Curso de desarrollo de software

Práctica Calificada 4

Instrucciones

- Construye un repositorio llamado PracticaCalificada5 donde coloques tus respuestas
- Evita copiar y pegar de otros repositorios. Cualquier evidencia de copia implica la anulación de la evaluación. No puedes usar ningún repositorio en la evaluación.
- Evita solo copiar imágenes como respuesta y añade un archive Readme en formato markdown que indique como se han desarrollado cada una de las preguntas. La presentación de código sin explicación no será revisado.
- Documenta con palabras técnicas las respuestas a todas las preguntas.
- Pueden utilizar la documentación de las herramientas utilizadas.
- Debes entregar en la plataforma del curso, el URL donde se alojan todas tus respuestas.

Ejercicio 1 (7 puntos)

Esta actividad es para completar algunas lineas de código, por lo que para que puntue debes resolver todos los ítems. Sino se puede puntuar.

En muchos de los ejemplos una clase de objeto de acceso a datos https://en.wikipedia.org/wiki/Data_access_object (DAO) es responsable de recuperar o conservar información en la base de datos. Sin embargo, en algún momento, debes probar estas clases. Estas DAO a menudo realizan consultas SQL complejas y encapsulan una gran cantidad de conocimiento comercial, lo que requiere que los desarrolladores gasten algo de energía para asegurarse de que produzcan los resultados esperados.

SQL es un lenguaje robusto y contiene muchas funciones diferentes que podemos usar. Podemos las consultas como una composición de predicados.

Como desarrollador, un criterio posible es ejercitar los predicados y verificar si la consulta SQL devuelve los resultados esperados cuando los predicados se evalúan con resultados diferentes. Prácticamente todas las técnicas de prueba se pueden aplicar aquí:

- Pruebas basadas en especificaciones: las consultas SQL surgen de un requisito. Un desarrollador puede analizar los requisitos y derivar particiones equivalentes que deben probarse.
- Análisis de límites: tales programas tienen límites. Debido a que esperamos que los límites sean lugares con una alta probabilidad de errores, es importante ejercitarlos.
- Pruebas estructurales: las consultas de SQL contienen predicados y un desarrollador puede usar la estructura de SQL para derivar casos de prueba.

En este ejercicio, nos centramos en las pruebas estructurales. Si miramos el siguiente ejemplo SQL y tratamos de hacer una analogía con lo que hemos discutido sobre las pruebas estructurales,

```
SELECT * FROM FACTURA WHERE VALOR > 50 AND VALOR < 200
```

vemos que la consulta SQL contiene una sola rama compuesta por dos predicados ¿Identifica cuales son?. Esto significa que hay cuatro posibles combinaciones de resultados en estos dos predicados: (true, true), (true, false), (false, true) y (false, false). Podemos apuntar a cualquiera de los siguientes:

- Cobertura de rama: en este caso, dos pruebas (una que haga que la decisión general se evalúe como true y otra que haga que se evalúe como false) serían suficientes para lograr una cobertura del 100 %.
- Condición + cobertura de rama: en este caso, tres pruebas serían suficientes para lograr el 100 % de condición + cobertura de rama: por ejemplo, T1 = 150, T2 = 40, T3 = 250.

Podemos usar JUnit para escribir pruebas SQL. Todo lo que necesitamos hacer es (1) establecer una conexión con la base de datos, (2) asegurarnos de que la base de datos esté en el estado inicial correcto, (3) ejecutar la consulta SQL y (4) verificar el resultado. Considera el siguiente escenario:

- Tenemos una tabla **Factura** compuesta por nombre (varchar, longitud 100) y un valor (doble).
- Tenemos una clase **FacturaDao** que usa una API para comunicarse con la base de datos. La API precisa no importa.
- Este DAO realiza tres acciones: **guardar()** conserva una factura en la base de datos, **todo()** devuelve todas las facturas de la base de datos y **todosConAlMenos** () devuelve todas las facturas con al menos un valor especificado. Específicamente,
    - guardar() ejecuta INSERT INTO factura (nombre, valor) VALORES(?,?)
    - todo() ejecuta SELECT * FROM factura.
    - todosConAlMenos() ejecuta SELECT * FROM factura WHERE valor >= ?.

Pregunta 1 (2 puntos): Escribe un listado (FacturaDao.java) donde se muestre una implementación JDBC simple de la clase Factura.java

Código Factura.java

```
import java.util.Objects;

public class Factura {
   public final String cliente;
   public final int valor;


   public Factura(String cliente, int valor) {
      this.cliente = cliente;
      this.valor = valor;
   }
   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      Factura factura = (Factura) o;
      return valor == factura.valor &&
```

```
            cliente.equals(factura.cliente);
    }
    @Override
    public int hashCode() {
        return Objects.hash(cliente, valor);
    }
    @Override
    public String toString() {
        return "Factura{" +
                "cliente='" + cliente + '\'' +
                ", valor=" + valor +
                '}';
    }
    public int obtenerValor() {
        return valor;
    }
}
```

Probemos la clase FacturaDao. Recuerda, queremos aplicar las mismas ideas que hemos visto hasta ahora. La diferencia es que tenemos una base de datos en el bucle.

Comencemos con **todo().**

Este método envía SELECT * FROM factura la base de datos y obtiene el resultado. Pero para que esta consulta devuelva algo, primero debemos insertar algunas facturas en la base de datos. La clase FacturaDao que has implementado debe proporcionar un método **guardar(),** que envía una consulta INSERT. Esto es suficiente para la primera prueba.

Pregunta 2 (1 punto): Implementa esta primera prueba en un archivo llamado FacturaDaoIntegracionTest.java.

En términos de pruebas, tu implementación debe ser correcta. Sin embargo, dada la base de datos, tenemos algunas preocupaciones adicionales. En primer lugar, no debemos olvidar que la base de datos conserva los datos de forma permanente. Supongamos que comenzamos con una base de datos vacía. La primera vez que ejecutamos la prueba, persistirán dos facturas en la base de datos.

La segunda vez que ejecutemos la prueba, persistirán dos facturas nuevas, por un total de cuatro facturas. Esto hará que nuestra prueba falle, ya que se espera que la base de datos tenga una y dos facturas, respectivamente.

Al probar con una base de datos real, debemos asegurar un estado limpio: antes de que se ejecute la prueba, abrimos la conexión de la base de datos, limpiamos la base de datos y (opcionalmente) la ponemos en el

estado que necesitamos antes de ejecutar la consulta SQL bajo prueba. Después de que se ejecute la prueba, cerramos la conexión de la base de datos.

Pregunta 3 (1 punto): Implementa en código y utilizando @BeforeEach y @AfterEach de JUnit y muestra los resultados.Utiliza el método openConnectionAndCleanup() s qie e anota como @BeforeEach, lo que significa que JUnit ejecutará la limpieza antes de cada método de prueba. En este momento, tu implementación es simple.

También abrimos la conexión a la base de datos manualmente, usando la llamada API más rudimentaria de JDBC, getConnection. (En un sistema de software real, probablemente usamos Hibernate o Spring Data una conexión de base de datos activa). No olvides de cerrar la conexión en el método close() (que ocurre después de cada método de prueba) para este caso.

Continuamos con el método **todosConAlMenos** ()

Aquí la consulta SQL contiene un predicado, where valor >= ?. Esto significa que tenemos diferentes escenarios para hacer el ejercicio. Aquí podemos usar todo nuestro conocimiento sobre las pruebas de límites.

Por ejemplo:

Explora el límite: valor >= x

Punto1: x

Punto2:x-1

Punto3: x+1

La estrategia que usamos para derivar el caso de prueba es muy similar a lo que hemos visto anteriormente. Ejercitamos los puntos1 y punto 2 y luego nos aseguramos de que el resultado sea correcto. Dado where valor >= ?, ¿dónde reemplazamos concretamente ? con 50, tenemos 50 como punto1 y 49 como punto2 (valor - 1 en inv1 usa es variable). Además, probamos un solo punto.

Tus pruebas deben ejecutarse en una base de datos de prueba, una base de datos configurada exclusivamente para tus pruebas. No hace falta decir que no deseas ejecutar tus pruebas en la base de datos de producción.

Una estrategia que se aplica a menudo es crear una clase base para las pruebas de integración: por ejemplo, SQLIntegrationTestBase. Esta clase base maneja toda la magia, como crear una conexión, limpiar la base de datos y cerrar la conexión. Luego, la clase de prueba, como FacturaDaoTest, que extendería SQLIntegrationTestBase, se enfoca solo en probar las consultas SQL.

Pregunta 4 (2 punto): Implementa la clase SQLIntegrationTestBase con las siguientes características:

- Hacer que FactureDao esté protegido para que podamos acceder a él desde las clases hijos.
- Los métodos son los mismos que antes.
- FacturaDaoTest ahora amplía SqlIntegrationTestBase.
- La clase de prueba se centra en las pruebas mismas, ya que la clase base maneja la infraestructura de la base de datos.

Pregunta 5 (1 punto): Puedes agregar alguna característica adicional como la apertura y confirmación de la transacción, restablecer el estado de la base de datos o algunos métodos auxiliares que reducen la cantidad de código en las prueba .

Ejercicio 2 (4 puntos)

En este ejercicio debes mostrar los resultados y archivos generados en tus resultados. Evita presentar solo imágenes. Explica tus configuraciones.

Método 1 (2 puntos): Usando un Dockerfile

El primer método que veremos para construir una base de imágenes de contenedor es crear un Dockerfile y luego ejecutaremos un comando docker image build para obtener una imagen NGINX.

Escribe el siguiente Dockerfile una vez más:

```
FROM alpine:latest

LABEL maintainer=" Cesar Lara Avila<checha@claraa.io>"

LABEL description="Este Dockerfile de ejemplo instala NGINX"

RUN apk add --update nginx && \

   rm -rf /var/cache/apk/* && \

   mkdir -p /tmp/nginx/

COPY files/nginx.conf /etc/nginx/nginx.conf

COPY files/default.conf /etc/nginx/conf.d/default.conf ADD

files/html.tar.gz /usr/share/nginx/

EXPOSE 80/tcp

ENTRYPOINT ["nginx"]

CMD ["-g", "daemon off;"]
```

No olvides que también necesitarás los archivos default.conf, html.tar.gz y nginx.conf en la carpeta de archivos. Una vez que esté construida la image , debes ejecutar el comando apropiado para verificar si la imagen está disponible, así como el tamaño de su imagen:.

Puedes lanzar un contenedor con tu imagen recién construida ejecutando este comando:

$ docker container run -d --name dockerfile-example -p 8080:80 local:dockerfile-example

Esto lanzará un contenedor llamado dockerfile-example. Puedes comprobar si se está ejecutando mediante el siguiente comando:

$ docker container ls

Abriendo tu navegador y yendo a http: /localhost: 808/ debería mostrarte una página web extremadamente simple.

A continuación, ejecutaremos rápidamente algunos de los comandos:

$ docker container run --name nginx-version local: dockerfile- example -v

El siguiente comando que veremos en ejecución muestra las etiquetas que insertamos en el momento de la compilación. Para ver esta información, ejecuta el siguiente comando:

$ docker image inspect -f {{.Config.Labels}} local: dockerfile- example

Antes de continuar, puedes detener y eliminar los contenedores que lanzamos con los siguientes comandos:

$ docker container stop dockerfile-example

$ docker container rm dockerfile-example nginx-version

Método 2 (1 punto) : Usando un contenedor existente

La forma más sencilla de crear una imagen base es comenzar utilizando una de las imágenes oficiales de Docker Hub. Docker también mantiene el Dockerfile para estas compilaciones oficiales en sus repositorios de GitHub. Por lo tanto, hay al menos dos opciones para usar imágenes existentes que otros ya han creado. Al usar Dockerfile, puedes ver exactamente lo que se incluye en la compilación y agregar lo que necesitas. Luego, puedes controlar la versión de ese Dockerfile si deseas cambiarlo o compartirlo más tarde.

Primero, debemos descargar la imagen que queremos usar como base; como hicimos anteriormente, usaremos Alpine Linux:

$ docker image pull alpine: latest

A continuación, necesitamos ejecutar un contenedor en primer plano para que podamos interactuar con él:

$ docker container run -it --name alpine-test alpine / bin / sh

Una vez que se ejecuta el contenedor, puedes agregar los paquetes según sea necesario usando el comando apk, o cualquiera que sean los comandos de administración de paquetes para su versión de Linux.

Por ejemplo, los siguientes comandos instalarán NGINX:

$ apk update

$ apk upgrade

$ apk add --update nginx

$ rm -rf /var/cache/apk/*

$ mkdir -p /tmp/nginx/

$ exit

Después de haber instalado los paquetes que necesita, debes guardar el contenedor. El comando exit al final del conjunto de comandos anterior detendrá el contenedor en ejecución, ya que el proceso de shell del que nos estamos separando resulta ser el proceso que mantiene el contenedor en ejecución en primer plano.

Es en este punto que realmente debería detenerse.

$ docker container commit <container_name> <REPOSITORY>: <TAG>

Por ejemplo, ejecuta el siguiente comando para guardar una copia del contenedor que lanzamos y personalizamos:

$ docker container commit alpine-test local: broken-container

¿Por qué lo llamé broken-container? Dado que uno de los casos de uso para adoptar este enfoque es que si, por alguna razón, tienes un problema con un contenedor, entonces es extremadamente útil guardar el contenedor fallido como una imagen, o incluso exportarlo como un archivo TAR para compartir con otros si necesita ayuda para llegar a la raíz del problema.

Para guardar el archivo de imagen, simplemente ejecute el siguiente comando:

$ docker image save -o <name_of_file.tar> <REPOSITORY>:<TAG>

Entonces, para el ejemplo, ejecuta el siguiente comando:

$ docker image save -o broken-container.tar local:broken-container

Método 3 (1 punto) : Desplegando la imágen desde cero

Es importante desplegar tus propias imágenes desde cero. Ahora, cuando sueles escuchar la frase desde cero, literalmente significa que empiezas de cero. Ahora, esto puede ser un beneficio porque mantendrá el tamaño de la imagen muy pequeño, pero también puede ser perjudicial si no has realizado las actividades de clase en Docker, ya que puede ser complicado.

Docker ya ha hecho parte del trabajo duro por nosotros y ha creado un archivo TAR vacío en Docker Hub llamado scratch; que puedes usarlo en la sección FROM de tu Dockerfile.

Puedes basar toda tu compilación de Docker en esto y luego agregar partes según sea necesario. Nuevamente, usaremos Alpine Linux como el sistema operativo base para la imagen.

Para descargar una copia, simplemente seleccione la descarga apropiada de la página de descargas, que se puede encontrar en https://www.alpinelinux.org/downloads/.

Una vez que haya terminado de descargarse, debes crear un Dockerfile que use **scratch** y luego agregar el archivo tar.gz, asegurándose de usar el archivo correcto, como se muestra en el siguiente ejemplo:

FROM scratch

ADD files/alpine-minirootfs-3.11.3-x86_64.tar.gz /

CMD [“/bin/sh”]

En este ejercicio ¿por qué acabo de descargar el archivo alpine-minirootfs-3.11.3-x86_64.tar.gz? ¿No podrías haber usado http://dl-cdn.alpinelinux.org/alpine/v3.11/releases/x86_64/alpine-minirootfs-3.11.3-x86_64.tar.gz en su lugar?

Ahora que tienes el Dockerfile, puedes construir la imagen como lo hubiéramos hecho en cualquier otra imagen de Docker, ejecutando el siguiente comando:

$ docker image build --tag local:fromscratch .

Puedes comparar el tamaño de la imagen con las otras imágenes de contenedor que hemos creado ejecutando el siguiente comando:

$ docker image ls

Ahora que se ha creado la propia imagen, podemos probarla ejecutando este comando:

$ docker container run -it --name alpine-test local: fromscratch /bin/sh

Si recibes un error, es posible que ya tenga un contenedor llamado alpine-test creado o en ejecución. Elimínalo ejecutando docker container stop alpine-test, seguido de docker container rm alpine-test. Esto debería lanzarnos a un shell en la imagen de Alpine Linux.

Puedes verificar esto ejecutando el siguiente comando:

$ cat / etc / * release

Esto mostrará información sobre la versión que se está ejecutando el contenedor. Si bien todo parece sencillo, esto es solo gracias a la forma en que Alpine Linux empaqueta su sistema operativo. Puede comenzar a complicarse más cuando elige usar otras distribuciones que empaquetan sus sistemas operativos de una manera diferente. Hay varias herramientas que se pueden utilizar para generar un paquete de un sistema operativo.

Nota: en este ejercicio puedes elegir otra imagen si deseas, pero deben ser distintas a las utilizadas en las actividades.

Ejercicio 4 (7 puntos)

Esta actividad es para completar algunas lineas de código, por lo que para que puntue debes resolver todos los ítems. Sino se puede puntuar.

Como parte de la aparición de Spring Boot, surgió un sitio web relacionado (mantenido por el equipo de Spring): Spring Initializr (start.spring.io).

start.spring.io viene con las siguientes características clave:

- Podemos seleccionar la versión de Spring Boot que deseamos utilizar
- Podemos elegir nuestra herramienta de compilación preferida (Maven o Gradle)
- Podemos ingresar las coordenadas de un proyecto (artefacto, grupo, descripción, etc.)
- Podemos seleccionar en qué versión de Java se construirá un proyecto
- Podemos elegir los distintos módulos (Spring y de terceros) para usar en el proyecto

Comienza creando un prototipo de acuerdo a la realizado en clase, seleccionando la herramienta de compilación, el idioma y la versión de Spring Boot que deseamos usar, etc

Haz clic en ADD DEPENDENCIES escribe la palabra web y observa cómo Spring Web sube a la parte superior de la lista. Al hacer clic en el botón Generate, se iniciará la descarga de un archivo ZIP que contiene un proyecto vacío con un archivo de compilación que contiene todas las configuraciones para nuestro proyecto.

Descomprime el archivo ZIP del proyecto y abre el archivo dentro de tu IDE favorito,

La pieza de Spring que permite escribir controladores web es Spring MVC.

Spring MVC es el módulo de Spring Framework que permite crear aplicaciones web sobre contenedores basados en servlets utilizando el paradigma Model-View-Controller (MVC).

De hecho, si le das un vistazo al archivo pom.xml en la raíz del proyecto, se verá una dependencia crítica:

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

Esta dependencia coloca a Spring MVC en el classpath del proyecto. Esto da acceso a las anotaciones de Spring MVC y otros componentes, lo que permite definir controladores web. Su mera presencia activará los ajustes de configuración automática de Spring Boot para activar cualquier controlador web que se crea.

Antes de crear un nuevo controlador, es importante tener en cuenta que el proyecto ya tiene un paquete base creado según tú configuración.

Pregunta 1: Comienza creando una nueva clase dentro de este paquete y llámela ControladorBase. A partir de ahí, escribe el siguiente código:

```
@Controller
public class ControladoBase {
  @GetMapping("/")
  public String index() {
  return "index";
  }
}
```

¿Qué crees que significan @Get Mapping @Controller, index?

Es posible retomar un proyecto ya existente y hacer modificaciones usando start.spring.io. Si bien podemos escribir HTML a mano, hoy en día es más fácil usar motores de plantillas para hacerlo por nosotros. Como buscamos algo liviano, escoge Moustache (mustache. github.io).

La mejor manera de aumentar un proyecto existente es volver a visitar el sitio Spring Initializr y marcar todas las configuraciones, así como elegir los módulos que necesitas (especialmente los nuevos paa agregar al proyecto existente).

Suponiendo que se ingresa en la misma configuración anterior, solo necesitamos hacer clic en el botón DEPENDENCIES e ingresar mustache y repetir el proceso anterior solo que en lugar de GENERATE usa EXPLORE, que permite visualizar el proyecto obtenido directamente en el navegador.

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

¡Ahora puedes cambiar y escribir una plantilla de mustache!,

Después de crear la clase de controlador, no tienes que hacer mucho más. La función de escaneo de componentes de Spring Boot hará todo el trabajo.

De forma predeterminada, Spring Boot espera que todas las plantillas estén ubicadas en src/main/resources/templates.

Pregunta 2: Crea index.mustache dentro de src/main/resources/templates. Luego, agrega el siguiente código:

```
<h1>Hola fans de CC3S2!</h1>

<p>

    Estamos resolviendo una tarea de spring boot

 </p>
```

Dentro de tu IDE, solo tienes que hacer clic con el botón derecho en la clase ...Application que Spring Initializr creó y seleccionar Run. Una vez que esté activo, puedes navegar a localhost:8080 y ver los resultados.

Puedes ajustar el ControladorBase que acabamos de hacer de la siguiente manera (completa los comentarios)

```
@Controller

public class ControladorBase {

record Video(String name) {}

 List<Video> videos = List.of(

   new Video(" ….algun comentario ?"),

   new Video("...!"),

   new Video("….!"));

 @GetMapping("/")

 public String index(Model model) {

   model.addAttribute("videos", videos);

   return "index";

 }

}
```

Para pasar estos datos a la plantilla, necesitamos un objeto que Spring MVC entienda. Para hacer eso, necesitas agregar un parámetro Model al método de index.

Spring MVC tiene un puñado de atributos opcionales que puedes agregar a cualquier método web. Model es el tipo que usamos si necesitamos transferir datos al motor de plantillas.

El código mostrado anteriormente tiene un atributo llamado videos y se proporciona con List<Video>. Con eso en su lugar, podemos mejorar index.mustache para mostrarlo a los espectadores agregando el siguiente código:

```
<ul>
```

```
  {{#videos}}
    <li>{{name}}</li>
  {{/videos}}
</ul>
```

Pregunta 3: Vuelve a ejecutar la aplicación y luego visita localhost:8080 y muestra la plantilla actualizada en acción.

La última aplicación que construimos fue bastante ingeniosa. Rápidamente modelamos algunos datos y los presentamos en una plantilla liviana.

El problema persistente es que el diseño no es muy reutilizable. En el momento en que necesitemos otro controlador, nos encontraremos en una situación complicada por las siguientes razones:

- Los controladores no deberían administrar definiciones de datos. Dado que responden a llamadas web y luego interactúan con otros servicios y sistemas, estas definiciones deben estar en un nivel más bajo.
- Los controladores web pesados que también manejan datos dificultarán la realización de ajustes a medida que evolucionen nuestras necesidades web. Es por eso que es mejor si la gestión de datos se lleva a un nivel inferior.

Pregunta4: Realiza, la migración este registro de Video a su propia clase, Video.java.

Pregunta5: Mueve esa lista de objetos Video a un servicio separado. Crea una clase llamada ServicioVideo de la siguiente manera (completa la parte faltante). Debes implementar un método de utilidad para devolver la colección actual de objetos Video.

```
@Service
public class VideoService {
  private List<Video> videos = List.of(//
 new Video(
….
}
```

Pregunta6: Realiza las actualizaciones correspondiente para pivotar ControladorBasico para comenzar a usar esta nueva clase ServicioVideo.

Cada vez que creas una clase de Java que es recogida por la función de escaneo de componentes de Spring Boot. Spring Boot verificará si hay puntos de inyección y, si encuentra alguno, buscará en el contexto de la aplicación beans del tipo coincidente... ¡y los inyectará!

Esto se conoce como autowiring (ya vimos esto en la actividad). Dejamos que Spring maneje el problema de encontrar las dependencias de Spring Bean en el contexto de la aplicación y conectarlas por nosotros.

Pero con el surgimiento de Spring Boot y sus beans generados por configuración automática que aprovecharon masivamente el autowiring, el autowiring, a su vez, se volvió agradable para casi todos.

Las siguientes son tres formas en que puedes inyectar dependencias en una clase:

- Opción 1: la clase en sí se puede marcar con una de las anotaciones @Component de Spring Framework (o simplemente @Component en sí) como @Service, @Controller, @RestController o @Configuration.
- Opción 2: la anotación @Autowired Spring Framework marca puntos para inyectar dependencias.Se puede aplicar a constructores, métodos setter y campos (¡incluso privados!).
- Opción 3: si una clase tiene un solo constructor, no es necesario aplicar la anotación @Autowired. Spring simplemente asumirá que se va a conectar automáticamente.

Pregunta7: Con ServicioVideo inyectado en ControladorBase, actualiza el método index().

Pregunta 8: Agrega lo siguiente a index.mustache

```
<form action="/nuevo-video" method="post">
  <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

Pregunta9: Para hacer que la aplicación Spring Boot responda a POST /nuevo-video, escribe otro método de controlador en el ControladorBase (completa)

```
@PostMapping("/nuevo-video")
public String nuevoVideo(@ModelAttribute Video nuevoVideo) {
 …..create(nuevoVideo);
 return "redirect:/";
}
```

Es importante reconocer que hasta ahora, hemos estado usando el operador List.of() de Java 17 para construir la colección de videos, lo que produce una lista inmutable. Esta lista inmutable respeta la interfaz List de Java, dándonos acceso a un método add(). Si intentamos usarlo, solo generará una UnsupportedOperationException.

No, debes tomar un par de pasos adicionales si vamos a mutar esta colección inmutable.

La jugada para agregar algo inmutable es crear una nueva instancia inmutable a partir de los contenidos originales combinados con nuevos contenidos. Aquí es donde puedees aprovechar las API basadas en listas más familiares. Adiciona el siguiente código a ServicioVideo.

```
public Video crear(Video nuevoVideo) {
 List<Video> extend = new ArrayList<>(videos);
 extend.add(nuevoVideo);
```

```
  this.videos = List.copyOf(extend);

  return nuevoVideo;

}
```

Pregunta10: Explica el código anterior y con todo esos cambos re-ejecuta la aplicación y muestra los resultados.

Si haces clic en Summit, el controlador emitirá una redirección de regreso a /. El navegador navegará de regreso a la ruta raíz, lo que hará que obtenga los datos y los reproduzca el último Video.

Con eso, debes teenr una página web funcional que ofrece contenido dinámico que también nos permite agregar más contenido. Pero esto no es de ninguna manera completo.

Un ingrediente clave en la creación de cualquier aplicación web es la capacidad de proporcionar una API. En los viejos tiempos, esto era complejo y difícil de garantizar la compatibilidad.

En la actualidad, el mundo ha convergido principalmente en un puñado de formatos, muchos basados en estructuras basadas en JSON.

Una de las características poderosas de Spring Boot es que cuando agrega Spring Web a un proyecto, como hicimos al comienzo , agrega a Jackson a la ruta de clase. Jackson es una biblioteca de serialización/deserialización JSON que ha sido ampliamente adoptada por la comunidad de Java.

Referencia: https://github.com/FasterXML/jackson

La capacidad de Jackson de permitir definir cómo traducir las clases de Java de un lado a otro con la forma preferida de JSON combinado con la capacidad de Spring Boot para configurar automáticamente las cosas significa que no tienes que mover un dedo más de la configuración para comenzar a codificar una API.

Para empezar, crea una nueva clase en el mismo paquete que hemos estado usando en este ejercicio.

Llámalo ControladorApi. En la parte superior, aplica la anotación @RestController.

@RestController es similar a la anotación @Controller que usamos anteriormente. Le indica a Spring Boot que esta clase debe seleccionarse automáticamente para el escaneo de componentes como un bean Spring. Este bean se registrará en el contexto de la aplicación y también con Spring MVC como una clase de controlador para que pueda enrutar llamadas web.

Pero tiene una propiedad adicional: cambia todos los métodos web de estar basados en plantillas a estar basados en JSON. En otras palabras, en lugar de que un método web devuelva el nombre de una plantilla que Spring MVC representa a través de un motor de plantillas, serializa los resultados utilizando Jackson.

Utiliza el siguiente código:

```
@RestController

public class ControladorApi {

  private final ServicioVideo serviciovideo;

  public ControladorAPi(ServicioVideo serviciovideo) {

    this.serviciovideo = serviciovideo;
```

```
 }
 @GetMapping("/api/videos")
 public List<Video> all() {
   return serviciovideo....();  //completa
 }
}
```

Pregunta 11: Ejecuta la la aplicación ahora mismo y curl ese punto final, para ver los resultados.

curl (https://curl.se/) es una popular herramienta de línea de comandos que te permite interactuar con las API web.

Por supuesto, una API que no hace nada más que producir JSON no es una API en absoluto. Naturalmente, también necesitamos consumir JSON.

Pregunta 12: Para hacer eso, crea un método web en la clase ControladorAPi que responda a las llamadas HTTP POST.

Pregunta 13: Comprueba curl -v -X POST localhost:8080/api/videos -d '{"name": "... 3"}' -H 'Content-type:application/json'. Explica los resultados.

Pregunta 14: ¿Qué sucede si ejecutas curl localhost:8080/api/videos?

Ejercicio 4 (2puntos) Del artículo https://martinfowler.com/articles/microservices.html explica en que consiste la automatización de Infraestructura y comos se relaciona con los componentes de la arquitectura de miscroservicios detallada allí.