Geekbrains

**Разработка web-сервиса для обеспечения топо-геодезических работ исходными данными для построения опорных и съёмочных сетей.**

IT-специалист:

Веб-разработка на Java

Низовкин А.В.

Шахты

2024

# Содержание.

1. Титульный лист

2. Содержание

3. Введение

4. Основная часть

5. Заключение

6. Список литературы и ресурсов

7. Приложения

# Введение.

Целью топо-геодезических работ является создание цифровой модели местности, которая в свою очередь определяется в пространстве набором точек с трёхмерными координатами.

Для этих целей используется различное современное оптико-электронное и спутниковое оборудование, но в любом случае исходными данными для этих работ является опорная геодезическая сеть.

Геодезическая сеть — совокупность специально обозначенных (закреплённых) точек земной поверхности (геодезических пунктов), положение которых определено в общей для них системе координат. Применяется в целях установления, распространения и связи предусмотренных геодезических систем координат и высот. Геодезические сети создают по принципу перехода от сетей более высокоточных и масштабных к сетям с меньшими расстояниями и менее точными измерениями

Геодезический пункт — точка, особым образом закреплённая на местности (в грунте, на строении или другом искусственном сооружении) и являющаяся носителем координат, определённых геодезическими методами.

Настоящий проект ставит перед собой целью разработку бэкэнда веб-приложения для работы с базой данных опорных геодезических пунктов, со следующей функциональностью:

* получение списка всех пунктов
* добавление пункта в базу данных
* изменение атрибутов пункта с заданным идентификатором
* удаление пункта с заданным идентификатором
* получение пункта по уникальному идентификатору
* получение списка пунктов по названию
* получение списка пунктов по номенклатуре листа
* получение списка пунктов в заданном районе

В проекте используются следующие инструменты и технологии:

**-** рабочая машина под управлением операционной системы **Ubuntu 20.04**

- разработка пояснительной записки к проекту в программе **LibreOffice Writer**

- формат пояснительной записки Word 2007-365(.docx)

- среда разработки **Intellij Idea(Community Edition)**

- система контроля версий **git**

- хранение файлов проекта публичный репозиторий на онлайн платформе **Github**

- реализация клиент-серверной архитектуры программы средствами фрэймворка **Spring**

- база данных **Postgres**

- эмуляция клиентской части средствами **Postman**

- документирование API средствами **Swagger**

- разработка Unit-тестов средствами библиотеки **Junit-5**

- разработка интеграционных тестов средствами фрэймворка **Mockito**

# Основная часть.

Разделы основной части.

## Хранение данных.

Для хранения данных о геодезических пунктах проектом предусматривается использовать базу данных Postgres.

Чтобы развернуть базу данных на рабочей машине, будем использовать контейнер docker. Для этого подбираем подходящий образ на сайте hub.docker.com.

Команду, запускающую наш контейнер прописываем в файле run-postgres.sh:

docker run -d --name postgres-container -e POSTGRES\_DB=basepointsdb -e POSTGRES\_USER=andrew -e POSTGRES\_PASSWORD=1234 -p 5432:5432 -v /home/andrew/basepointsdb:/var/lib/postgresql/data postgres:alpine3.19

где:

-d – ключ, запускающий контейнер в режиме демона

--name postgres-container – название создаваемого контейнера

-e POSTGRES\_DB=basepointsdb – переменная окружения, определяющая название создаваемой базы данных

-e POSTGRES\_USER=andrew – переменная окружения, определяющая имя пользователя

-e POSTGRES\_PASSWORD=1234 – переменная окружения, определяющая пароль для входа в субд для пользователя andrew

-p 5432:5432 – указываем соответствие порта в контейнере порту на хост-машине

-v /home/andrew/basepointsdb:/var/lib/postgresql/data – указываем место на хост-машине, в котором будет сохраняться наша база данных. Это позволит сохранять данные между сеансами работы с бд

postgres:alpine3.19 – название выбранного на docker.hub образа

Для запуска созданного скрипта необходимо добавить права доступа на выполнение для файла. Для этого выполним в терминале команду:

sudo chmod +x run-postgres.sh

После этого запускаем скрипт терминальной командой из папки в которой он расположен:

./run-postgres.sh

Результат работы скрипта можно посмотреть с помощью терминальной команды, позволяющей увидеть список запущенных контейнеров:

docker ps

После этого, входим в контейнер в интерактивном режиме и убеждаемся в работе субд:

docker exec -it postgres-container psql -U andrew basepointsdb

\dt

где:

\dt – отображает содержимое базы данных

## Разработка приложения.

### 1. Проектирование API.

Проектом предусматривается наличие в приложении следующих эндпоинтов для работы с базой данных геодезических пунктов:

1. GET /point: Получение списка всех геодезических опорных пунктов (далее пунктов), хранящихся в базе данных
2. GET /point/{id}: Получение информации о конкретном пункте. Здесь {id} – это уникальный идентификатор пункта.
3. GET /point/name/{name}: Получение списка всех пунктов с названием, совпадающим с переданным параметром {name}. Название пункта не является уникальным. Пункты с одинаковым названием могут иметь разные координаты, разный класс точности и расположены на разных листах карт.
4. GET /point/sheet/{sheet}: Получение списка всех пунктов, расположенных на листах карты с номенклатурой (см. Приложение А), совпадающей с переданным параметром {sheet}.
5. PUT /point/{id}: Обновление информации о существующем пункте. Здесь {id} – это уникальный идентификатор пункта, а обновлённые данные о пункте будут отправляться в теле запроса в формате JSON.
6. POST /point: Добавление нового пункта. Здесь данные о новом пункте будут отправляться в теле запроса в формате JSON.
7. POST/point/area: Получение списка всех пунктов, расположенных в конкретном районе. Здесь данные о районе, заданном координатами юго-западного угла и его размерами, отправляются в теле запроса в формате JSON.
8. DELETE /point/{id}: Удаление пункта из базы данных. Здесь {id} – это уникальный идентификатор пункта.

### 2. Проектирование структуры данных для пункта.

Каждый геодезический пункт будет иметь следующие атрибуты:

* id: Уникальный идентификатор пункта.
* name: Название пункта.
* x: Координата X пункта.
* y: Координата Y пункта.
* z: Координата Z пункта.
* sheet: Номенклатура листа карты масштаба 1:100 000, на котором расположен пункт (или наименование объекта для крупномасштабных съёмок).
* accuracyClass: Класс точности сети пункта.(см. Приложение Б)
* coordinateSystem: Система координат пункта.

### 3. Создание и настройка нового Spring проекта.

На сайте Spring Initializr выбираем следующие параметры:

- Project: Maven Project

- Language: Java

-Spring Boot: Выбираем последнюю стабильную версию

- Project Metadata: Вводим информацию о своём проекте

Добавляем следующие зависимости:

- Spring Web: Для создания веб-приложения с использованием Spring MVC.

- Spring Data JPA: Для работы с базой данных через JPA(Java Persistence API). Это спецификация для управления, доступа и сохранения Java объектов в базу данных. Это способ взаимодействия с базой данных с помощью объектно-ориентированного подхода.

- Lombok: Библиотека аннотаций Java, которая помогает сократить количество шаблонных кодов.

- PostgreSQL: Драйвер JDBC и R2 ODBC, который позволяет Java-программам подключаться к базе данных PostgreSQL, используя стандартный, независимый от базы данных Java-код.

После выбора всех параметров, генерируем наш проект, нажав кнопку «Generate», скачиваем и распаковываем архив в рабочую папку проекта.

Для подключения нашего проекта к базе данных, сервер которой уже запущен в docker-контейнере добавляем в файл srс/main/resources/application.yml следующие строки:

datasource:  
 url: jdbc:postgresql://localhost:5432/basepointsdb  
 username: andrew  
 password: 1234  
 driver-class-name: org.postgresql.Driver  
jpa:  
 hibernate:  
 ddl-auto: create-drop  
 show-sql: true

Здесь мы указываем параметры, использованные при запуске контейнера с базой данных:

url: адрес, порт и название базы данных

username: имя пользователя

password: пароль

ddl-auto: параметр библиотеки Hibernate, контролирующий автоматическое создание и обновление схемы базы данных. В нашем случае, при разработке и отладке приложения используем значение «create-drop», при котором Hibernate будет удалять существующую схему базы данных при каждом запуске и после завершения работы.

show-sql: этот параметр позволит отображать sql-запросы, что удобно при разработке и отладке приложения.

### 4. Слой model

Для работы с базой данных через JPA, создадим класс-сущность, который соответствует таблице в базе данных.

Класс BasePoint.java

import jakarta.persistence.\*;  
import lombok.Data;  
import lombok.NoArgsConstructor;

@Entity  
@Data  
@Table(name = "points")  
@NoArgsConstructor  
public class BasePoint {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private long id;  
  
 @Column(name = "name", nullable = false)  
 private String name;  
  
 @Column(name = "x", nullable = false)  
 private long x;  
  
 @Column(name = "y", nullable = false)  
 private long y;  
  
 @Column(name = "z", nullable = false)  
 private long z;  
  
 @Column(name = "sheet")  
 private String sheet;  
  
 @Column(name = "accuracy\_class")  
 private String accuracyClass;  
  
 @Column(name = "coordinate\_system")  
 private String coordinateSystem;

Здесь используются следующие аннотации:

@Entity: указывает, что этот класс является JPA сущностью и должен быть отображён на таблицу базы данных.

@Table(name = «points»): указывает название таблицы в базе данных, с которой связана наша сущность.

@Data: генерирует шаблонный код, добавляя геттеры для всех полей, сеттеры для всех нефинальных полей, правильные реализации toString, equals, hashCode, конструктор для всех нефинальных полей, формируя таким образом объект POJO, необходимый для работы с JPA технологией.

@NoArgsConstructor: генерирует пустой конструктор.

@Id: указывает на то, что это поле является идентификатором, первичным ключом для базы данных.

@GeneratedValue(strategy = GenerationType.IDENTITY): этой аннотацией обычно помечается то же поле, что и аннотацией @Id. У неё есть четыре возможных стратегии (AUTO, IDENTITY, SEQUENCE, TABLE). В нашем случае ( GenerationType.IDENTITY), Hibername делегирует установку ID на уровень базы данных. При этом используется колонка, помеченная как PRIMARY KEY, AUTOINCREMENT.

@Column: позволяет задать имя колонки и другие параметры, такие как nullable и length. В нашем случае (nullable = false) означает, что для этих полей невозможно пустое значение.

### 5. Слой Repositoty

JpaRepositoty предоставляет множество полезных встроенных методов для работы с базой данных, а также позволяет создать пользовательские методы на базе SQL-запросов.

Интерфейс BasePointRepository

public interface BasePointRepository extends JpaRepository<BasePoint, Long> {

@Query("select b from BasePoint b where b.name like %?1%")  
 List<BasePoint> findByName(String name);

@Query("select b from BasePoint b where b.sheet like %?1%")  
 List<BasePoint> findBySheet(String sheet);

@Query("select p from BasePoint p where (p.x between :xSouth and :xNorth) and (p.y between :yWest and :yEast)")  
 List<BasePoint> findByArea(@Param("xSouth") long xSouth, @Param("yWest") long yWest, @Param("xNorth") long xNorth, @Param("yEast") long yEast);

}

Здесь:

@Query: аннотация, позволяющая создать SQL-запрос к базе данных и внедрить в него параметр, передаваемый в пользовательский метод.

С помощью конструкции ?1, указываем порядковый номер параметра в описании пользовательского метода, который передаём в SQL-запрос.

С помощью конструкции :xNorth, передаём в SQL-запрос параметр метода, помеченный аннотацией @Param(«xNorth»).

Дополняем JpaRepository следующими методами:

findByName: Выбирает в базе данных записи о пунктах, название которых содержит фрагмент, переданный аргументом.

findBySheet: Выбирает в базе данных записи о пунктах, название листа карты (название объекта) содержит фрагмент, переданный аргументом.

findByArea: Выбирает в базе данных записи о пунктах, которые попадают в область, параметры которой передаются аргументами. Прямоугольная область описывается плановыми (xSouth, yWest) координатами юго-западного угла и координатами северо-восточного угла (xNorth, yEast).

### 6. Слой Service

Слой сервиса – место, где выполняется основная бизнес-логика нашего приложения. В соответсвии с принципами SOLID, вначале создадим интерфейс, что сделает наш код более гибким, менее связанным.

Интерфейс BasePointService

public interface BasePointService {

List<BasePointDto> getAll();

BasePointDto getById(long id);

List<BasePointDto> getByName(String name);

List<BasePointDto> getBySheet(String sheet);

List<BasePointDto> getByArea(AreaDto areaDto);

BasePointDto createBasePoint(BasePointDto basePointDto);

BasePointDto removeById(long id);

BasePointDto updatePoint(long id, BasePointDto basePointDto);

}

Здесь мы определяем основные операции, которые проектом предусматривается выполнять над нашими данными.

Методы, описанные в интерфейсе BasePointService реализуем в классе BasePointServiceImpl:

@Service  
@RequiredArgsConstructor  
public class BasePointServiceImpl implements BasePointService{  
 private static final String NOT\_FOUND\_POINT\_MESSAGE = "Couldn't find the point with the ";

private final BasePointRepository basePointRepository;

private final BasePointMapper basePointMapper;

@Override  
 public List<BasePointDto> getAll() {  
 return basePointMapper.toListBasePointDto(basePointRepository.findAll());  
 }

@Override  
 public BasePointDto getById(long id) {  
 BasePoint basePoint = basePointRepository.findById(id).orElse(null);  
 if (basePoint != null) {  
 return basePointMapper.toBasePointDto(basePoint);  
 } else {  
 throw new NoSuchElementException(NOT\_FOUND\_POINT\_MESSAGE + "id = " + id);  
 }  
 }

@Override  
 public List<BasePointDto> getByName(String name) {   
 List<BasePoint> basePoints = basePointRepository.findByName(name);  
 if (basePoints.isEmpty()) {  
 throw new NoSuchElementException(NOT\_FOUND\_POINT\_MESSAGE + "name = " + name);  
 }  
 return basePointMapper.toListBasePointDto(basePoints);  
 }

@Override  
 public List<BasePointDto> getBySheet(String sheet) {   
 List<BasePoint> basePoints = basePointRepository.findBySheet(sheet);  
 if (basePoints.isEmpty()) {  
 throw new NoSuchElementException(NOT\_FOUND\_POINT\_MESSAGE + "sheet = " + sheet);  
 }  
 return basePointMapper.toListBasePointDto(basePoints);  
 }

@Override  
 public List<BasePointDto> getByArea(AreaDto areaDto) {  
 long xNorthEast = areaDto.getX() + areaDto.getAreaHeight();  
 long yNorthEast = areaDto.getY() + areaDto.getAreaWidth();  
 List<BasePoint> basePoints = basePointRepository  
 .findByArea(areaDto.getX(),  
 areaDto.getY(),  
 xNorthEast,  
 yNorthEast);  
 if (basePoints.isEmpty()) {  
 throw new NoSuchElementException("No points were found inside the specified area");  
 }  
 return basePointMapper.toListBasePointDto(basePoints);  
 }

@Override  
 @Transactional  
 public BasePointDto createBasePoint(BasePointDto basePointDto) {  
 BasePoint basePoint = basePointMapper.toBasePoint(basePointDto);  
 return basePointMapper.toBasePointDto(basePointRepository.saveAndFlush(basePoint));  
 }

@Override  
 @Transactional  
 public BasePointDto removeById(long id) {  
 BasePoint basePoint = basePointRepository.findById(id).orElse(null);  
 if (basePoint == null) {  
 throw new NoSuchElementException(NOT\_FOUND\_POINT\_MESSAGE + "id = " + id);  
 }  
 basePointRepository.deleteById(id);  
 return basePointMapper.toBasePointDto(basePoint);  
 }

@Override  
 @Transactional  
 public BasePointDto updatePoint(long id, BasePointDto basePointDto) {  
 BasePoint basePoint = basePointRepository.findById(id).orElse(null);  
 if (basePoint == null) {  
 throw new NoSuchElementException(NOT\_FOUND\_POINT\_MESSAGE + "id = " + id);  
 }  
 basePoint.setName(basePointDto.getName());  
 basePoint.setX(basePointDto.getX());  
 basePoint.setY(basePointDto.getY());  
 basePoint.setZ(basePointDto.getZ());  
 basePoint.setSheet(basePointDto.getSheet()); basePoint.setAccuracyClass(basePointDto.getAccuracyClass());  
 return basePointMapper.toBasePointDto(basePointRepository.saveAndFlush(basePoint));  
 }  
}

Здесь:

@Service: сообщает фреймворку, что класс является компонентом приложения и указывает, что он содержит бизнес-логику

@RequiredArgsConstructor: позволяет получить конструктор с final полями. Таким образом, в нашем случае, внедряем в сервис BasePointRepository и BasePointMapper.

@Override: маркерная аннотация, указывающая компилятору на то, что компонент подкласса переопределяет элемент родительского класса, в нашем случае интерфейса BasePointService.

@Transactional: помечает метод, который исполняется в рамках транзакции и позволяет установить различные параметры, такие как Propagation и Isolation. В нашем случае устанавливаются значения по умолчанию.

propogation = Propagation.REQUIRED: Spring проверяет, есть ли активная транзакция, и если ничего не существует, создает новую. В противном случае бизнес-логика добавляется к текущей активной транзакции.

Isolation это одно из свойств ACID. Описывает как изменения, применяемые параллельными транзакциями, видны друг другу.

Isolation = Isolation.READ\_COMMITTED: уровень изоляции по умолчанию для Postgres. Этот уровень изоляции предтвращает dirty reads (грязное чтение). Остальные побочные эффекты параллелизма все еще могут иметь место. Таким образом, незафиксированные изменения в параллельных транзакциях никак на нас не влияют, но если транзакция зафиксирует свои изменения, наш результат может измениться при повторном запросе. Здесь речь идёт о следующих, возможных эффектах параллелизма:

- Dirty read: считывание незафиксированного изменения параллельной транзакции.

- Nonrepeatable read: получение другого значения при повторном чтении строки, если параллельная транзакция обновляет ту же строку и фиксирует.

-Phantom read: получение других строк после повторного выполнения запроса диапазона, если другая транзакция добавляет или удаляет некоторые строки в диапазоне и фиксирует.

В случае возникновения ошибки, например отсутствия или некорректного запроса имени пункта сервис пробрасывает исключение на уровень контроллера.

Для большей гибкости, меньшей связности и взаимодействия между слоями добавляем в проект классы BasePointDto и AreaDto.

Класс BasePointDto:

@Data  
public class BasePointDto {

private long id;

private String name;

private long x;

private long y;

private long z;

private String sheet;

private String accuracyClass;

private String coordinateSystem;

}

Здесь:

@Data: генерирует шаблонный код, добавляя геттеры для всех полей, сеттеры для всех нефинальных полей, правильные реализации toString, equals, hashCode, конструктор для всех нефинальных полей, формируя таким образом объект POJO, необходимый для работы с JPA технологией.

Класс AreaDto:

@Data  
@NoArgsConstructor  
public class AreaDto {

private long x;

private long y;

private long areaHeight;

private long areaWidth;

}

Здесь:

x: Координата X юго-западного угла прямоугольной области.

y: Координата Y юго-западного угла прямоугольной области.

areaHeight: Высота области.

areaWidth: Ширина области.

@NoArgsConstructor: генерирует пустой конструктор.

Для преобразования объектов между слоями Model и Dto служит класc BasePointMapperImpl, который, следуя принципам SOLID, внедряем в реализацию сервиса через интерфейс BasePointMapper.

Интерфейс BasePointMapper:

public interface BasePointMapper {

BasePoint toBasePoint(BasePointDto basePointDto);

BasePointDto toBasePointDto(BasePoint basePoint);

List<BasePointDto> toListBasePointDto(List<BasePoint> basePoints);

List<BasePoint> toListBasePoint(List<BasePointDto> basePointsDto);

}

Класс BasePointMapperImpl:

@Component  
public class BasePointMapperImpl implements BasePointMapper{

public BasePoint toBasePoint(BasePointDto basePointDto) {

BasePoint basePoint = new BasePoint();

basePoint.setName(basePointDto.getName());

basePoint.setX(basePointDto.getX());

basePoint.setY(basePointDto.getY());

basePoint.setZ(basePointDto.getZ());

basePoint.setSheet(basePointDto.getSheet());

basePoint.setAccuracyClass(basePointDto.getAccuracyClass());

basePoint.setCoordinateSystem(basePointDto.getCoordinateSystem());

return basePoint;

}

public BasePointDto toBasePointDto(BasePoint basePoint) {

BasePointDto basePointDto = new BasePointDto();

basePointDto.setId(basePoint.getId());

basePointDto.setName(basePoint.getName());

basePointDto.setX(basePoint.getX());

basePointDto.setY(basePoint.getY());

basePointDto.setZ(basePoint.getZ());

basePointDto.setSheet(basePoint.getSheet());

basePointDto.setAccuracyClass(basePoint.getAccuracyClass());

basePointDto.setCoordinateSystem(basePoint.getCoordinateSystem());

return basePointDto;

}

public List<BasePointDto> toListBasePointDto(List<BasePoint> basePoints) {

return basePoints.stream().map(this::toBasePointDto).toList();

}

public List<BasePoint> toListBasePoint(List<BasePointDto> basePointsDto) {

return basePointsDto.stream().map(this::toBasePoint).toList();

}

}

Здесь:

@Component: Указывает, что этот класс является компонентом. Такие классы автоматически сканируются Spring и регистрируются в контейнере IoC.

В методах toListBasePointDto и toListBasePoint используем возможности Java Stream Api, отображая и преобразуя данные с помощью функции map в нужный нам тип. В качестве функционального интерфейса типа Function, передаём аргументом ссылку на соответствующий метод класса.

### Слой Controller.

Слой контроллеров отвечает за обработку HTTP-запросов и взаимодействует со слоем сервисов для выполнения бизнес-логики.

Класс BasePointController:

@RestController

@RequestMapping("point")

@RequiredArgsConstructor

public class BasePointController {

private final BasePointService basePointService;

@GetMapping

public ResponseEntity<List<BasePointDto>> getAllBasePoints() {

return ResponseEntity.ok().body(basePointService.getAll());

}

@GetMapping("{id}")

public ResponseEntity<BasePointDto> getBasePointById(@PathVariable long id) {

try {

return ResponseEntity.ok().body(basePointService.getById(id));

} catch (NoSuchElementException e) {

throw new ResponseStatusException(HttpStatus.NOT\_FOUND, e.getMessage());

}

}

@GetMapping("name/{name}")

public ResponseEntity<List<BasePointDto>> getBasePointByName(@PathVariable String name) {

try {

return ResponseEntity.ok().body(basePointService.getByName(name));

} catch (NoSuchElementException e) {

throw new ResponseStatusException(HttpStatus.NOT\_FOUND, e.getMessage());

}

}

@GetMapping("sheet/{sheet}")

public ResponseEntity<List<BasePointDto>> getBasePointBySheet(@PathVariable String sheet) {

try {

return ResponseEntity.ok().body(basePointService.getBySheet(sheet));

} catch (NoSuchElementException e) {

throw new ResponseStatusException(HttpStatus.NOT\_FOUND, e.getMessage());

}

}

@DeleteMapping("{id}")

public ResponseEntity<BasePointDto> removeBasePointById(@PathVariable long id) {

try {

return ResponseEntity.ok().body(basePointService.removeById(id));

} catch (NoSuchElementException e) {

throw new ResponseStatusException(HttpStatus.NOT\_FOUND, e.getMessage());

}

}

@PostMapping

public ResponseEntity<BasePointDto> createBasePoint(@RequestBody BasePointDto basePointDto) {

return new ResponseEntity<>(basePointService.createBasePoint(basePointDto),HttpStatus.CREATED);

}

@PostMapping("area")

public ResponseEntity<List<BasePointDto>> getByArea(@RequestBody AreaDto areaDto) {

try {

return ResponseEntity.ok().body(basePointService.getByArea(areaDto));

} catch (NoSuchElementException e) {

throw new ResponseStatusException(HttpStatus.NOT\_FOUND, e.getMessage());

}

}

@PutMapping("{id}")

public ResponseEntity<BasePointDto> updateBasePoint(@PathVariable long id, @RequestBody BasePointDto basePointDto) {

try {

return ResponseEntity.ok().body(basePointService.updatePoint(id, basePointDto));

} catch (NoSuchElementException e) {

throw new ResponseStatusException(HttpStatus.NOT\_FOUND, e.getMessage());

}

}

}

Здесь:

**@RestController**: объединяет в себе аннотации @Controller и @ResponseBody. Она не только помечает класс как Spring MVC Controller, но и автоматически преобразует возвращаемые данные в формат JSON или XML. REST – архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Таким образом наш микросервис легко внедрить в другие системы.

**@RequestMappping(«point»)**: эта аннотация задаёт базовый маршрут для всех методов в этом контроллере. Все эндпоинты будут начинаться с «/point».

**@GetMapping, @PostMapping, @PutMapping, @DeleteMapping**: Эти аннотации задают HTTP-метод для каждого из методов контроллера. Главным образом, они соответствуют стандартным CRUD-операциям: получение, создание, обновление и удаление.

**@PathVariable**: эта аннотация используется для передачи элемента пути запроса в виде параметра метода контроллера.

**@RequestBody**: эта аннотация помечает параметр в методе контроллера, сопоставляя его с телом запроса.

Методы контроллера обращаются к соответствующим методам сервиса для выполнения операции и возвращает результат в теле объекта ResponseEntity<T>.

Контроллер обрабатывает исключения вбрасываемые на уровне сервисного слоя и возвращает исключение типа ResponseStatusException.

**ResponseStatusException**: является программной альтернативой @ResponseStatus и базовым классом для исключений, используемых для применения кода состояния к HTTP-ответу. Это RuntimeException, и, следовательно, его не требуется явно добавлять в сигнатуру метода.

# Заключение.

Текст заключения.

# Приложения.

## Приложение А. Номенклатура топографических карт.

В основу номенклатуры топографических карт положена карта масштаба 1:1 000 000 (10 км в 1см; так называемые «миллионки»).

Вся поверхность Земли делится параллелями на ряды (через 4°), а меридианами — на колонны (через 6°); стороны образовавшихся трапеций служат границами листов карты масштаба 1:1000 000. Ряды обозначаются заглавными латинскими буквами от А до V, начиная от экватора к обоим полюсам, а колонны — арабскими цифрами, начиная от меридиана 180°с запада на восток. Номенклатура листа карты состоит из буквы ряда и номера колонны. Например, лист с городом Москва обозначается N-37.

Приполярные круглые области (с широтой свыше 88°) обозначаются буквой Z без указания номера колонны.

Карты масштаба 1:100 000 (1 км в 1 см) получаются делением листа миллионной карты на 144 части (рис. 3); их номенклатура состоит из обозначения листа карты масштаба 1:1000000 с добавлением одного из чисел 1, 2, 3, 4, …, 143, 144. Например, лист стотысячной карты с городом Рязань имеет номенклатуру N-37-56.

Во избежание путаницы с картами масштабов 1:500 000 и 1:200 000 (для которых существуют сходные альтернативные обозначения), карты этого масштаба могут обозначаться трёхзначными арабскими цифрами без потери начальных нулей, например: О-37-050.

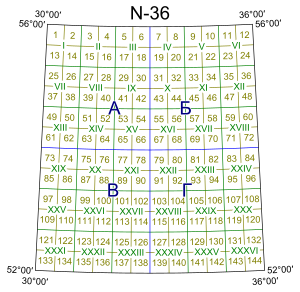


Рис. 3. Пример разграфки листа миллионной карты на карты масштаба 1:500 000 (синие), 1:200 000 (зелёные) и 1:100 000 (жёлтые)

## Приложение Б. Классификация геодезических сетей.

Все геодезические сети по назначению и точности построения подразделяются на три большие группы:

* государственные геодезические сети (ГГС),
* геодезические сети сгущения (ГСС),
* геодезические съемочные сети.

Государственные плановые сети имеют плотность в среднем 1 пункт на 5 — 15 км², высотные — 1 пункт на 5 — 7 км², дальнейшее сгущение геодезических сетей осуществляется путем создания сетей местного значения — сетей сгущения и съемочных сетей. Минимальное число пунктов съемочного обоснования для незастроенной территории 12 шт. на 1 км

У пунктов геодезической сети есть такая характеристика как класс точности.

В порядке убывания от более точных к менее точным:

* 1 класс
* 2 класс
* 3 класс
* 4 класс
* I разряд
* II разряд

С точки зрения геометрии любая геодезическая сеть – это группа зафиксированных на местности точек, для которых определены плановые координаты (X и Y или B и L) в принятой двухмерной системе координат и отметки H в принятой системе высот или три координаты X, Y и Z в принятой трехмерной системе пространственных координат.

По своей сущности геодезические пункты подразделяются на грунтовые и стенные. Грунтовые (закладываемые в грунт на незастроенной территории) состоят непосредственно из самого центра, являющегося носителем координат, наружного знака — обозначающего положение центра на местности и обеспечивающего взаимную видимость смежных пунктов сети и подземного сооружения (скрытая часть). Основная часть — Центр. Предназначен надежно и долговременно сохранять координаты пункта всех классов кроме I-го. Пункты I-го класса представляют группы центров, т. н. «кусты». Стенные г. п. закладываются в стены сооружений на застроенной территории и предназначены для сохранения координат III, IV классов, 1 и 2 разрядов. В отдельный класс выделяются пункты технических сетей, не имеющие капитального закрепления и наружного знака. В таких случаях применяются разборные знаки (переносимые или перевозимые), а сам пункт имеет характер временного (утрачивается в течение незначительного отрезка времени — 1 или 2 сезонов). Все виды геодезических пунктов не противопоставляются друг другу, а взаимно дополняют — чем выше класс сети, тем солидней конструкция и выше надежность центра.



Рис. Наружный знак геодезического пункта

# Список использованной литературы.

1. Строительные нормы и правила
2. «Java полное руководство» Герберт Шилдт
3. «Разработка веб-приложений на платформе Spring» Суханов В.И.
4. «Чистая архитектура. Искусство разработки программного обеспечения.» Роберт Мартин.