответ не устраивает, мы будет обрабатывать исключения вручную и возвращать такой ответ:

{

**"status":** "BAD\_REQUEST",

**"message":** "Malformed JSON request",

**"debugMessage":** "JSON parse error: Unexpected character ('l' (code 108)): was expecting comma to separate Object entries; nested exception is com.fasterxml.jackson.core.JsonParseException: Unexpected character ('l' (code 108)): was expecting comma to separate Object entries\n at [Source: (PushbackInputStream); line: 2, column: 18]"

}

Суть в том, что мы его немного поменяли, и ниже будет показано, как это сделать. В реальной жизни тут можно вставить любые поля под свои нужды.

**Класс обработки исключений**

1. Для того, чтобы обрабатывать все исключения, выбрасываемые из контроллеров, создадим класс *RestExceptionHandler*, который расширяет имеющийся класс *ResponseEntityExceptionHandler.* Это класс в котором уже есть 17 готовых исключений *(имена которых начинаются с “handle…”)*. 2. Кроме того есть еще класс PersistenceException c 10 исключениями

3. и класс NestedRuntimeException с десятками готовых исключений.

@ControllerAdvice

**public** **class** RestExceptionHandler **extends** ResponseEntityExceptionHandler {

...

}

**@ControllerAdvice**

Обратите внимание, что наш класс обработки исключений необходимо аннотировать с помощью *@ControllerAdvice*

**Класс *ResponseEntityExceptionHandler***

В этом классе, который мы расширили, уже есть ряд методов, обрабатывающих исключения, в том числе сэмулированное выше исключение  *HttpMessageNotReadableException.*Чтобы вернуть свой JSON, надо переопределить метод *handleHttpMessageNotReadable()*:

@Override

**protected** ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex,

HttpHeaders headers, HttpStatus status, WebRequest request) {

ApiError apiError = **new** ApiError(status, "Malformed JSON request", ex);

**return** **new** ResponseEntity(apiError, status);

}

Как видите, ошибка здесь возвращается в *ApiError* – этот класс как раз и проектируется с учетом требований к формату возвращаемой ошибке. В нем есть все поля, которые нужны именно нам.

**ApiError**

У нас он выглядит так:

@JsonInclude(Include.NON\_NULL)

**public** **class** ApiError {

**private** HttpStatus status;

**private** **String** message;

**private** **String** debugMessage;

**private** List<FieldValidationError> fieldValidationErrors;

...

}

Обратите внимание, что мы создали еще поле *fieldValidationErrors(*это поле понадобится для метода **handleMethodArgumentNotValid**, написанного ниже*)*, которое будет содержать ошибки валидиции полей JPA-сущностей, в нашем случае полей сущности *Person.*

Мы не видели это поле в ответе, поскольку *ApiError* аннотировано *@JsonInclude(Include.NON\_NULL),* а значит нулевые поля не сериализуются и не включаются в ответ. А поскольку в предыдущем запросе ошибок валидации не возникало, это поле было нулевым.

Валидация инициируется только в том случае, если в котроллере перед аргументом стоит аннотация *@Valid*. В нашем контроллере в методе createPerson() она как раз стоит:

@PostMapping

**public** Person createPerson(@RequestBody @Valid Person person) {

**return** personRepository.save(person);

}

**Еще один некорректный запрос**

Давайте теперь сделаем запрос с корректным JSON но невалидной JPA-сущностью, чтобы ответ содержал ошибки валидации. Например, пусть сущность не содержит поле *name:*

POST http://localhost:8080/persons

{

**"namet":** "Jane"

}

Поскольку поле *name* сущности *Person* аннотированно @NotNull, такой запрос должен повлечь ошибку валидации поля. А исключения, касающиеся валидации полей, обрабатываются в методе *handleMethodArgumentNotValid()*, вот его то мы и переопределим для создания нашего JSON с перечнем ошибок для каждого поля.

**handleMethodArgumentNotValid()**

@Override

**protected** ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,

HttpHeaders headers, HttpStatus status, WebRequest request) {

ApiError apiError = **new** ApiError(status, "method arg not valid", ex);

apiError.addValidationErrors(ex.getBindingResult().getFieldErrors());

**return** **new** ResponseEntity<Object>(apiError, HttpStatus.BAD\_REQUEST);

}

Как видите, здесь мы еще заполнили список *fieldValidationErrors* нашего класса *ApiError*, и взяли мы его из *ex.getBindingResult().getFieldErrors()* исключения Spring *MethodArgumentNotValidException.* Вот такое дополнительное перекладывание ошибок в нашу пользовательскую *ApiError в*пользовательском методе обработки исключения *MethodArgumentNotValidException*.

Выглядит класс *FieldValidationError* так:

**public** **class** FieldValidationError {

**private** **String** object;

**private** **String** field;

**private** Object rejectedValue;

**private** **String** message;

// constructor/getters/setters и

}

Итак, какой же теперь мы получим ответ на ошибочный запрос:

{

**"status":** "BAD\_REQUEST",

**"message":** "method arg not valid",

**"debugMessage":** "Validation failed for argument at index 0 in method: public ru.sysout.springsecurity.model.Person ru.sysout.springsecurity.controller.PersonController.createPerson(ru.sysout.springsecurity.model.Person), with 1 error(s): [Field error in object 'person' on field 'name': rejected value [null]; codes [NotNull.person.name,NotNull.name,NotNull.java.lang.String,NotNull]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [person.name,name]; arguments []; default message [name]]; default message [должно быть задано]] ",

**"apiValidationErrors":** [

{

**"object":** "person",

**"field":** "name",

**"rejectedValue":** **null**,

**"message":** "должно быть задано"

}

]

}

**Пользовательское исключение**

В классе *ResponseEntityExceptionHandler*  обрабатывается довольно много стандартных исключений, убедитесь, что вы переопределили все нужные методы, иначе ошибка внезапно вернется не в нашем формате, если какое-то из исключений не учесть, а оно возникнет и обработается в родительском классе.

Но у нас же есть еще и собственные пользовательские исключения. Как быть с ними? Переопределить можно только методы, обрабатывающие исключения Spring. Для обработки пользовательского исключения надо дописать метод **1.** и аннотировать его **2.**. Ниже покажу как. Но сначала само исключение – мы будем его выбрасывать в том случае, если сущности не найдено. Например, мы получаем в контроллере список *Person* и выбрасываем исключение, если он пустой:

@GetMapping

**public** ResponseEntity<List<Person>> listAllPersons() **throws** EntityNotFoundException {

// personRepository.save(new Person("Kate"));

List<Person> persons = personRepository.findAll();

**if** (persons.isEmpty()) {

**throw** **new** EntityNotFoundException();

}

**return** **new** ResponseEntity<List<Person>>(persons, HttpStatus.OK);

}

Дальше будет создано собственное исключение для данного контроллера. Если создавать исключение для использования во всех контроллерах, то нужно будет пользоваться аннотацией @ControllerAdvice, а не @ExceptionHandler.

**1.** Пользовательское исключение конечно должно передавать дополнительную информацию, но для простоты пусть оно будет таким:

**public** **class** EntityNotFoundException **extends** Exception{

}

**2.** Итак, как же его обрабатывать в нашем классе? Для этого добавим в него новый метод, аннотированный *@ExceptionHandler(EntityNotFoundException.class)*

@ExceptionHandler(EntityNotFoundException.class)

**protected** ResponseEntity&lt;Object&gt; handleEntityNotFoundEx(EntityNotFoundException ex, WebRequest request) {

ApiError apiError = **new** ApiError(HttpStatus.NOT\_FOUND, "entity not found ex", ex);

**return** **new** ResponseEntity&lt;&gt;(apiError, HttpStatus.NOT\_FOUND);

}

 Он будет вызываться, когда контроллер выбросит исключение *EntityNotFoundException* и формировать JSON-ответ.

Проверим, как это работает, сделав запрос:

GET http://localhost:8080/persons/

Получим ответ:

{

**"status":** "NOT\_FOUND",

**"message":** "entity not found ex"

}

Обратите внимание, что тип исключения в аргументе и аннотации должны соответствовать.

**Исключение *MethodArgumentTypeMismatchException***

Полезно знать еще исключение *MethodArgumentTypeMismatchException*, оно возникает, если тип аргумента неверный. Например, наш контроллер заточен на получение Person по id:

@GetMapping(value = "/{personId}")

**public** Person getPerson(@PathVariable("personId") **Long** personId) **throws** EntityNotFoundException {

**return** personRepository.getOne(personId);

}

А мы передаем не целое, а строковое значение id:

GET http://localhost:8080/persons/**mn**

Тут то и возникает исключение *MethodArgumentTypeMismatchException*. Давайте его обработаем:

@ExceptionHandler(MethodArgumentTypeMismatchException.class)

**protected** ResponseEntity<Object> handleMethodArgumentTypeMismatch(MethodArgumentTypeMismatchException ex,

WebRequest request) {

ApiError apiError = **new** ApiError(BAD\_REQUEST);

apiError.setMessage(**String**.format("The parameter '%s' of value '%s' could not be converted to type '%s'",

ex.getName(), ex.getValue(), ex.getRequiredType().getSimpleName()));

apiError.setDebugMessage(ex.getMessage());

**return** **new** ResponseEntity<>(apiError, HttpStatus.BAD\_REQUEST);

}

Проверим ответ сервера:

{

"status": "BAD\_REQUEST",

"message": "The parameter 'personId' of value 'mn' could not be converted to type 'Long'",

"debugMessage": "Failed to convert value of type 'java.lang.String' to required type 'java.lang.Long'; nested exception is java.lang.NumberFormatException: For input string: \"mn\""

}

**Исключение *handleNoHandlerFoundException***

Еще одно полезное исключение *handleNoHandlerFoundException* – оно возникает, если на данный запрос не найдено обработчика.

Например, сделаем запрос:

GET http://localhost:8080/pers

По данному адресу у нас нет контроллера, так что возникнет *handleNoHandlerFoundException*.  Добавим обработку исключения:

@Override

**protected** ResponseEntity<Object> handleNoHandlerFoundException(NoHandlerFoundException ex, HttpHeaders headers,

HttpStatus status, WebRequest request) {

**return** **new** ResponseEntity<Object>(**new** ApiError(status, "no handler found", ex), HttpStatus.NOT\_FOUND);

}

Только учтите, для того, чтобы обработчик вызывался, надо поменять файл *application.properties*:

spring.mvc.throw-exception-**if**-no-handler-found=true

spring.resources.add-mappings=false

Проверим ответ сервера:

{

**"status":** "NOT\_FOUND",

**"message":** "no handler found",

**"debugMessage":** "No handler found for GET /pers"

}

**Обработчик по умолчанию**

Этот обработчик будет ловить исключения, не пойманные предыдущими обработчиками (но не все):

@ExceptionHandler(Exception.class)

**protected** ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request) {

ApiError apiError = **new** ApiError(HttpStatus.INTERNAL\_SERVER\_ERROR, "prosto exception", ex);

**return** **new** ResponseEntity<>(apiError, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

**Заключение**

Мы рассмотрели, как переопределить формат  JSON-ответа, выдаваемого при возникновении исключения в Spring Boot, и как закодировать обработку исключений в едином классе. Код приведенного проекта доступен на [GitHub](https://github.com/myluckagain/sysout/tree/master/spring-exceptions).