Introducción

A continuación se presentan una serie de ejercicios divididos por bloques. Para poder realizar los ejercicios es necesario leer antes la teoría del bloque correspondiente. Algunos bloques no tienen ejercicios.

Los ejercicios serán revisados por el profesor correspondiente al alumno. Es importante leer detenidamente cada ejercicio e intentar realizarlo individualmente. Puede que sea necesario buscar información adicional en internet en algunos de los ejercicios. Si buscando en internet tampoco puedes resolver el problema, hay que preguntar las dudas que tengas al profesor o a otros compañeros. **Otro compañero te puede ayudar a resolver el problema, pero es importante entender lo que estás haciendo**.

Requisitos para aceptar un ejercicio:
- El módulo correspondiente al ejercicio **debe compilar**.
- El programa **debe realizar las acciones pedidas** por el ejercicio.
- Se deben incluir en el repositorio las configuraciones del proyecto necesarias para probar el programa. En los ejercicios **donde haya llamadas HTTP será necesario adjuntar el proyecto de Postman usado para testear el programa**.

Hay que intentar no pasar al siguiente ejercicio si no has resuelto el actual. Si finalmente vas a hacerlo, **informar de ello al profesor**.

1. Java Development Kit (JDK)

Lectura de ficheros y filtrado con Stream

Nombre proyecto Maven: *block1-process-file-and-streams*

Tiempo estimado: 12 horas

Crear un método que reciba la ruta de un fichero CSV (por ejemplo: `people.csv`) y devuelva una lista de personas. El formato del fichero es el siguiente:

- Cada línea del fichero corresponde a una persona (clase `Person`).
- Cada línea puede contener hasta 3 campos separados por dos puntos (:). Los campos serán los siguientes:

  `name:town:age`

- El campo `name` es obligatorio, el resto de campos son opcionales. **Nota**: cuando no se informe el campo `age` guardaremos la persona con 0 años. Si hay alguna línea en el fichero en la que el campo `age` sea 0, consideraremos que esa persona tiene edad desconocida.
- Si cualquiera de las líneas no tiene formato correcto el método debe lanzar una excepción `InvalidLineFormatException`. Esta excepción debe incluir un mensaje descriptivo incluyendo la línea que provocó el error y la causa de la excepción.

Nota: no usamos Stream en este método porque no permite tratar correctamente la excepción `InvalidLineFormatException`. Stream no se recomienda cuando hay que tratar excepciones.

Crear un segundo método que filtre las personas obtenidas al leer el fichero y devuelve esa lista de personas. Usar **Stream** para filtrar.

Requisitos:

- Para crear el método que lee el fichero, no usar Stream, usar las clases NIO: https://www.baeldung.com/reading-file-in-java
- Para crear el segundo método, usar Stream.
- La ruta al fichero se le pasa al programa vía argumentos y será una ruta relativa.
- Tratar la excepción `InvalidLineFormatException` capturándola desde el método que invoca la función, imprimiendo su mensaje e imprimiendo el Stack Trace.

Contenido de un fichero de ejemplo:

```
Jesús:Logroño:41
Andres:Madrid:19
Angel Mari:Valencia:
Laura Saenz::23
Fernando:Zaragoza
Noelia:Sevilla:28
Maria Calvo::38
Roberto:Madrid:20
Carles:Barcelona:37
```

Ejemplos de líneas incorrectas:

```
Fernando:Zaragoza   -> Falta el último delimitador de campo (:)
Noelia   -> Faltan dos delimitadores de campo
   :Sevilla:28   -> El nombre es obligatorio. Hay 3 espacios en el campo y esto se considera como blank.
```

a) Invocar a la función con un filtro que conserve únicamente las personas menores de 25 años. Imprimir la lista devuelta en la consola. Para los campos opcionales vacíos, imprimir: unknown. **Nota**: no deben aparecer las personas con 0 años, es decir, con edad desconocida.

Salida esperada según el fichero de ejemplo:

```
Name: Andres. Town: Madrid. Age: 19
Name: Laura Sáenz. Town: unknown. Age: 23
Name: Roberto. Town: Madrid. Age: 20
```

b) Invocar a la función con un filtro que elimine las personas cuyo nombre empiece con la letra A. Imprimir la lista devuelta en la consola. Para los campos opcionales vacíos, imprimir: unknown.

c) Usando el resultado del apartado a), crear un Stream a partir de la lista y obtener el primer elemento cuya ciudad sea Madrid. Si existe algún elemento imprimirlo. Tratar correctamente el `Optional`.

d) Usando el resultado del apartado a), crear un Stream a partir de la lista y obtener el primer elemento cuya ciudad sea Barcelona. Si existe algún elemento imprimirlo. Tratar correctamente el `Optional`.

3. Herramientas: Maven, Git

Empaquetar proyecto Maven

Nombre proyecto Maven: *block3-maven-package*

Tiempo estimado: 3 horas

- Crear un programa simple que imprima por consola: `Hello world!`.
- Compilar nuestro proyecto con Maven desde IntelliJ. Esto nos deberá crear un fichero JAR, en el directorio 'target'. Conseguir ejecutar el programa java desde línea de comandos con la instrucción:
    - java –jar XXX.jar

Hint: Para poder ejecutar nuestro JAR, hay que incluir este plugin en el fichero pom.xml.

```
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <archive>
            <manifest>
              <mainClass>com.bosonit.prueba.Prueba</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

Esto es porque para que Java sepa dónde está nuestro main, dentro del fichero JAR, hay que especificarlo de algún modo. Este plugin indica en qué clase está nuestra función main. En el ejemplo, es la clase Prueba que está en el paquete "com.bosonit.prueba".

Recordar que Java deberá estar en el path de windows.

Crear repositorio Git

Nombre proyecto Maven: *block3-create-repo-git*

Tiempo estimado: 4 horas

Nota: No es necesario utilizar GIT en línea de comandos. Utilizar el IDE con el más cómodo nos sintamos, si es necesario.

En el directorio 'PROYECTOGIT' (estará donde queramos en nuestro disco duro) crear los siguientes ficheros: 'fichero1.txt' y 'fichero2.txt'

- Editar fichero1.txt y escribir el texto "Primera línea de fichero1"
- Crear repositorio **GIT** , realizar un **commit**.
- Editar fichero2.txt y escribir el texto "Primera línea de fichero2"
- Realizar un nuevo commit.
- Crear un repositorio en github y conseguir subir estos dos ficheros. (git push)
- Volver al punto donde el fichero1 no tiene la línea "Primera línea de fichero1"
- Volver al estado final.

- Hacer un clon desde el repositorio de github en otro directorio de nuestro ordenador. Le llamaremos 'COPIA PROYECTOGIT".
- Modificar el fichero1 añadiendo la línea "Segunda línea de fichero1"
- Realizar un **commit** y **push**.

- Volver al directorio 'PROYECTOGIT'.
- Modificar el fichero1 añadiendo la línea "Tercera línea de fichero1" (no tendremos la segunda línea)
- Hacer un commit y push. **Dará un error.**
- Solucionar conflictos y subir cambios.

- En el directorio ‘COPIA PROYECTOGIT” crear la rama ‘cambio1’ y cambiarse a ella.
- Modificar el fichero2 añadiendo la línea “Segunda línea de fichero2”
- Realizar un commit y un push.

- Volver al directorio ‘‘PROYECTOGIT’.
- Hacer un pull (no es necesario) Volcar (mergear) los cambios de la rama “cambio1” en la rama principal (main).
- Subir los cambios al repositorio de Github

5. Spring Boot (Básico)

CommandLineRunner

Nombre proyecto Maven: *block5-command-line-runner*

Tiempo estimado: 4 horas.

1) Realizar programa con tres funciones que se deberán ejecutar al arrancar el programa. Una mostrará el texto “Hola desde clase inicial”, otra escribirá “Hola desde clase secundaria” y la tercera función pondrá “Soy la tercera clase”. Se deberá utilizar @Postconstruct en la primera función y la interface **CommandLineRunner** en los dos siguientes. ¿En qué orden se muestran los mensajes? ¿Por qué?

Por ejemplo:

```
@SpringBootApplication
public class Main {

          @Bean
          CommandLineRunner ejecutame()
          {
             return p ->
             {
                    System.out.println("Linea a ejecutar cuando arranque");
             };
       }
}
```

2) Modificar la tercera función para que imprima los valores pasados como parámetro al ejecutar el programa.

Carga de propiedades

Nombre proyecto Maven: *block5-properties*

Tiempo estimado: 4 horas.

1) Realizar aplicación que tenga en el fichero 'application.properties' los siguientes valores:

`greeting=Hello world!`

`my.number=100`

El programa debe imprimir por consola:

`El valor de greeting es: (valor de 'greeting')`

`El valor de my.number es: (valor de 'my.number')`

Intentar leer el valor '`new.property`', que no existe en `application.properties`. Deberá asignar a la variable el texto '`new.property no tiene valor`".

Imprime por consola:

`El valor de new.property es: (valor de 'new.property')`

Poner la variable '`new.property`' dentro del S.O.

Lanzar aplicación y ver como la presenta.

2) Hacer el ejercicio anterior, pero cambiando el fichero **`application.properties`** por "**`application.`**`yml`" (adaptando el formato) Deberían obtenerse los mismos resultados que anteriormente.

Yml es abreviatura de YAML:

Propiedades de Spring Boot:
https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

Logging

Nombre proyecto Maven: *block5-logging*

Tiempo estimado: 4 horas + 3 horas parte opcional.

Crear programa que escriba los logs tipo **Error** en un fichero y los de tipo **Warning** o inferiores solo a consola.

Recordar que todas las configuraciones posibles estan en esta URL:
https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

Pista: el filtro `ThresholdFilter` puede ser de ayuda.

Parte opcional:

Los ficheros se deberán llamar ‘spring-logging-NN.log', donde NN es el número consecutivo. Hacer que genere un fichero de log nuevo cada vez que se inicie la aplicación y que como máximo haya 5 ficheros.

Por ejemplo, el primer fichero será: spring-logging.log, el segundo spring-logging-01.log, etc.

Cada vez que se genere uno nuevo, se renombrara spring-logging.log a spring-logging-01.log, spring-logging-01.log a spring-logging-02.log y así sucesivamente. spring-logging-05.log, si existe será borrado y spring-logging-04.log será renombrado a spring-logging-05.log

Configurar el sistema de logs para que si un fichero excede los 5K de longitud también deberá rotar.

Aclaración importante: Para hacer este ejercicio completamente habría que crear un fichero ‘logback-spring.xml” o “logback.xml”.

Como crear ese fichero está fuera del ámbito del curso solo se requiere conocer muy por encima las posibilidades que tenemos a la hora de configurar nuestro sistema de logging.

Existe documentación en la página oficial del proyecto Logback (https://logback.qos.ch/) pero NO es necesario hacerlo. Estableciendo propiedades en el fichero application.properties como “logging.file.name” o "logging.logback.rollingpolicy.max-file-size" normalmente valdrá para configurar nuestro sistema de log. (ver https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#application-properties.core.logging.file.name)

Sí se requiere utilizar los diferentes niveles de log (warning, info, etc) y sacar los warning o inferiores por consola y los de error dejarlos en un fichero.

NO estar más de 3 horas con este ejercicio.

Perfiles

Nombre proyecto Maven: *block5-profiles*

Tiempo estimado: 4 horas.

Crear programa que reciba el nombre del entorno vía JVM y cargue el fichero de propiedades correspondiente al entorno. El programa debe tener 3 ficheros de propiedades, uno para cada entorno: local, INT y PRO.

Crear el programa modo consola, usando CommandLineRunner.

Cada fichero de propiedades debe contener las mismas propiedades pero diferente información. Para el ejercicio, deben contener únicamente las siguientes propiedades: “spring.profiles.active” y “bd.url”.

Para local:

```
spring.profiles.active=local

bd.url=localhost:5432
```

Para INT:

```
spring.profiles.active=INTEGRATION
bd.url=int.bosonit.com:5432
```

Para PRO:

```
spring.profiles.active=PRODUCTION
bd.url=pro.bosonit.com:5432
```

En formato YAML (YML)

```
spring:
  profiles:
     active: PRODUCTION
bd.url: pro.bosonit.com:5432
```

Nota: Recordad guardar las Run Configuration del ejercicio.

6. Spring Web

Controladores

Nombre proyecto Maven: *block6-simple-controllers*

Tiempo estimado: 4 horas.

Crear una aplicación que tenga los siguientes endpoints.

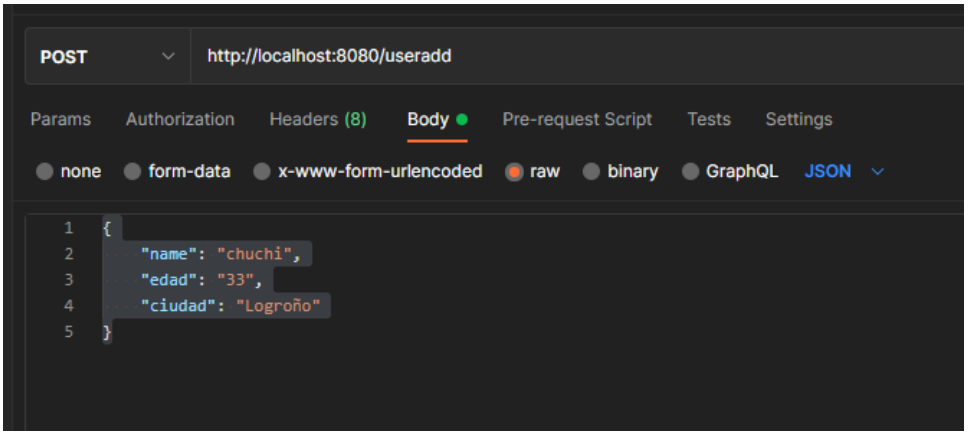
- /*user*/{nombre} (GET) //Debe devolver un String poniendo “Hola” + el contenido de la variable nombre

- /useradd (POST) //Recibe un objeto en formato JSON

Recibe un objeto de la clase `Persona`, con los campos: nombre, población y edad,

Devolver un objeto tipo persona cuya edad sea la recibida más una.

Ejemplo de cómo mandar hacer una petición POST, mandando un objeto JSON utilizando Postman.



Controladores de Persona

Nombre proyecto Maven: *block6-person-controllers*

Tiempo estimado: 4 horas.

1) Crear aplicación con dos clases de controlador. Controlador1.java y Controlador2.java

En Controlador1, en la URL /controlador1/addPersona, tipo GET, en los headers tiene que recibir, el nombre, población y edad.

Utilizando una clase de servicio, se creará un objeto Persona. La llamada devolverá el objeto Persona creado.

En otra clase, crear el Controlador2, en la URL /controlador2/getPersona tiene que devolver el objeto Persona recibido en el Controlador1, multiplicando la edad por dos.

Importante: Utilizar inyección de dependencias.

2) Al arrancar el programa crear una lista de objetos tipo Ciudad. Ciudad tendrá dos campos: nombre(String) y numeroHabitantes (int)

En controlador1, en la URL /controlador1/addCiudad, petición tipo POST, se añadirá una ciudad a la lista.

En controlador2, en la URL /controlador1/getCiudad, petición tipo GET, se devolverá la lista de las ciudades existentes.

Importante: Utilizar la anotación @Bean.

3) Crear 3 objetos Persona diferentes con funciones que tengan la etiqueta @Bean. La primera función pondrá el nombre a 'bean1', el segundo a "bean2" y el tercero a "bean3". Usar @Qualifiers

En un controlador con la URL /controlador/bean/{bean} dependiendo del parámetro mandado devolver cada uno de los Beans. Asi: /controlador/bean/bean1 devolverá el objeto cuyo nombre sea bean1 y así sucesivamente.

Path variables y headers

Nombre proyecto Maven: *block6-path-variable-headers*

Tiempo estimado: 8 horas.

1º)

Programa ejemplo: https://spring.io/guides/gs/rest-service/

Hacer aplicación como la del ejemplo. Incluyendo:

- Petición POST: mandando un objeto JSON en el body y recibiendo ese mismo objeto JSON en la respuesta (en el body).

- Petición GET: mandando parámetros en el path (http://localhost:8080/user/{id}) y devolver el mismo id mandado

- Petición PUT: mandando Request Params (http://localhost:8080/post?var1=1&var2=2) devolver un HashMap con los datos mandados . Por ejemplo: [{var1: 1}, {var2: 2}]

7. Spring Data (Básico)

CRUD

Nombre proyecto Maven: *block7-crud*

Tiempo estimado: 16 horas.

Realizar un CRUD sobre el objeto Persona. Para ello se creará 4 clases de controladores, una para cada una de las opciones:

- Añadir Persona. Petición POST. Body: { “name”:”Jesús” …. } http://URL/persona
- Modificar por id. Petición PUT. http://URL/persona/{id} - Body se manda los datos.

  Tener en cuenta que si no se manda un campo este está a nulo y NO queremos modificar a NULO los campos.

  En el caso de que el ID no exista devolver un 404 - Persona no encontrada.
- Borrar (por id). Petición DELETE. http://URL/persona/{id} - Devolver un objeto Persona o un 404: Persona no encontrada.
- Consultar.
  - por Id Petición GET http://URL/persona/{id} Devolver Persona o un código HTTP 404 si no existe.
  - Por nombre http://URL/persona/nombre/{nombre} Devolver List<Persona>
  - Devolver todos los registros . http://URL/persona. Devolver List<Persona>

El objeto **Persona** tendrá los campos: ‘nombre’, ‘edad’ y ‘población’.

Se deberá crear una clase de servicio que será la que inyectaremos en los controladores. Esa clase tendrá la lógica para el mantenimiento de los datos.

Usar **@RequestMapping(“persona”)** para que todas las llamadas sean tipo http://URL/persona/...

CRUD con validación

Nombre proyecto Maven: *block7-crud-validation*

Tiempo estimado: 8 horas parte 1 + 4 horas parte 2 + 8 horas parte 3.

Realizar commit al acabar cada apartado del ejercicio.

Parte 1 - Crear CRUD de Persona

```
table persona
{
  id_persona int [pk, increment]
  usuario string [not null max-length: 10 min-length: 6]
  password string  [not null]
  name string [not null]
  surname string
  company_email string  [not null ]
  personal_email string [not null]
  city string [not null]
  active boolean  [not null]
  created_date date  [not null]
  imagen_url string
  termination_date date
}
```

Realizar validaciones necesarias con lógica en java (no usar etiqueta @Valid)

if (usuario==null) {throw new Exception(“Usuario no puede ser nulo”); }

if (usuario.length()>10) { throw new Exception(“Longitud de usuario no puede ser superior a 10 caracteres)

…

EI ID autoincremental se puede hacer con estas simples instrucciones:

```
@GeneratedValue
private int id;
```

Poner 3 endpoints en la búsqueda.

- Buscar por ID
- Buscar por nombre de usuario (campo usuario)
- Mostrar todos los registros.

Usar **DTOS**, interfaces y clases de servicio.

Nota: No es necesario crear la carpeta repository. Para hacer más simple el ejercicio se pueden poner todos los servicios en application.

Parte 2 - Crear excepciones

Crear dos tipos de excepción al CRUD anteriormente realizado:

- **EntityNotFoundException** que generará un código HTTP 404. Se lanzará cuando no se encuentren registros en un findById o si al borrar o modificar un registro este no existe.
- **UnprocessableEntityException** que devolverá un 422 (**UNPROCESSABLE** ENTITY) cuando la validación de los campos no cumpla los requisitos establecidos, al modificar o insertar un registro.

Ambas excepciones deberán devolver un objeto llamado **CustomError** con los campos:

Date timestamp;

Int HttpCode;

String mensaje; // Devolverá el mensaje de error.

Parte 3 - Relaciones entre entidades

Añadir las siguientes tablas:

table student

{

id_student string [pk, increment]

id_persona string [ref:- persona.id_persona] -- Relación OneToOne con la tabla persona.

num_hours_week int [not null] -- Número de horas semanales del estudiante en formación

coments string

id_profesor string [ref: > profesor.id_profesor] -- Un estudiante solo puede tener un profesor.

branch string [not null] -- Rama principal delestudiante (Front, Back, FullStack)

}

table profesor

{

id_profesor string [pk, increment]

id_persona string [ref:- persona.id_persona] -- Relación OneToOne con la tabla persona.

coments string

branch string [not null] -- Materia principal que imparte. Por ejemplo: Front

}

table estudiante_asignatura

{

id_asignatura String [pk, increment]

id_student string [ref: > student.id_student] -- Un estudiante puede tener N asignaturas

asignatura string -- Ejemplo: HTML, Angular, SpringBoot...

coments string

initial_date date [not null], -- Fecha en que estudiante empezó a estudiar la asignatura

finish_date date -- Fecha en que estudiante termina de estudiar la asignatura

}

--- Ejemplo de entidades --

```java
@Entity
@Table(name = "estudiantes")
@Getter
@Setter
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    Integer id_student;
    @OneToOne
    @JoinColumn(name = "id_persona")
    Persona persona;
    @Column(name = "horas_por_semana")
    Integer num_hours_week;
    @Column(name = "comentarios")
    String coments;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "id_profesor")
    Profesor profesor;
    @Column(name = "rama")
    String branch;
    @OneToMany
    List<Alumnos_Estudios> estudios;
}

@Entity
@Table(name = "estudios")
```

```java
@Getter
@Setter

public class Alumnos_Estudios {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    Integer id_study;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "profesor_id")
    Profesor profesor;
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "id_student")
    Student student;
    @Column(name = "asignatura")
    String asignatura;
    @Column(name = "comentarios")
    String comment;
    @Column(name = "initial_date")
    Date initial_date;
    @Column(name = "finish_date")
    Date finish_date;

...
```

La relación gráfica entre las tablas la tenéis en este enlace:
https://dbdiagram.io/d/60938b29b29a09603d139d64

La relación de la tabla student y profesor con Persona es one-to-one

La relación entre estudiante y profesor es tipo one-to-many. Un estudiante puede tener un solo profesor y un profesor puede tener N estudiantes.

Un estudiante puede estar en N estudios por lo cual la relación es many-to-many.

Realizar CRUD sobre TODAS las tablas. Cuando se busque por ID la persona, si es estudiante, debe devolver los datos de 'estudiante', 'profesores' y los estudios realizados por el estudiante. Si la persona es profesor, sacará los datos de la entidad profesor relacionada, los estudiantes a su cargo, y para cada estudiante los estudios realizados

Como se puede apreciar los índices son tipo String y autoincrementales. Recordar que en la primera parte de este ejercicio tenéis cómo crear campos autoincrementales del tipo String.

a) Realizar CRUD de estudiante con endpoints similares a los de persona. De momento, no poner relación con profesor (id_profesor)

   Modificar endpoint "/estudiante/{id}" para que acepte el QueryParam 'ouputType'. Por defecto deberá tener el valor 'simple' (es decir si no se manda cogerá como valor el literal 'simple').

En el caso de que el valor de ‘outputType’ sea ‘simple‘, la consulta devolverá sólo los datos del estudiante. En el caso de que sea ‘full’ devolverá los datos del estudiante y de la persona asociada.

Ejemplo:

GET “http://localhost:8080/{id}?outputType=full”

```
{
 id_student : 1,
 num_hours_week : 40,
 coments: “No puede trabajar por las tardes”,
id_persona: 1,
user: ‘francisco’,
password: “secreto”,
Name: “Francisco”,
Surname: “perez”,
company_email: “francisco@bosonit.com”,
personal_email: “francisco@gmail.com”,
city : “zaragoza”,
Active: true
created_date: ‘2021-10-03'
imagen_url: “http://pictures.com/imagen1.png”,
termination_date: null
}
```

Como se puede observar habrá que crear diferentes DTOs de salida.

b) Realizar CRUD de las tablas restantes.

Tener en cuenta que UNA persona solo puede ser o profesor o estudiante.

En TODOS los endpoints de búsqueda de la entidad persona (por ID, por nombre o todas las personas), aceptar un parámetro que indique si debe devolver solo los datos de la persona o también los de estudiante o profesor si fuera alguno de ellos.

c) Crear endpoint en ‘estudiante_asignatura’ que permita buscar por id de estudiante. Este endpoint sacará todas las asignaturas que tiene un estudiante.

Realizar chequeos lógicos: ¿Borrar persona si tiene un estudiante/profesor asignado? ¿Borrar asignatura si tiene estudiantes asignados?

d) Realizar endpoints de estudiante para asignarle una o más asignaturas. Crear otro endpoint para desasignar una o más asignaturas (deberá poder recibir una lista de IDs de asignaturas).

Parte 4 - Feign

En el CRUD anteriormente realizado incluir en la entidad Persona el siguiente endpoint.

@RestControler(“/profesor/{id})
ProfesorOutputDto GET getProfesor(@PathVariable int id)

Este endpoint deberá llamar al de la entidad profesor que devuelve el profesor por id, en el puerto 8081 usando RestTemplate.

Usando Feign:

Crear la llamada anterior usando feign.

Para hacer la prueba deberemos tener dos instancias de nuestro programa lanzado. Una corriendo en el puerto 8080 y otra en el puerto 8081. La prueba consistirá en llamar a GET localhost:8080/persona/profesor/1 y que ésta llamada llame a localhost:8081/profesor/1 devolviéndonos los datos del profesor.

10. Docker

Dockerizar aplicación

Nombre proyecto Maven: *block10-dockerize-app*

Tiempo estimado: 2 horas.

Crear imagen Docker que incluya una aplicación Spring Boot que se conecte a una base de datos PostgreSQL.

Levantar un servidor de PostgreSQL que no sea accesible desde nuestro Windows, es decir, no mapear el puerto de postgres a nuestro host (“-p 5432:5432”). El usuario para conectarse será ‘postgres’ y el password: ‘contrasena’. Se creará la base de datos ‘test’.

Deberemos tener en un contenedor la aplicación de Spring Boot y el servidor de PostgreSQL deberá estar en otro contenedor, dentro de la misma red virtual para que ambos contenedores Docker se vean entre ellos.

Desde Windows deberemos poder hacer peticiones con postman a la aplicación en Spring Boot.

Pistas:

- Poner el driver de PostgreSQL y configurar fichero application.properties para conectarte a Postgresql. Mirar este articulo como guía: https://dzone.com/articles/bounty-spring-boot-and-postgresql-database
- Crear una red interna: docker network create mynetwork

- Ejecutar postgres especificando que este en una red definida por nosotros: docker run --network mynetwork --name postgres_test -ePOSTGRES_USER=postgres -e POSTGRES_PASSWORD=contrasena -e POSTGRES_DB=test postgres

11. Spring Web (Avanzado)

CORS

Nombre proyecto Maven: *block7-crud-validation* (hay que modificar el proyecto existente)

Tiempo estimado: 2 horas.

Permitir realizar peticiones entre dominios modificando el módulo del ejercicio *CRUD con validación* para la ruta '/person'.

Para probar ir a la página: https://codepen.io/de4imo/pen/VwMRENP .

Actualizar el back para que funcionen estas llamadas:

Alta: Tipo. POST - Ruta: http://localhost:8080/addperson

Objeto mandado:

```
'usuario': document.getElementById('username').value,
'password': document.getElementById('passwd').value,
'name': document.getElementById('name').value,
'surname': document.getElementById('surname').value,
'company_email': document.getElementById('emailcomp').value,
'personal_email': document.getElementById('emailpers').value,
'city': document.getElementById('city').value,
'active': document.getElementById('active').checked,
'created_date': document.getElementById('created_date').value,
'imagen_url': document.getElementById('imagen_url').value,
'termination_date': document.getElementById('finish_date').value,
})
```

Consulta: http://localhost:8080/getall

List<Person> (Mismos campos que en el add)

Subir y bajar ficheros

Nombre proyecto Maven: *block11-upload-download-files*

1)

- Permitir subir un fichero incluyendo como metadato la categoría. Guardar el fichero y en una tabla el metadato, nombre de fichero, fecha de subida, etc. Devolver Entidad ‘Fichero’ con los datos, incluyendo un ID único.
- Descargar fichero, buscándolo por diferentes métodos (id y nombre de fichero).

2) Crear programa con estos endpoints.

- Petición POST. /upload/{tipo} (@RequestParam("file") MultipartFile file,

RedirectAttributes redirectAttributes).

Solo aceptara ficheros con la extensión indicada en la URL

- Petición GET /setpath?path={directorio_a_guardar}

El programa al arrancar permite mandar un parámetro que es el directorio donde debe guardar los ficheros. Ejemplo: java –jar MIPROGRAMA.jar “/DIRECTORIO_A_GUARDAR”. Si no se especifica esta variable ponerlo en el directorio donde se lanza java.

12. Mensajería

Kafka

Nombre proyecto Maven: *block12-kafka*

Usando Kafka, montar dos aplicaciones: una que envíe mensajes y la segunda le responda con otros mensajes.

13. Spring Data (Avanzado)

CriteriaBuilder

Nombre proyecto Maven: *block7-crud-validation* (hay que modificar el proyecto existente)

1) Sobre el ejercicio Crear endpoint que permita buscar aquellas personas cuyo ‘user’, ‘name’, ‘surname‘ o ‘fecha creación’ superior a una dada y/o inferior a una dada. Incluir otro campo que indicará si se quiere ordenar el resultado por ‘user’ o ‘name’. Tener en cuenta que cualquiera de esos campos puede ser mandados, pero ninguno es obligatorio.

2) Incluir paginación. Es decir que se pueda mostrar N elementos a partir de la página dada. Así si se define que el tamaño de la página es 10 y que se quiere mostrar la página 1, se mostrará del registro 11 al 20 (ambos inclusive). Por defecto el tamaño de la página será 10 y el número de página será obligatorio mandarlo.

MongoDB

Nombre proyecto Maven: *block13-mongodb*

Realizar CRUD de personas desde cero, utilizando MongoTemplate. Cuando recuperas la lista de objetos que no te la devuelva completa, que la devuelva en páginas.

14. Testing con JUnit avanzado

Testing CRUD

Nombre proyecto Maven: *block7-crud-validation* (hay que modificar el proyecto existente).

Hacer testing de ejercicio CRUD básico.

Ejecutar SonarQube sobre el ejercicio y que de un 70% de cobertura del código.

15. Spring Security

Login

Nombre proyecto Maven: *block7-crud-validation* (hay que modificar el proyecto existente)

Se añadirá a la tabla personas el campo

admin boolean [not null]

Después se creará el endpoint /login que pedirá usuario y password devolviendo un token JWT firmado si los datos son válidos comprobando contra la tabla 'personas'.

Si el usuario es admin podrá acceder a todos los endpoints, si no lo es, solo a los endpoints de consulta.

Tener en cuenta que en este ejercicio NO estamos usando OAUTH2.

16. Spring Cloud

Proyecto Cloud

Nombre proyecto Maven: *block16-spring-cloud*

Primero se creará una aplicación backend con dos entidades:

Cliente.

Con las variables:

Id, nombre, apellido, edad, email y teléfono.

Los endpoints serán un crud básico, crear borrar, actualizar y buscar tanto por id como obtener todos los clientes.

Viaje (Es un autobús de 40 plazas).

Con las variables:

Id, origin, destination, departureDate, arrivalDate, passenger y status.

Los endpoints serán los siguientes:

- Crud básico para cada viaje.
- Añadir pasajero a viaje. Usaremos el id de cada uno, se verá de la siguiente forma:
    - localhost:8080/trip/addPassenger/4/8

- Haremos una cuenta de pasajeros que hay en cada viaje, ya que al añadir un pasajero a cada viaje deberá de limitarse a la cuenta de 40 pasajeros.
    - localhost:8080/passenger/count/{tripId}
- Un endpoint para cambiar la disponibilidad del viaje ya que es posible que el autobús se averíe.
    - localhost:8080/trip/{tripId}/{tripStatus}
- Finalmente crearemos un endpoint que nos indique la disponibilidad del viaje.
    - localhost:8080/trip/verify/{tripId}

Hemos de tener en cuenta que ambas entidades están conectadas a una base de datos.

Segundo haremos la aplicación backend-Front que constará de la siguiente entidad:

Ticket. (Se guarda en base de datos diferente a la de viajes o clientes)

Con las variables:

id, passengerId, passengerName, passenger Lastname, passgener Email, tripOrigin, tripDestination, departureDate y arrivalDate.

Tendremos que crear dos entidades a parte exactamente iguales que las del backend pero sin acceso a la base de datos, ya que con RestTemplate recogeremos variables de tipo cliente y viaje.

Como es un caso de práctica resumimos los endpoint a uno solo, que será el que cree el ticket, obteniendo el pasajero a través del id y añadiendolo al viaje.

- localhost:8080/generateTicket/{userId}/{tripId}

Para esta parte necesitaremos conocer el uso de RestTemplate o Feign, que hará uso de la aplicación backend ejecutándose de forma simultánea a esta.

Para esta aplicación se podrá usar la misma base de datos u otra diferente (si es así mejor, se entenderá mejor por que los microservicios son tan usados).

Tercero crearemos la aplicación eureka-naming-server.

Como has visto en la teoría poco hay que explicar en la hora de su desarrollo.

Cuarto crearemos la aplicación api-gateway, que nos abrirá los puertos y permitirá relacionar nuestros microservicios con eureka.

Finalmente dockeriza esta aplicación, respetando como dependen de las bases de datos y entre sí, es recomendable utilizar docker-compose.