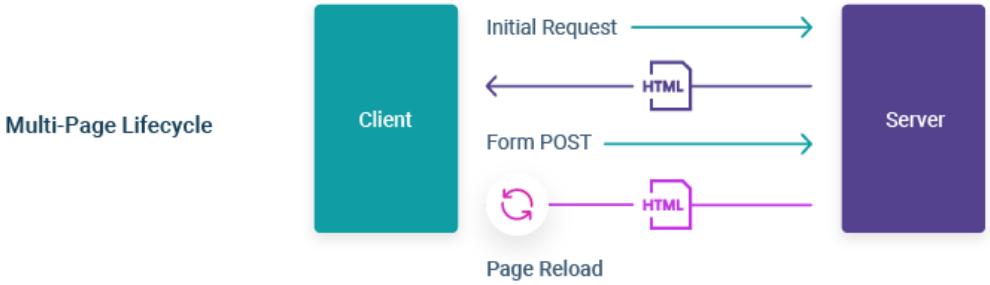
Лекций №4 REST API

Архитектурный стиль REST API, одностраничные приложения (SPA) и многостраничные приложения (MPA). Применение POSTMAN.

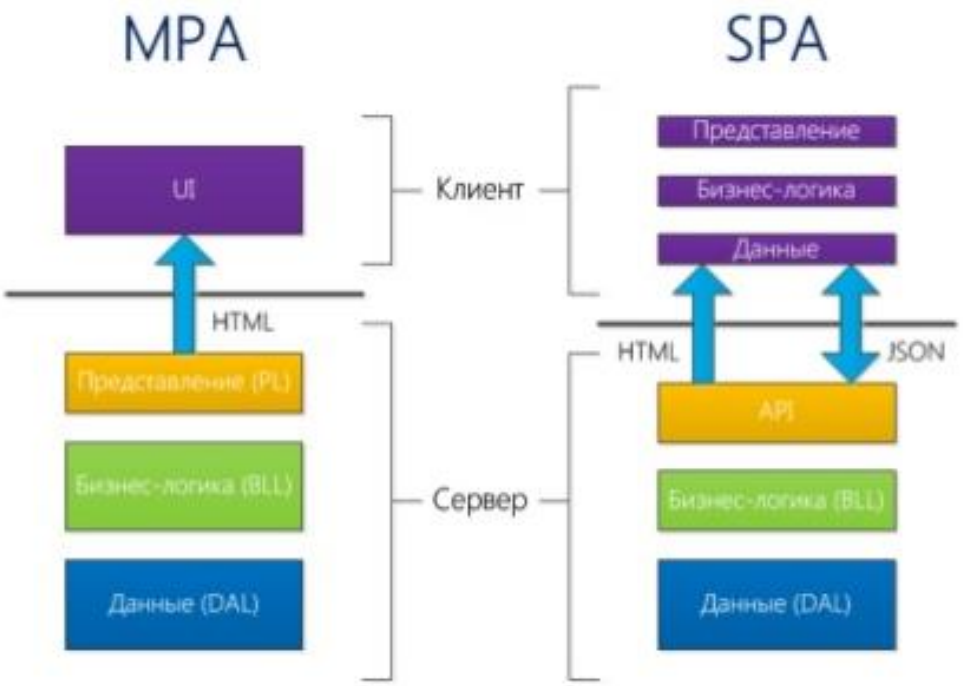
SPA и MPA

Существует два способа разработки веб приложений: одностраничные приложения (SPA) и многостраничные приложения(MPA). Single Page Application (SPA) и Multi Page Application (MPA)



Архитектура SPA устроена таким образом, что при переходе на новую страницу, обновляется только часть контента. Таким образом, нет необходимости повторно загружать одни и те же элементы. Для разработки SPA используется – javascript, jQuery React.js, Angular.js, Vue.js и другие фреймворки/библиотеки. SPA общается с сервером только чистыми данными. Разметка хранится на стороне клиента в шаблонах. Перезагрузки страницы не происходит. Примеры динамических приложений: Gmail, Google Maps, Facebok, GitHub

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.



Прогрессивные приложения или **Progressive Web Application** (PWA) взаимодействуют с пользователем, как приложение. Они могут устанавливаться на главный экран смартфона, отправлять push-уведомления и работать в офлайн-режиме.

Пример: Google Docs.

Введение в REST

SOAP (Simple Object Access Protocol)

SOAP - это метод передачи сообщений или небольших объемов информации через Интернет. Сообщения SOAP форматируются в XML и обычно отправляются с использованием HTTP (протокол передачи гипертекста).

Некоторые свойства:

- WSDL - Web Service Definition Language определяет контракт между клиентом и сервисом и является статическим по своей природе.
- создает протокол на основе XML поверх HTTP или иногда TCP / IP.
- описывает функции и типы
- является преемником XML-RPC и очень похож, но описывает стандартный способ связи.
- Некоторые языки программирования имеют встроенную поддержку SOAP

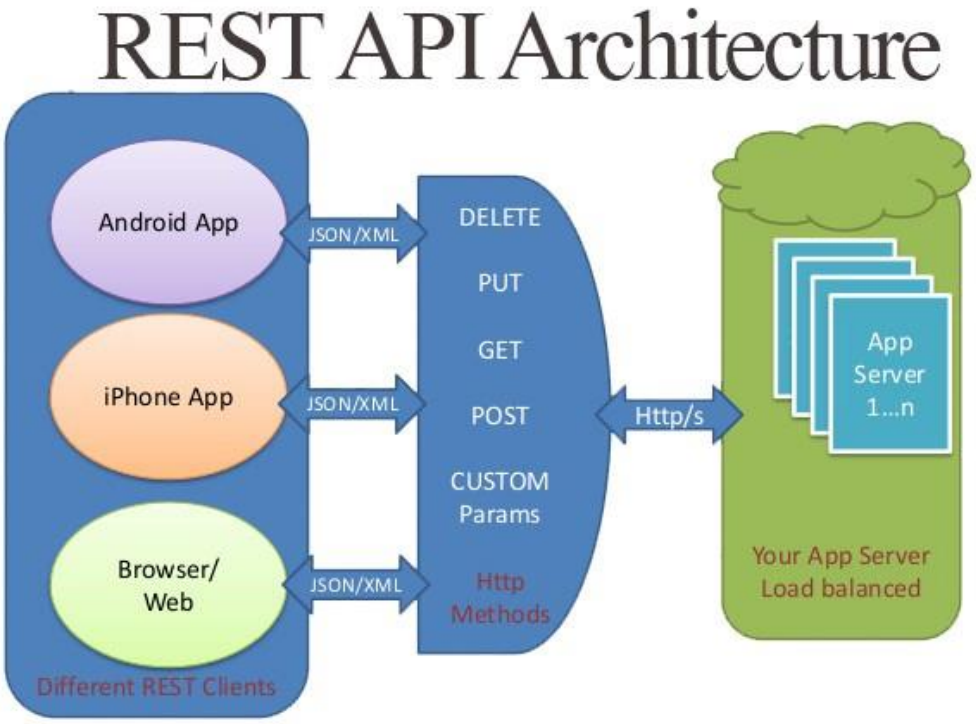
REST (REpresentational State Transfer):

REST — это архитектурный стиль взаимодействия приложений в сети. Но в отличии от SOAP, у REST отсутствует какой-либо стандарт, а

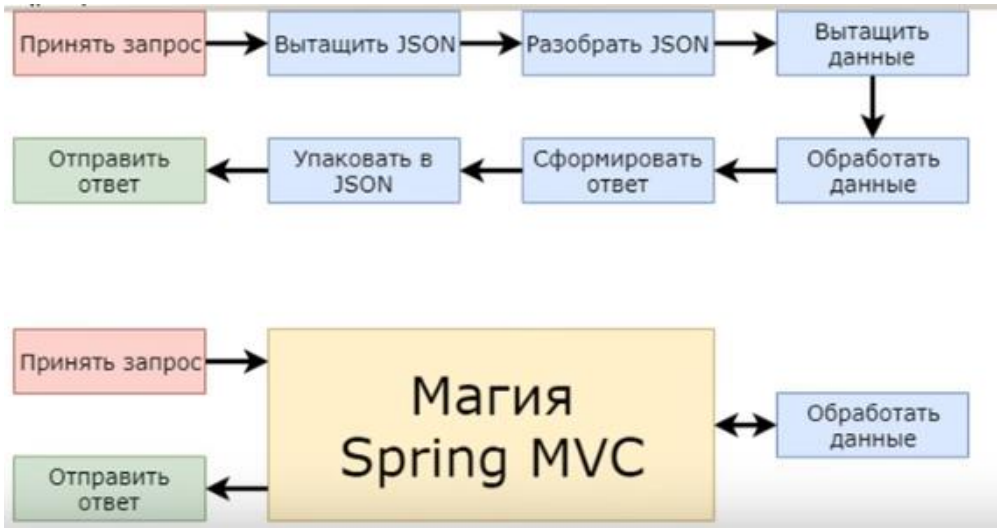
данные между клиентом и сервером могут предаваться в любом виде, будь то JSON, XML, YAML и т.д.

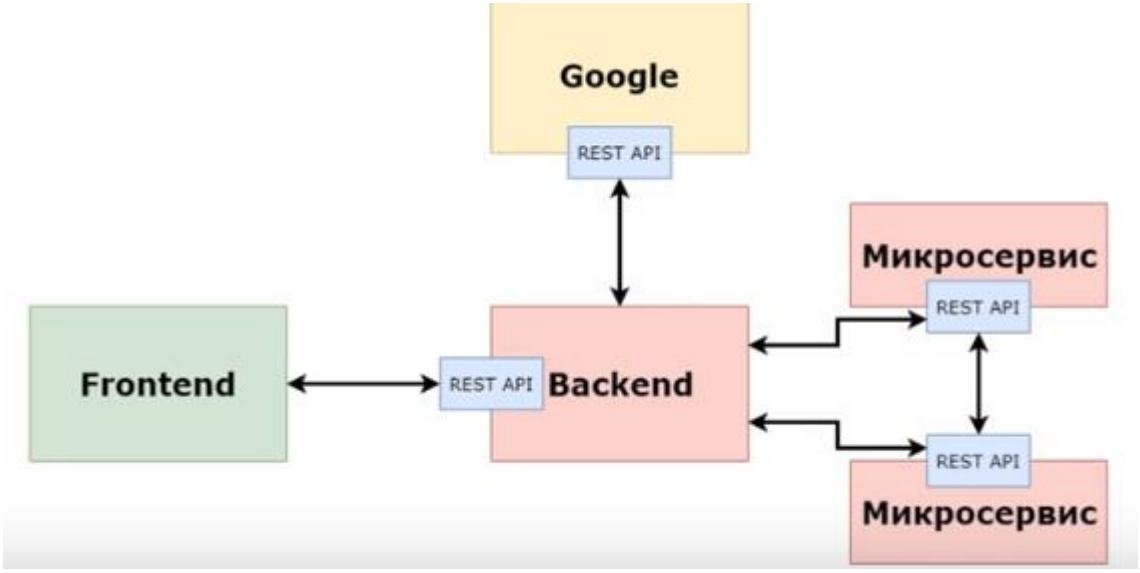
Свойства

- В случае REST контракт между клиентом и сервисом несколько усложняется и определяется HTTP, URI, медиаформатами и протоколом координации для конкретного приложения. Более динамично в отличие от WSDL.
- REST легок
- Обычно использует стандартные методы HTTP вместо формата XML, описывающего все. Например, для получения ресурса вы используете HTTP GET, для размещения ресурса на сервере вы используете HTTP PUT. Для удаления ресурса на сервере вы используете HTTP DELETE.
- REST обычно лучше всего использовать с ресурсно-ориентированной архитектурой (ROA). В этом способе мышления все является ресурсом, и вы будете оперировать этими ресурсами.
- Двоичные данные или двоичные ресурсы могут быть легко доставлены по запросу.



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





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*). В период между запросами клиента никакая информация о состоянии клиента на сервере не хранится (Stateless protocol).

3. **Кэширование:** клиенты, а также промежуточные узлы, могут выполнять кэширование ответов сервера.

4. **Единообразие интерфейса**: наличие унифицированного интерфейса является фундаментальным требованием дизайна REST-сервисов. К унифицированным интерфейсам предъявляются следующие четыре ограничительных условия:

- Идентификация ресурсов - Все ресурсы идентифицируются в запросах, например, с использованием URI в интернет-системах.
- 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. .
- 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).

5. **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

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.

Контроллер

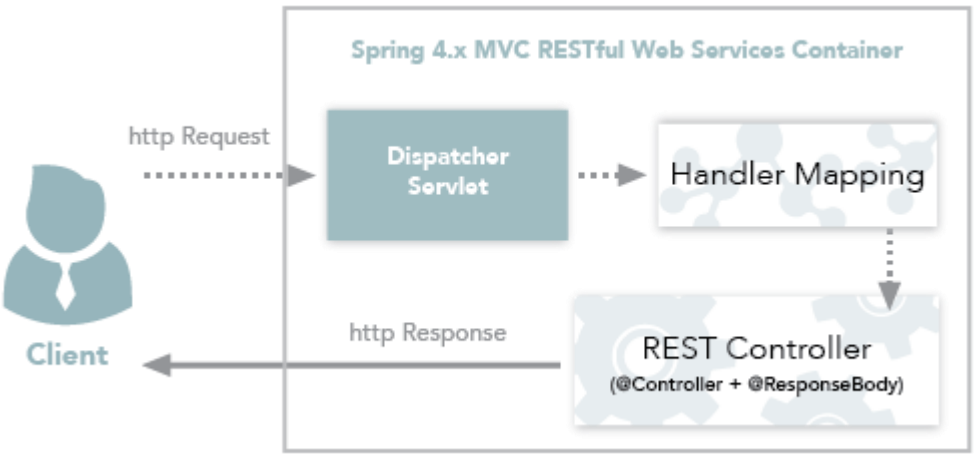
@RestController является центральным артефактом для API RESTful. Промоделируем простой ресурс.

```
@RestController
@RequestMapping("persons")
 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 = {"/all"})
    public List<PersonDto> personList() {
        return Mapper.mapAll(personService.getAllPerson(), PersonDto.class);
    }

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

```
    }

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

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

    }

}
```

@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**.



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.

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**.

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

```
@PostMapping("/add")
@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.

Использование @ExceptionHandler

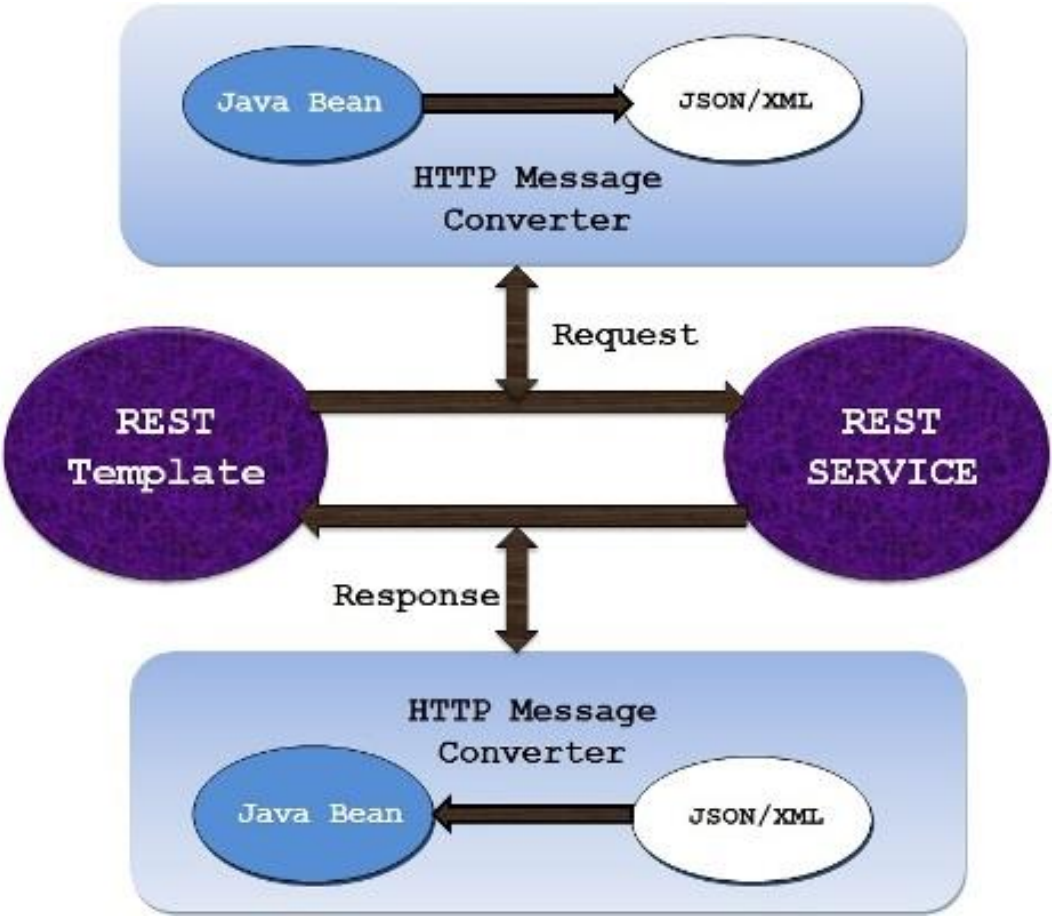
Другой вариант сопоставления пользовательских исключений с конкретными кодами состояния - использовать аннотацию **@ExceptionHandler** в контроллере.

Формат данных

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

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.



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. HttpMessageConverter работает в обоих направлениях: тела входящих запросов конвертируются в Java объекты, а Java объекты конвертируются в тела HTTP ответов.

Существуют следующие конверторы.

- *ByteArrayHttpMessageConverter* – byte arrays
- *StringHttpMessageConverter* – Strings
- *ResourceHttpMessageConverter* – converts *org.springframework.core.io.Resource* for any type of octet stream
- *SourceHttpMessageConverter* – *javax.xml.transform.Source*
- *FormHttpMessageConverter* – form data to/from a *MultiValueMap<String, String>*.

- *Jaxb2RootElementHttpMessageConverter* – Java objects to/from XML (added only if JAXB2 is present on the classpath)
- *MappingJackson2HttpMessageConverter* – JSON (added only if Jackson 2 is present on the classpath)
- *MappingJacksonHttpMessageConverter* – JSON (added only if Jackson is present on the classpath)
- *AtomFeedHttpMessageConverter* – Atom feeds (added only if Rome is present on the classpath)
- *RssChannelHttpMessageConverter* – RSS feeds *(added only if Rome is present on the classpath)*

```
@Override
public void configureMessageConverters(
        List<HttpMessageConverter<?>> converters) {

    messageConverters.add(createXmlHttpMessageConverter());
    messageConverters.add(new MappingJackson2HttpMessageConverter());
}
private HttpMessageConverter<Object> createXmlHttpMessageConverter() {
    MarshallingHttpMessageConverter xmlConverter =
            new MarshallingHttpMessageConverter();

    XStreamMarshaller xstreamMarshaller = new XStreamMarshaller();
    xmlConverter.setMarshaller(xstreamMarshaller);
    xmlConverter.setUnmarshaller(xstreamMarshaller);

    return xmlConverter;
}
```

API Design

1) **Конечные точки** – имя существительное. Множественное число
/frames/{frame_id}
/persons

2) Документирование

Документация с перечисленными в ней конечными точками, и описывающая список операций для каждой из них.

Swagger
https://swagger.io/
Apiary
https://apiary.io/

https://www.drfdocs.com/

3) Версия вашего приложения

Существует два общих способа для управления версиями REST приложений:

1. URI версии. host/v2/farmers
2. Мультимедиа версии.

4) Пагинация

Best practice является разбиение результатов на части, а не отправка всех записей сразу.

5) SSL

Вы всегда должны применять SSL для своего REST приложения.

Стандартные протоколы проверки аутентификации облегчают работу по защите вашего приложения.

6) HTTP методы

Ниже представлены две характеристики, которые должны быть определены перед использованием HTTP метода:

Безопасность: HTTP метод считается безопасным, когда вызов этого метода не изменяет состояние данных. Например, когда вы извлекаете данные с помощью метода GET, это безопасно, потому что этот метод не обновляет данные на стороне сервера.

Идемпотентность: когда вы получаете один и тот же ответ, сколько раз вы вызываете один и тот же ресурс, он известен как идемпотентный. Например, когда вы пытаетесь обновить одни и те же данные на сервере, ответ будет таким же для каждого запроса, сделанного с одинаковыми данными.

HTTP Method	Safe	Idempotent
GET	✅	✅
POST	❌	❌
PUT	❌	✅
DELETE	❌	✅
OPTIONS	✅	✅
HEAD	✅	✅

7) Эффективное использование кодов ответов HTTP

Проверка REST API

Для проверки можно использовать броузер

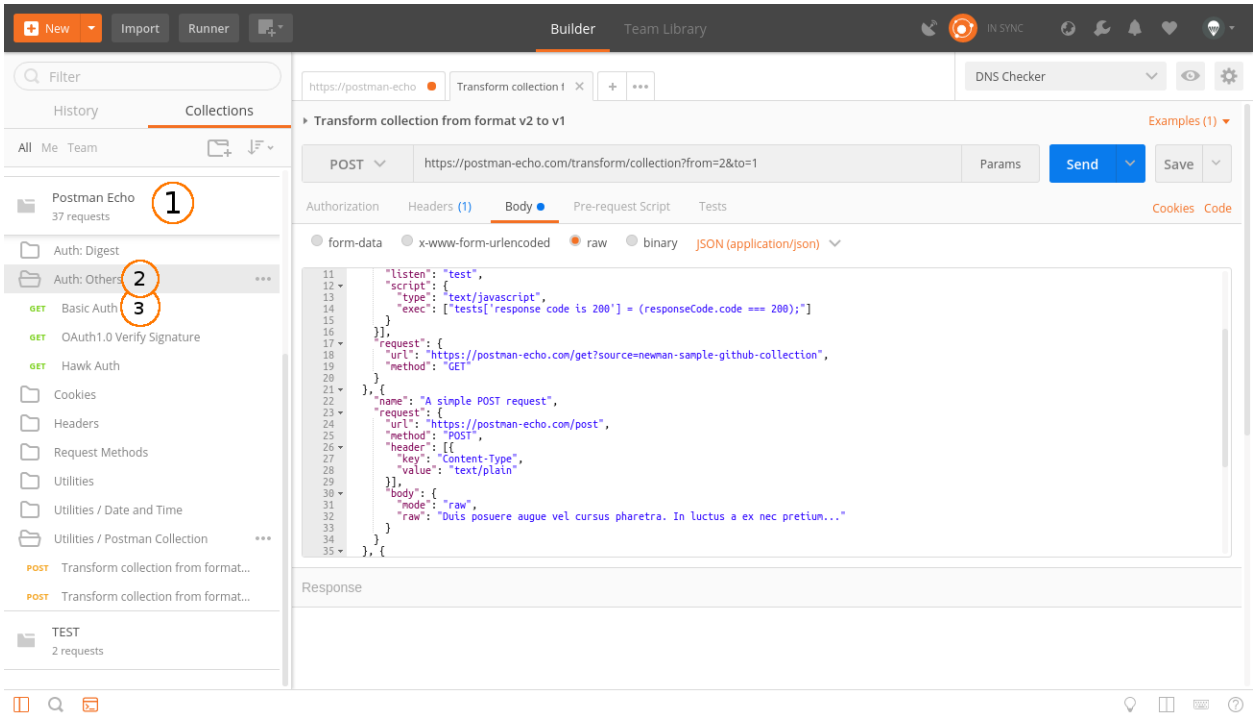
Или утилиту curl с терминала:
curl -v http://localhost:8080/persons/all

Команды curl можно посмотреть по ссылке
https://curl.se/

POSTMAN

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

https://www.getpostman.com/downloads/



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

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.

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.

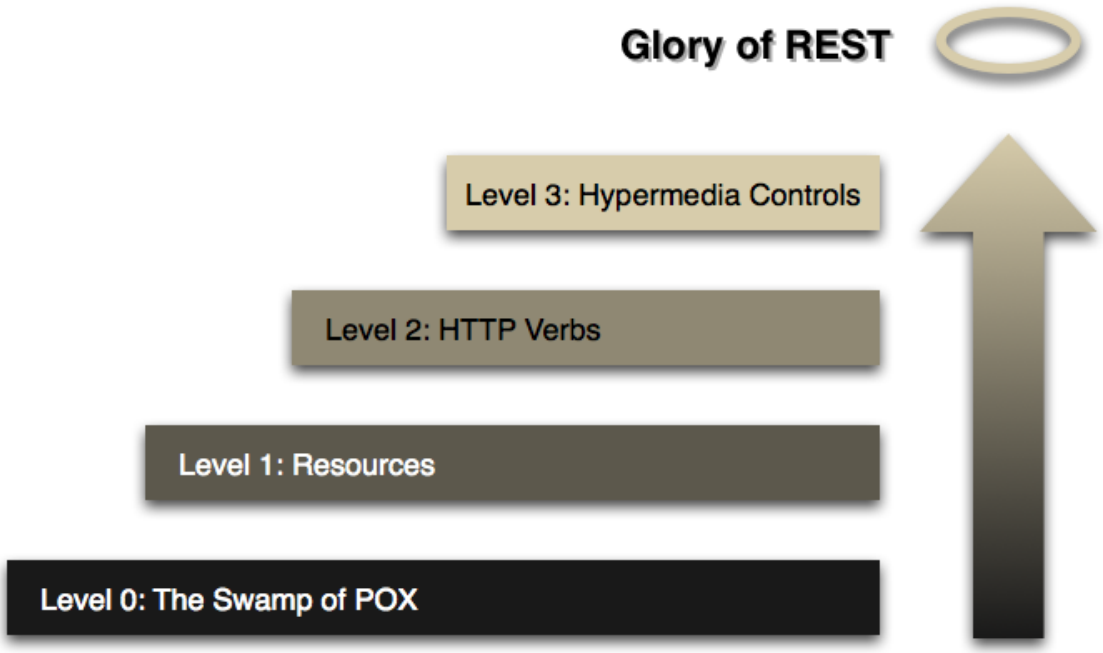
Leonard Richardson собрал воедино модель, которая объясняет различные уровни соответствующих понятий REST и сортирует их. Она описывает 4 уровня:

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

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

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.

```
EntityModel -|> RepresentationModel
CollectionModel -|> RepresentationModel
PagedModel -|> CollectionModel
```

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.

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.

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.

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:

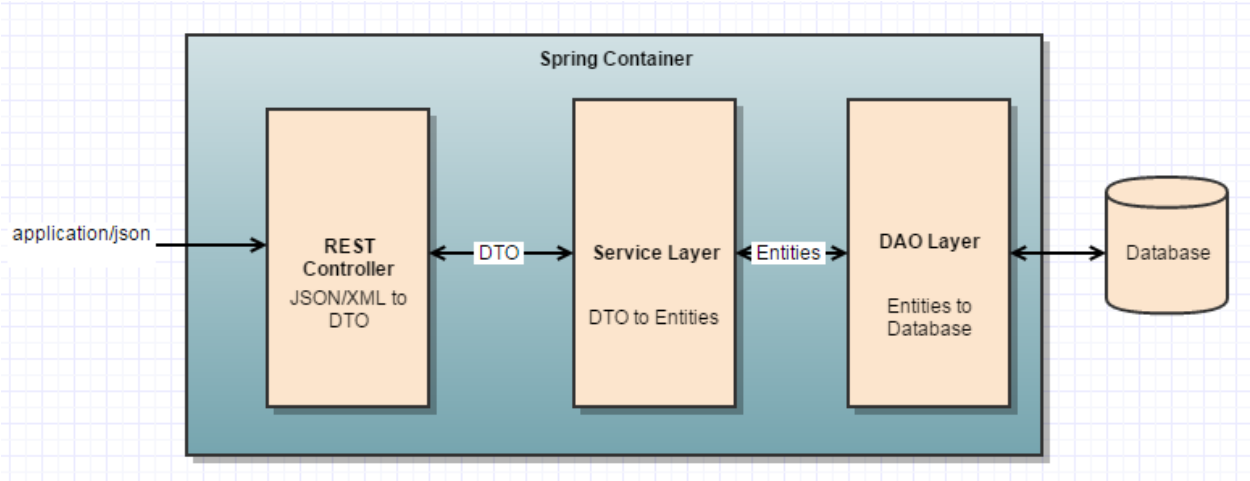
- @Api - Отмечает класс как ресурс Swagger.
- @ApiModel - Предоставляет дополнительную информацию о моделях Swagger.
- @ApiModelProperty - Добавляет и манипулирует данными свойства модели.
- @ApiOperation - Описывает операцию или, как правило, метод HTTP для определенного пути.
- @ApiParam - Добавляет 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

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

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.

Требуется

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