



СПбГЭТУ «ЛЭТИ»  
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

# Создание базы данных PostgreSQL

**PostgreSQL** – это объектно-реляционная система управления базами данных с открытым исходным кодом.

## **Основные возможности PostgreSQL:**

- ✓ поддержка различных типов данных;
- ✓ обеспечение целостности данных;
- ✓ параллелизм и производительность;
- ✓ надежность и аварийное восстановление;
- ✓ безопасность;
- ✓ расширяемость;
- ✓ интернационализация и полнотекстовый поиск;
- ✓ ...

Загрузка PostgreSQL

<https://www.postgresql.org/download/>

Официальная документация на английском языке:

<https://www.postgresql.org/docs/>

Русскоязычная версия официальной документации:

<https://postgrespro.ru/docs/postgresql>

# Особенности установки



## Выбор ОС

### Packages and Installers

Select your operating system family:



## Выбор версии на основании платформы

PostgreSQL Version	64 Bit Windows Platforms	32 Bit Windows Platforms
16	2022, 2019	
15	2019, 2016	
14	2019, 2016	
13	2019, 2016	
12	2019, 2016, 2012 R2	
11	2019, 2016, 2012 R2	
10	2016, 2012 R2 & R1, 7, 8, 10	2008 R1, 7, 8, 10

# Загрузка инсталлятора для ОС Windows



## Windows installers

### Interactive installer by EDB

**Download the installer** certified by EDB for all supported PostgreSQL versions.

**Note!** This installer is hosted by EDB and not on the PostgreSQL community servers. If you have issues with the website it's hosted on, please contact [webmaster@enterprisedb.com](mailto:webmaster@enterprisedb.com).

This installer includes the PostgreSQL server, pgAdmin, a graphical tool for managing and developing your databases, and StackBuilder, a package manager that can be used to download and install additional PostgreSQL tools and drivers. Stackbuilder includes management, integration, migration, replication, geospatial, connectors and other tools.

This installer can run in graphical or silent install modes.

The installer is designed to be a straightforward, fast way to get up and running with PostgreSQL on Windows.

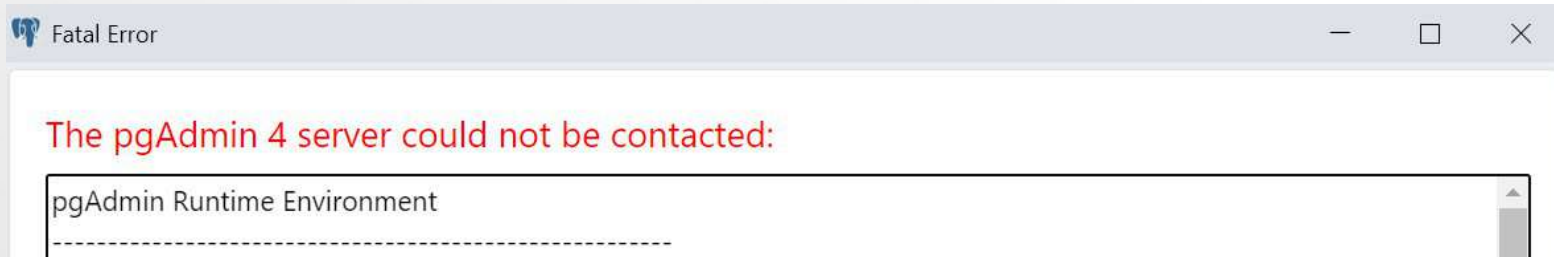
*Advanced users* can also download a **zip archive** of the binaries, without the installer. This download is intended for users who wish to include PostgreSQL as part of another application installer.

С целью администрирования и разработки для PostgreSQL и связанных с ней СУБД используется платформа с открытым исходным кодом – pgAdmin.

# Решение проблемы запуска pgAdmin4



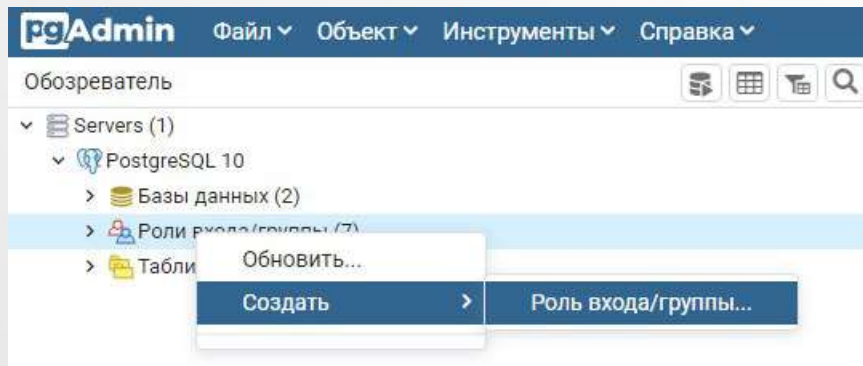
## Ошибка при старте pgAdmin4



## Решение:

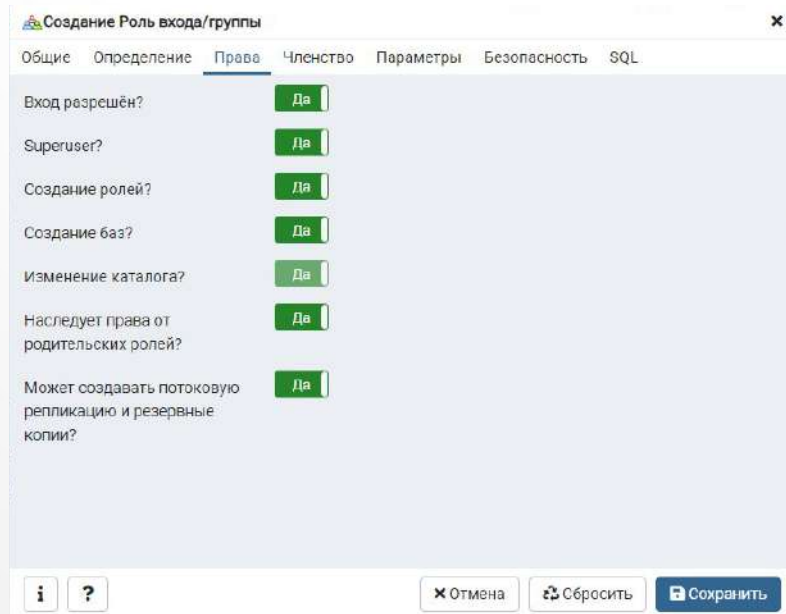
- ✓ удалить PostgreSQL;
- ✓ очистить содержимое папки C:\Users\{UserName}\AppData\Roaming\pgadmin;
- ✓ установить PostgreSQL без pgAdmin4;
- ✓ Установить версию pgAdmin4 в соответствии с выбранной платформой с сайта <https://www.pgadmin.org/download/pgadmin-4-windows/>

# Создание роли

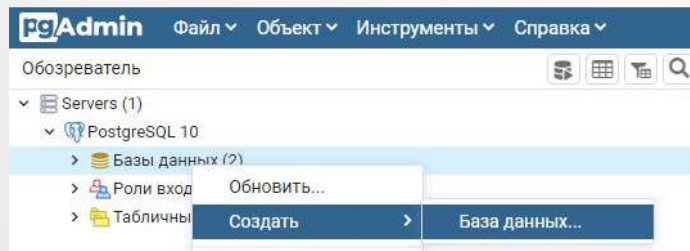


В процессе создания роли необходимо задать:

- ✓ имя;
- ✓ пароль;
- ✓ уровень прав (выбрать права на все возможные действия).



# Создание БД и таблицы в БД

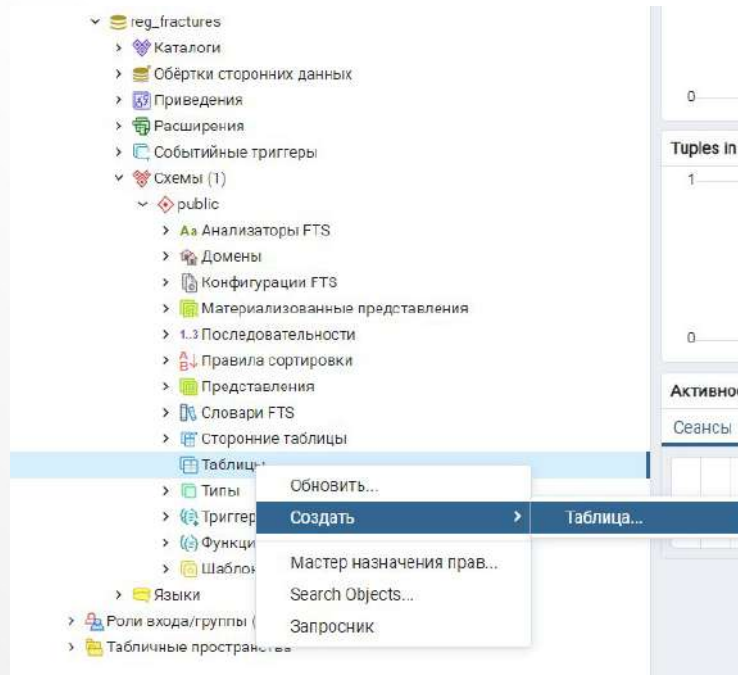


В процессе создания БД необходимо задать:

- ✓ имя БД;
- ✓ имя владельца (в соответствии с созданной ролью).

В процессе создания таблицы в БД необходимо задать:

- ✓ имя таблицы;
- ✓ имя владельца (в соответствии с созданной ролью);
- ✓ информацию о каждом из столбцов таблицы.





1. Скачать, установить и запустить PostgreSQL.
2. Создать новую роль с полными правами.
3. Создать базу данных, указав в качестве владельца созданную роль.
4. Создать таблицу в базе данных, указав в качестве владельца созданную роль. Столбцы таблицы должны соответствовать переменным класса модели.



СПбГЭТУ «ЛЭТИ»  
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

## Создание сервера конфигураций

# Разделение конфигурации и кода приложения



**Конфигурация приложения** – это совокупность настроек приложения, задаваемая пользователем, а также процесс изменения этих настроек в соответствии с нуждами пользователя.

Конфигурация может существенно различаться между развёртываниями, код различаться не должен.

Полное отделение конфигурации от кода приложения позволяет разработчикам и сопровождающим специалистам изменять конфигурацию без необходимости повторно компилировать и развертывать приложение.

1. Экземпляр микросервиса в момент запуска обращается к конечной точке службы для получения своей конфигурации, которая зависит от окружения, в котором запускается микросервис.
2. Конфигурация хранится в репозитории (файлы в системе управления версиями, реляционные базы данных или хранилища данных типа ключ/значение).
3. Управление конфигурацией приложения осуществляется независимо от развертывания приложения.
4. Приложения должны получать уведомления об изменении конфигурации.

# Spring Cloud Configuration Server



Spring Cloud Configuration Server – решение для управления конфигурациями с открытым исходным кодом, предлагающее несколько базовых механизмов хранения (общие файловые системы, Eureka, Consul и Git).

Spring Cloud Configuration Server предоставляет поддержку для внешней конфигурации на стороне сервера и клиента. Экземпляр Spring Cloud Configuration Server обеспечивает централизованное управление внешними свойствами приложений во всех средах.

# Создание проекта с помощью Spring Initializr



## Project

☐ Gradle - Groovy ☐ Gradle - Kotlin

## Language

☒ Java ☐ Kotlin ☐ Groovy

☒ Maven

## Spring Boot

☐ 3.2.0 (SNAPSHOT) ☐ 3.2.0 (RC2) ☐ 3.1.6 (SNAPSHOT) ☒ 3.1.5

☐ 3.0.13 (SNAPSHOT) ☐ 3.0.12 ☐ 2.7.18 (SNAPSHOT) ☐ 2.7.17

## Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☒ 21 ☐ 17 ☐ 11 ☐ 8

## Dependencies

[ADD DEPENDENCIES...](#) CTRL + B

### Config Server SPRING CLOUD CONFIG

Central management for configuration via Git, SVN, or HashiCorp Vault.

### Spring Boot Actuator OPS

Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.



[GENERATE](#) CTRL + G

[EXPLORE](#) CTRL + SPACE

[SHARE...](#)

# Коррекция версий в файле pom.xml



```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.1.3</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
...
<properties>
  <java.version>20</java.version>
  <spring-cloud.version>2022.0.4</spring-cloud.version>
</properties>
```

# Сведения о Spring Cloud в файле pom.xml



## Версия Spring Cloud

<properties>

<java.version>20</java.version>

<spring-cloud.version>2022.0.4</spring-cloud.version>

</properties>

## Зависимость Spring Cloud

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>



# Определение Spring Cloud BOM в файле pom.xml



```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

**Spring Cloud Config BOM** (Bill of Materials) – это родительская ведомость материалов, которая включает все сторонние библиотеки и зависимости, используемые в проекте, а также номера версий отдельных проектов.

## Преимущества:

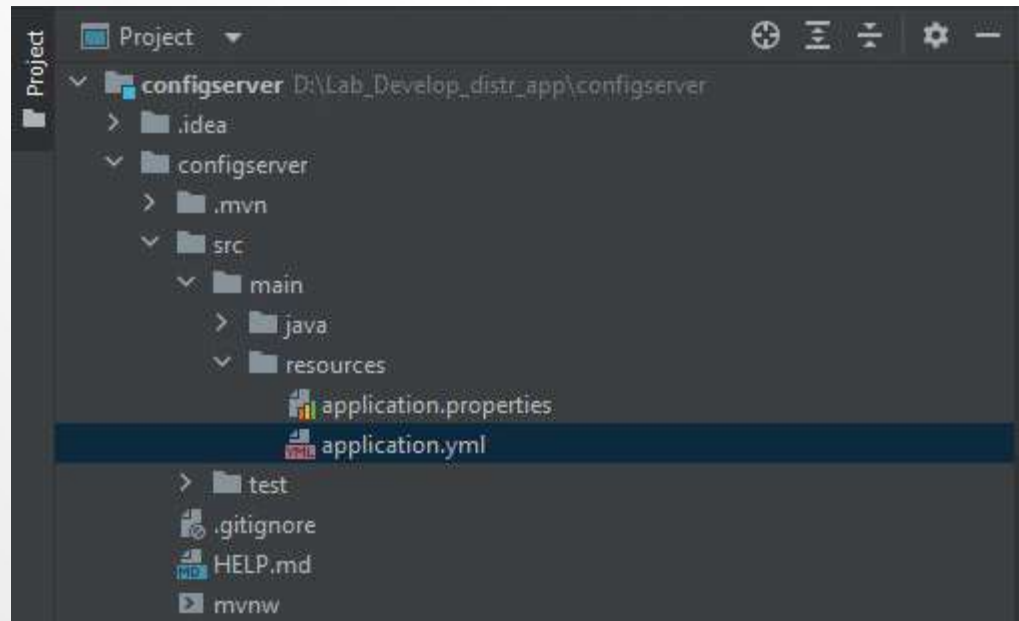
- ✓ гарантия использования совместимых версий подпроектов Spring Cloud;
- ✓ отсутствие необходимости объявления номеров версий для вложенных зависимостей.

# Создание файла свойств с настройками сервера



Щелкнуть ПКМ на пакете `src/main/resources` и создать файл `application.yml`.

Spring Boot позволяет хранить настройки приложения в файле и получать к ним доступ. Файл свойств может иметь один из трёх форматов: `properties`, `xml` и `yaml`.



# Содержимое файла свойств



```
spring:
  application:
    name: config-server
  profiles:
    active:
      native
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/config
server:
  port: 8071

management:
  endpoints:
    web:
      exposure:
        include: "*"

```

**config-server** – имя приложения, используемое для обнаружения службы.

**native** – профиль для Spring Cloud Configuration Server, который указывает на извлечение файлов конфигурации приложения из пути поиска классов (classpath) или из файловой системы.

**classpath:/config** – указание конкретного пути к классам.

**8071** - порт, который прослушивает Spring Configuration Server с целью ожидания запросов на получение конфигурации.

**include: "\*" -** включение всех конечных точек Spring Boot Actuator.

# Обновление класса инициализации



```
package com.hospitals.configserver;
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.cloud.config.server.EnableConfigServer;
```

```
@SpringBootApplication
```

```
@EnableConfigServer
```

```
public class ConfigServerApplication {
```

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

```
        SpringApplication.run(ConfigServerApplication.class, args);
```

```
    }
```

```
}
```

**@EnableConfigServer** -  
это аннотация, которая  
объявляет данную службу  
службой Spring Cloud  
Config

# Создание конфигураций для службы



Щелкнуть ПКМ на пакете `src/main/resources` и создать папку `config`. В данной папке необходимо создать три файла:

- ✓ `fractures.properties` – конфигурация для окружения по умолчанию в целях локального запуска службы;
- ✓ `fractures-dev.properties` – конфигурация для окружения разработки;
- ✓ `fractures-prod.properties` – конфигурация для промышленного окружения.

# Содержимое файла fractures.properties



example.property= Default environment

**# Инициализация базы данных с помощью Hibernate**

**spring.jpa.hibernate.ddl-auto=none**

spring.jpa.database=POSTGRESQL

spring.datasource.platform=postgres

spring.jpa.show-sql = true

**# Указание стратегии для сопоставления сущности java и имени атрибута с соответствующей реляционной базой данных и именем столбцов**

**spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy**

**# Указание диалекта для использования (сопоставление между типами данных в Java и типами данных в БД)**

**spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect**

spring.database.driverClassName= org.postgresql.Driver

**#Указание на сохранение соединения, даже в случае длительного простоя**

**spring.datasource.testWhileIdle = true**

**spring.datasource.validationQuery = SELECT 1**

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

management.endpoints.enabled-by-default=true

# Содержимое файлов конфигураций



## Файл fractures-dev.properties

example.property= Development environment

spring.datasource.url = jdbc:postgresql://localhost:5432/reg\_fractures

Имя БД

spring.datasource.username = Elena

spring.datasource.password = tatlip13542

Имя и пароль роли

## Файл fractures-prod.properties

example.property= Production environment

spring.datasource.url = jdbc:postgresql://localhost:5432/reg\_fractures

spring.datasource.username = Elena

spring.datasource.password = tatlip13542

# Запрос конфигурации для окружения по умолчанию

Fractures / GET\_ConfServer\_default

GET http://localhost:8071/fractures/default

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

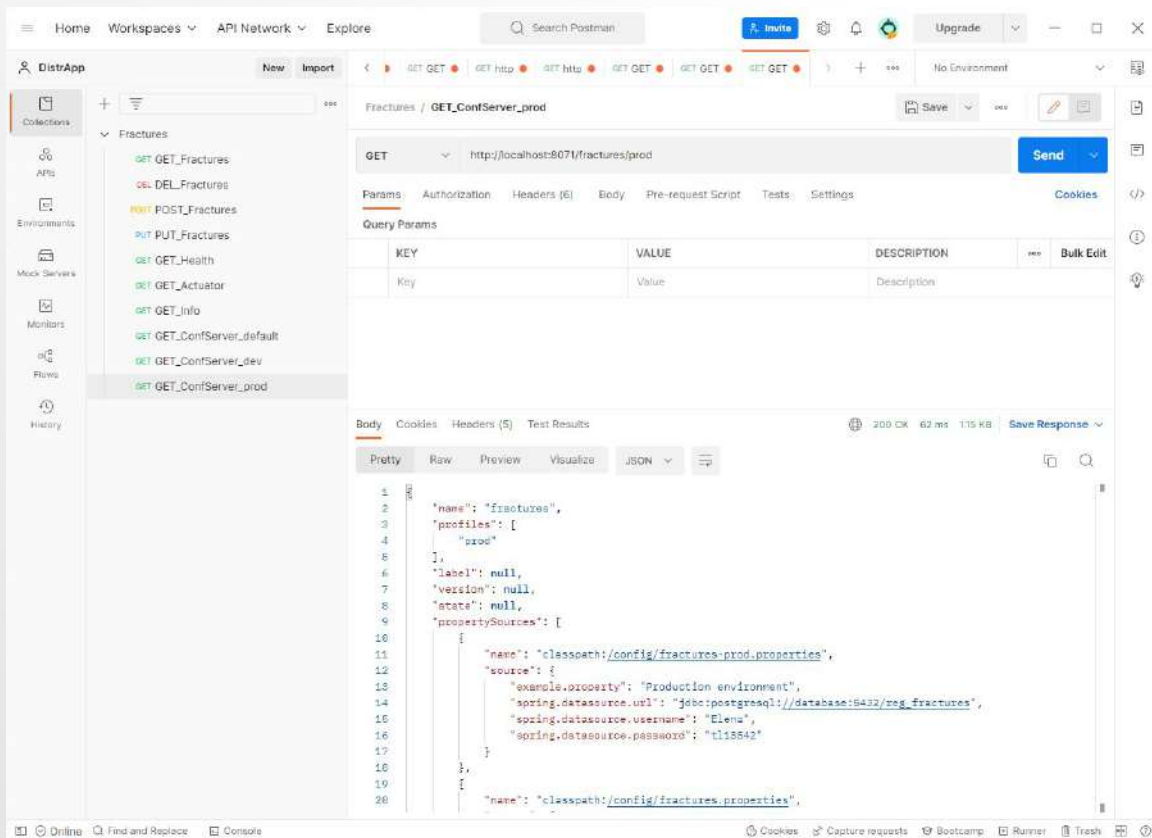
Body Cookies Headers (5) Test Results 200 OK 441 ms 922 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "name": "fractures",
3   "profiles": [
4     "default"
5   ],
6   "label": null,
7   "version": null,
8   "state": null,
9   "propertySources": [
10    {
11      "name": "classpath:/config/fractures.properties",
12      "source": {
13        "example.property": "Default environment",
14        "spring.jpa.hibernate.ddl-auto": "none",
15        "spring.jpa.database": "POSTGRESQL",
16        "spring.datasource.platform": "postgres",
17        "spring.jpa.show-sql": "true",
18        "spring.jpa.hibernate.naming-strategy": "org.hibernate.cfg.ImprovedNamingStrategy",
19        "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",
20        "spring.database.driverClassName": "org.postgresql.Driver",
```



# Запрос конфигурации для промышленного окружения



The screenshot shows the Postman interface with a GET request to `http://localhost:8071/fractures/prod`. The response body is a JSON object representing configuration data for a production environment.

```
1 {
2   "name": "fractures",
3   "profiles": [
4     "prod"
5   ],
6   "label": null,
7   "version": null,
8   "state": null,
9   "propertySources": [
10    {
11      "name": "classpath:/config/fractures-prod.properties",
12      "source": {
13        "example.property": "Production environment",
14        "spring.datasource.url": "jdbc:postgresql://database:5432/reg_fractures",
15        "spring.datasource.username": "Elena",
16        "spring.datasource.password": "t1s18542"
17      }
18    },
19    {
20      "name": "classpath:/config/fractures.properties",
21      "source": {
22        "example.property": "Development environment",
23        "spring.datasource.url": "jdbc:postgresql://database:5432/reg_fractures",
24        "spring.datasource.username": "Elena",
25        "spring.datasource.password": "t1s18542"
26      }
27    }
28  ]
29 }
```

1. Создать проект Spring Boot с помощью Spring Initializr для сервера конфигурации.
2. Создать файл свойств приложения.
3. Создать файлы конфигурации для трех различных окружений.
4. Реализовать класс инициализации.
5. Выполнить запросы GET к конечным точкам с использованием Postman, ответы на которые содержат сведения о конфигурации для трех различных окружений.



СПбГЭТУ «ЛЭТИ»  
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

# Взаимодействие микросервисного приложения с базой данных PostgreSQL через сервер конфигураций

# Версия Spring Cloud



Добавить в файл pom.xml сведения о версии Spring Cloud:

```
<properties>
```

```
    <java.version>20</java.version>
```

```
    <spring.cloud-version>2022.0.4</spring.cloud-version>
```

```
</properties>
```

Подробнее о совместимости версий Spring Cloud и Spring Boot:

<https://spring.io/projects/spring-cloud>

# Добавление зависимостей в файл pom.xml



```
<dependency>
```

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

```
  <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.springframework.cloud</groupId>
```

```
  <artifactId>spring-cloud-starter-config</artifactId>
```

```
</dependency>
```

```
<dependency>
```

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

```
  <artifactId>spring-boot-configuration-processor</artifactId>
```

```
  <optional>true</optional>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.postgresql</groupId>
```

```
  <artifactId>postgresql</artifactId>
```

```
</dependency>
```

Использование Spring Data Java Persistence API (JPA)

Загрузка зависимости для взаимодействий со службой Spring Cloud Config Server

Использование процессора аннотаций для генерирования метаданных конфигурации

Загрузка драйверов JDBC для PostgreSQL

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>2022.0.4</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

# Соединение с Spring Cloud Configuration Server



Щелкнуть ПКМ на пакете `src/main/resources` и создать файл `application.yml`.

spring:

application:

name: fractures

Имя приложения, которое использует Spring Cloud Config  
для его поиска

profiles:

active: dev

Профиль по умолчанию, определяющий конфигурацию окружения, в  
котором запускается служба

config:

import: optional:configserver:http://localhost:8071

Местоположение службы Spring Cloud  
Config Server

# Обновление класса модели



Класс модели – это класс, который хранит данные, полученные из таблицы базы данных PostgreSQL.

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

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;
import org.springframework.hateoas.RepresentationModel;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Getter @Setter @ToString
@Entity
@Table(name="info_fractures")

public class Fractures extends RepresentationModel<Fractures> {

    @Id
    @Column(name = "fractures_id", nullable = false)
    private int id;
```

```
    @Column(name = "hospital_name", nullable = false)
    private String hospitalName;
    @Column(name = "bone_type", nullable = false)
    private String boneType;
    @Column(name = "bone_segment", nullable = false)
    private String segmentBone;
    @Column(name = "fracture_type", nullable = false)
    private String fractureType;
    @Column(name = "infection", nullable = false)
    private boolean infection;
    @Column(name = "number_patients", nullable = false)
    private int patientsNumber;
    @Column(name="comment")
    private String comment;
    public com.hospitals.fractures.model.Fractures withComment(String comment) {
        this.setComment(comment);
        return this;
    }
}
```



Аннотации JPA указывают фреймворку Spring Data на необходимость отобразить данные из таблицы в базе данных PostgreSQL в объект Java.

**@Entity** – это аннотация уровня класса, указывающая, что данный класс является классом сущности.

**@Table** – это аннотация, указывающая таблицу в базе данных, которая сопоставляется с этой сущностью.

**@Id** – это аннотация, которая указывает на то, что соответствующий столбец в базе данных является первичным ключом.

**@Column** – это аннотация, которая определяет столбец в таблице, соответствующий данному свойству. Если свойство имеет то же имя, что и столбец в базе данных, то аннотация @Column не добавляется.

Spring проводит анализ имен методов при запуске из интерфейса Repository, преобразует их в оператор SQL в соответствии с именами и генерирует динамический прокси-класс для выполнения работы.

Создание класса FracturesService производится следующим образом:

- щелкнуть ПКМ на пакете `com.hospitals.fractures` и создать новый пакет `repository`;
- щелкнуть ПКМ на пакете `repository` и создать новый класс `FracturesRepository`.

# Код интерфейса Repository



Spring проводит анализ имен методов при запуске из интерфейса Repository, преобразует их в оператор SQL в соответствии с именами и генерирует динамический прокси-класс для выполнения работы.

```
package com.hospitals.fractures.repository;
```

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

```
import org.springframework.stereotype.Repository;
```

```
import org.springframework.data.repository.CrudRepository;
```

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

```
import java.util.List;
```

```
@Repository
```

```
public interface FracturesRepository
```

```
    extends CrudRepository<Fractures,String> {
```

```
    public List<Fractures> findByHospitalName
```

```
        (String hospitalName);
```

```
    public Fractures findByHospitalNameAndBoneType
```

```
        (String hospitalName,
```

```
        String boneType);
```

```
}
```

**@Repository** - это аннотация, которая указывает фреймворку Spring, что интерфейс необходимо рассматривать как интерфейс репозитория и сгенерировать для него прокси-класс.

**Интерфейс CrudRepository** – это интерфейс, который определяет стандартные методы CRUD для создания, чтения, обновления и удаления сущностей.

**Методы findByHospitalName и findByHospitalNameAndBoneType** – это методы запросов, которые извлекают данные из таблицы. Фреймворк Spring Data преобразует методы запросов в запросы SELECT...FROM.

# Обновление кода в классе сервиса



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

```
import ...
```

```
@Service
```

```
public class FracturesService {
```

```
    @Autowired
```

```
    MessageSource messages;
```

```
    @Autowired
```

```
    private FracturesRepository fracturesRepository;
```

```
    @Autowired
```

```
    ServiceConfig config;
```

```
    public Fractures getFractures(String hospitalName, String boneType){
```

```
        Fractures fractures = fracturesRepository.findByIdAndLicenseId(boneType, hospitalName);
```

```
        if (null == fractures) {
```

```
            throw new IllegalArgumentException(String.format(messages.getMessage("fractures.search.error.message", null, null), boneType, hospitalName));
```

```
        }
```

```
        return fractures.withComment(config.getProperty());
```

```
    }
```

# Обновление кода в классе сервиса

```
public Fractures createFractures(Fractures fractures){  
    fracturesRepository.save(fractures);  
    return fractures.withComment(config.getProperty());  
}
```

Сохраняет указанную сущность fractures

```
public String deleteFractures(String hospitalName, String boneType, Locale locale) {  
    String responseMessage = null;  
    Fractures fractures = new Fractures();  
    fractures.setHospitalName(hospitalName);  
    fractures.setBoneType(boneType);  
    fracturesRepository.delete(fractures);  
    responseMessage = String.format(messages.getMessage("fractures.delete.message", null, locale), boneType, hospitalName);  
    return responseMessage;  
}
```

Удаляет указанную сущность fractures

## **messages\_en.properties**

fractures.search.error.message = Unable to find entry in the fracture register with the type of %s bone in the %s

fractures.delete.message = Deleting entry in the fracture register with the type of %s bone in the %s

## **messages\_rus.properties**

fractures.search.error.message = Не найдена запись в реестре переломов с типом кости - %s - в медицинской организации %s

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

# Создание класса ServiceConfig



Класс ServiceConfig – это класс, который используется для хранения настроек микросервисного приложения.

Создание класса FracturesService производится следующим образом:

- щелкнуть ПКМ на пакете com.hospitals.fractures и создать новый пакет config;
- щелкнуть ПКМ на пакете repository и создать новый класс ServiceConfig.



# Код класса ServiceConfig



```
package com.hospitals.fractures.config;

import lombok.Getter;
import lombok.Setter;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
@ConfigurationProperties(prefix = "example")
```

```
@Getter
@Setter
public class ServiceConfig {
    private String property;
}
```

**@Configuration** – это аннотация, которая находится на уровне класса и указывает Spring, что класс является классом конфигурации, содержит один или несколько **@Bean** и может обрабатываться контейнером Spring для генерации определений bean-компонентов.

**@ConfigurationProperties** – это аннотация уровня класса или метода, которая используется с целью доступа к связанным группам свойств путем сохранения конфигурации в объект Java и использования объекта для дальнейшего доступа.

В примере **@ConfigurationProperties(prefix= "example")** извлекает все свойства **example** из Spring Cloud Configuration Server и вставляет их в атрибут **property** класса **ServiceConfig**.

@SpringBootApplication

@RefreshScope

```
public class FracturesApplication {
```

*код методов внутри класса остается неизменным*

```
}
```

Аннотация @RefreshScope – это аннотация, которая позволяет получить доступ к конечной точке /refresh и указать приложению Spring Boot на необходимость проведения повторного чтения своей конфигурации с целью её обновления после внесения изменений.

Приложения Spring Boot читают конфигурацию только в момент запуска, поэтому изменения в конфигурации из Spring Cloud Configuration Server, не отражаются в приложении автоматически.

# Код класса контроллера

```
import ...

@RestController
@RequestMapping(value="hospitals/{hospitalName}/fractures")
public class FracturesController {
    @Autowired
    private FracturesService fracturesService;

    @GetMapping(value="/{boneType}")
    public ResponseEntity<Fractures> getFractures(
        @PathVariable("hospitalName") String hospitalName,
        @PathVariable("boneType") String boneType) {
        Fractures fractures = fracturesService.getFractures(hospitalName, boneType);
        fractures.add(
            linkTo(methodOn(FracturesController.class)
                .getFractures(boneType, fractures.getHospitalName()))
                .withSelfRel(),
            linkTo(methodOn(FracturesController.class)
                .createFractures(fractures))
                .withRel("Create an entry in the fracture register"),
            linkTo(methodOn(FracturesController.class)
                .updateFractures(fractures))
                .withRel("Update an entry in the fracture register"),
            linkTo(methodOn(FracturesController.class)
                .deleteFractures(fractures.getHospitalName(), boneType,null))
                .withRel("Delete an entry in the fracture register"));
        return ResponseEntity.ok(fractures);
    }
}
```

# Код класса контроллера



```
@PostMapping
```

```
public ResponseEntity<Fractures> createFractures(  
    @RequestBody Fractures request) {  
    return ResponseEntity.ok(fracturesService.createFractures(request));  
}
```

```
@DeleteMapping(value="/{boneType}")
```

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

```
}
```

1. Выполнить запуск сервера конфигураций.
2. Выполнить запуск сервера PostgreSQL, используя pgAdmin.
3. Добавить в файл pom.xml сведения о версии Spring Cloud и необходимые зависимости.
4. Создать файл с настройками для подключения к серверу конфигураций.
5. Создать интерфейс Repository и класс ServiceConfig.
6. Модифицировать код в классах инициализации, модели, сервиса и контроллера.
7. Выполнить запуск приложения.
8. Выполнить обращение к конечным точкам (запросы POST, GET, PUT, DELETE) с использованием Postman.
9. Продемонстрировать изменение данных в таблице базы данных с использованием pgAdmin.