



UNIVERSIDAD PRIVADA FRANZ TAMAYO

DEFENSA HITO 3 - TAREA FINAL

Nombre Completo: **Nicolas Gonzalo Aguilar Arimoza**

Asignatura: **PROGRAMACIÓN III**

Carrera: **INGENIERÍA DE SISTEMAS**

Paralelo: **PROG (1)**

Docente: **Lic. William R. Barra Paredes**

fecha: **06/05/2020**

github: <https://github.com/shierf/prograiii/tree/master/Hito3/tareas/CoronaVirusWeb>

Introducción

Se tiene como visión general a la capacidad de poder implementar y generar un servicio REST desarrollado en java bajo el framework Spring accesible desde cualquier cliente web.

parte teorica

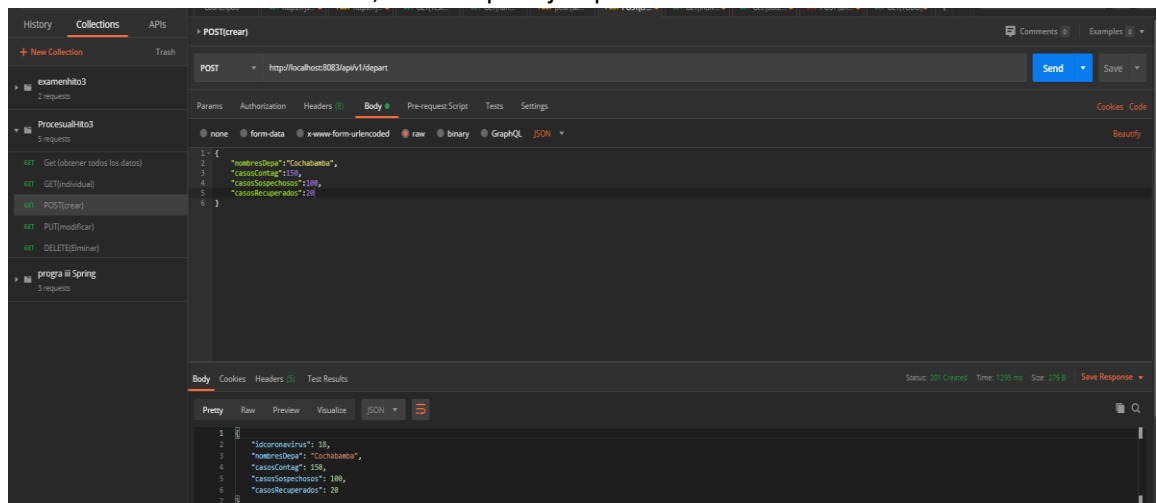
- **pregunta 1 (Defina y muestre ejemplo de un servicio REST)**

REST es una interfaz para conectar varios sistemas basados en el protocolo HTTP (uno de los protocolos más antiguos) y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON.

El formato más usado en la actualidad es el formato **JSON**, ya que es más ligero y legible en comparación al formato XML. Elegir uno será cuestión de la lógica y necesidades de cada proyecto.

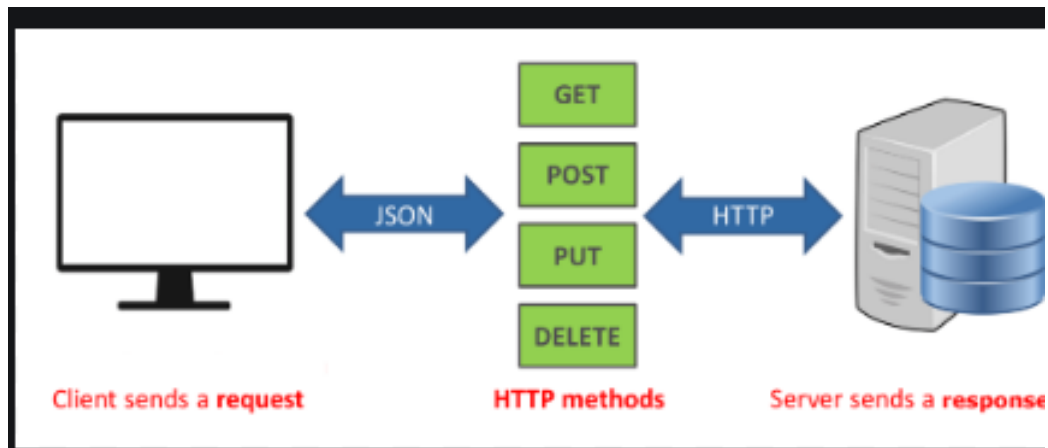
REST se apoya en HTTP, los verbos que utiliza son exactamente los mismos, con ellos se puede hacer GET, POST, PUT y DELETE. De aquí surge una alternativa a SOAP.

1. **Post** : Para crear recursos nuevos.
2. **Get** : Para obtener un recurso en concreto.
3. **Put** : Para modificar.
4. **Patch** : Para modificar un recurso que no es un recurso de un dato, por ejemplo.
5. **Delete** : Para borrar un recurso, un dato por ejemplo de nuestra base de datos.



REST nos beneficia con varias de sus ventajas las cuales son:

1. **Nos permite separar el cliente del servidor**. Esto quiere decir que nuestro servidor se puede desarrollar en Node y Express, y nuestra API REST con Vue por ejemplo, no tiene por qué estar todos dentro de un mismo.
2. En la actualidad tiene una gran comunidad como proyecto en **Github**.
3. Podemos hacer nuestra **API pública**, permitiendo darnos visibilidad si la hacemos pública.
4. Nos da **escalabilidad**, porque tenemos la separación de conceptos de CLIENTE y SERVIDOR, por tanto, podemos dedicarnos exclusivamente a la parte del servidor.



- **pregunta 2 (Que es JPA y como configurar en un entorno Spring.)**

JPA es la propuesta estándar que ofrece Java para implementar un Framework Object Relational Mapping (ORM), que permite interactuar con la base de datos por medio de objetos, de esta forma, JPA es el encargado de convertir los objetos Java en instrucciones para el Manejador de Base de Datos (MDB).

También puede utilizarse directamente en aplicaciones web y aplicaciones clientes; incluso fuera de la plataforma **Java EE**, por ejemplo, en aplicaciones **Java SE**.

En su definición, se han combinado ideas y conceptos de los principales frameworks de persistencia como **Hibernate**, **Toplink** y **JDO**, y de las versiones anteriores de **EJB**. Todos estos cuentan actualmente con una implementación **JPA**.

Para lograr la configuracun tenemos que configurar las dependencias priero en el fichero pom.xml del proyecto:

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema
2 <modelversion>4.0.0</modelversion>
3 <groupid>com.autentia</groupid>
4 <artifactid>model-tutoriales</artifactid>
5 <version>1.0-SNAPSHOT</version>
6 <name>model-tutoriales</name>
7 <description>Libreria de acceso a datos para la base de datos "tutoriales"</description>
8 <build>
9 <plugins>
10 <plugin>
11 <artifactid>maven-compiler-plugin</artifactid>
12 <configuration>
13 <encoding>UTF-8</encoding>
14 <source>1.6</source>
15 <target>1.6</target>
16 </configuration>
17 </plugin>
18 </plugins>
19 </build>
20 <dependencies>
21 <dependency>
22 <groupid>commons-lang</groupid>
23 <artifactid>commons-lang</artifactid>
24 <version>2.5</version>
25 </dependency>
26 <dependency>
27 <groupid>javax.persistence</groupid>
28 <artifactid>persistence-api</artifactid>
29 <version>1.0</version>
30 </dependency>
31 <dependency>
32 <groupid>org.springframework</groupid>
33 <artifactid>spring-core</artifactid>
34 <version>3.0.4.RELEASE</version>
35 </dependency>
36 <dependency>
37 <groupid>org.springframework</groupid>
38 <artifactid>spring-tx</artifactid>
39 <version>3.0.4.RELEASE</version>
40 </dependency>

```

```

42     <groupId>org.springframework</groupId>
43     <artifactId>spring-orm</artifactId>
44     <version>3.0.4.RELEASE</version>
45 </dependency>
46 <dependency>
47     <groupId>org.springframework</groupId>
48     <artifactId>spring-hibernate3</artifactId>
49     <version>2.0.8</version>
50 </dependency>
51 <dependency>
52     <groupId>org.hibernate</groupId>
53     <artifactId>hibernate-entitymanager</artifactId>
54     <version>3.3.1.ga</version>
55 </dependency>
56 <dependency>
57     <groupId>postgresql</groupId>
58     <artifactId>postgresql</artifactId>
59     <version>9.0-801.jdbc4</version>
60 </dependency>
61 <dependency>
62     <groupId>concurrent</groupId>
63     <artifactId>concurrent</artifactId>
64     <version>1.3.3</version>
65 </dependency>
66 <dependency>
67     <groupId>javax.annotation</groupId>
68     <artifactId>jsr250-api</artifactId>
69     <version>1.0</version>
70 </dependency>
71 <dependency>
72     <groupId>org.springframework</groupId>
73     <artifactId>spring-test</artifactId>
74     <version>3.0.4.RELEASE</version>
75     <scope>test</scope>
76 </dependency>
77 <dependency>
78     <groupId>junit</groupId>
79     <artifactId>junit</artifactId>
80     <version>4.8.2</version>
81     <scope>test</scope>
82 </dependency>
83 <dependency>
84     <groupId>org.hsqldb</groupId>
85     <artifactId>hsqldb</artifactId>
86     <version>1.8.0.10</version>
87     <scope>test</scope>
88 </dependency>

```

- **pregunta 3(Que es MAVEN - POM.)**

Maven se utiliza en la gestión y construcción de software. Posee la capacidad de realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. Es decir, hace posible la creación de software con dependencias incluidas dentro de la estructura del JAR.

POM (Project Object Model). Este es un archivo en formato XML que contiene todo lo necesario para que a la hora de generar el fichero ejecutable de nuestra aplicación este contenga todo lo que necesita para su ejecución en su interior.



Ejemplo de maven pom

```
</plugins>
<finalName>MavenEnterpriseApp-ear</finalName>
</build>
<dependencies>
  <dependency>
    <groupId>com.mycompany</groupId>
    <artifactId>MavenEnterpriseApp-ejb</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>ejb</type>
  </dependency>
  <dependency>
    <groupId>com.mycompany</groupId>
    <artifactId>MavenEnterpriseApp-web</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>war</type>
  </dependency>
</dependencies>
</project>
```

- **Pregunta 4(Qué son los Spring estereotipos y anotaciones muestre ejemplos)**

Spring es un framework alternativo al stack de tecnologías estándar en aplicaciones JavaEE.

Spring popularizó ideas como la inyección de dependencias o el uso de objetos convencionales (POJOs) como objetos de negocio, que suponían un soplo de aire fresco. Estas ideas permitían un desarrollo más sencillo y rápido y unas aplicaciones más ligeras. Eso permitió que de ser un framework inicialmente diseñado para la capa de negocio pasara a ser un completo stack de tecnologías para todas las capas de la aplicación.

Estereotipos

Spring ofrece una serie de anotaciones estándar para los objetos de nuestra aplicación: por ejemplo, **@Service** indica que la clase es un bean de la capa de negocio, mientras que **@Repository** indica que es un DAO. Si simplemente queremos especificar que algo es un bean sin decir de qué tipo es podemos usar la anotación **@Component**. Por ejemplo:

```
1 package es.ua.jtech.spring.negocio;
2
3 import org.springframework.stereotype.Service;
4
5 @Service("usuariosBO")
6 public class UsuariosBOSimple implements IUsuariosBO {
7     public UsuarioTO login(String login, String password) {
8         ...
9     }
10    public boolean logout() {
11        ...
12    }
13    ...
14 }
```

El parámetro "usuariosBO" de la anotación sirve para darle un nombre o identificador al bean.

Para que nuestro bean funcione, falta un pequeño detalle. Spring necesita de un fichero XML de configuración mínimo. Cuando los beans no se definen con anotaciones sino con XML, aquí es donde se configuran, en lugar de en el fuente Java.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xmlns:context="http://www.springframework.org/schema/context"
5        xsi:schemaLocation="http://www.springframework.org/schema/beans
6                             http://www.springframework.org/schema/beans/spring-beans.xsd
7                             http://www.springframework.org/schema/context
8                             http://www.springframework.org/schema/context/spring-context.xsd">
9
10     <context:component-scan base-package="es.ua.jtech.spring"/>
11 </beans>
```

La etiqueta **<context:component-scan>** es la que especifica que usaremos anotaciones para la definición de los beans, y que las clases que los definen van a estar en una serie de paquetes.

Anotaciones

stereotipos: @Component, @Service, y @Controller. @Component es un estereotipo genérico para cualquier componente administrado por Spring. @Repository, @Service, y @Controller son especializaciones de @Component para casos de uso más específicos, por ejemplo, en las capas de persistencia, servicio y presentación, respectivamente.

Annotation	Meaning
@Component	generic stereotype for any Spring-managed component
@Repository	stereotype for persistence layer
@Service	stereotype for service layer
@Controller	stereotype for presentation layer (spring-mvc)

• **pregunta 5 (Describe las características principales de REST)**

1. Cliente-Servidor

Los servicios REST deben estar basados en una arquitectura Cliente-Servidor. Un servidor que contiene los recursos y estados de los mismos, y unos clientes que acceden a ellos.

2. Sin estado

Los Servicios REST pueden ser escalados hasta alcanzar grandes rendimientos para abarcar la demanda de todos los posibles clientes. Esto implica que sea necesario crear granjas de servidores con balanceo de cargas y failover o diferentes niveles de servidores para minimizar el tiempo de respuesta a los clientes

3. Información cacheable

Para mejorar la eficiencia en el tráfico de red, las respuestas del servidor deben tener la posibilidad de ser marcadas como cacheables. Esta información es utilizada por los clientes REST para decidir si hacer una copia local del recurso con la fecha y hora del último cambio de estado del recurso

4. Interfaz uniforme

Una de las características principales de los servicios Web REST es el uso explícito de métodos HTTP (HyperText Transfer Protocol). Estos métodos son indicados en la cabecera HTTP por parte del cliente y son los siguientes:

- GET: recoge información de un recurso
- PUT: modifica o actualiza el estado de un recurso
- POST: crea un nuevo recurso en el servidor
- DELETE: elimina un recurso del servidor.

5. Acceso a recursos por nombre

Un sistema REST está compuesto por recursos que son accedidos mediante URL, y éstas deben ser intuitivas, predecibles y fáciles de entender y componer. Una manera de conseguirlo es mediante una estructura jerárquica, similar a directorios. Puede existir un nodo raíz único, a partir del cual se crean los subdirectorios que expongan las áreas principales de los servicios, hasta formar un árbol con la información de los recursos.

6. Recursos relacionados

Los recursos accesibles en el servidor suelen estar relacionados unos con otros. Por tanto, la información de estado de un recurso debería permitir acceder a otros recursos. Esto se consigue añadiendo en el estado de los recursos links o URL de otros recursos.

7. Respuesta a un formato conocido

La representación de un recurso refleja el estado actual del mismo y sus atributos en el instante en el que el cliente ha realizado la solicitud. Este resultado puede representar simplemente el valor de una variable en un instante de tiempo, un registro de una Base de Datos o cualquier otro tipo de información. En cualquiera de los casos, la información debe ser entregada al cliente en un formato comprensible para ambas partes y contenida dentro del cuerpo HTTP.

Parte practica

1. Crear los PACKAGES necesarios(debe reflejarse el modelo MVC).
 2. Debe de crear los servicios REST para el siguiente escenario.
 3. Actualmente toda la humanidad está pasando por una pandemia conocida como Corona Virus COVID19. En bolivia se pretende crear una plataforma en tiempo real para mostrar estos datos a cada habitante.
 4. Es decir mostrar casos contagiados, casos sospechosos, casos recuperados, etc.
 5. Para este propósito la primera fase de de desarrollo es la creación de un servicio rest que pueda crear, modificar y retornar estos datos.
 6. Se tiene como base principal la siguiente tabla que nos servirá para poder generar toda esta información.
- Primero se crea las tablas mediante la aplicación intellij DEA y almacenándolas en datagrip para la base de datos

Código clase VirusModel

```
package com.procesualHito3.CoronaVirusWeb.Model;

import javax.persistence.*;
import java.util.Date;
@Entity
@Table(name = "Virus")
public class VirusModel {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int idcoronavirus;
    @Column(name = "nombresDepartamento", length = 50, nullable = false)
    private String nombresDepa;
    @Column(name = "CasosContagiados")
    private int casosContag;
    @Column(name = "CasosSostepochosos")
    private int casosSospechosos;
    @Column(name = "CasosRecuperados")
    private int casosRecuperados;

    public int getIdcoronavirus() {
        return idcoronavirus;
    }

    public void setIdDepar(int idcoronavirus) {
        this.idcoronavirus = idcoronavirus;
    }

    public String getNombresDepa() {
        return nombresDepa;
    }

    public void setNombresDepa(String nombresDepa) {
        this.nombresDepa = nombresDepa;
    }

    public int getCasosContag() {
        return casosContag;
    }
}
```



```

public void setCasosContag(int casosContag) {
    this.casosContag = casosContag;
}

public int getCasosSospechosos() {
    return casosSospechosos;
}

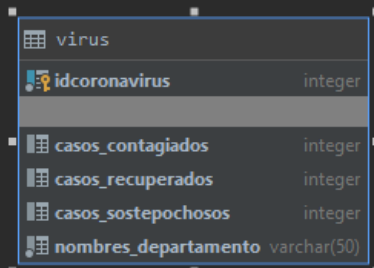
public void setCasosSospechosos(int casosSospechosos) {
    this.casosSospechosos = casosSospechosos;
}

public int getCasosRecuperados() {
    return casosRecuperados;
}

public void setCasosRecuperados(int casosRecuperados) {
    this.casosRecuperados = casosRecuperados;
}
}

```

- Gracias a la clase model podemos seleccionar que cosa tendrá cada columna de la tabla y creándola en el datagrip al igual los get y set de cada uno.



virus	
idcoronavirus	integer
casos_contagiados	integer
casos_recuperados	integer
casos_sostepochosos	integer
nombres_departamento	varchar(50)

Clase VirusService

```
package com.procesualHito3.CoronaVirusWeb.Services;

import com.procesualHito3.CoronaVirusWeb.Model.VirusModel;
import com.procesualHito3.CoronaVirusWeb.Repo.VirusRepo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

@Service
public class VirusService implements VirusInterfaceService{
    @Autowired
    private VirusRepo virusRepo;
    @Override
    public VirusModel save(VirusModel virusModel){
        return virusRepo.save(virusModel);
    }
    @Override
    public VirusModel update(VirusModel virusModel,Integer idcoronavirus){
        Optional<VirusModel> virus = virusRepo.findById(idcoronavirus);
        VirusModel virusUpdate = null;
        if(virus.isPresent()){
            virusUpdate = virus.get();
            virusUpdate.setNombresDepa(virusModel.getNombresDepa());
            virusUpdate.setCasosContag(virusModel.getCasosContag());
            virusUpdate.setCasosRecuperados(virusModel.getCasosRecuperados());
            virusUpdate.setCasosSospechosos(virusModel.getCasosSospechosos());
        }
        return virusUpdate;
    }
    @Override
    public Integer delete(Integer idcoronavirus){
        virusRepo.deleteById(idcoronavirus);
        return 1;
    }
    @Override
    public List<VirusModel>getAllDepar(){
        List<VirusModel> coronavirus = new ArrayList<VirusModel>();
        virusRepo.findAll().forEach(coronavirus::add);
        return coronavirus;
    }
    @Override
    public VirusModel getDeparByIdPer(Integer idcoronavirus){
        Optional<VirusModel> coronavirus = virusRepo.findById(idcoronavirus);
        VirusModel virusModel = null;
        if(coronavirus.isPresent()){
            virusModel = coronavirus.get();
        }
        return virusModel;
    }
}
```

- Con esta clase se puede mandar cada función que utilizaremos en el postman guardar, mostrar, modificar, eliminar, como.
SAVE,UPDATE,DELETE,GETALLDEPAR,GETDEPARBYIDPER

Class userController

```
package com.procesualHito3.CoronaVirusWeb.Controller;

import com.procesualHito3.CoronaVirusWeb.Model.VirusModel;
import com.procesualHito3.CoronaVirusWeb.Services.VirusService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping(value = "/api/v1/")
public class userController {
    @Autowired
    private VirusService virusService;

    @GetMapping("/depart")
    public ResponseEntity<List<VirusModel>> getAllDepart() {
        try {
            List<VirusModel> depar = virusService.getAllDepar();

            if (depar.isEmpty()) {
                return new ResponseEntity<>(HttpStatus.NO_CONTENT);
            } else {
                return new ResponseEntity<>(depar, HttpStatus.OK);
            }
        } catch (Exception e) {
            return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    @PostMapping("/depart")
    public ResponseEntity save(@RequestBody VirusModel VIRUS) {
        try {
            return new ResponseEntity<>(virusService.save(VIRUS),
HttpStatus.CREATED);
        } catch (Exception e) {
            return new ResponseEntity<>(null, HttpStatus.EXPECTATION_FAILED);
        }
    }

    @GetMapping("/depart/{idDepar}")
    public ResponseEntity<VirusModel> getDeparByIdPer(@PathVariable("idDepar")
Integer idDep) {
        try {
            VirusModel DepModel = virusService.getDeparByIdPer(idDep);

            if ( DepModel!= null) {
                return new ResponseEntity<>(DepModel, HttpStatus.OK);
            } else {
                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
            }
        } catch (Exception e) {
            return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    @PutMapping("/depart/{idDepar}")
    public ResponseEntity<VirusModel> update(@PathVariable("idDepar") Integer
```

```

idDep, @RequestBody VirusModel DepModel) {
    try {
        VirusModel DUpdate = virusService.update(DepModel, idDep);
        if (DUpdate != null) {
            return new ResponseEntity<>(DUpdate, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    } catch (Exception e) {
        return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
}
@DeleteMapping("/depart/{idDepar}")
public ResponseEntity<String> delete(@PathVariable("idDepar") Integer idDep)
{
    try {
        virusService.delete(idDep);
        return new ResponseEntity<>("person successfully deleted",
HttpStatus.OK);
    } catch (Exception e) {
        return new ResponseEntity<>(null, HttpStatus.EXPECTATION_FAILED);
    }
}
}
}

```

- Con esta clase logramos la conexión con el postamn mediante ("**/depart**") y pidiendo que es lo que queremos que realice en cada momento esto nos ayuda a utilizar las diferentes funciones del postamn que son GET,POST,PUT,DELETE .
- Con esto ingresa a la base de datos y puede ingresar extraer o modificar cada información colocada en las tablas del datagrip.

Interfaces

Interface virus repo

```

package com.procesualHito3.CoronaVirusWeb.Repo;

import com.procesualHito3.CoronaVirusWeb.Model.VirusModel;
import org.springframework.data.jpa.repository.JpaRepository;

public interface VirusRepo extends JpaRepository<VirusModel, Integer> {
}

```

Interface VirusInterfaceService

```

package com.procesualHito3.CoronaVirusWeb.Services;

import com.procesualHito3.CoronaVirusWeb.Model.VirusModel;

import java.util.List;

public interface VirusInterfaceService {

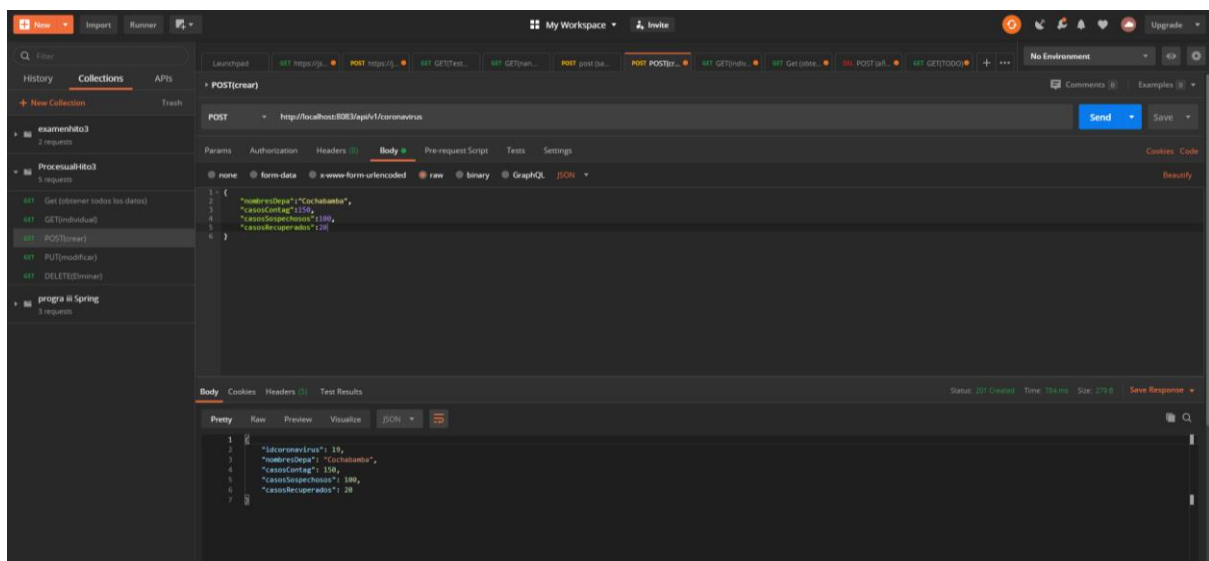
```

```

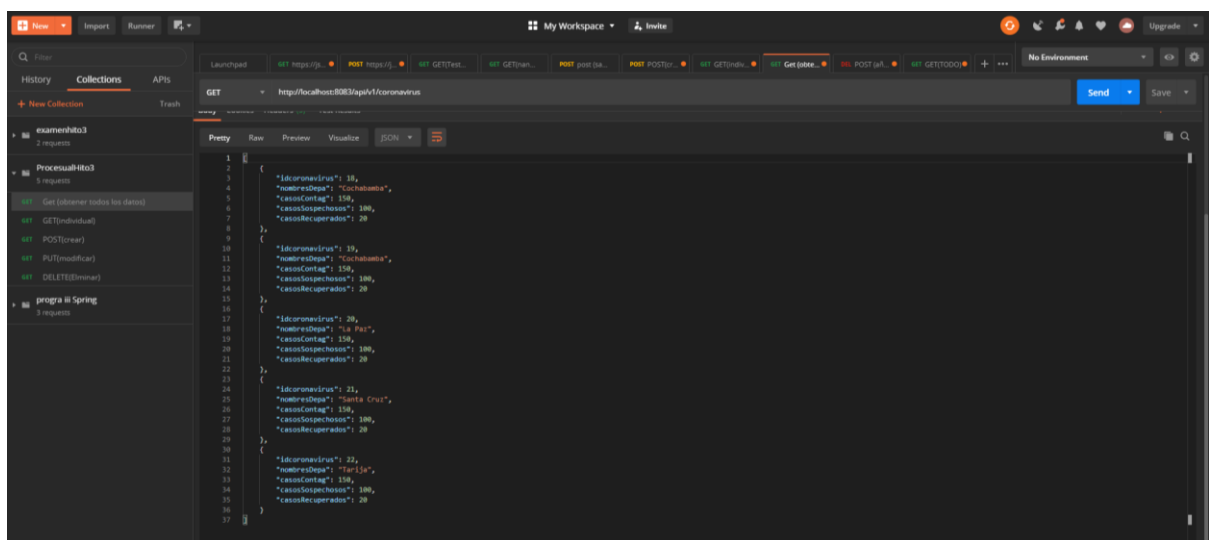
public VirusModel save(VirusModel virusModel);
public VirusModel update(VirusModel virusModel, Integer idDepar);
public Integer delete(Integer idPer);
public List<VirusModel> getAllDepar();
public VirusModel getDeparByIdPer(Integer idDep);
}

```

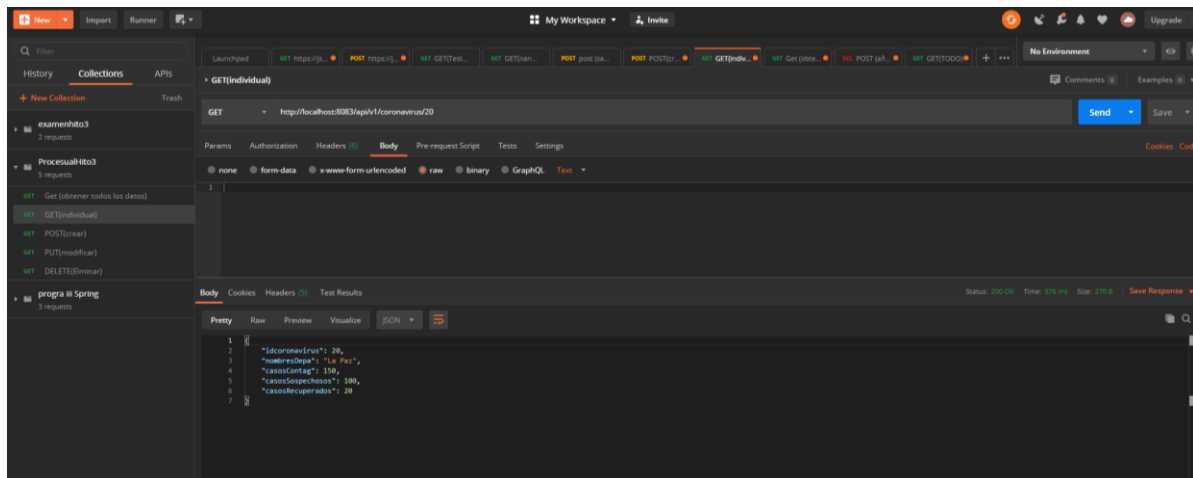
- Al finalizar todo en el IntelliJ IDEA nos pasamos a la utilización del Postman para poder realizar las diferentes funciones que tiene este.
- POST para ingresar cada uno de los departamentos



- GET para obtener todos los departamentos ingresados

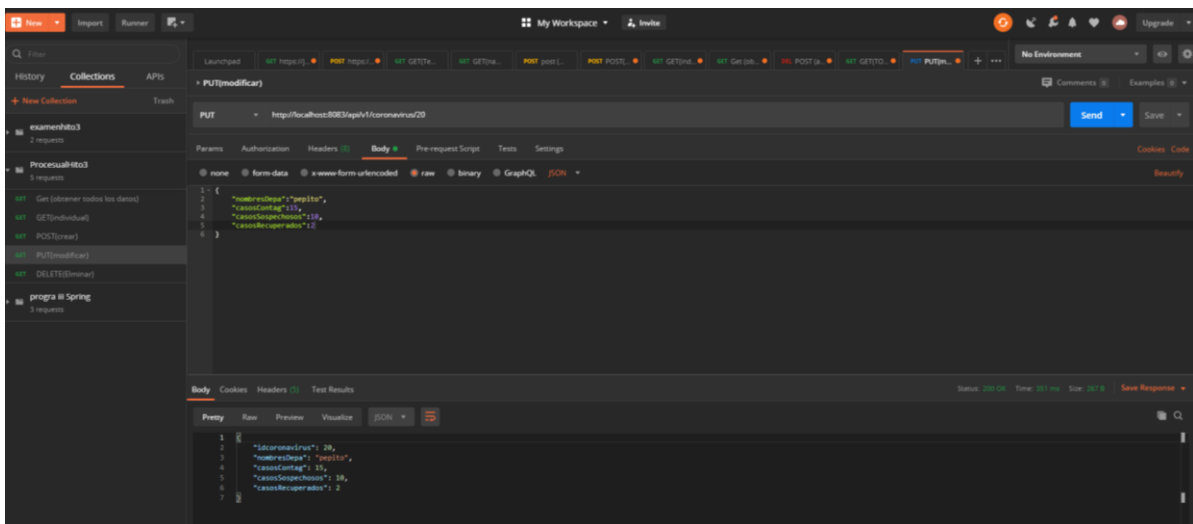


- GET para obtener un departamento mediante su idcoronavirus

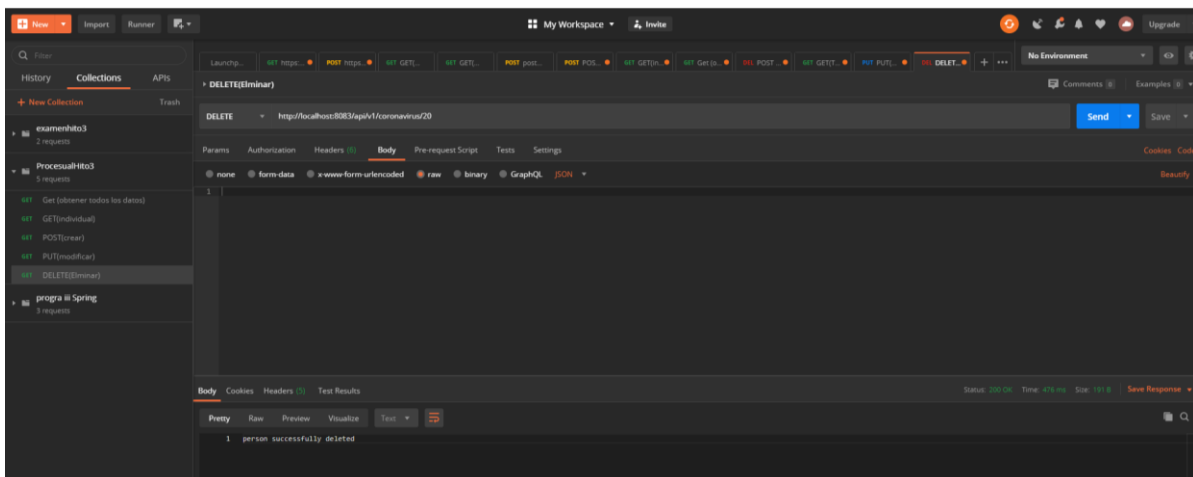


- PUT para poder modificar uno de los departamentos ingresados

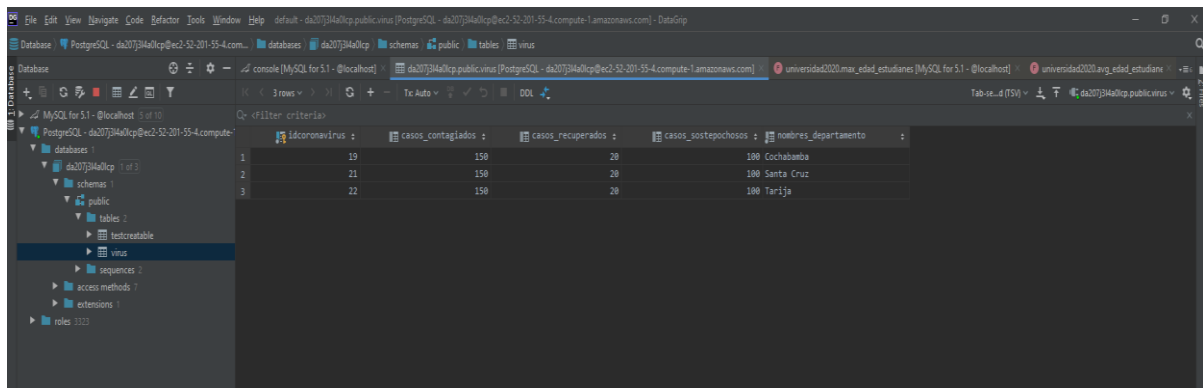
Se puede observar como el id 20 de La Paz fue cambiado por pepito y por menos variables



- DELETE para eliminar uno de los departamentos de las tablas



Al lograr la eliminación de uno de los departamentos se observa en las tablas que se elimino exitosamente



The screenshot shows a PostgreSQL DataGrip interface. On the left, a tree view displays the database structure: 'databases' > 'da20734a0kcp' > 'schemas' > 'public' > 'tables' > 'virus'. The main pane shows a table with 3 rows and 5 columns. The columns are: 'idcoronavirus', 'casos_contagiados', 'casos_recuperados', 'casos_sostepochosos', and 'nombres_departamento'. The data is as follows:

idcoronavirus	casos_contagiados	casos_recuperados	casos_sostepochosos	nombres_departamento
19	150	20	100	Cochabamba
21	150	20	100	Santa Cruz
22	150	20	100	Tarija