



СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

Интернационализация, локализация и реализация HATEOAS в микросервисах

Интернационализация (англ. internationalization, i18n) – технологические приёмы разработки программного обеспечения, упрощающие его адаптацию к языковым и культурным особенностям региона (регионов), отличного от того, в котором разрабатывался продукт.

Локализация (англ. localization, L10n) – процесс адаптации интернационализированного программного обеспечения к языковым и культурным особенностям конкретного региона.

Параметры для i18n и L10n



- Язык
- Текст (шрифты)
- Алфавиты, направление письма, системы нумерации.
- Форматы даты и времени
- Часовой пояс
- Валюта
- Единицы мер и весов
- ...

Объект класса `Locale` представляет определенный географический, политический или культурный регион.

Конструкторы в классе `Locale`:

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

Параметры конструктора: *language* – язык, *country* – страна (регион), *variant* – правила правописания.

Создание объекта класса Locale



- Константы для создания объекта класса Locale (есть не для всех языков и стран)

Locale.US (указание языка и страны, en_US)

Locale.FRENCH (указание языка без указания страны, fr).

- Создание объекта с помощью конструктора

Locale rus = new Locale("ru", "RU");

Коды стран, языков и вариантов приведены в IANA Language Subtag Registry (Type: language, Type: region, Type: variant).

<https://www.iana.org/assignments/language-subtag-registry/language-subtag-registry>

Обновление класса инициализации

```
package com.hospitals.fractures;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
import org.springframework.context.support.ResourceBundleMessageSource;
import java.util.Locale;

@SpringBootApplication
public class FracturesApplication {

    public static void main(String[] args) {
        SpringApplication.run(FracturesApplication.class, args);
    }

    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver localeResolver = new SessionLocaleResolver();
        Locale rus = new Locale("ru", "RU");
        localeResolver.setDefaultLocale(rus);
        return localeResolver;
    }

    @Bean
    public ResourceBundleMessageSource messageSource() {
        ResourceBundleMessageSource messageSource =
            new ResourceBundleMessageSource();
        messageSource.setUseCodeAsDefaultMessage(true);
        messageSource.setBasenames("messages");
        messageSource.setDefaultEncoding("UTF-8");
        return messageSource;
    }
}
```

Установка регионального стандарта



@Bean - это аннотация Java уровня метода. Она сообщает, что метод вернет объект, который Spring должен зарегистрировать как компонент в контексте приложения.

Интерфейс LocaleResolver – это интерфейс, который позволяет определить текущий региональный стандарт (Locale) пользователя через запрос и изменить его через запрос и ответ.

Класс SessionLocaleResolver – это класс, который позволяет хранить Locale в сеансе текущего пользователя с использованием имени атрибута. Объект класса Locale rus установлен в коде в качестве текущего регионального стандарта по умолчанию.

Обновление класса инициализации

```
package com.hospitals.fractures;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
import org.springframework.context.support.ResourceBundleMessageSource;
import java.util.Locale;

@SpringBootApplication
public class FracturesApplication {

    public static void main(String[] args) {
        SpringApplication.run(FracturesApplication.class, args);
    }

    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver localeResolver = new SessionLocaleResolver();
        Locale rus = new Locale("ru", "RU");
        localeResolver.setDefaultLocale(rus);
        return localeResolver;
    }

    @Bean
    public ResourceBundleMessageSource messageSource() {
        ResourceBundleMessageSource messageSource =
            new ResourceBundleMessageSource();
        messageSource.setUseCodeAsDefaultMessage(true);
        messageSource.setBasenames("messages");
        messageSource.setDefaultEncoding("UTF-8");
        return messageSource;
    }
}
```


Класс ResourceBundleMessageSource - это класс, который служит для преобразования текстовых сообщений из файла свойств (*.properties) в зависимости от выбранных Locale.

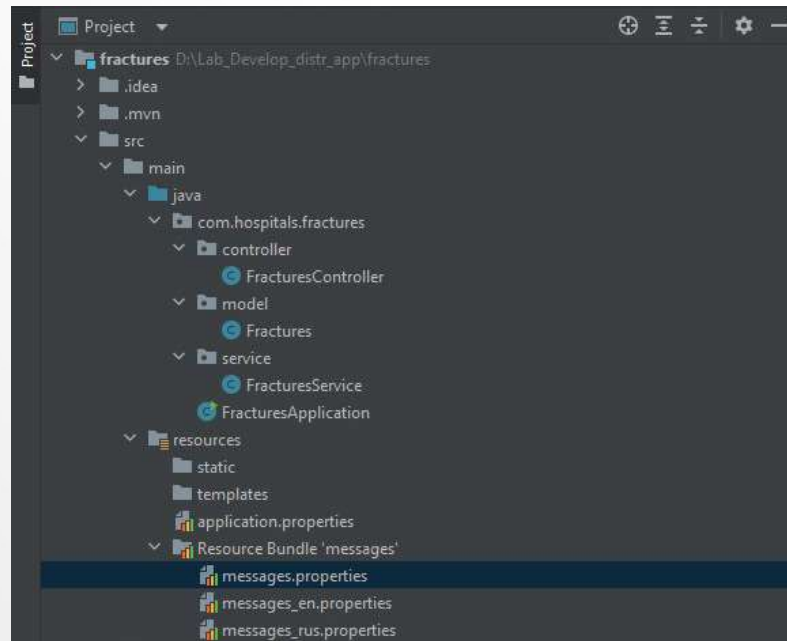
Метод `setUseCodeAsDefaultMessage(boolean useCodeAsDefaultMessage)` – метод, который указывает, следует ли использовать код сообщения в качестве сообщения по умолчанию вместо создания исключения `NoSuchMessageException` (возврат кода сообщения `'fractures.create.message'` в примере).

Метод `setBasename(String basename)` – это метод, который указывает, что источником сообщения является файл свойств, начинающийся с `basename`.

Создание файлов свойств

Щелкнуть ПКМ на пакете `src/main/resources` и создать файлы:

- `messages.properties`
- `messages_rus.properties`
- `messages_en.properties`



Файл `messages_rus.properties`

`fractures.create.message` = Запись в реестре переломов создана %s

`fractures.update.message` = Запись в реестре переломов обновлена %s

`fractures.delete.message` = Удалена запись в реестре переломов с типом %s
кости для %d пациентов из %s больницы

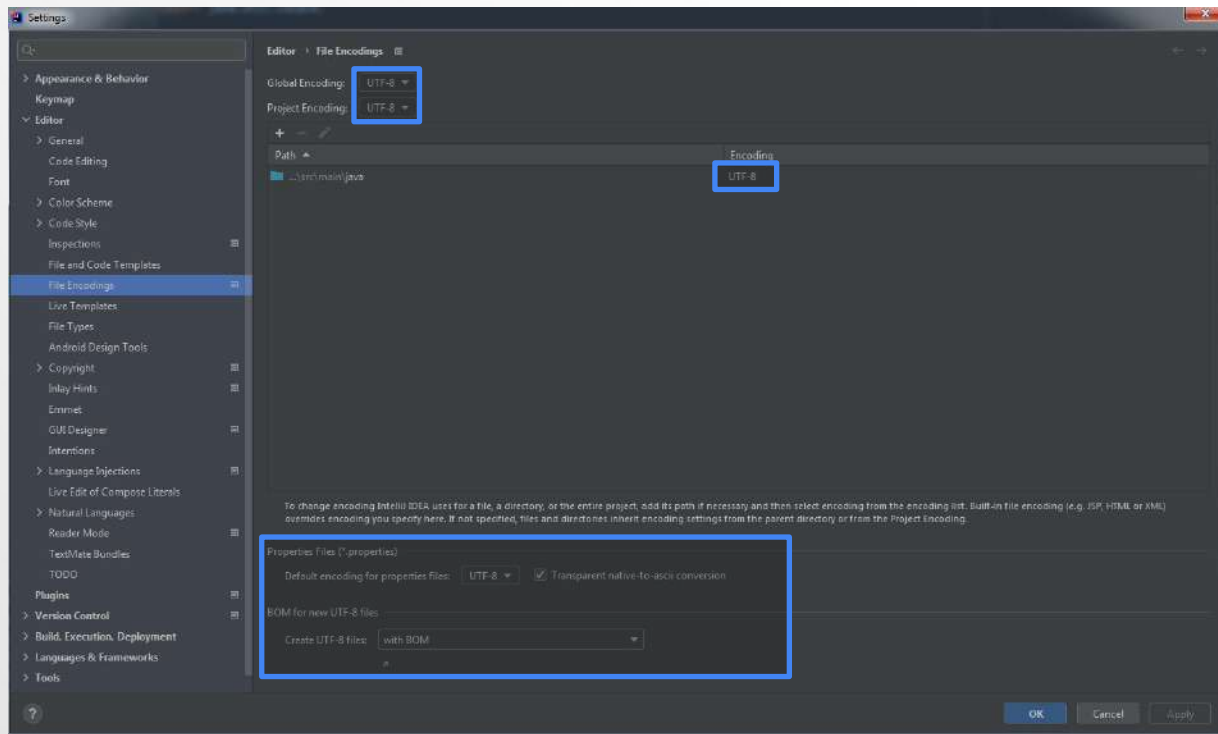
Файл `messages_en.properties`

`fractures.create.message` = Entry in the fracture register created %s

`fractures.update.message` = Entry in the fracture register updated %s

`fractures.delete.message` = Deleting entry in the fracture register with the type of %s
bone for %d patients in the %s hospital

Настройка кодировок



Обновление класса службы



```
package com.hospitals.fractures.service;
```

```
import java.util.Locale;  
import java.util.Random;
```

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.MessageSource;  
import org.springframework.stereotype.Service;
```

```
import com.hospitals.fractures.model.Fractures;
```

```
@Service  
public class FracturesService {
```

```
    @Autowired  
    MessageSource messages;
```

*

```
public String createFractures(Fractures fractures, String hospitalName, Locale locale) {  
    String responseMessage = null;  
    if(fractures != null) {  
        fractures.setHospitalName(hospitalName);  
        responseMessage = String.format(messages.getMessage("fractures.create.message", null, locale), fractures.toString());  
    }  
  
    return responseMessage;  
}
```

* Метод `getFractures` остается в коде без изменений.
Он не представлен на слайде .

Интерфейс `MessageSource` – это интерфейс, который служит для преобразования сообщений с поддержкой их интернационализации.

Метод `getMessage` – метод, который предпринимает попытку преобразовать сообщение. Он возвращает сообщение по умолчанию, если сообщение не найдено.

Обновление метода в классе контроллера



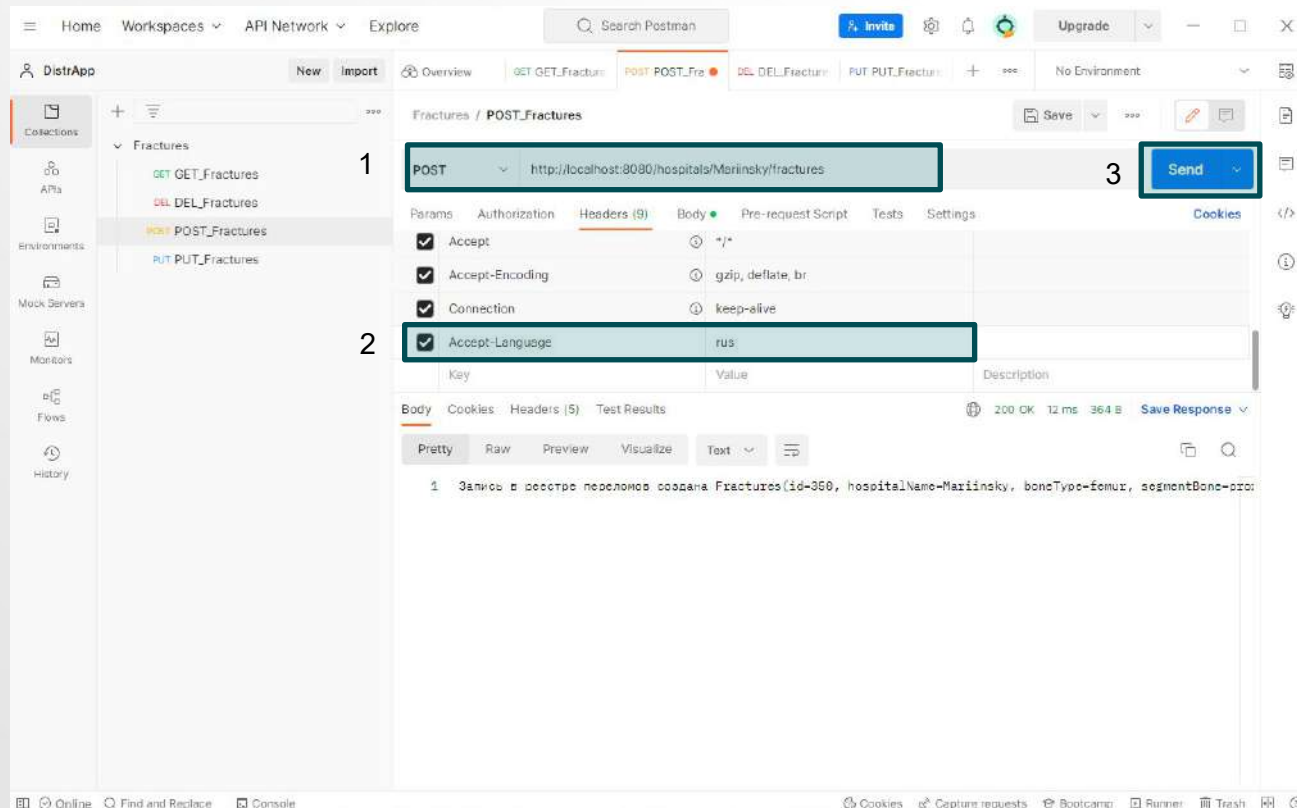
@PostMapping

```
public ResponseEntity<String> createFractures(  
    @PathVariable("hospitalName") String hospitalName,  
    @RequestBody Fractures request,  
    @RequestHeader(value = "Accept-Language", required = false)  
    Locale locale) {  
    return ResponseEntity.ok(fracturesService.createFractures(request, hospitalName, locale));  
}
```

@RequestHeader – аннотация, указывающая, что параметр метода должен быть связан с заголовком веб-запроса.

В примере выполнено отображение значения заголовка Accept-Language запроса в параметр метода locale.

Вызов конечной точки POST через URL



The screenshot displays the Postman interface for a REST client. The left sidebar shows a collection named 'Fractures' with several endpoints. The main panel shows the configuration for a POST request to 'http://localhost:8080/hospitals/Mariinsky/fractures'. The 'Headers' tab is active, showing 'Accept-Language' set to 'rus'. The 'Body' tab shows a JSON response. The 'Send' button is highlighted.

1. POST method selected

2. Accept-Language header set to rus

3. Send button

Response (JSON):

```
1 { "Занесён в историю болезни созданная Fractures(id=360, hospitalName=Mariinsky, boneType=femur, segmentBone=prox"
```


1. Реализовать в приложении перевод сообщений, уведомляющих о создании, модификации и удалении объекта модели на три языка:
 - русский;
 - английский;
 - язык согласно номеру варианта.
2. Выполнить запуск приложения.
3. Выполнить обращение к конечным точкам (запросы POST, PUT, DELETE) с использованием заголовка Accept-language в Postman.

Варианты

Вариант	Язык
1	Немецкий
2	Французский
3	Итальянский
4	Испанский
5	Португальский
6	Болгарский
7	Чешский
8	Финский
9	Шведский
10	Норвежский
11	Латышский
12	Литовский
13	Польский

HATEOAS (Hypermedia as the Engine of Application State, гипермедиа как средство изменения состояния приложения) – это архитектурное ограничение для REST-приложений.

Используя HATEOAS, клиент взаимодействует с сетевым приложением, сервер которого обеспечивает динамический доступ через гипермедиа.

Гипермедиа

Гипермедиа – это любой контент, который содержит ссылки на другие формы мультимедиа, такие как изображения, видео и текст. Ответ на запрос REST представляет собой данные, а не действия с ними. С помощью HATEOAS ответ на запрос REST возвращает не только данные, но и действия, которые можно выполнить с ресурсом.



Spring HATEOAS – это проект, который позволяет создавать API, следующие принципу HATEOAS отображения связанных ссылок для данного ресурса.

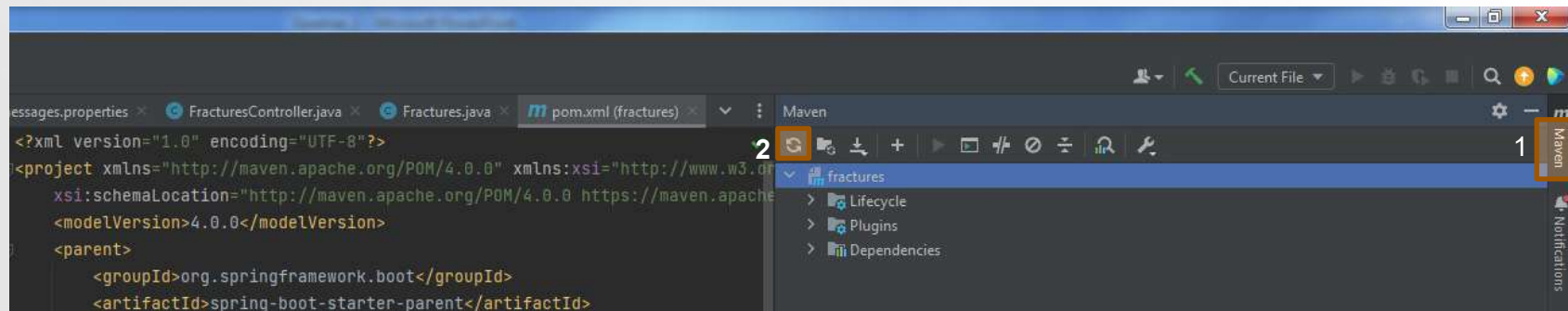
Добавление зависимости



Добавить зависимость HATEOAS в файл pom.xml

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

При возникновении «Dependency 'org.springframework.boot:spring-boot-starter-hateoas:3.1.3' not found»



Обновление класса модели



```
package com.hospitals.fractures.model;
```

```
import lombok.Getter;
```

```
import lombok.Setter;
```

```
import lombok.ToString;
```

```
import org.springframework.hateoas.RepresentationModel;
```

```
@Getter
```

```
@Setter
```

```
@ToString
```

```
public class Fractures extends RepresentationModel<Fractures> {
```

```
    private int id;
```

```
    private String hospitalName;
```

```
    private String boneType;
```

```
    private String segmentBone;
```

```
    private String fractureType;
```

```
    private boolean infection;
```

```
    private int patientsNumber;
```

```
}
```

Класс Fractures наследуется от класса RepresentationModel, который содержит набор ссылок и предоставляет API для добавления этих ссылок в модель.

Обновление класса контроллера

```
package com.hospitals.fractures.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;

import com.hospitals.fractures.model.Fractures;
import com.hospitals.fractures.service.FracturesService;

import java.util.Locale;

@RestController
@RequestMapping(value="hospitals/{hospitalName}/fractures")
public class FracturesController {

    @Autowired
    private FracturesService fracturesService;

    @GetMapping(value="/{boneType}/{patientsNumber}")
    public ResponseEntity<Fractures> getFractures(
        @PathVariable("hospitalName") String hospitalName,
        @PathVariable("boneType") String boneType,
        @PathVariable("patientsNumber") int patientsNumber) {

        Fractures fractures = fracturesService.getFractures(hospitalName, boneType, patientsNumber);
        fractures.add(linkTo(methodOn(FracturesController.class)
            .getFractures(hospitalName, boneType, patientsNumber))
            .withSelfRel(),
            linkTo(methodOn(FracturesController.class)
                .createFractures(hospitalName, fractures, null))
                .withRel("Create an entry in the fracture register"),
            linkTo(methodOn(FracturesController.class)
                .updateFractures(hospitalName, fractures, null))
                .withRel("Update an entry in the fracture register"),
            linkTo(methodOn(FracturesController.class)
                .deleteFractures(hospitalName, boneType, patientsNumber, null))
                .withRel("Delete an entry in the fracture register"));
        return ResponseEntity.ok(fractures);
    }
}
```

* Методы `createFractures`, `updateFractures`, `deleteFractures` остаются в коде без изменений. Они не представлены на слайде.

Класс `WebMvcLinkBuilder` – класс, предназначенный для создания ссылок на классы контроллеров.

Метод `methodOn` – метод, который позволяет динамически генерировать путь к заданному ресурсу путем фиктивного обращения к методу в контроллере. В примере `WebMvcLinkBuilder` использует базовый путь `FracturesController` и путь к методу `getFractures` для формирования URL.

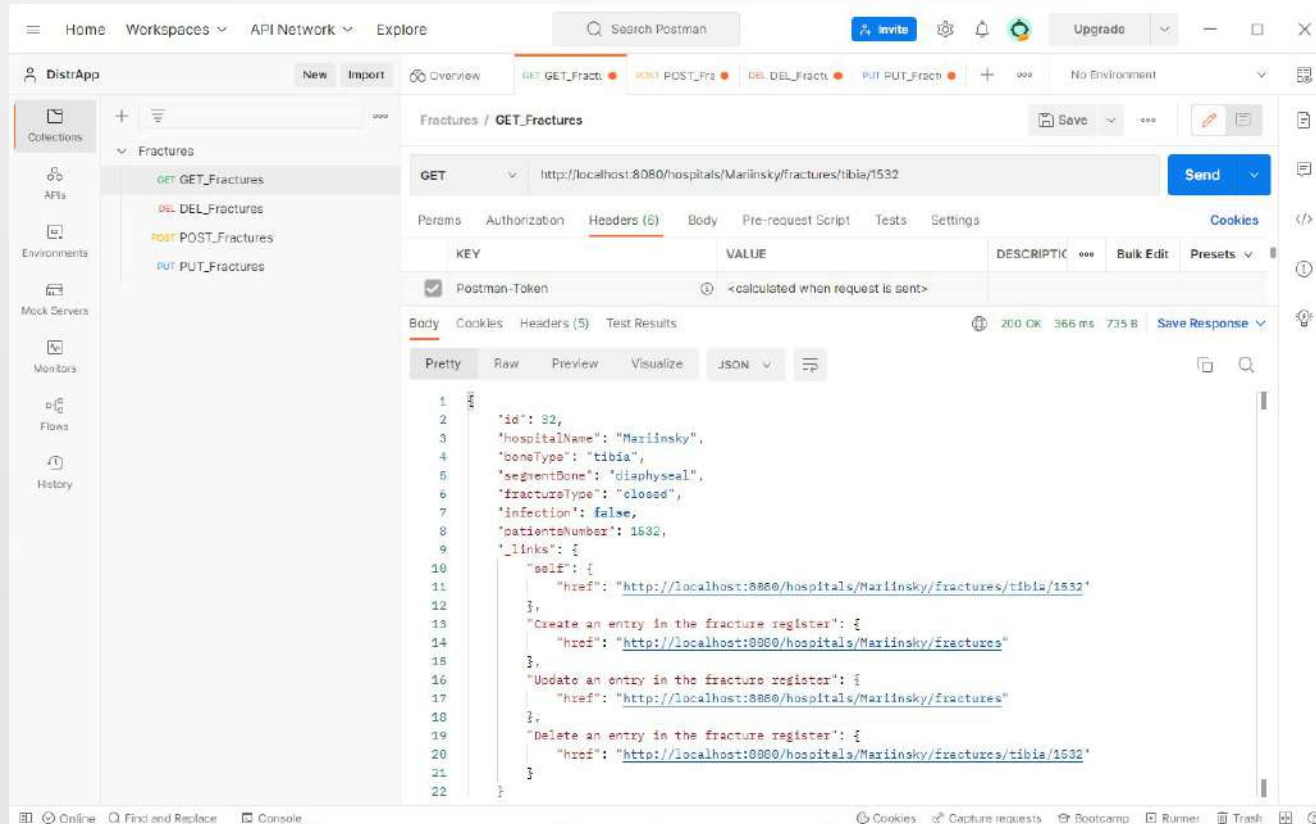
Метод `linkTo` – метод, который отвечает за создание ссылки. В примере `WebMvcLinkBuilder` использует базовый путь `FracturesController` как базовый путь создаваемой ссылки.

Метод `add` – метод, который отвечает за добавление указанной ссылки на ресурс.

Метод `withRef` – метод, который позволяет задать имя для URL-адреса.

Метод `withSelfRef` – метод, который позволяет задать имя для URL-адреса в виде собственной ссылки по умолчанию.

Вызов конечной точки GET через URL



The screenshot displays the Postman application interface. The top navigation bar includes 'Home', 'Workspaces', 'API Network', and 'Explore'. The main workspace is titled 'DistrApp' and shows a collection of API endpoints under the 'Fractures' folder. The selected endpoint is 'GET GET_Fractures' with the URL 'http://localhost:8080/hospitals/Mariinsky/fractures/tibia/1532'. The 'Headers' tab is active, showing a 'Postman-Token' header. The 'Body' tab is also visible, showing a JSON response. The response is displayed in the 'Pretty' view, showing a JSON object with fields like 'id', 'hospitalName', 'boneType', 'segmentBone', 'fractureType', 'infection', 'patientsNumber', and '_links'.

Fractures / GET_Fractures

GET <http://localhost:8080/hospitals/Mariinsky/fractures/tibia/1532> Send

Params Authorization Headers (5) Body Pre-request Script Tests Settings Cookies

KEY	VALUE	DESCRIPTION	Bulk Edit	Presets
<input checked="" type="checkbox"/> Postman-Token	<calculated when request is sent>			

Body Cookies Headers (5) Test Results 200 OK 366 ms 735 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 32,
3   "hospitalName": "Mariinsky",
4   "boneType": "tibia",
5   "segmentBone": "diaphyseal",
6   "fractureType": "closed",
7   "infection": false,
8   "patientsNumber": 1532,
9   "_links": {
10    "self": {
11      "href": "http://localhost:8080/hospitals/Mariinsky/fractures/tibia/1532"
12    },
13    "Create an entry in the fracture register": {
14      "href": "http://localhost:8080/hospitals/Mariinsky/fractures"
15    },
16    "Update an entry in the fracture register": {
17      "href": "http://localhost:8080/hospitals/Mariinsky/fractures"
18    },
19    "Delete an entry in the fracture register": {
20      "href": "http://localhost:8080/hospitals/Mariinsky/fractures/tibia/1532"
21    }
22  }
23 }
```

1. Реализовать в приложении вывод ссылок на выполнение действий с ресурсом (получение, создание, модификация, удаление) в ответе на вызов конечной точки GET с использованием Spring HATEOAS.
2. Предусмотреть наличие перевода имени для URL-адреса на следующие языки:
 - русский;
 - английский;
 - язык согласно номеру варианта (из первой части работы).
2. Выполнить запуск приложения.
3. Выполнить обращение к конечным точкам (запрос GET) с использованием заголовка Accept-language в Postman.



СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

Мониторинг состояния микросервисов

Проект Spring Boot Actuator предоставляет готовые возможности для управления и мониторинга микросервисов с помощью конечных точек HTTP или Java Management Extensions (JMX).

Конечные точки предлагают проверку работоспособности, мониторинг метрик, доступ к журналам, дампам потоков*, информации об окружающей среде и многое другое.

*Дамп потока — это моментальный снимок состояния всех потоков процесса Java.

Добавление зависимости



Добавить зависимость в файл pom.xml (если не выполнено на первом занятии):

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

Виды доступных конечных точек



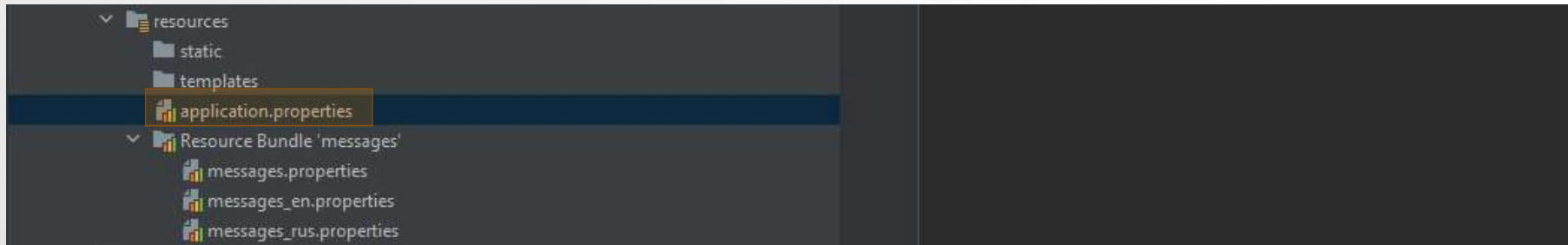
Доступ к большинству конечных точек Spring Boot Actuator через HTTP по умолчанию отключен из соображений безопасности, поскольку они могут содержать конфиденциальную информацию. Доступные конечные точки представлены ниже.

- /actuator
- /actuator/health

Описания всех конечных точек: <https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/>

Файл свойств приложения

Spring Actuator позволяет изменить конфигурацию по умолчанию через файл свойств приложения `application.properties`.



Свойства для включения конечных точек



Включение всех конечных точек:

*management.endpoints.web.exposure.include=**

Включение конкретных конечных точек:

management.endpoints.web.exposure.include=<список конечных точек через запятую>

Включение всех конечных точек, исключая указанные:

*management.endpoints.web.exposure.include=**

management.endpoints.web.exposure.exclude=<список конечных точек через запятую>

Отключение конечных точек и настройка URL



Отключение всех конечных точек:

*management.endpoints.web.exposure.exclude=**

Настройка URL для доступа к конечным точкам (по умолчанию все конечные точки доступны по URL /actuator):

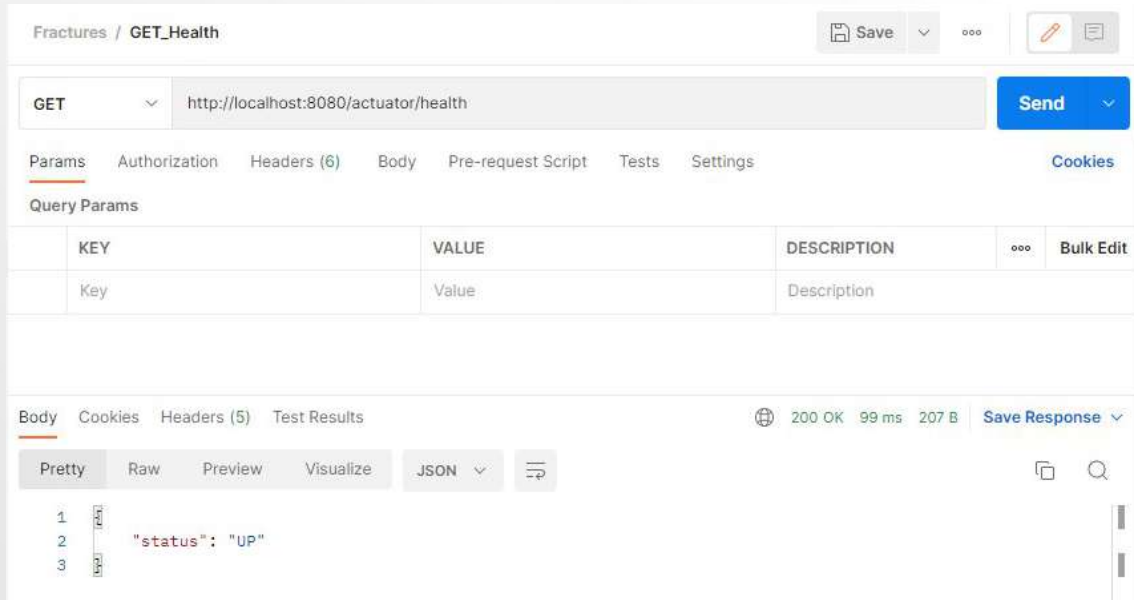
management.endpoints.web.base-path=<новый URL>

Конечная точка /actuator является префиксом для всех конечных точек Actuator.



```
1  {
2    "_links": {
3      "self": {
4        "href": "http://localhost:8080/actuator",
5        "templated": false
6      },
7      "health": {
8        "href": "http://localhost:8080/actuator/health",
9        "templated": false
10     },
11     "health-path": {
12       "href": "http://localhost:8080/actuator/health/{*path}",
13       "templated": true
14     }
15   }
16 }
```

Конечная точка `/actuator/health` предоставляет доступ к основной информации о состоянии приложения.



The screenshot shows a REST client interface with the following details:

- Request:** Method `GET`, URL `http://localhost:8080/actuator/health`.
- Response:** Status `200 OK`, Time `99 ms`, Size `207 B`.
- Response Body (JSON):**

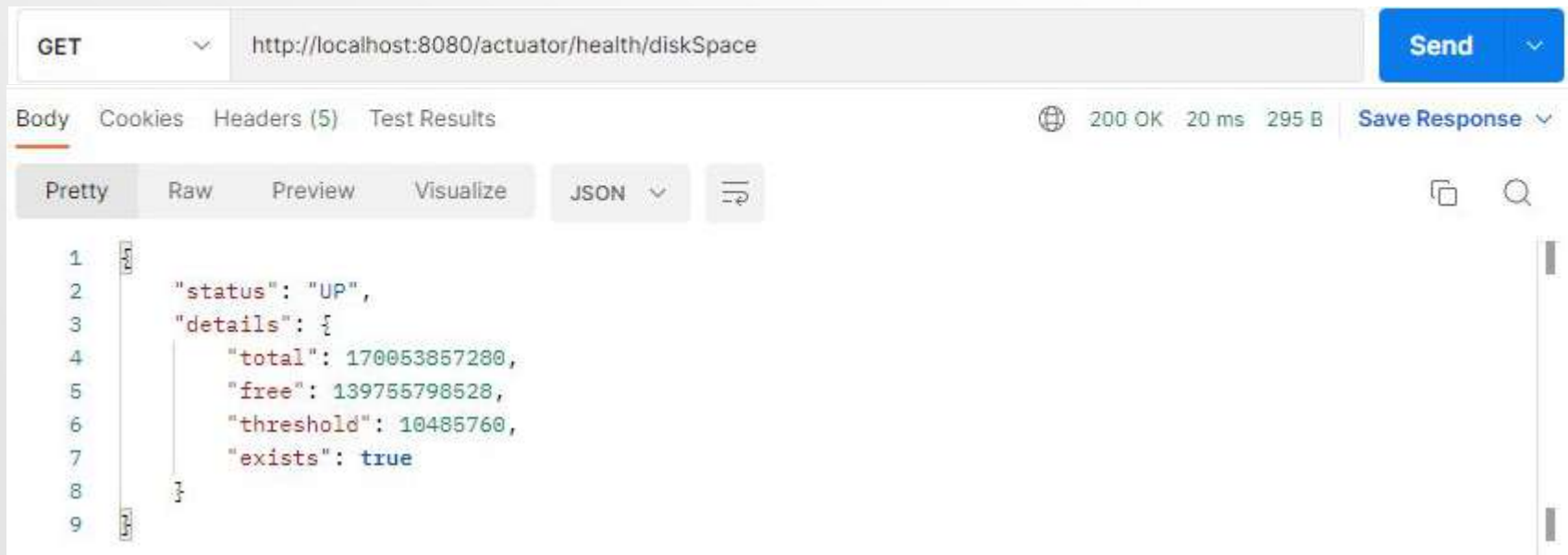
```
1 {  
2   "status": "up"  
3 }
```

Объем информации при использовании конечной точки */actuator/health* регулируется следующими свойствами:

- *management.endpoint.health.show-details=never* (дополнительная информация не отображается);
- *management.endpoint.health.show-details=always* (дополнительная информация доступна для просмотра всем пользователям);
- *management.endpoint.health.show-details=when-authorized* (дополнительная информация доступна для просмотра только авторизованным пользователям с применением свойства *management.endpoint.health.roles*).

/actuator/health/{component}

Конечная точка /actuator/health/{*path} предоставляет сведения о работоспособности конкретного компонента.



Конечная точка /actuator/info предоставляет общую информацию о приложении (по умолчанию выдается пустой JSON).

Способы отображения общей информации:

- информация о сборке;
- информация о пользовательских данных;
- информация о Git;
- ...

Добавить конфигурацию в плагин Spring Boot Maven в файле pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>build-info</id>
          <goals>
            <goal>build-info</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```


Информация о пользовательских данных



С целью получения информации о пользовательских данных выполняется следующее обновление кода в классе инициализации.

@Bean

```
InfoContributor getInfoContributor() {  
    Map<String, Object> details = new HashMap<>();  
    details.put("nameApp", "Fractures");  
    details.put("developer", "Elena Belova");  
    Map<String, Object> wrapper = new HashMap<>();  
    wrapper.put("info", details);  
    return new MapInfoContributor(wrapper);  
}
```

GET

http://localhost:8080/actuator/info

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results

200 OK 272 ms 251 B Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "info": {
3     "nameApp": "Fractures",
4     "developer": "Elena Belova"
5   }
6 }
```

1. Выполнить запросы GET в Postman к следующим конечным точкам:
 - /actuator;
 - /actuator/health с отображением дополнительной информации, доступной для просмотра всем пользователям;
 - /actuator/info с информацией о сборке и пользовательских данных (название приложения, описание, разработчик, e-mail).
2. Изучить описания конечных точек (<https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/>) и вывести следующую информацию через запросы GET в Postman:
 - список метрик и значения конкретных метрик (время для запуска приложения, длительность обработки запроса HTTP-сервером, число доступных процессоров для JVM);
 - данные об окружении;
 - данные о bean-компонентах приложения.