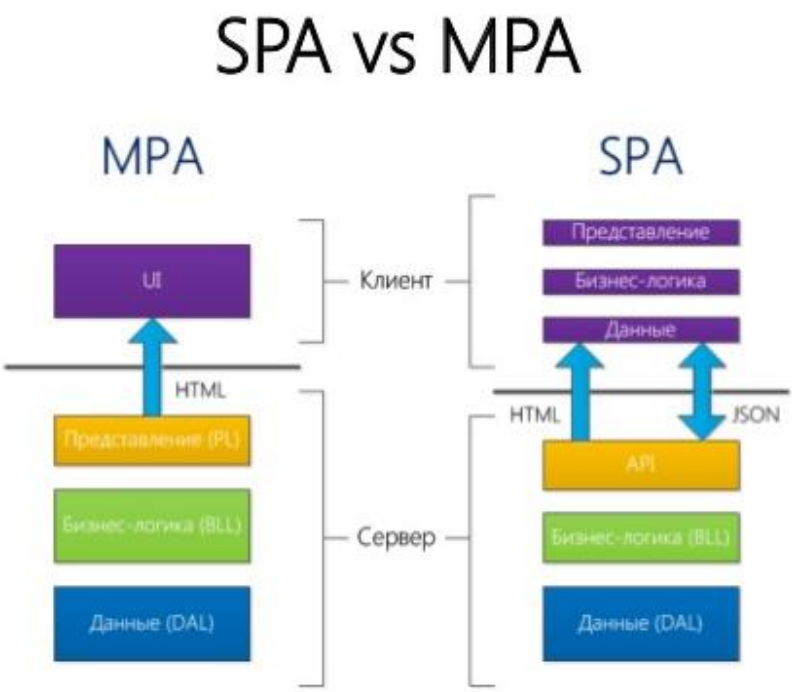
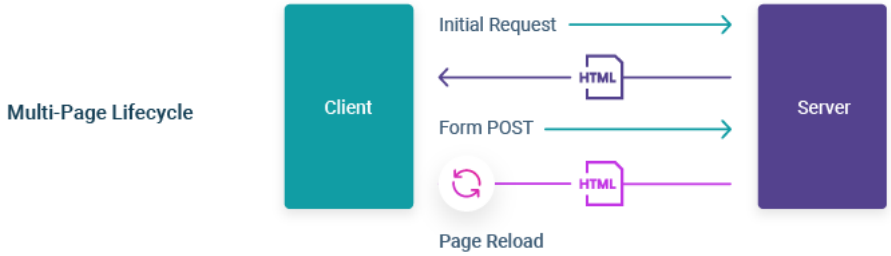
Задание №7 Создание REST API

7.1 Введение в REST



Чтобы обеспечить стабильную работу сложных веб приложений, желательно использовать технологии которые дадут наилучшую производительность и скорость. Существует два способа разработки веб приложений: одностраничные приложения (SPA) и многостраничные приложения(MPA). Single Page Application (SPA) и Multi Page Application (MPA)



Одностраничные приложения позволяют имитировать работу десктоп

приложений. Архитектура устроена таким образом, что при переходе на новую страницу, обновляется только часть контента. Таким образом, нет необходимости повторно загружать одни и те же элементы. Это очень удобно для разработчиков и пользователей. Для разработки SPA используется - javascript. Небольшое веб приложение можно сделать с библиотекой jQuery React.js, Angular.js, Vue.js и другие фреймворки/библиотеки. Их архитектура позволяет разрабатывать гибкие веб приложения.

SPA базируется на HTML/JS/CSS/JSON. SPA жестко делит клиентскую и серверную логику. SPA общается с сервером только чистыми данными. Разметка хранится на стороне клиента в шаблонах. Перезагрузки страницы не происходит.

Многостраничные приложения имеют более классическую архитектуру. Каждая страница отправляет запрос на сервер и полностью обновляет все данные. тратится производительность на отображение одних и тех же элементов. Соответственно это влияет на скорость и производительность. Многие разработчики, для того чтобы повысить скорость и уменьшить нагрузку, используют JavaScript/jQuery..

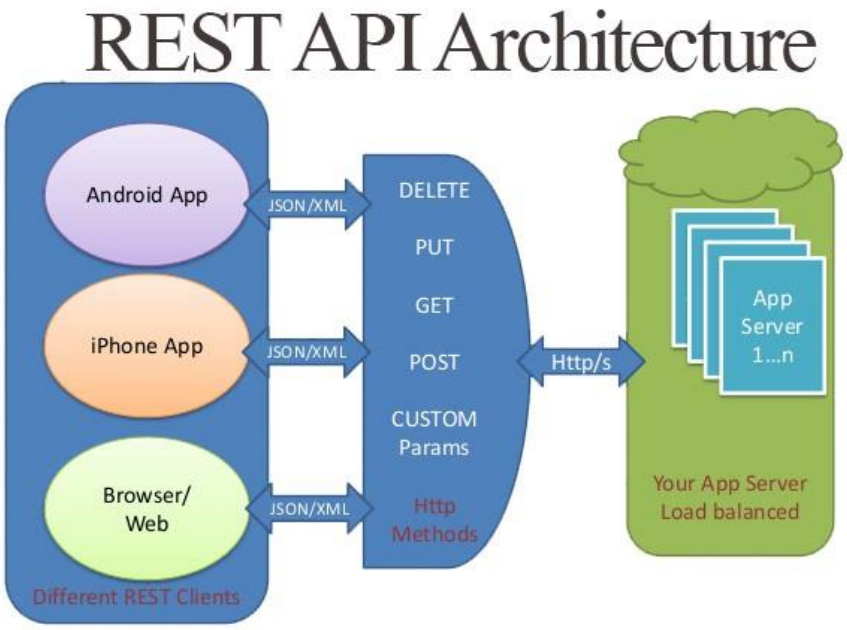
однако, архитектура MPA позволяет достаточно легко оптимизировать каждую страницу под поисковые системы.

Во вторых Как правило для разработки многостраничного приложения требуется меньший стек технологий. И существует Множество решений.

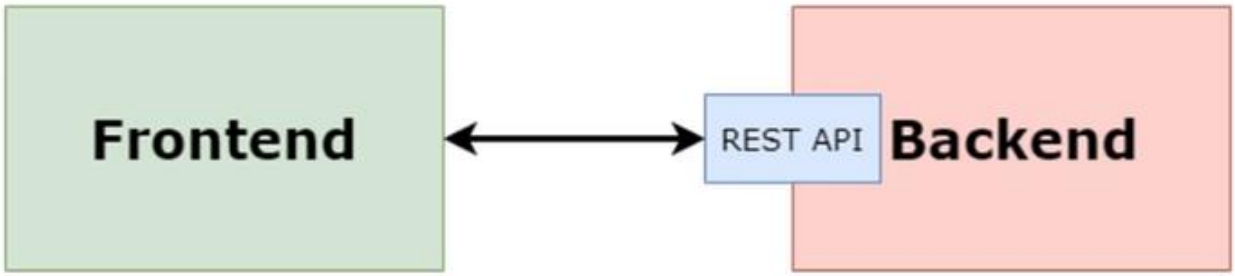
Из недостатков - Сложно разделить front-end и back-end. Как правило они очень тесно взаимодействуют друг с другом. Усложняется работа front-end и back-end разработчиков.

Каждая архитектура имеет свои преимущества и недостатки и хорошо подходит для определенного типа проекта. SPA отличается своей скорость . данная архитектура отлично подход для SaaS платформ, социальных сетей, закрытых сообществ, где поисковая оптимизация не имеет значения.

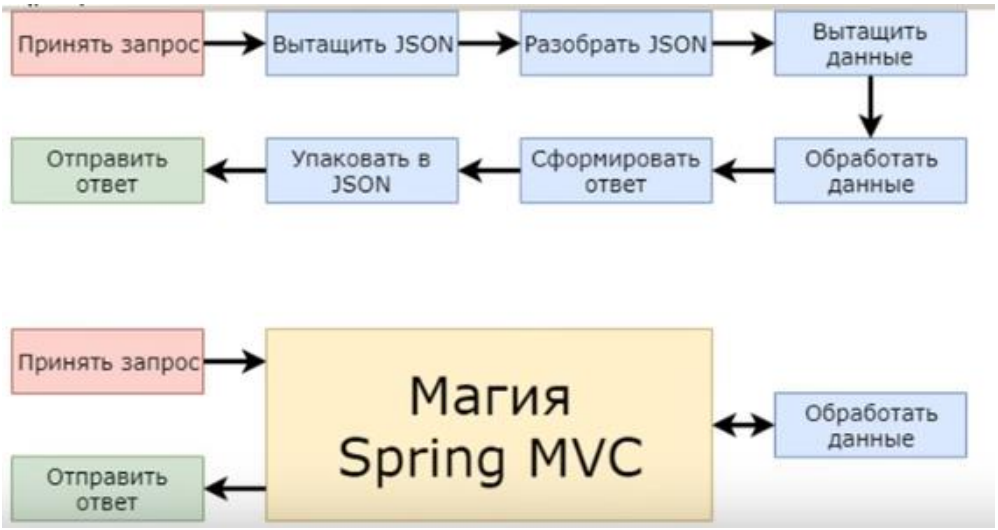
MPA больше подходит для интернет магазинов, бизнес сайтов, каталогов, маркетплейсов . Хорошо оптимизированная MPA имеет высокую скорость.



REST — это архитектурный стиль взаимодействия приложений в сети. Но в отличии от SOAP, у REST отсутствует какой-либо стандарт, а данные между клиентом и сервером могут предаваться в любом виде, будь то JSON, XML, YAML и т.д.



Spring Framework предоставляет богатый набор инструментов, упрощающий разработку REST API: инструменты для маршрутизации запросов, классы-кодеки для преобразования JSON/XML в объекты требуемых типов и т.д.



RESP оказался удобным не только для взаимодействия front и back но для взаимодействия между собой разный back end, что помогает делать модульные веб приложения и даже микро сервисные архитектуры. Можно получать данные из общедоступных web сервисов или использовать свои микро сервисы в нескольких проектах одновременно.

7.2 Entity – DTO конвертация

В результате разработки API нам понадобится обрабатывать преобразования, которые должны произойти между внутренними объектами приложения Spring и внешними DTO (объектами передачи данных), которые передаются обратно клиенту.

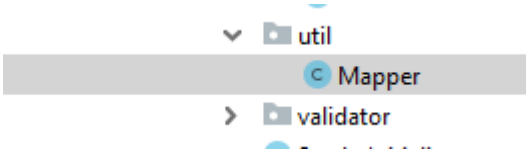
Модель Mapper

Начнем с введения основной библиотеки, которую мы собираемся использовать для выполнения преобразования Entity-DTO - *ModelMapper*.

Нам понадобится зависимость в pom.xml:

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.3.5</version>
</dependency>
```

Создадим пакет util с классом Mapper:



Давайте посмотрим на операции уровня serivce, которые работают с сущностями (не с DTO).

Теперь посмотрим на уровень выше сервисов - уровень контроллера. Именно здесь на самом деле происходит конверсия.

Логика преобразования проста - используем map API и преобразуем данные, не записывая ни одной строки логики преобразования:

```
private static ModelMapper modelMapper = new ModelMapper();
modelMapper.map(source, targetClass);
```

Когда вызывается метод map, типы источника и назначения анализируются, чтобы определить, какие свойства неявно совпадают в соответствии со стратегией сопоставления и другой конфигурацией. Затем данные отображаются в соответствии с этими совпадениями. Даже когда исходный и целевой объекты и их свойства различаются, *ModelMapper* сделает все возможное, чтобы определить разумные соответствия между свойствами.

Конфигурация по умолчанию использует стандартную стратегию сопоставления для общедоступных методов источника и назначения, которые названы согласно соглашению JavaBeans.Применяются следующие правила:

- Токены могут быть сопоставлены в любом порядке;
- Все токены имени свойства назначения должны совпадать;
- Все имена свойств источника должны иметь хотя бы один совпадающий токен.

Стандартная стратегия сопоставления хотя и не является точной, она идеально подходит для большинства сценариев.

Определим конфигурацию в статическом конструкторе:

```
public class Mapper {

    private static ModelMapper modelMapper;

    static {
        modelMapper = new ModelMapper();
        modelMapper
                .getConfiguration()
                .setFieldMatchingEnabled(true)
                .setSkipNullEnabled(false)
                .setMatchingStrategy(MatchingStrategies.STANDARD);
    }
```

Более подробно про настройку конфигурации можно посмотреть здесь http://modelmapper.org/user-manual/configuration/

Таким образом, у нас получилось:

```
import org.modelmapper.ModelMapper;
import org.modelmapper.convention.MatchingStrategies;
import java.util.Collection;
import java.util.List;
import java.util.stream.Collectors;

    public class Mapper {

        private static ModelMapper modelMapper;

        static {
```

```
            modelMapper = new ModelMapper();
            modelMapper
                    .getConfiguration()
                    .setFieldMatchingEnabled(true)
                    .setSkipNullEnabled(false)
                    .setMatchingStrategy(MatchingStrategies.STANDARD);
        }

        public static <S, T> T map(S source, Class<T> targetClass) {
            return modelMapper.map(source, targetClass);
        }

        public static <S, T> List<T> mapAll(Collection<? extends S>
sourceList, Class<T> targetClass) {
            return sourceList.stream()
                    .map(e -> map(e, targetClass))
                    .collect(Collectors.toList());
        }

    }
```

map – для отображения одного типа, mapAll – для коллекции.

7.3 REST

REST (сокращение от Representational State Transfer — «передача состояния представления»)— это архитектурный стиль взаимодействия приложений. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры. REST является альтернативой RPC (remote procedure call – вызов удаленной процедуры), JMS.

Для веб-служб, построенных с учётом REST (то есть не нарушающих накладываемых им ограничений), применяют термин «RESTful».

В отличие от веб-сервисов (веб-служб) на основе SOAP, не существует «официального» стандарта для RESTful веб-API. Дело в том, что REST является архитектурным стилем, в то время как SOAP является протоколом. SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) — протокол обмена структурированными сообщениями в распределённой вычислительной среде.

Требования к архитектуре REST

Существует шесть обязательных ограничений для построения распределённых REST-приложений. Выполнение этих ограничительных требований обязательно для REST-систем.

Обязательными условиями-ограничениями являются:

1. Модель клиент-сервер

2. Отсутствие состояния (*не сохраняет контекст клиента на сервере между запросами*) : протокол взаимодействия между клиентом и сервером требует соблюдения следующего условия: в период между запросами клиента никакая информация о состоянии клиента на сервере не хранится (Stateless protocol). Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Состояние сессии при этом сохраняется на стороне клиента. Информация о состоянии сессии может быть передана сервером какому-либо другому сервису (например, в службу базы данных) для поддержания устойчивого состояния, например, на период установления аутентификации. Клиент инициирует отправку запросов, когда он готов (возникает необходимость) перейти в новое состояние.

3. Кэширование: клиенты, а также промежуточные узлы, могут выполнять кэширование ответов сервера. Ответы сервера, в свою очередь, должны иметь явное или неявное обозначение как кэшируемые или некэшируемые с целью предотвращения получения клиентами устаревших или неверных данных в ответ на последующие запросы.

4. Единообразие интерфейса : наличие унифицированного интерфейса является фундаментальным требованием дизайна REST-сервисов. Унифицированные интерфейсы позволяют каждому из сервисов развиваться независимо.

К унифицированным интерфейсам предъявляются следующие четыре ограничительных условия:

- Идентификация ресурсов - Все ресурсы идентифицируются в запросах, например, с использованием URI в интернет-системах. Ресурсы концептуально отделены от представлений, которые возвращаются клиентам. Например, сервер может отсылать данные из базы данных в виде HTML, XML или JSON, ни один из которых не является типом хранения внутри сервера.
- Манипуляция ресурсами через представление - Если клиент хранит представление ресурса, включая метаданные — он обладает достаточной информацией для модификации или удаления ресурса.
- «Самоописываемые» сообщения - Каждое сообщение содержит достаточно информации, чтобы понять, каким образом его обрабатывать. К примеру, обработчик сообщения (parser), необходимый для извлечения данных, может быть указан в списке MIME-типов.
- Гипермедиа как средство изменения состояния приложения (HATEOAS) - Клиенты изменяют состояние системы только через действия, которые динамически определены в гипермедиа на сервере (к примеру, гиперссылки в гипертексте). Исключая простые точки входа в приложение, клиент не может предположить, что доступна какая-то операция над каким-то

ресурсом, если не получил информацию об этом в предыдущих запросах к серверу.

5. Слои - Клиент обычно не способен точно определить, взаимодействует он напрямую с сервером или же с промежуточным узлом, в связи с иерархической структурой сетей (подразумевая, что такая структура образует слои).

6. Код по требованию (необязательное ограничение) - REST может позволить расширить функциональность клиента за счёт загрузки кода с сервера в виде апплетов или сценариев.

HTTP-методы REST

В отличии от классических веб-приложений, в которых используются только методы GET и POST, в REST API используются практически все HTTP-методы, например:

GET — для получения объекта или списка объектов

HEAD — проверка существования объекта

OPTIONS — проверка доступных методов для указанного пути

POST — для создания нового объекта

PUT — для полного изменения объекта или помещения объекта в список

PATCH — для частичного изменения объекта

DELETE — для удаления объекта

Стоит помнить, что хорошей практикой является использование **GET** только для запросов, которые не изменяют состояние объекта. Если запрос должен изменить состояние объекта, то должен использоваться соответствующий HTTP-метод, хотя бы POST.

Для маршрутизации запросов будем использовать аннотации **@RequestMapping** с указанием HTTP-метода при помощи свойства *method* или более простые аннотации вроде **@GetMapping**, **@PostMapping**, **@DeleteMapping** и т.д.

Платформа Spring поддерживает два способа создания сервисов RESTful:

- На основе MVC с ModelAndView
- Используя конвертеры HTTP-сообщений

Подход ModelAndView более старый и лучше документированный, но тяжелый по конфигурации. Он пытается внедрить парадигму REST в старую модель.

Новый подход, основанный на HttpMessageConverter и аннотациях, более легкий и простой в реализации. Конфигурация минимальна, и она обеспечивает значения по умолчанию.

Аннотирование

Аннотация @EnableWebMvc делает несколько полезных вещей - в частности, в случае REST она обнаруживает существование Jackson и JAXB и автоматически создает и регистрирует конвертеры JSON и XML по умолчанию.

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
```

Когда требуется более сложная конфигурация, удалите аннотацию и расширьте WebMvcConfigurationSupport напрямую.

Так как мы используем аннотацию @SpringBootApplication, то аннотация @EnableWebMvc добавляется автоматически с автоконфигурацией по умолчанию.

```
@SpringBootApplication
public class SpringProject1Application {
```

Можно добавить функциональность MVC в эту конфигурацию, реализовав интерфейс WebMvcConfigurer в аннотированном @Configuration классе (см выше). Можно использовать экземпляр WebMvcRegistrationsAdapter для предоставления собственных реализаций RequestMappingHandlerMapping, RequestMappingHandlerAdapter или ExceptionHandlerExceptionResolver.

Т.е. можно отказаться от функций MVC Spring Boot и объявить пользовательскую конфигурацию с помощью аннотации @EnableWebMvc.

Контроллер

@RestController является центральным артефактом для API RESTful. Изменим проект и перепишем контроллер PersonController, который моделирует простой ресурс REST.

```
@RestController
@RequestMapping
class PersonController {


    private  final PersonService personService;

    @Value("${welcome.message}")
    private String message;

    @Value("${error.message}")
    private String errorMessage;
```

```
    @Autowired
    public PersonController(PersonService personService) {
        this.personService = personService;
    }


//, produces = { "application/json" , "application/xml"}
    @GetMapping(value = {"/personList"})
    public List<PersonDto> personList() {
        return Mapper.mapAll(personService.getAllPerson(), PersonDto.class);
    }

    @GetMapping(value = {"/personList/{id}"})
    public PersonDto findById(@PathVariable("id") Long id) throws
ResourceNotFoundException {
        return Mapper.map(personService.getById(id), PersonDto.class);
    }


    @PutMapping(value = "/editPerson/{id}")
    @ResponseStatus(HttpStatus.OK)
    public void editPerson(@PathVariable("id") Long id, @Valid @RequestBody
PersonDto persondto) throws ResourceNotFoundException {
        personService.editPerson(Mapper.map(persondto, Person.class),id);

    }

    @PostMapping("/addPerson")
    public void  savePerson( @Valid @RequestBody NewPersonDto personDto) {
        personService.addNewPerson(Mapper.map(personDto, Person.class));
    }


    @DeleteMapping(value = "/{id}")
    @ResponseStatus(HttpStatus.OK)
    public void deletePerson(@PathVariable("id") Long id) throws
ResourceNotFoundException {
         personService.deletePerson(personService.getById(id));
    }
}
```

Обратите внимание реализация контроллера не является общедоступной. Обычно контроллер является последним в цепочке зависимостей. Он получает HTTP-запросы от фронт-контроллера Spring (DispatcherServlet) и просто передает их на сервисный уровень. Если нет варианта использования, когда контроллер должен вводиться или манипулировать с помощью прямой ссылки, тогда лучше не объявлять его публичным.

Отображения запросов просты. Как и в любом контроллере, фактическое значение отображения, а также метод HTTP определяют целевой метод для запроса.

@RequestBody будет привязывать параметры метода к телу HTTP-запроса, тогда как **@ResponseBody** делает то же самое для ответа и типа возврата.

Контекст приложения создаст и зарегистрирует экземпляр класса, поскольку указана аннотация **@RestController**. **@RestController** - это сокращение, включающее в класс аннотации **@ResponseBody и @Controller**.

Кроме этого аннотация подсказывает, что возвращаемые методами значения являются телом ответов на запросы. Они также гарантируют, что ресурс будет распакован с использованием правильного конвертера HTTP. Будет выполнено согласование содержимого, чтобы выбрать, какой из активных преобразователей будет использоваться, в основном на основе заголовка Accept, хотя для определения представления могут также использоваться другие заголовки HTTP.

Клиент может установить Accept в application/json, если он запрашивает ответ в JSON. И наоборот, когда отправляются данные, установленный Content-Type в application/xml говорит клиенту, что данные были отправлены в XML форме.

Отображение кодов ответа HTTP

Коды состояния ответа HTTP являются одной из наиболее важных частей службы REST.

Не сопоставленные запросы

Если Spring MVC получает запрос, который не имеет сопоставления, он считает, что запрос не разрешен, и возвращает **405 METHOD NOT ALLOWED** обратно клиенту.

Также рекомендуется включать заголовок *Allow HTTP* при возврате 405 клиенту, чтобы указать, какие операции разрешены. Это стандартное поведение Spring MVC и не требует дополнительной настройки.

Сопоставленные запросы

Для любого запроса, который имеет сопоставление, Spring MVC считает запрос действительным и отвечает **200 OK**, если другой код состояния не указан.

Поэтому контроллер объявляет разные **@ResponseStatus** для действий создания, обновления и удаления, но не для получения, что должно действительно возвращать значение по умолчанию **200 OK**.

Давайте посмотрим метод контроллера

```
@PostMapping(value = {"/addPerson"})
public void savePerson(@Valid @RequestBody NewPersonDto personDto) {
    personService.addNewPerson(Mapper.map(personDto, Person.class));
     }
```

Метод создания персоны по ее идентификатору должен быть вызван при POST-запросе по пути */addPerson*; он должен обратиться к сервису и вернуть персону с HTTP-кодом *Created*. Добавим код статуса

```
@PostMapping("/addPerson")
@ResponseStatus(HttpStatus.CREATED)
public void  savePerson( @Valid @RequestBody NewPersonDto personDto) {
```

```
    personService.addNewPerson(Mapper.map(personDto, Person.class));
}
```

Ошибки клиента

В случае ошибки клиента настраиваемые исключения определяются и сопоставляются с соответствующими кодами ошибок.

Генерация исключения из любого веб-уровня гарантирует, что Spring отобразит соответствующий код состояния в ответе HTTP:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException {
    //
}
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    //
}
```

Эти исключения являются частью REST API и должны использоваться на соответствующих уровнях (лучше слой DAO/DAL) но, контроллер не должен использовать исключения напрямую.

Также обратите внимание, что это не проверяемые исключения, а исключения времени выполнения – что соответствует практике и идиомам Spring.

Использование @ExceptionHandler

Другой вариант сопоставления пользовательских исключений с конкретными кодами состояния - использовать аннотацию **@ExceptionHandler** в контроллере. Проблема этого подхода заключается в том, что аннотация применяется только к контроллеру, в котором он определен. Это означает, что нужно объявлять в каждом контроллере индивидуально.

В Spring и Spring Boot существует больше способов обработки ошибок, которые обеспечивают большую гибкость.

Формат данных

Если нужно получить ресурсы в формате JSON, Spring Boot предоставляет поддержку для различных библиотек: Jackson, Gson и JSON-B.

Автоконфигурирование выполняется путем простого включения любой из библиотек сопоставления в путь к классам.

Spring Boot уже включает все необходимые артефакты в проект:

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

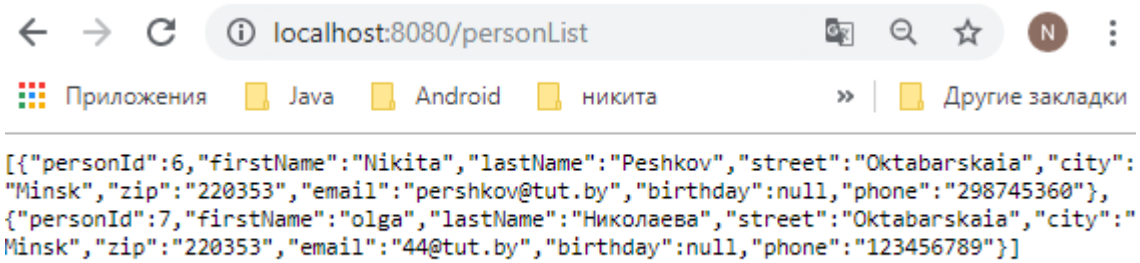
Jackson используется по умолчанию.

Если нужно сериализовать ресурсы в формате XML, нужно будет добавить расширение Jackson XML (jackson-dataformat-xml) к зависимостям или использовать реализацию JAXB (предоставляется по умолчанию в JDK) с помощью аннотации **@XmlRootElement** на нашем ресурсе.

Как это работает. Все методы контроллера возвращают простые POJO List <PersonDto> и PersonDto. Когда HTTP запрос приходит с указанным заголовком Accept, Spring MVC перебирает настройки HttpMessageConverter до тех пор, пока не найдет того, кто сможет конвертировать из типов POJO доменной модели в указанный тип заголовка Accept. Spring Boot автоматически инициализирует HttpMessageConverter, который может конвертировать общий для всех тип Object в JSON, при отсутствии каких-либо других указанных конвертеров. HttpMessageConverter работает в обоих направлениях: тела входящих запросов конвертируются в Java объекты, а Java объекты конвертируются в тела HTTP ответов.

7.4 Проверка REST API

Для проверки можно использовать броузер



Или утилиту curl с терминала:

```
C:\NATALLIA\Training_center_IBA\Projects>curl -v  http://localhost:8080/personList
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /personList HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 14 Jul 2019 19:06:49 GMT
<
[{"personId":6,"firstName":"Nikita","lastName":"Peshkov","street":"Oktabarskaia","city":"Minsk","zip":"220353","email":"p
98745360"},{"personId":7,"firstName":"olga","lastName":"[illegible]","street":"Oktabarskaia","city":"Minsk","zip":'
11,"phone":"123456789"}]* Connection #0 to host localhost left intact
```

```
C:\NATALLIA\Training_center_IBA\Projects>curl http://localhost:8080/personList/6
{"personId":null,"firstName":null,"lastName":null,"street":null,"city":null,"zip":null,"email":null,"birthday":null,"pho
C:\NATALLIA\Training_center_IBA\Projects>
```

Команды curl можно посмотреть по ссылке

https://curl.haxx.se/docs/history.html

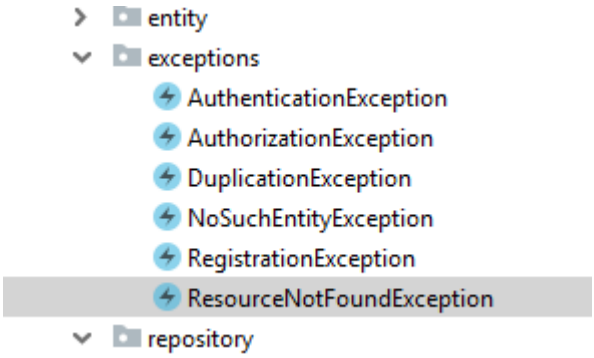
Запросим данные которых нет

```
C:\NATALLIA\Training_center_IBA\Projects>curl -v  http://localhost:8080/personList/11
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /personList/11 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 14 Jul 2019 19:09:14 GMT
<
{"personId":null,"firstName":null,"lastName":null,"street":null,"city":null,"zip":null,"email":null,"birthday":null,"pl
left intact
```

Возврат не правильный, вместо код ошибки нет и был возвращен пустой объект поэтому сделаем обработку ошибок

Ошибки могут быть совершенно разными, и клиенту нашего сервиса придется ставить кучу условных операторов и исследовать, что у нас пошло не так и как это можно поправить. Как раз для таких случаев и была придумана аннотация ResponseStatus.

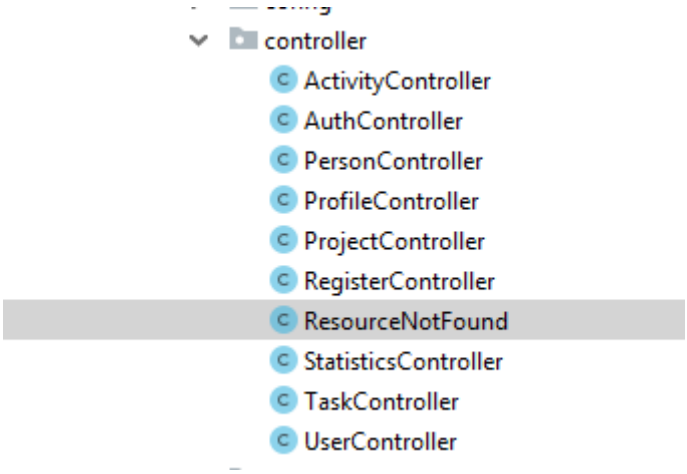
В проект добавьте новый класс исключений ResourceNotFoundExcetion:



Со следующим содержимым

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends Exception {
    public ResourceNotFoundException() {
    }

    public ResourceNotFoundException(String message) {
        super(message);
    }

    public ResourceNotFoundException(long id) {
        super("Resource not found " + Long.toString(id));
    }
}
```

Определим еще один контроллер, который будет обрабатывать ошибки в случае отсутствия ресурса ResourceBotFound:




```
import by.patsei.springproject1.exceptions.ResourceNotFoundException;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
public class ResourceNotFound {

        @ResponseBody
        @ExceptionHandler(ResourceNotFoundException.class)
```

```
        @ResponseStatus(HttpStatus.NOT_FOUND)
        String recourceNotFoundHandler  (ResourceNotFoundException ex) {
            return ex.getMessage();
        }
    }
```

Здесь используется

- Аннотация ExceptionHandler. - для обработки собственных и каких-то специфичных исключений. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html
- Аннотация ControllerAdvice. Данная аннотация дает «совет» группе контроллеров по определенным событиям. В нашем случае — это обработка ошибок. По умолчанию применяется ко всем контроллерам, но в параметрах можно указать определенную группу.

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html

- Класс ResponseEntityExceptionHandler. Данный класс занимается обработкой ошибок. У него куча методов, название которых построенно по принципу handle + название исключения. Если мы хотим обработать какое-то базовое исключение, то наследуемся от этого класса и переопределяем нужный метод.

Изменяем метод в PersonController:

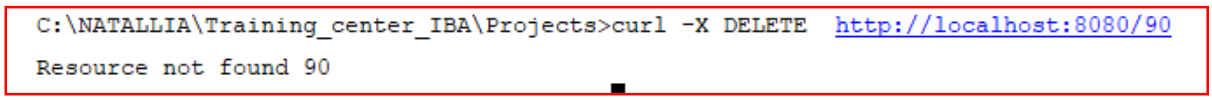
```
@GetMapping(value = {"/personList/{id}"})
public PersonDto findById(@PathVariable("id") Long id) throws
ResourceNotFoundException {
    return Mapper.map(personService.getById(id)
            .orElseThrow(() -> new ResourceNotFoundException(id)),
PersonDto.class);
}
```

Проверяем еще раз

Допишем генерацию исключений для удаления

```
@DeleteMapping(value = "/{id}")
    @ResponseStatus(HttpStatus.OK)

    public void deletePerson(@PathVariable("id") Long id) throws
ResourceNotFoundException {
        personService.deletePerson(personService.getById(id)
                .orElseThrow(() -> new ResourceNotFoundException(id)));
    }
}
```

Наберите запрос удаления



Теперь проверим добавление:

curl --header "Content-Type: application/json" --header "Accept: application/json" --request POST --data "{\"firstName\":\"Oleg\", \"lastName\":\"Link\",\"street\":\"Oktabarskaia\",\"city\":\"Minsk\",\"zip\":\"220353\",\"email\":\"link@tut.by\",\"birthday\":null,\"phone\":\"298745360\" }" http://localhost:8080/addPerson

curl http://localhost:8080/personList/11

```
C:\NATALLIA\Training_center_IBA\Projects>curl http://localhost:8080/personList/11
{"personId":11,"firstName":"Oleg","lastName":"Link","street":"Oktabarskaia","city":"Minsk","zip":"220353","email":"link@tut.by","birthday":null,"phone":
"298745360"}
```

Запись была добавлена.

Проверим валидацию. Допустим, сделаем имя пустым и повторим запрос

```
C:\NATALLIA\Training_center_IBA\Projects>curl --header "Content-Type: application/json" --header "Accept: application/json"  --request POST    --data "{
\"firstName\":\"\", \"lastName\":\"Link\",\"street\":\"Oktabarskaia\",\"city\":\"Minsk\",\"zip\":\"220353\",\"email\":\"link@tut.by\",\"birthday\":null,
\"phone\":\"298745360\" }"    http://localhost:8080/addPerson
{"timestamp":"2019-07-15T06:14:32.021+0000","status":400,"error":"Bad Request","errors":[{"codes":["Size.newPersonDto.firstName","Size.firstName","Size.
java.lang.String","Size"],"arguments":[{"codes":["newPersonDto.firstName","firstName"],"arguments":null,"defaultMessage":"firstName","code":"firstName"}
,2147483647,3],"defaultMessage":"\"Name must be at least 3 characters long\"","objectName":"newPersonDto","field":"firstName","rejectedValue":"","bindin
gFailure":false,"code":"Size"}],"message":"Validation failed for object='newPersonDto'. Error count: 1","trace":"org.springframework.web.bind.MethodArgu
```

Осталось проверить редактирование

Добавим генерацию исключения в контроллер

```
@PutMapping(value = "/editPerson/{id}")
@ResponseStatus(HttpStatus.OK)
public void editPerson(@PathVariable("id") Long id, @Valid @RequestBody
PersonDto persondto) throws ResourceNotFoundException {
    personService.getById(id).orElseThrow(() -> new
ResourceNotFoundException(id));

    personService.editPerson(Mapper.map(persondto, Person.class),id);

}
```

И поменяем редактирование в сервисе:

```
@Service
@Transactional
public class PersonServiceImpl implements PersonService{

    private final PersonRepository personRepository;


    @Autowired
    public PersonServiceImpl(PersonRepository personRepository) {

        this.personRepository = personRepository;
    }
...
    public  void editPerson(Person person, Long id){
        person.setId(id);
         personRepository.save(person);
    }
```

Введем команду (меняем имя на ANNA):

curl -v --header "Content-Type: application/json" --header "Accept: application/json" --request PUT   --data "{ \"personId\":10, \"firstName\":\"ANNA\",

\"lastName\":\"Link\",\"street\":\"Oktabarskaia\",\"city\":\"Minsk\",\"zip\":\"220353\",\"email\":\"link@tut.by\",\"birthday\":null,\"phone\":\"298745360\" }" http://localhost:8080/editPerson/10

```
C:\NATALLIA\Training_center_IBA\Projects>curl -v --header "Content-Type: application/json" --header "Accept: application/json"  --request PUT    --da
"{ \"personId\":10, \"firstName\":\"ANNA\", \"lastName\":\"Link\",\"street\":\"Oktabarskaia\",\"city\":\"Minsk\",\"zip\":\"220353\",\"email\":\"link@
.by\",\"birthday\":null,\"phone\":\"298745360\" }"    http://localhost:8080/editPerson/10
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> PUT /editPerson/10 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.55.1
> Content-Type: application/json
> Accept: application/json
> Content-Length: 168
>
* upload completely sent off: 168 out of 168 bytes
< HTTP/1.1 200
< Content-Length: 0
< Date: Mon, 15 Jul 2019 06:47:53 GMT
<
* Connection #0 to host localhost left intact
```

Редактирование прошло.

Перенос исключений из уровня контроллера на уровень сервисов.

Как уже говорили ранее контроллер не должен генерировать исключения. Поэтому перенесем их на уровень ниже:

```
public interface PersonService {
    List<Person> getAllPerson();
    void addNewPerson(Person person);
    void deletePerson(Person person );
    void editPerson(Person person, Long id);
    Person getById(long id) throws ResourceNotFoundException;


}


@Service
@Transactional
public class PersonServiceImpl implements PersonService{

    private final PersonRepository personRepository;


    @Autowired
    public PersonServiceImpl(PersonRepository personRepository) {

        this.personRepository = personRepository;
    }

    public List<Person> getAllPerson() {
        return personRepository.findAll();
    }

    public  void addNewPerson(Person person){
        personRepository.save(person);
```

```
    }
    public  void deletePerson(Person person ){
        personRepository.delete(person);

    }

    public  void editPerson(Person person, Long id){
        person.setId(id);
         personRepository.save(person);
    }
    public Person getById(long id) throws ResourceNotFoundException {
            return personRepository.findAllById(id)
                    .orElseThrow(() -> new ResourceNotFoundException(id));

    }
}
```

И из контроллера уберем дублирующийся код.

```
@RestController
@RequestMapping
class PersonController {


    private  final PersonService personService;

    @Value("${welcome.message}")
    private String message;

    @Value("${error.message}")
    private String errorMessage;

    @Autowired
    public PersonController(PersonService personService) {
        this.personService = personService;
      //  this.personRepository = personRepository;
    }


//, produces = { "application/json" , "application/xml"}
    @GetMapping(value = {"/personList"})
    public List<PersonDto> personList() {
        return Mapper.mapAll(personService.getAllPerson(), PersonDto.class);
    }

    @GetMapping(value = {"/personList/{id}"})
    public PersonDto findById(@PathVariable("id") Long id) throws
ResourceNotFoundException {
        return Mapper.map(personService.getById(id), PersonDto.class);
    }


    @PutMapping(value = "/editPerson/{id}")
    @ResponseStatus(HttpStatus.OK)
    public void editPerson(@PathVariable("id") Long id, @Valid @RequestBody
PersonDto persondto) throws ResourceNotFoundException {
        personService.getById(id);

        personService.editPerson(Mapper.map(persondto, Person.class),id);

    }
```

```
    @PostMapping("/addPerson")
    @ResponseStatus(HttpStatus.CREATED)
    public void  savePerson( @Valid @RequestBody NewPersonDto personDto) {
        personService.addNewPerson(Mapper.map(personDto, Person.class));
    }


    @DeleteMapping(value = "/{id}")
    @ResponseStatus(HttpStatus.OK)
    public void deletePerson(@PathVariable("id") Long id) throws
ResourceNotFoundException {
         personService.deletePerson(personService.getById(id));
    }
}
```

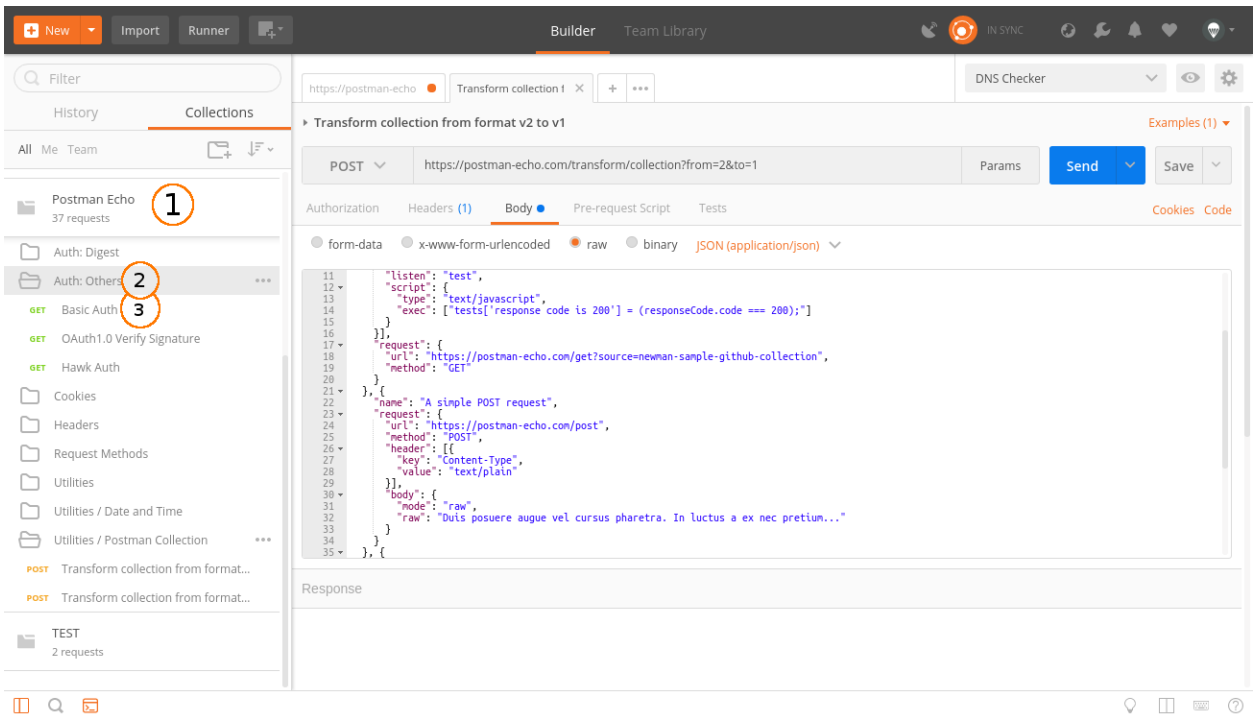
Таким образом получился простой сервис. Данные можно доставать AJAX в html.

```
<script
src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"></script>
<script type="text/javascript">
    var prefix = '/personList';

    var RestGet = function() {
        $.ajax({
            type: 'GET',
            url:  prefix + '/' + 10,
            dataType: 'json',
            async: true,
            success: function(result) {
                alert('сообщение: ' + result.message);
            },
            error: function(jqXHR, textStatus, errorThrown) {
                alert(jqXHR.status + ' ' + jqXHR.responseText);
            }
        });

    }
</script>
```

Для разработанных endpoint надо написать тесты.

7.5 POSTMAN

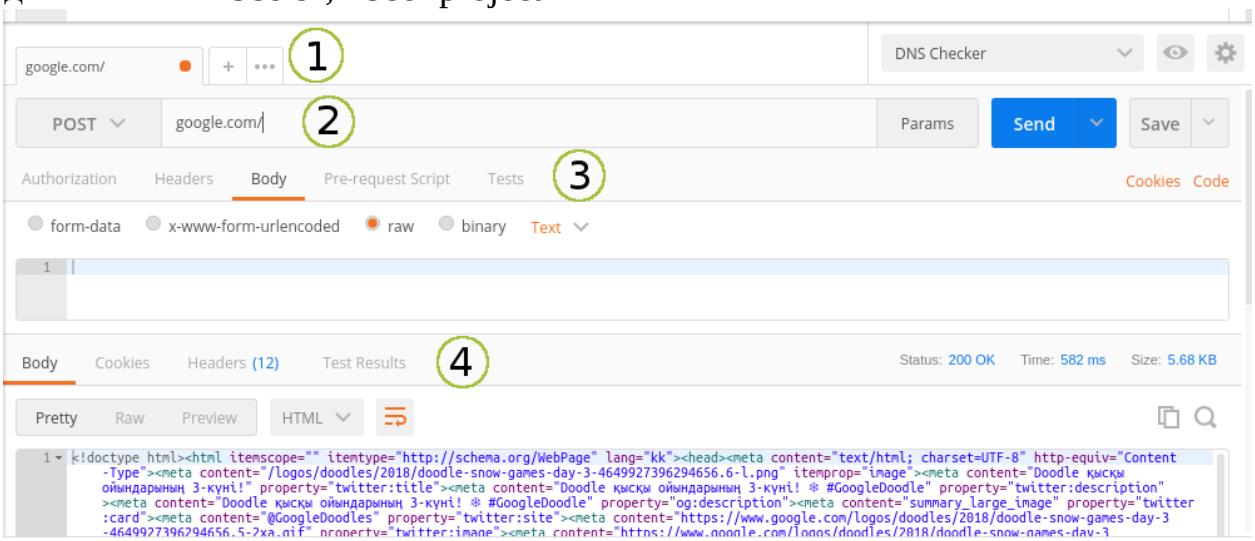
В процессе тестирования необходимо использовать тестовые феймворки. Это требует времени на написание. Однако в вначале можно выполнить ручное тестирование. Для этого подходит POSTMAN.



Здесь на рисунке *1 — коллекция, 2 — папка, 3 — запрос*

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. Коллекция может иметь свои скрипты и переменные

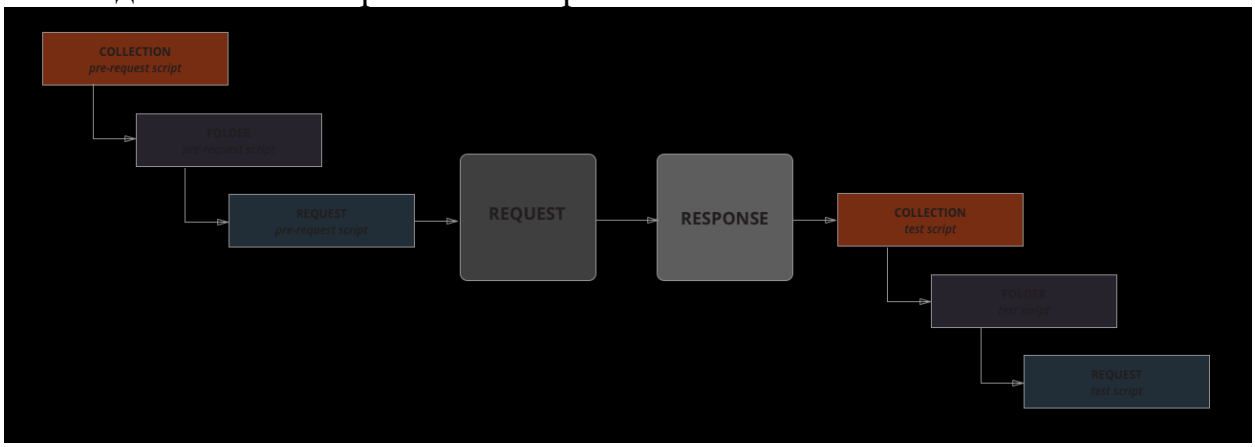
Папка — используется для объединения запросов в одну группу внутри коллекции. К примеру, вы можете создать папку для первой версии своего API — "v1", а внутри сгруппировать запросы по смыслу выполняемых действий — «Users", "User project" и т. п



1 — вкладки с запросами, 2 — URL и метод, 3 — параметры запроса, 4 — параметры ответа

Запрос — основная составляющая коллекции,. Запрос создается в конструкторе. Конструктор запросов это главное пространство, с которым

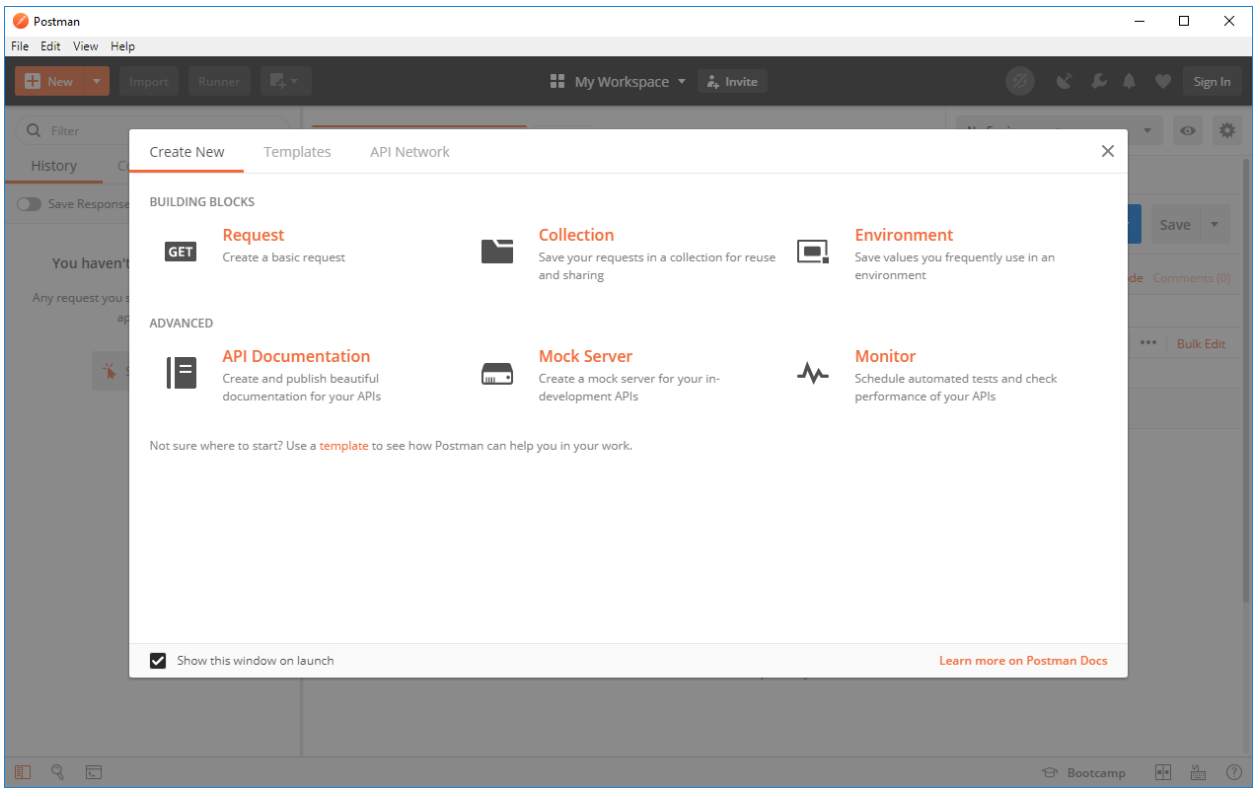
придётся работать. Postman умеет выполнять запросы с помощью всех стандартных HTTP методов, все параметры запроса.
Обратите внимание на вкладки "Pre-request Script" и "Tests" среди параметров запроса. Они позволяют добавить скрипты перед выполнением запроса и после
"Pre-request Script" используется для проведения необходимых операций перед запросом, например, можно сделать запрос к другой системе и использовать результат его выполнения в основном запросе. "Tests" используется для написания тестов, проверки результатов, и при необходимости их сохранения в переменные.



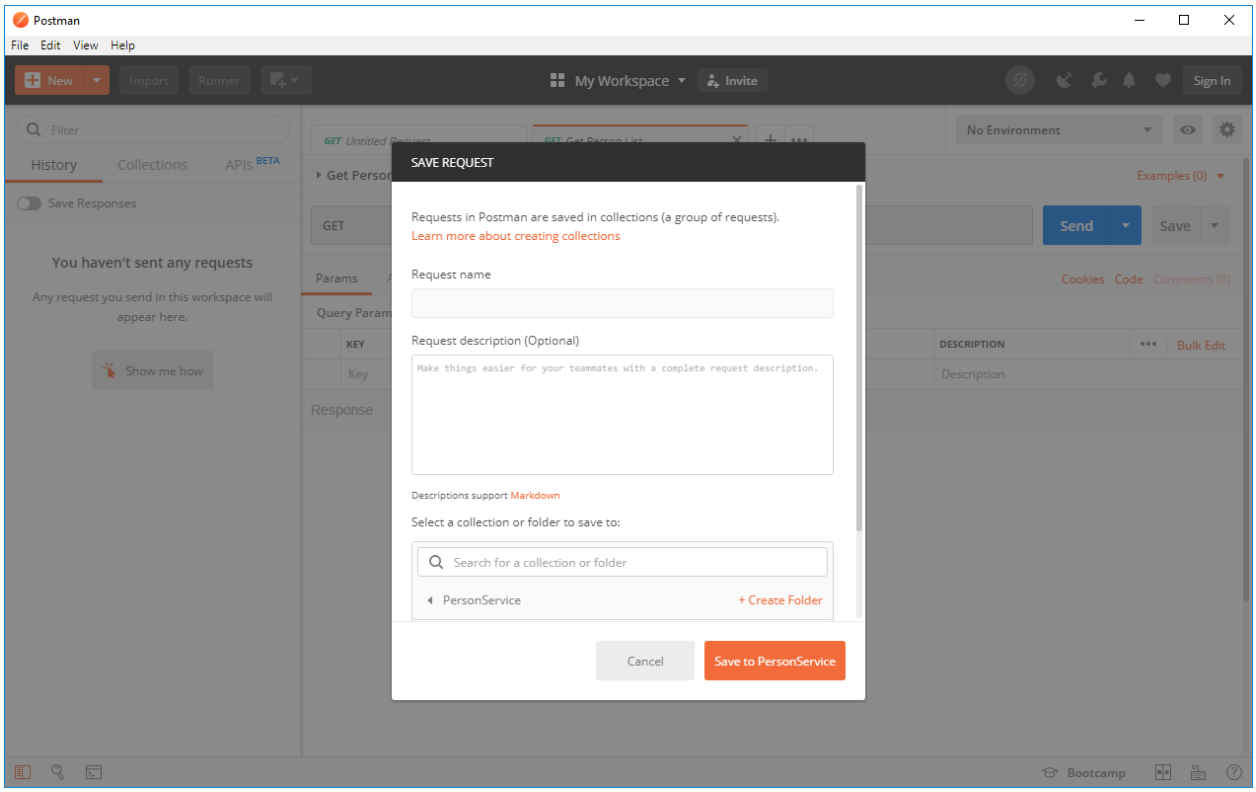
Помимо скриптов на уровне запроса, мы можем создавать скрипты на уровне папки, и, даже, на уровне коллекции. Они называются также — "Pre-request Script" и "Tests", но их отличие в том, что они будут выполняться перед каждым и после каждого запроса в папке, или, как вы могли догадаться, во всей коллекции.

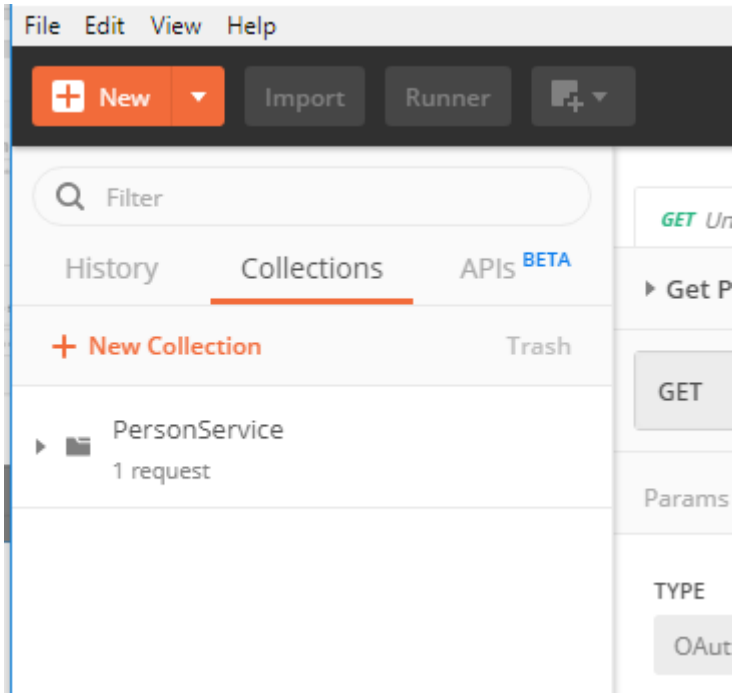
Основное предназначение приложения — создание коллекций с запросами к вашему API. Любой разработчик или тестировщик, открыв коллекцию, сможет с лёгкостью разобраться в работе вашего сервиса. Postman позволяет проектировать дизайн API и создавать на его основе Mock-сервер. Реализацию сервера и клиента можно запустить одновременно. Есть инструменты для автоматического документирования по описаниям из ваших коллекций. Еще возможность создания коллекций для мониторинга сервисов.

Скачаем и установим приложение
https://www.getpostman.com/downloads/

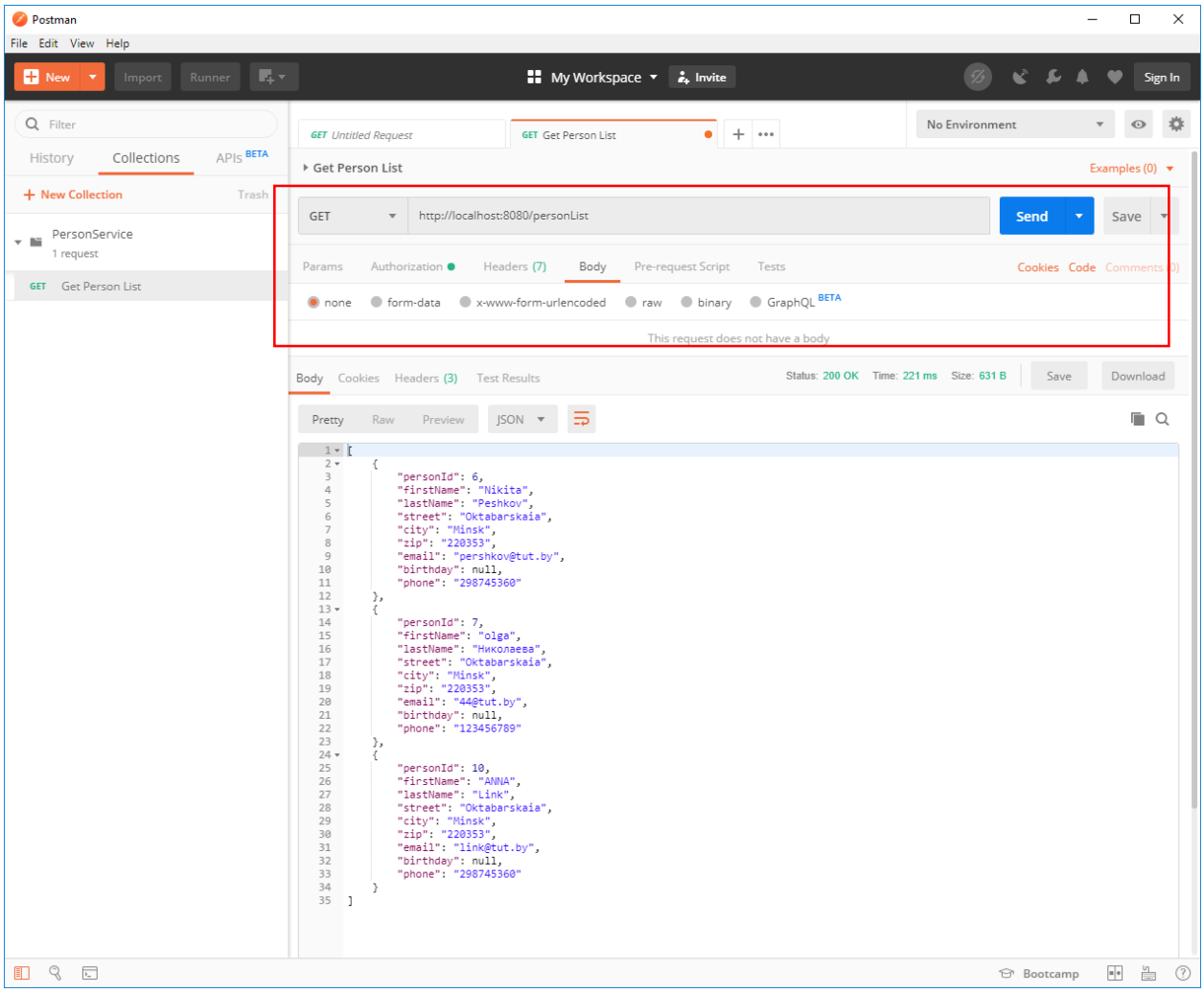


Протестируем наш сервис. Перейдите на Request .
Введите имя запроса, описание (необязательно) и выберите коллекцию, в которой вы хотите разместить эти запросы. Можно свою собственную новую коллекцию, чтобы хранить все ваши запросы на приложение в одном месте.

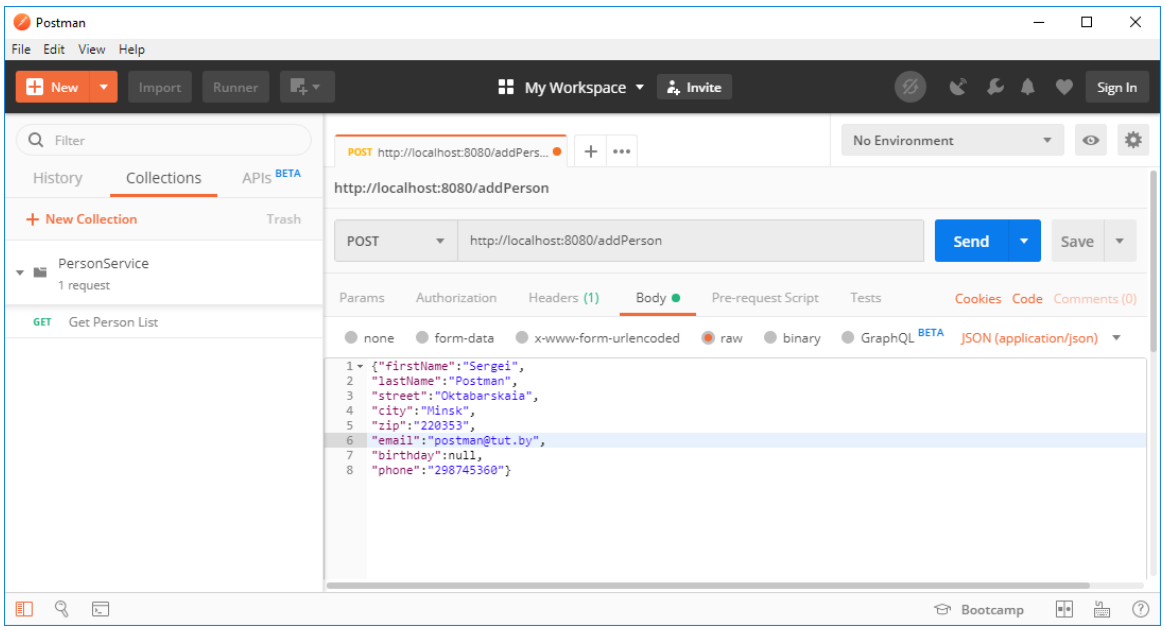




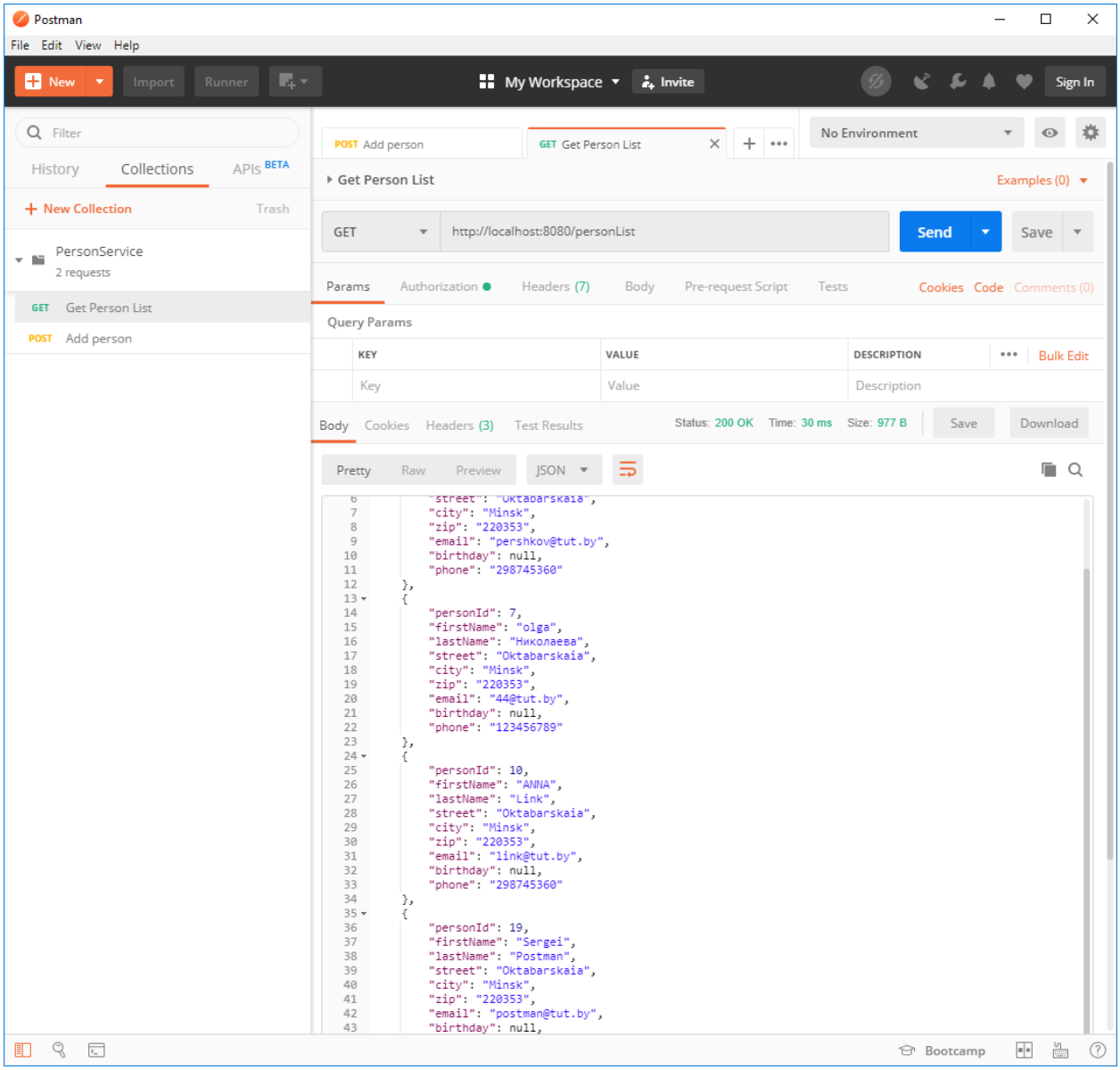
Проверим get запрос на список



Проверим запрос на добавление:

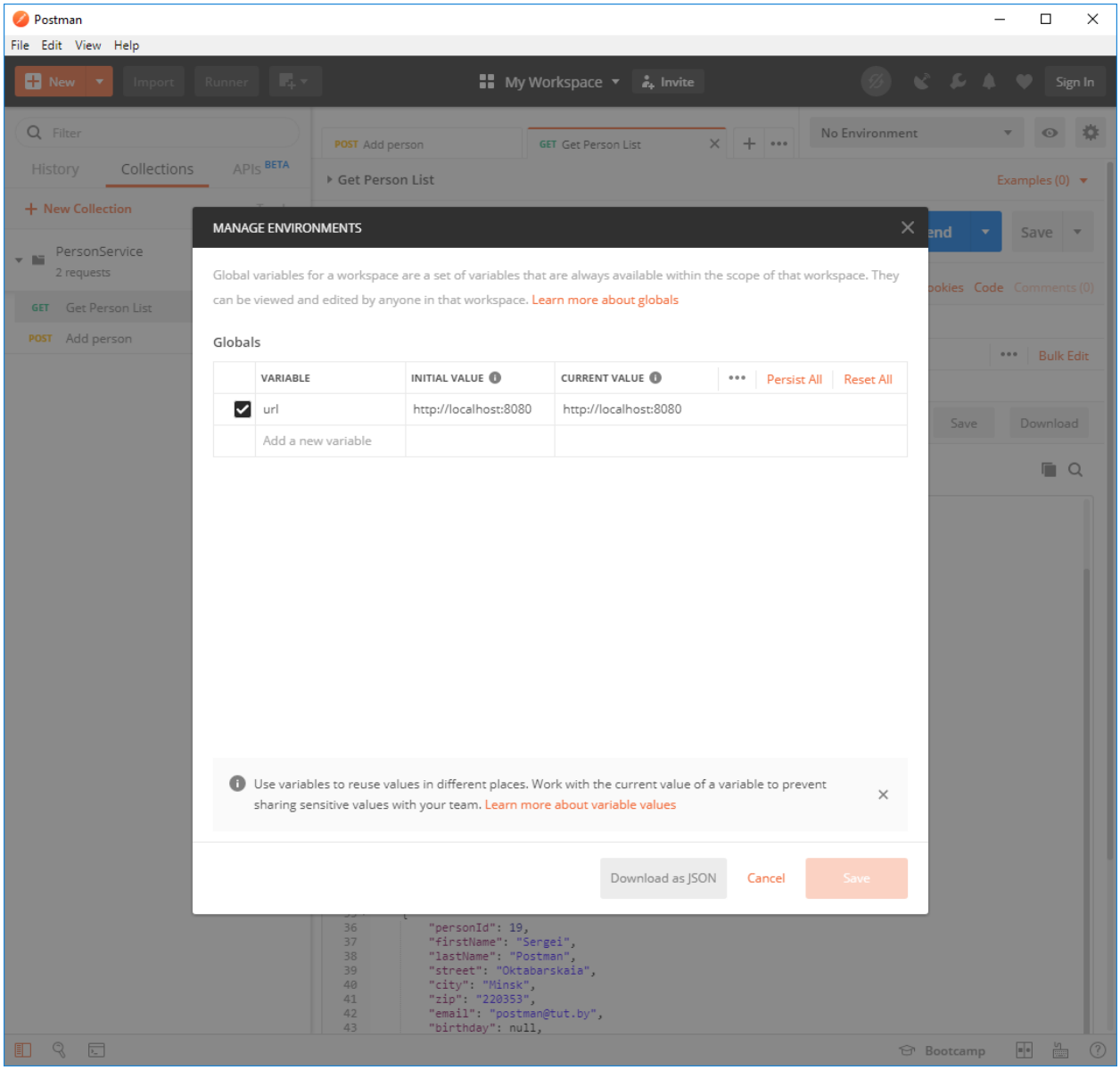


Сохраним, запустим и просмотрим

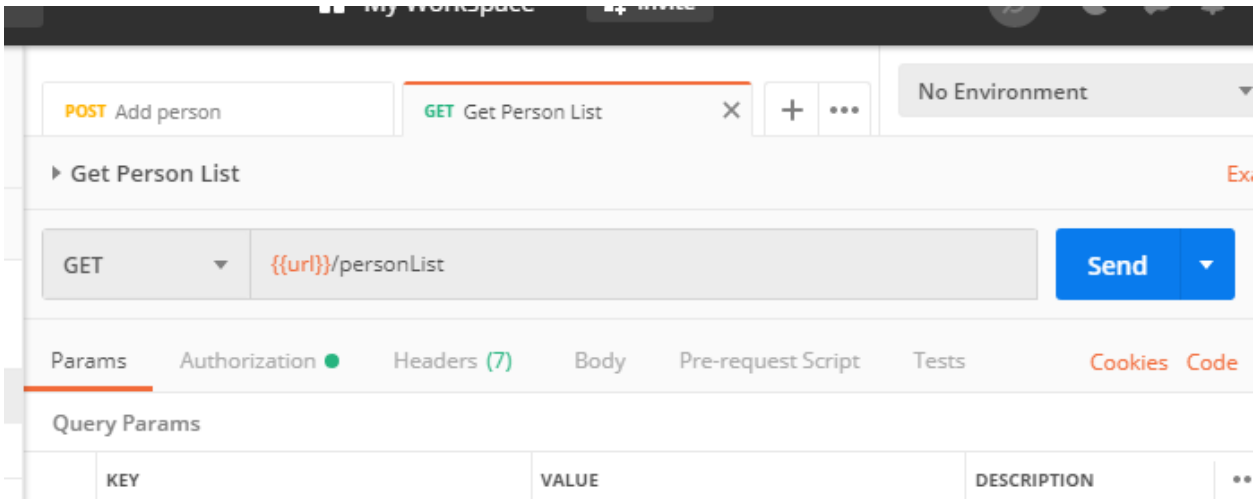


Чем еще можно пользоваться ?

Используйте переменные среды. Например, если **url** используется больше одного раза то важно использовать переменные, что упрощает редактирование и зависимости.



Когда вы сохранили свою переменную в таком енв. то в самом запросе вы можете использовать просто название ключа.



Еще одной полезной возможностью являются динамические переменные. Это **$guid** , **$randomint** - от 0 до 1000, и **$timestamp**

Изучите дополнительные возможности самостоятельно.

7.6 Создание HATEOAS REST сервиса

REST, несмотря на повсеместность использования, не является стандартом, как таковой, а подходом, стилем, ограничением HTTP протокола. Его реализация может различаться в стиле, подходе. Качество REST сервисов варьируется.

Leonard Richardson собрал воедино модель, которая объясняет различные уровни соответствующих понятий REST и сортирует их. Она описывает 4 уровня:

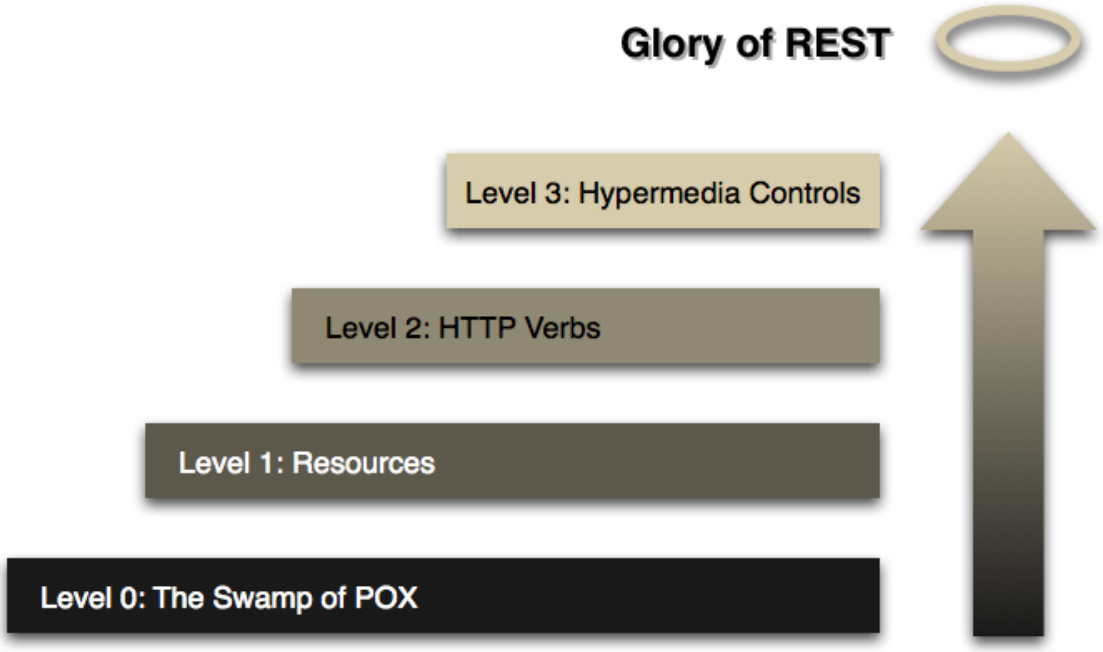
Level 0 Один URI, один HTTP метод: Swamp of POX - это уровень, где мы просто используем HTTP как транспорт. Вы можете вызвать SOAP технологию. Она использует HTTP, но как транспорт. SOAP, соответственно не является RESTful. Он всего лишь HTTP-aware.

Level 1 Несколько URI, один HTTP метод: Resources - на этом уровне сервисы могут использовать HTTP URI для отличия между сущностями в системе. К примеру, вы можете направить запросы на /customers, /users и т.д. XML-RPC является примером **Level 1** технологии: он использует HTTP и он может использовать URI для различия точек выхода. В конечном счете, несмотря на то, что XML-RPC не является RESTful, он использует HTTP как транспорт для чего-нибудь ещё(удаленный вызов процедур).

Level 2 Несколько URI, каждый поддерживает разные HTTP методы (Правильное использование HTTP): HTTP Verbs - это уровень, на котором мы сделали предыдущий проект. На этом уровне сервисы используют преимущества нативных HTTP возможностей, таким как заголовки, коды статуса, методы, определенные URI и другие. Этот уровень мы сделали.

Level 3 HATEOAS. Ресурсы сами описывают свои возможности и взаимосвязи : Hypermedia Controls - это заключительный уровень, к которому

стремимся. Гипермедиа как практическое применение HATEOAS ("HATEOAS" является сокращением от "Hypermedia as the Engine of Application State") шаблона проектирования. Гипермедиа продлевает жизнь сервису, отделяя клиента сервиса от необходимости глубокого знания платформы и топологии сервиса. Она описывает REST сервисы. Сервис может ответить на вопрос о том, какой был вызов и когда.



Текущий вид API работает хорошо. Если этот сервис про документировать, он будет работоспособен для REST клиентов написанных на множестве различных языков. Особенностью API является соответствие *принципу единого интерфейса*. Каждое сообщение включает достаточно информации для описания того, как обрабатывать сообщение. К примеру, клиент может решить, какой парсер будет вызван на основе заголовка Content-type в сообщении запроса. Изменение состояния осуществляется через HTTP глаголы (POST, GET, DELETE, PUT и др.). Таким образом, когда клиент хранит представление ресурса, включая метаданные, он имеет достаточно информации для изменения или удаления ресурса.

Однако, как было сказано выше: **Клиенты должны знать API**. Изменения в API отталкивают клиентов и они не обращаются к документации сервиса. Гипермедиа как двигатель состояния приложения (HATEOAS) является ещё одним ограничением. Клиенты создают состояния переходов только через действия, которые динамически идентифицируются в гипермедиа сервером.

HATEOAS REST или Hypermedia RESTful Web-сервиса

Давайте разработаем **исправленный вариант API с использованием Spring HATEOAS**.

Дополнительно посмотрите тут:
https://docs.spring.io/spring-hateoas/docs/current/reference/html/

Все что мы будем делать, это обертывать наш ответ полезной нагрузкой, используя Spring HATEOAS тип **EntityModel (ранее ResourceSupport)**. Этот тип аккумулирует объекты **Link**, которые описывают ресурсы.

Добавим к нашему проекту зависимость

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

В версии 1.0 были внесены изменения в структуре пакета. Это вызвано введением API регистрации типов гипермедиа для поддержки дополнительных типов носителей в Spring HATEOAS. Теперь API-интерфейсы клиента и сервера (имена пакетов соответственно), а также реализация медиа-типов четко разделены.

Обратите внимание !!!!! Была переименована группа классов ResourceSupport / Resource / Resources / PagedResources.

Старый тип	Новый тип
ResourceSupport	PresentationModel
Resource	EntityModel
Resources	CollectionModel
PagedResources	PagedModel

ResourceAssembler был переименован в PresentationModelAssembler, а его методы toResource (…) и toResources (…) были переименованы в toModel (…) и toCollectionModel (…) соответственно. Также изменения имени были отражены в классах, содержащихся в TypeReferences.

Фундаментальная идея гипермедиа заключается в обогащении представления ресурса элементами гипермедиа. Самая простая форма этого - ссылки. Они указывают клиенту, что он может перейти к определенному ресурсу. Семантика связанного ресурса определяется в так называемом отношении ссылки.

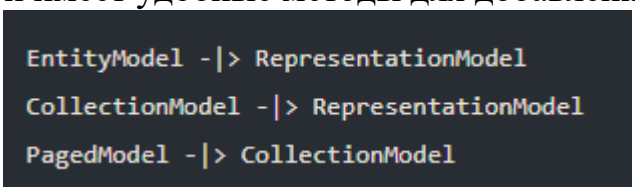
Spring HATEOAS позволяет работать со ссылками через его неизменный тип Link. Его конструктор принимает гипертекстовую ссылку и отношение ссылки.

```
Link link = new Link("http://localhost:8080/personList/{id}");
```

Есть набор предварительно определенных отношений ссылок. На них можно ссылаться через IanaLinkRelations.

```
Link link = new Link("http://localhost:8080/personList/{id}",
IanaLinkRelations.COLLECTION);
```

Чтобы легко создавать представления, обогащенные гипермедией, Spring HATEOAS предоставляет набор классов, в которых корневая структура представляет PresentationModel. Это в основном контейнер для коллекции ссылок и имеет удобные методы для добавления их в модель.



Стандартный способ работы с PresentationModel это создать его подкласс, содержащий все свойства, которые должно содержать представление, создать экземпляры этого класса, заполнить свойства и добавить ссылки.

Внесем изменения в `PersonDto`.

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.hateoas.RepresentationModel;
import java.util.Date;
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PersonDto extends RepresentationModel {

         private Long personId;
        private String firstName;
        private String lastName;
        private String street;
        private String city;
        private String zip;
        private String email;
        private Date birthday;
        private String phone;

}
```

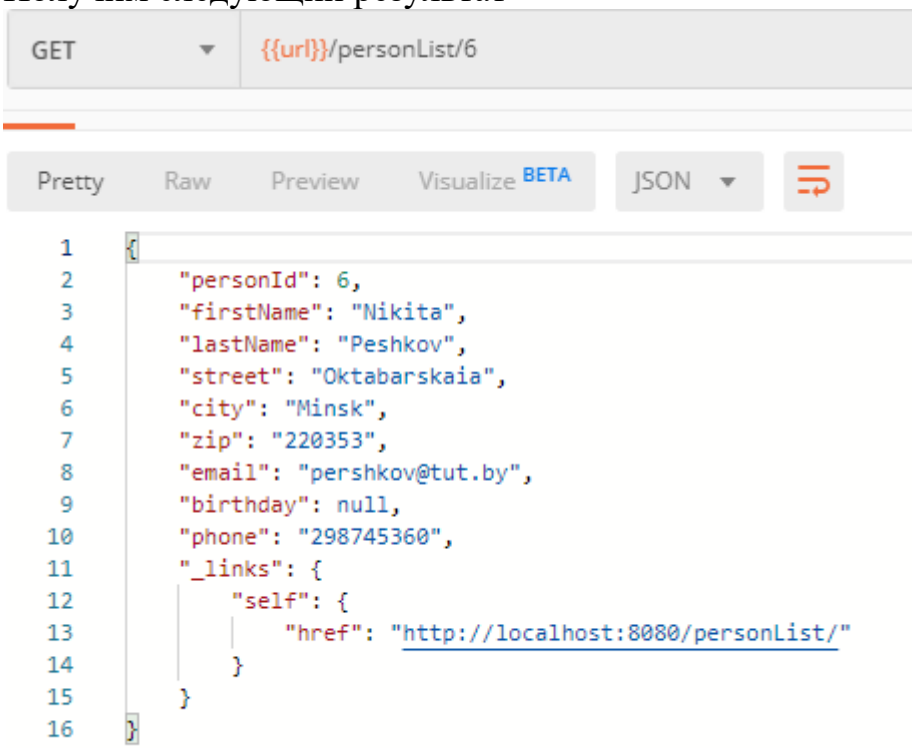
JSON представление ресурса будет дополнен списком гипермедиа элементов в свойстве _links. Самыми элементарными являются ссылки, указывающие на самих себя.

Тогда соответственно метод контроллера.

```
@GetMapping(value = {"/personList/{id}"})
public EntityModel<PersonDto> findById(@PathVariable("id") Long id) throws
ResourceNotFoundException {
```

```
    PersonDto personid = Mapper.map(personService.getById(id),
PersonDto.class);
    Link link = new Link("http://localhost:8080/personList/").withSelfRel();
    EntityModel <PersonDto> rezult =  new EntityModel <PersonDto>(personid,
link);
    return  rezult;
}
```

Получим следующий результат



Еще одна очень важная абстракция, предлагаемая библиотекой, - ControllerLinkBuilder - которая упрощает создание URI, избегая жестко закодированных ссылок. В следующем фрагменте показано построение собственной ссылки клиента с использованием класса ControllerLinkBuilder:

```
@GetMapping(value = {"/personList/{id}"})
    public EntityModel<PersonDto> findById(@PathVariable("id") Long id) throws
    ResourceNotFoundException {

 PersonDto personid = Mapper.map(personService.getById(id), PersonDto.class);
 Link link = linkTo(methodOn(PersonController.class).findById(id)).withSelfRel();

EntityModel <PersonDto> rezult =  new EntityModel <PersonDto>(personid, link);

    return  rezult;
    }
```

Метод linkTo () проверяет класс контроллера и получает его корневое отображение, метод withSelfMethod () квалифицирует отношение как самостоятельную ссылку. Однако более сложные системы могут включать и другие отношения.

Результат


```
{
    "personId": 6,
    "firstName": "Nikita",
    "lastName": "Peshkov",
    "street": "Oktabarskaia",
    "city": "Minsk",
    "zip": "220353",
    "email": "pershkov@tut.by",
    "birthday": null,
    "phone": "298745360",
    "_links": {
        "self": {
            "href": "http://localhost:8080/personList/6"
        }
    }
}
```


На этом этапе мы можем расширить метод `personList` PersonController:

```
@GetMapping(value = {"/personList"})
public CollectionModel<EntityModel<PersonDto>> personList() {

  //  return Mapper.mapAll(personService.getAllPerson(), PersonDto.class);

    CollectionModel<EntityModel<PersonDto>> resource =  CollectionModel.wrap(
            Mapper.mapAll(personService.getAllPerson(), PersonDto.class));

    for (final EntityModel<PersonDto> res : resource) {
        Link selfLink = linkTo(PersonController.class)
                .slash(res.getContent().getPersonId()).withSelfRel();
        res.add(selfLink);
    }
    resource.add(linkTo(methodOn(PersonController.class)
            .personList()).withRel("all"));
      return resource;

}
```

Результат


```
{
    "_embedded": {
        "personDtoList": [
            {
                "personId": 6,
                "firstName": "Nikita",
                "lastName": "Peshkov",
                "street": "Oktabarskaia",
                "city": "Minsk",
                "zip": "220353",
                "email": "pershkov@tut.by",
                "birthday": null,
                "phone": "298745360",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/6"
                    }
                }
```

```
            },
            {
                "personId": 7,
                "firstName": "olga",
                "lastName": "Николаева",
                "street": "Oktabarskaia",
                "city": "Minsk",
                "zip": "220353",
                "email": "44@tut.by",
                "birthday": null,
                "phone": "123456789",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/7"
                    }
                }
            },
            {
                "personId": 10,
                "firstName": "ANNA",
                "lastName": "Link",
                "street": "Oktabarskaia",
                "city": "Minsk",
                "zip": "220353",
                "email": "link@tut.by",
                "birthday": null,
                "phone": "298745360",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/10"
                    }
                }
            },
            {
                "personId": 22,
                "firstName": "Sergei",
                "lastName": "Postman",
                "street": "Oktabarskaia",
                "city": "Minsk",
                "zip": "220353",
                "email": "postman@tut.by",
                "birthday": null,
                "phone": "298745360",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/22"
                    }
                }
            }
        ]
    },
    "_links": {
        "all": {
            "href": "http://localhost:8080/personList"
        }
    }
}
```

В каждом представлении ресурса есть собственная ссылка и ссылка all для извлечения всех персон. Лучше делать ссылки на связанные представления. Например для персоны – список задач.

Этот пример демонстрирует, как Spring HATEOAS способствует обнаружению API в остальном веб-сервисе. Если ссылка существует, клиент может перейти по ней.

О так как создавать ссылки и какие типы ссылок бывают читайте
https://docs.spring.io/spring-hateoas/docs/current/reference/html/

Таким образом, клиент может иметь единую точку входа в приложение, и дальнейшие действия будут определяться на основе метаданных в ответном представлении.

Это позволяет серверу изменять свою схему URI. Кроме того, приложение может рекламировать новые возможности, помещая новые ссылки или URI в представление.

7.7 Документирование REST API на основе Open API

Документация является неотъемлемой частью построения REST API.
https://github.com/springdoc/springdoc-openapi

Каждое изменение в API должно быть одновременно описано в справочной документации. Выполнение этого вручную - утомительное занятие, поэтому автоматизация процесса была неизбежна.

The OpenAPI Specification (с англ. — «спецификация OpenAPI»; изначально известная как Swagger Specification[1]) — формализованная спецификация и экосистема множества инструментов, предоставляющая интерфейс между front-end системами, кодом библиотек низкого уровня и коммерческими решениями в виде API.

Изначально разработка спецификации под названием Swagger Specification проводилась с 2010 года компанией SmartBear. В ноябре 2015 года SmartBear объявила, что она работает над созданием Open API Initiative при спонсорской поддержке Linux Foundation. 1 января 2016 года спецификация была переименована в The OpenAPI Specification, а её развитие ведётся в рамках Open API Initiative. Поскольку инструменты Swagger были разработаны командой, участвовавшей в создании оригинальной спецификации Swagger, эти инструменты часто по-прежнему считаются синонимами спецификации. Таким образом, Swagger по-прежнему сохраняет свое название для наиболее известных и широко используемых инструментов для реализации спецификации OpenAPI, таких как Swagger Core, Swagger UI и многие другие.

Вкратце:

OpenAPI = Specification

Swagger = Tools for implementing the specification

OpenAPI позволяет вам описать весь ваш API, включая:

- Доступные конечные точки (/ users) и операции на каждой конечной точке (GET / users, POST / users)
- Параметры ввода и вывода для каждой операции
- Методы аутентификации
- Контактная информация, лицензия, условия использования и другая информация.
-

Чтобы сгенерировать документацию предлагается набор аннотаций для объявления и манипулирования выводом.

Краткий обзор аннотаций Swagger-Core:

- @Api - Отмечает класс как ресурс Swagger.
- @ApiModel - Предоставляет дополнительную информацию о моделях Swagger.
- @ApiModelProperty - Добавляет и манипулирует данными свойства модели.
- @ApiOperation - Описывает операцию или, как правило, метод HTTP для определенного пути.
- @ApiParam - Добавляет дополнительные метаданные для параметров операции.
- @ApiResponse - Описывает возможный ответ операции.
- @ApiResponses - Оболочка, чтобы разрешить список нескольких объектов ApiResponse.

Более подробную информацию о аннотация смотрите на

https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md

Перейдем к практической части. Для начала мы должны добавить в проект Maven зависимость

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.1.49</version>
</dependency>
```

Springdoc-openapi работает, исследуя приложение во время выполнения, чтобы вывести семантику API на основе конфигураций Spring, структуры классов и различных аннотаций.

Добавьте конфигурацию

```
@Configuration
public class OpenApiConfig {

    @Bean
```

```
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                .components(new Components())
                .info(new Info().title("Person Application API").description(
                        "This is Spring Boot RESTful service "));
    }
}
```

Запустите приложение и перейдите по адресу http://localhost:8080/v3/api-docs

Метаданные openapi, описывающие API, уже генерируются, но не удобочитаемы. Библиотека springdoc-openapi-ui автоматически развертывает swagger-ui в приложении spring-boot 2.

Документация будет доступна в формате HTML, используя swagger-ui.

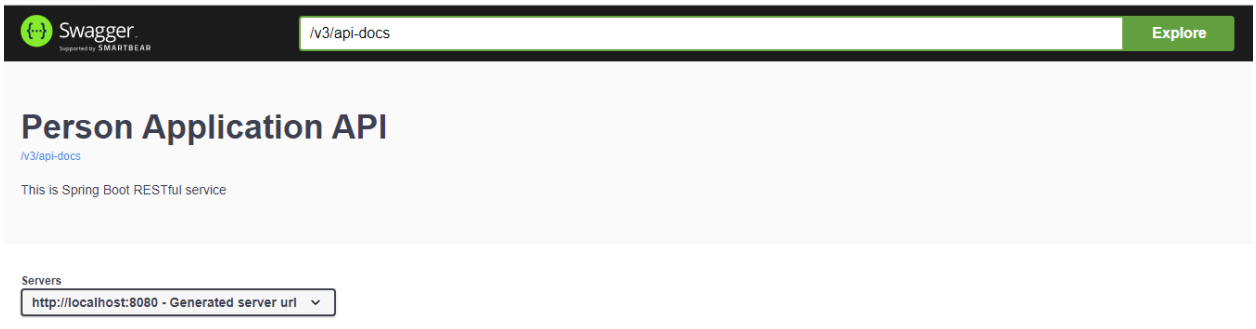
Страница пользовательского интерфейса Swagger должна быть доступна по адресу http: //server: port/context-path/swagger-ui.html, а описание OpenAPI будет доступно по следующему URL для формата json: http://server: port/context -path/v3/API-документы

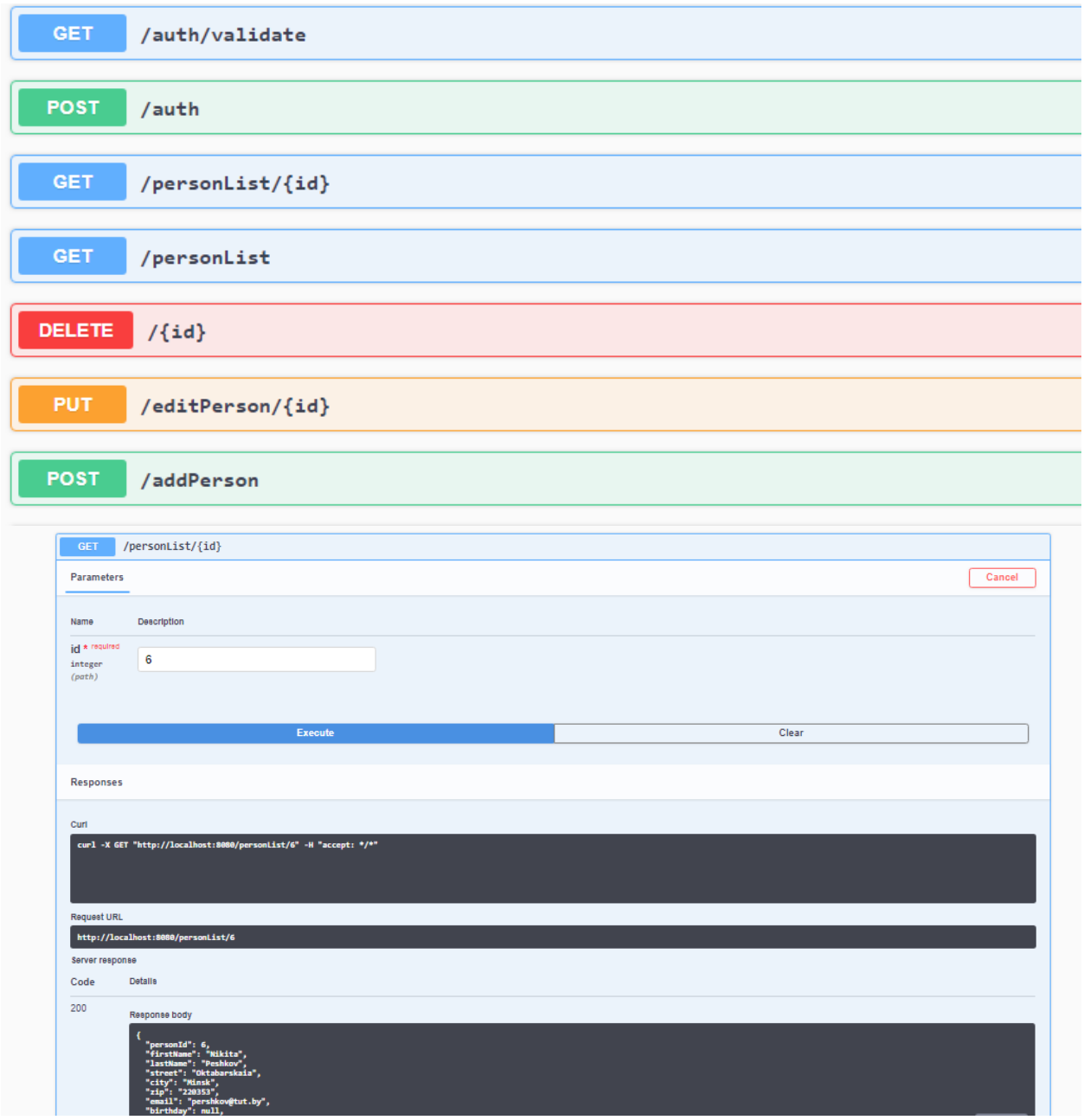
В нашем контексте, поскольку наш контекстный путь - /, то он будет доступен

http://localhost:8080/swagger-ui.html

который выполнит переадресацию на

http://localhost:8080/swagger-ui/index.html?url=/v3/api-docs&validatorUrl=#/default/findById





Здесь также описаны схемы

PersonDto {

personId	integer($int64)
firstName	string
lastName	string
street	string
city	string
zip	string
email	string
birthday	string($date-time)
phone	string

}

NewPersonDto {

personId	integer($int64)
firstName*	string
	maxLength: 2147483647
	minLength: 3
lastName*	string
street*	string
city*	string
zip	string
email*	string
birthday	string($date-time)
phone	string

}

Добавим аннотации

```
@Operation(summary = "Find person by ID", description = "Returns a single person", tags = {
"person" })
@ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "successful operation",
                content = @Content(schema = @Schema(implementation = Person.class))),
        @ApiResponse(responseCode = "404", description = "Person not found") })
@GetMapping(value = {"/personList/{id}"})
public PersonDto findById(
        @Parameter(description="Id of the person to be obtained. Cannot be empty",
                required=true)
                   @PathVariable("id") Long id) throws ResourceNotFoundException {

    return Mapper.map(personService.getById(id), PersonDto.class);

}
```