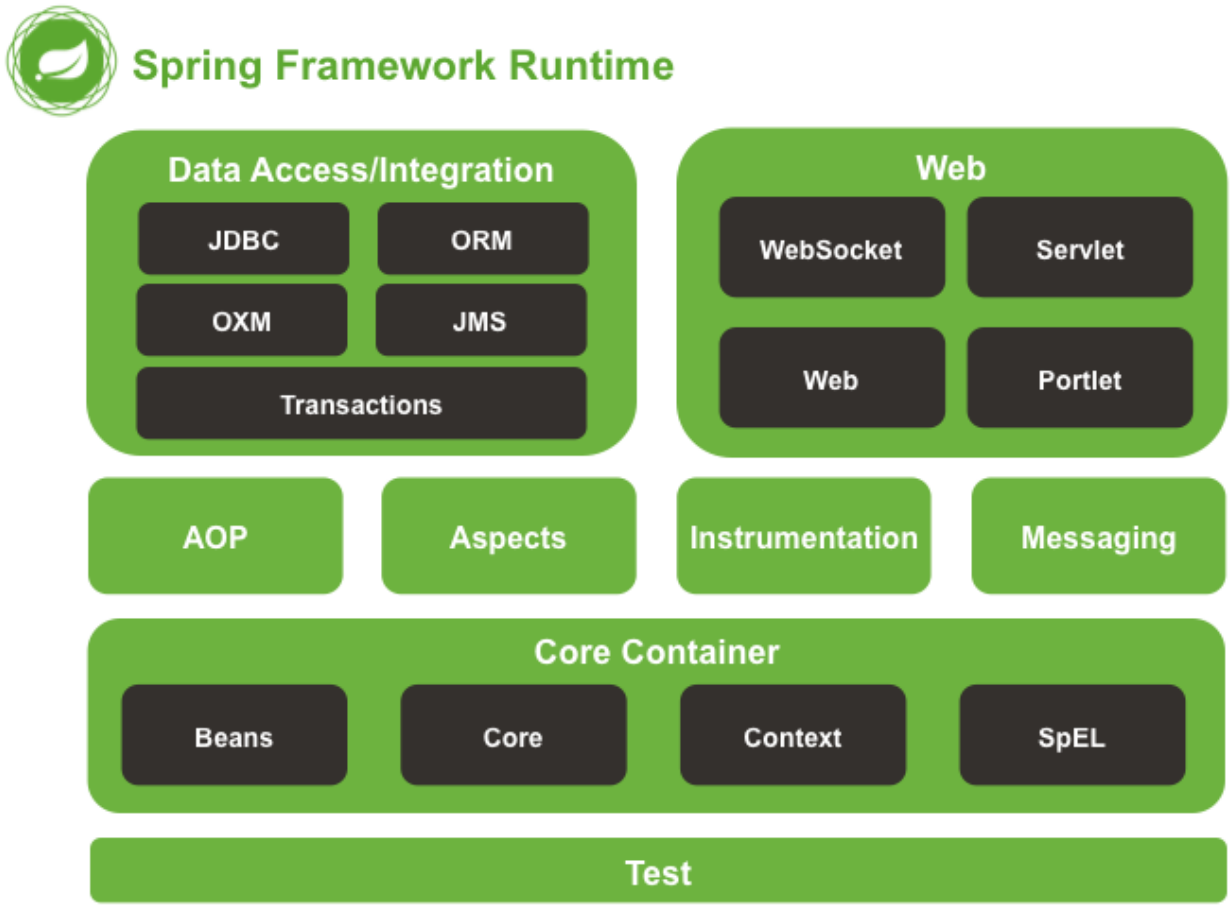
Apunte 13 - Spring Framework

Spring es un framework de código abierto que ha sido creado por miembros de la comunidad de código abierto y se ha convertido en un estándar ampliamente utilizado en el desarrollo de aplicaciones Java.

Spring se destaca por su capacidad para agilizar el proceso de desarrollo de aplicaciones, especialmente en el ámbito web. Aunque no es parte de la biblioteca estándar de Java, su popularidad entre los desarrolladores ha llevado a que sea considerado como un estándar de facto en la comunidad.

El enfoque principal de Spring es brindar herramientas que simplifiquen el desarrollo de aplicaciones Java en diferentes contextos. En particular, se ha destacado en el desarrollo web, tanto en la creación de sitios web basados en Java como en la construcción de servicios que exponen una API REST para ser consumida por un frontend desarrollado en otras tecnologías.



Spring Container

El contenedor de Spring es uno de los puntos centrales de Spring, se encarga de crear los objetos, conectarlos entre si e inicializar dichos objetos. Además controla los ciclos de vida de cada una de estas instancias mediante el patrón de Inyección de Dependencias (Dependency Injection ó DI). Si hay una función que

caracteriza al core de Spring es justamente la implementación de la Inyección de Dependencias y la Inversión de Control y vamos a abordar el tema más adelante en el presente material.

El contenedor de Spring se puede configurar mediante archivos de configuración (.xml) o código java en clases especialmente dedicadas a tal fin. En los ejemplos siguientes se verán ambos mecanismos. Algunos de los componentes que se configuran en lo que se conoce como **contexto de aplicación de Spring** son:

- Servicios que se usarán en la aplicación
- **Managed beans** o beans definidos administrados por el contenedor, individualmente o a través de un **introspección** de clases.

Los beans son la manera que tiene Spring de denominar a los objetos Java que se encuentren (viven) en su contenedor principal. Los beans se pueden declarar mediante anotaciones en POJO's (Plain Old Java Object , objetos normales de Java) o mediante XML. El siguiente ejemplo muestra como declarar un bean mediante configuración XML:

```
<bean id="service" class="org.springframework.example.services.ServiceImpl">
    <property name="itemData" ref="itemData"></property>
</bean>
```

En este ejemplo se crea un **bean** con id_service, y se le indica donde se encuentra la clase, junto con la propiedad itemData haciendo referencia a un bean creado previamente cuyo id es lo que se indica en el atributo ref.

En el contenedor Spring se suelen crear y almacenar objetos de servicio, objetos de acceso a datos (DAO's), y objetos que nos permitan conectarnos con otras partes del sistema como un sistema de colas de mensaje, por ejemplo. No se suelen configurar los objetos de dominio de nuestra aplicación para que se encargue el contenedor de Spring, ese sería el trabajo de los DAO's o los repositorios (que ser verán en detalle más adelante).

Inyección de dependencias

El patrón de Inyección de Dependencias, complementado por el patrón de Inversión de Control (IoC) es un patrón que tiene como finalidad conseguir un código más desacoplado, que facilita, entre otras cosas, tareas a la hora de hacer Tests y de cambiar partes del sistema en caso de que fuese necesario sin modificar el resto de los componentes. Esto se logra gracias a que los objetos son instanciados e inyectados por el framework (no se crean objetos mediante el operador **new**) según las relaciones de asociación definidas en las clases.

Tener el código desacoplado permite cambiar las dependencias en tiempo de ejecución basándose en cualquier factor que se considere, para ello se necesita un Inyector o Contenedor que sería el encargado de inyectar las dependencias correctas en el momento necesario.

Siguiendo el patrón de Inyección de Dependencias (DI, Dependency Injection) los componentes declaran sus dependencias, pero no se encargan de conseguirlas, ahí es donde entra el Contenedor de Spring, que será el encargado de conseguir e inyectar las dependencias a los objetos.

El siguiente código muestra un ejemplo de una clase que no usa el patrón de Inyección de Dependencia, además de estar fuertemente acopladas las dependencias, es la propia clase la que se encarga de crear una

instancia de la dependencia:

```
public class GeneradorPlaylist {
    private BuscadorCanciones buscadorCanciones;

    public GeneradorPlaylist(){
        this.buscadorCanciones = new BuscadorCanciones();
    }
    //Resto de métodos de la clase
}
```

La clase GeneradorPlaylist necesita una instancia de la clase BuscadorCanciones para funcionar, por lo que la crea manualmente mediante el operador **new**. Para optimizar este código se puede pensar en que el propio Spring sea el responsable de crear el objeto, pudiendo a futuro cambiar la política de búsqueda de canciones, sin necesidad de cambiar la clase GeneradorPlaylist.

Inyección de dependencias mediante constructor

En el siguiente ejemplo se puede ver cómo el objeto declara sus dependencias en el constructor, podemos observar que no hay código que se encargue de buscar esa dependencia o crearla, simplemente la declara, esto ayuda a tener clases Java mucho más limpias a la vez que facilita el Testing, ya que en un entorno de Tests podríamos intercambiar ese objeto por un Mock sin cambiar el código (mediante la configuración de Spring).

```
public class GeneradorPlaylist {

    private BuscadorCanciones buscadorCanciones;

    public GeneradorPlaylist(BuscadorCanciones buscadorCanciones){
        this.buscadorCanciones = buscadorCanciones;
    }

    //Resto de métodos de la clase

}
```

Para informar a Spring cual es la dependencia que tiene que inyectar en GeneradorPlaylist se puede hacer mediante dos formas: XML o anotaciones. En el siguiente ejemplo se muestra cómo se configuraría mediante XML:

```
<bean id="buscadorCanciones" class="com.example.BuscadorCanciones">
        <bean id="generadorPlaylist" class="com.example.GeneradorPlaylist">
            <constructor-arg type="com.autentia.BuscadorCanciones"
ref="buscadorCanciones">
            </constructor-arg>
```

```
        </bean>
</bean>
```

La ubicación estándar para colocar los archivos de configuración XML en un proyecto de Spring es en el directorio "resources" (src/main/resources) del proyecto. Si se utiliza un sistema de construcción como Maven o Gradle, este directorio generalmente se considera como el directorio de recursos del proyecto.

Por ejemplo, si tu proyecto sigue la estructura típica de un proyecto Maven, el archivo XML debe colocarse en el siguiente directorio:

```
src
  └── main
      └── resources
          └── application-context.xml (nombre del archivo puede variar)
```

Una vez colocado el archivo XML con la configuración de beans en el directorio de recursos, Spring podrá cargar y utilizar esta configuración cuando se inicialice la aplicación.

Es importante tener en cuenta que, a partir de versiones recientes de Spring, se ha fomentado el uso de configuraciones basadas en anotaciones (por ejemplo, usando clases de configuración con anotaciones **@Configuration**). Para que el contexto de aplicación de Spring cargue el archivo XML correctamente durante el arranque de la aplicación existen dos posibilidades: la clase **ClassPathXmlApplicationContext** o mediante configuraciones en el archivo de configuración principal de Spring (por ejemplo, "applicationContext.xml" o "spring-config.xml").

Por ejemplo, si se utiliza la clase ClassPathXmlApplicationContext, se puede hacer lo siguiente en tu código de inicio de la aplicación:

```
Copy code
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("application-context.xml");
        // Aquí obtener y utilizar los beans definidos en el archivo XML.
    }
}
```

Es importante recordar que Spring buscará automáticamente los archivos de configuración en el classPath, que incluye el directorio "resources" por defecto. Si se coloca el archivo de configuración en otra ubicación, se deberá especificar la ruta correcta en la llamada a ClassPathXmlApplicationContext.

Inyección de dependencias mediante «Setter»

Spring también permite inyectar la dependencia mediante los Setter (métodos set*()), cada forma de inyectar las dependencias tiene sus ventajas y sus desventajas.

Para indicarle a Spring que inyecte la dependencia mediante un método de establecimiento o set se utiliza la anotación **@autowired**, tal como se muestra en e@@l siguiente código:

```
public class GeneradorPlaylist {

    @autowired
    private BuscadorCanciones buscadorCanciones;

    public setBuscadorCanciones(BuscadorCanciones buscadorCanciones){
        this.buscadorCanciones = buscadorCanciones;
    }

    //Resto de métodos de la clase
}
```

Mediante la anotación @autowired se indica a Spring que se tiene que encargar de buscar un Bean que cumpla los requisitos para ser inyectado, en este caso el único requisito es que sea del tipo BuscadorCanciones, en caso de que hubiese más de un Bean que cumpliese esos requisitos se tendría que indicar cuál es el correcto.

Otro mecanismo que con frecuencia suele utilizarse para indicar a Spring que una clase tiene que se gestionada es mediante la anotación **@Component**. Es como indicarle a Spring que la clase será utilizada como un bean. Para más detalle visitar el siguiente link.

Ámbito de los Beans en Spring

Agregar

Funcionalidades de Spring

Spring ofrece una amplia gama de funcionalidades y herramientas prácticas que son de gran utilidad en diversos escenarios de desarrollo. Entre estas funcionalidades se incluyen:

- **Desarrollo ágil de aplicaciones**: Spring proporciona mecanismos que aceleran el desarrollo de aplicaciones Java, lo que permite ahorrar tiempo y esfuerzo en la implementación de características comunes.
- **Acceso a datos simplificado**: Spring facilita el acceso a bases de datos al proporcionar clases e interfaces que reducen la cantidad de código necesario para realizar operaciones de persistencia. Esto se logra utilizando tecnologías como JDBC (Java Database Connectivity) y JPA (Java Persistence API).
- **Seguridad y autenticación**: Spring ofrece herramientas y mecanismos para agregar capas de seguridad a nuestras aplicaciones, incluyendo autenticación y autorización. Esto permite proteger nuestros recursos y restringir el acceso a funciones o datos sensibles.
- **Gestión de transacciones**: Spring proporciona soporte para transacciones, lo que facilita la gestión y control de operaciones que deben ejecutarse en conjunto o de forma atómica.

- **Integración con otros frameworks y tecnologías**: Spring se integra fácilmente con otros frameworks y tecnologías populares en el ecosistema Java, como Hibernate, Thymeleaf, JUnit, entre otros. Esto permite aprovechar las ventajas de estos componentes adicionales y trabajar de manera conjunta.
- **Arquitectura basada en microservicios**: Spring se adapta bien a la arquitectura de microservicios, que consiste en desarrollar aplicaciones como un conjunto de servicios pequeños e independientes que se comunican entre sí. Esta arquitectura ofrece mayor flexibilidad, escalabilidad y facilidad de mantenimiento.

Una de las dificultades que se presentaba en el pasado al utilizar Spring era la configuración inicial del proyecto. Esto implicaba crear y configurar varios archivos, lo cual podía llevar mucho tiempo. Sin embargo, se ha desarrollado Spring Boot, una biblioteca que simplifica la configuración inicial y proporciona proyectos pre-configurados para escenarios comunes de desarrollo.

En resumen, Spring es un framework completo y versátil que ofrece una amplia gama de herramientas para el desarrollo de aplicaciones Java. Su uso está extendido en la comunidad de desarrolladores debido a su eficacia y facilidad de uso.

Spring Boot

Spring Boot es una extensión de Spring Framework, que se enfoca en simplificar la configuración y el uso de Spring en aplicaciones Web. Permite a los desarrolladores crear aplicaciones web rápidamente mediante la provisión de una serie de características pre-configuradas y pre-empaquetadas. Esto incluye características como la gestión de dependencias, la configuración automática, la integración con bases de datos, la creación de servicios RESTful y la seguridad.

Spring Boot consta de varios módulos que ofrecen diferentes funcionalidades. A continuación, se presentan algunos sus principales módulos:

- **Spring Boot Starter**: este módulo proporciona un conjunto de dependencias para diferentes tipos de aplicaciones, como aplicaciones web, aplicaciones de datos, aplicaciones de seguridad, etc. El objetivo es reducir la configuración y permitir que los desarrolladores comiencen a escribir código rápidamente.
- **Spring Boot Actuator**: este módulo proporciona características de supervisión y administración de la aplicación, como el monitoreo de la salud de la aplicación, la exposición de métricas y estadísticas, y la posibilidad de realizar operaciones de gestión y monitoreo de la aplicación.
- **Spring Boot CLI**: es una herramienta de línea de comandos que permite a los desarrolladores crear y ejecutar aplicaciones Spring Boot de manera rápida y sencilla, sin la necesidad de configurar manualmente un proyecto.
- **Spring Boot Data**: este módulo proporciona una manera fácil y rápida de conectarse y trabajar con bases de datos a través de la capa de persistencia de Spring Data.
- **Spring Boot Security**: este módulo ofrece características de seguridad y autenticación para las aplicaciones, incluyendo la autenticación basada en formularios, la autenticación basada en tokens y la integración con proveedores de autenticación externos.
- **Spring Boot Web**: este módulo proporciona características para el desarrollo de aplicaciones web, como el soporte para la creación de servicios RESTful, la integración con servidores web embebidos, la

administración de solicitudes y respuestas HTTP, y la gestión de errores.

Estos son solo algunos de los principales módulos de Spring Boot, existen otros que proporcionan diferentes funcionalidades para el desarrollo de aplicaciones.

Spring Initializer

A continuación, se mostrará el proceso para la creación de un nuevo proyecto utilizando **Spring Boot**. La forma más sencilla de hacerlo es mediante una herramienta llamada **Spring Initializer**, que se encuentra en el sitio web.

El sitio permite seleccionar varias opciones relacionadas con el proyecto para luego descargarlo pre-configurado listo para ejecutar. Las opciones que se nos presentan generalmente no son difíciles de elegir y, en muchos casos, ni siquiera es necesario cambiarlas. Las opciones son las siguientes:

1. **Tipo de proyecto**: Aquí se puede elegir entre dos gestores de dependencias/proyectos: Maven y Gradle. En este caso, se utilizará Maven, que es una opción muy popular y ampliamente utilizada en la comunidad de desarrollo de Spring.

2. **Lenguaje de programación**: Se puede elegir programar en Java, Kotlin o Groovy. Estos tres lenguajes son compatibles con la máquina virtual de Java (JVM), lo que significa que se pueden ejecutar en un entorno Java. En este caso, seleccionaremos Java, ya que es el lenguaje principal utilizado en esta asignatura.

3. **Versión de Spring Boot**: Aquí se debe seleccionar la versión de Spring Boot que deseamos utilizar. Es importante tener en cuenta que algunas versiones pueden estar en desarrollo y aún no ser completamente estables. Se recomienda utilizar la última versión estable disponible.

A continuación, se solicitan algunos datos relacionados con el proyecto:

1. **Nombre del proyecto**: Aquí se ingresa un nombre descriptivo para el proyecto. Puede ser el nombre del producto que se está desarrollando o cualquier nombre que se considere adecuado.

2. **Descripción del proyecto**: En este campo, se puede agregar una descripción breve del proyecto, que puede incluir su propósito, funcionalidad principal, etc.

3. **Grupo (Group) y Artefacto (Artifact)**: Estos campos están relacionados con la estructura de paquetes del proyecto. El Grupo representa la organización o equipo que está desarrollando el proyecto, y el Artefacto es el nombre del producto o módulo dentro del proyecto. Estos nombres suelen seguir una convención basada en nombres de dominio inversos, como com.mi-empresa.mi-producto. Esta estructura de paquetes ayuda a mantener la organización y evitar conflictos de nombres con otras librerías o proyectos.

4. **Otras opciones**: por último se pueden elegir el tipo de empaquetado: Jar/War junto con la versión del JDK. Algo no menor que es posible seleccionar antes de descargar el proyecto son las dependencias a utilizar en el proyecto, siendo las más frecuentes:

- Spring Boot Dev Tools
- Spring Web
- Lombok
- Docker Compose Support

Es importante tener en cuenta que estas opciones de configuración se utilizan para generar el archivo de configuración de Maven (pom.xml) y la estructura de paquetes del proyecto. Sin embargo, muchos de estos detalles se pueden modificar posteriormente en el archivo de configuración o mediante la organización del proyecto.

Una vez completadas todas las opciones, haciendo clic en el botón "Generate" para descargar el proyecto pre-configurado en un archivo zip. Luego, se puede descomprimir el archivo y abrirlo en nuestro entorno de desarrollo preferido (por ejemplo, IntelliJ IDEA, Visual Code o Netbeans).

En resumen, utilizar **Spring Initializer** permite crear rápidamente un proyecto Spring Boot con una configuración inicial predefinida. Esto ahorra tiempo y esfuerzo al evitar la necesidad de configurar manualmente todas las dependencias y estructuras de proyecto desde cero.

Hola Mundo con Spring boot

Una vez descargado el proyecto, es posible ejecutarlo desde el IDE sin haber escrito ningún código hasta el momento. Este proyecto ya contiene cierto código, en particular el método main, que sirve como punto de entrada de la aplicación y se encargará de iniciar el proyecto.

Sin embargo, al intentar ejecutarlo, es posible encontrarse con un error que indica que no se puede encontrar la clase principal. Esto se debe a que es necesario indicarle a la herramienta de desarrollo, en este caso, NetBeans, cuál es la clase que contiene el método main. Esto es posible configurar desde las propiedades del proyecto. Haciendo clic derecho en el nombre del proyecto y luego en la opción "Properties". Desde la sección "Run", es posible definir un campo de entrada llamado "Main Class". Allí se debe indicar la clase que contiene el método main de la aplicación.

Una vez que hemos configurada correctamente la clase principal, se puede iniciar la aplicación y ver qué ocurre. Al ejecutarla, se puede observar en la salida del programa que se inicia un servidor web de Spring. Esto indica que el proyecto Spring Boot ya incluye un servidor web que se ejecutará junto con la aplicación. Por defecto, el servidor web estará configurado en el puerto 8080 y responderá a las solicitudes HTTP.

En este punto, si se intenta navegar a "localhost:8080" en un navegador, es posible que nos muestre un mensaje de error, ya que aún no se ha programado ninguna funcionalidad. Sin embargo, este mensaje de error es emitido por la aplicación Spring Boot y no por el navegador en sí. Esto demuestra que el servidor web integrado está funcionando correctamente.

> Nota: siempre es buena práctica tener actualizado el JDK de Java a la última versión disponible para evitar problemas de versionado con la herramienta de gestión de dependencias y las configuraciones iniciales del proyecto.

Rutas y Controladores

Con la aplicación corriendo, se puede comenzar a programar los endpoints de la API. Un ***endpoint*** es un punto de conexión o acceso específico dentro de una API que permite la comunicación entre una aplicación y un servidor. Representa una URL (Uniform Resource Locator) única a la cual se pueden enviar solicitudes HTTP para interactuar con un recurso o realizar una acción específica. Cada endpoint suele estar asociado a un método HTTP, como GET, POST, PUT o DELETE, que define la acción a realizar en el recurso. Al acceder a un endpoint, se puede enviar información adicional, como parámetros de consulta, datos en el cuerpo de la solicitud o encabezados, para personalizar y controlar la operación realizada por la API.

En Spring, los endpoints se definen mediante métodos en clases conocidas como controladores (**controllers**). Para crear un endpoint, se necesita anotar un método con la anotación `@RequestMapping` o una de sus variantes, como `@GetMapping`, `@PostMapping`, etc.

Si por ejemplo se necesita crear un endpoint que responda con un mensaje de saludo "Hola, mundo". Para hacerlo, se crea una nueva clase de Java llamada `HolaMundoController` (siguiendo la convención CamelCase) y se anota con: `@RestController`. Esto indica que esta clase será un controlador de la API. Luego, se crea un método en esta clase, por ejemplo, `saludar`, que retorna un `String` con el mensaje "Hola, mundo".

Para especificar qué tipo de petición HTTP debe activar este endpoint, se utiliza una anotación como `@GetMapping` y se proporciona la URI en la cual estará disponible. Por ejemplo, podemos anotar el método con `@GetMapping("/saludo")`, lo que significa que este método responderá a las peticiones GET en la URI "`<raíz del sitio>`/saludo".

Una vez programado el endpoint, es posible iniciar la aplicación nuevamente y probarlo. Al navegar a "localhost:8080/saludo" en un navegador, se verá que se muestra el mensaje "Hola, mundo" que retorna el endpoint.

Parámetros en URL

Cuando se desarrollan APIs que exponen endpoints, es muy común que estos endpoints requieran datos adicionales, equivalentes a los parámetros de un método o una función. Para que un endpoint pueda recibir estos datos adicionales, existen varias formas de hacerlo.

En el caso de los endpoints asociados al verbo **GET**, existen dos lugares comunes donde se pueden incluir datos adicionales. Uno de ellos es como parte de la URL, es decir, como parte de la dirección de la URI. Por ejemplo, si se tiene un endpoint asociado a la URI "/hola", se podría incluir datos extra como parte de la dirección. Supongamos que se quiere que este mismo endpoint reciba el nombre del interlocutor, por ejemplo, "Jorge". Podemos incluir ese dato extra como parte de la dirección, como "/hola/Jorge".

Es importante tener en cuenta que no existirá un endpoint para cada posible nombre que pueda recibir. En su lugar, se puede indicar en el método que ésta porción variable de la dirección no es parte del identificador único del endpoint, sino más bien un parámetro variable que el método que atiende la petición desea recibir. Para lograr esto, se utiliza una notación especial en la URI, colocando la porción variable entre llaves. Por ejemplo, podríamos definir la URI como "/hola/{nombre}".

Luego, en el método que atiende este endpoint, por ejemplo, "saludar", se indica que se desea recibir un parámetro que contendrá el nombre de la persona a saludar. Esto se logra anotando el parámetro con la anotación `@PathVariable`, donde se especifica el nombre de la porción variable de la URI. Por ejemplo, se puede tener un método como `public string saludar(@PathVariable String nombre)`.

De esta manera, cuando se realiza una petición a la URI "/hola/Jorge" (o cualquier otro nombre que se especifique), todo el contenido después de "/hola/" se almacenará automáticamente en el parámetro "nombre" del método. No es necesario que el programador realice ninguna otra operación, ya que esto ocurre automáticamente durante la ejecución.

Al reiniciar el servidor y realizar peticiones a este nuevo endpoint, se puede ver cómo el método que recibe el parámetro "nombre" reacciona de manera diferente y responde concatenando el contenido variable indicado

en la URI con el saludo. Por ejemplo, si se hace una petición a "/hola/María", obtendremos como respuesta "¡Hola, María!".

De esta manera, se puede utilizar la inclusión de datos adicionales en la URI para hacer que los endpoints sean más flexibles y puedan recibir información específica en cada solicitud. Esto permite construir APIs más dinámicas y versátiles, adaptadas a las necesidades de los clientes que las consumen.

Query string

En Spring boot, la anotación **@RequestParam** se utiliza para vincular los parámetros de una solicitud HTTP (comúnmente conocido como **query string**) a los parámetros de un método controlador. Esta anotación permite acceder y utilizar los valores de los parámetros proporcionados en la URL o en el cuerpo de la solicitud.

La anotación @RequestParam se puede aplicar a los parámetros de un método controlador y permite especificar diferentes atributos para personalizar cómo se vinculan los parámetros de la solicitud. Algunos de los atributos más comunes son:

value o name: Permite especificar el nombre del parámetro en la solicitud HTTP. Por defecto, la anotación asume que el nombre del parámetro del método coincide con el nombre del parámetro en la solicitud, pero se puede utilizar esta opción para especificar un nombre diferente.

required: Un booleano que indica si el parámetro es requerido o no. Si se establece en true y no se proporciona el parámetro en la solicitud, se lanzará una excepción.

defaultValue: Permite establecer un valor predeterminado para el parámetro en caso de que no se proporcione en la solicitud.

A continuación se muestra un ejemplo de cómo se utiliza la anotación @RequestParam en un método controlador de Spring Boot:

```
@GetMapping("/saludar")
public String saludarConParam(@RequestParam("nombre") String nombre) {

    return "Hola" + nombre;
}
```

En este ejemplo, se espera que se proporcione el parámetro nombre. El valor del parámetro se vinculará automáticamente a la variable nombre. Si por ejemplo se ingresa:

> http://localhost:8080/saludar?nombre=John

la parte **?nombre=John** después del endpoint es la parte de la consulta de URL. Aquí, nombre=John indica que el valor del parámetro nombre es John.

Es importante tener en cuenta que estos parámetros enviados a través de la query string pueden ser opcionales. Si no se envía un parámetro determinado, llegará como una cadena vacía o nula al método correspondiente. Por lo tanto, es responsabilidad del método que maneje estos parámetros utilizarlos de manera adecuada y considerar su posible ausencia.

La elección entre el uso de **path variables** y **query string** depende del caso y la lógica del sistema. En general, se recomienda utilizar path variables cuando el dato es obligatorio y representa un recurso existente en el sistema, mientras que se utiliza query string cuando los parámetros son opcionales o su orden puede variar. Sin embargo, ambas formas permiten capturar y utilizar los parámetros de manera similar en el método que los recibe.

El código completo del proyecto se encuentra en Spring boot - Primer API

Ejecutando endpoints

Cuando se desarrolla una API REST, es común que no todos los endpoints estén programados para ser accedidos mediante el verbo GET. Por lo tanto, no es suficiente ejecutar las peticiones desde un navegador web, ya que los navegadores normalmente solo permiten enviar solicitudes GET. Para probar y simular otros verbos como POST, PUT, DELETE, entre otros, se utilizan herramientas como **Bruno** o **Postman**.

Postman

Postman es una herramienta muy utilizada que permite enviar peticiones HTTP de cualquier verbo y recibir las respuestas correspondientes. Con Postman, se puede probar cualquier verbo HTTP y realizar diferentes tipos de solicitudes a un servidor web. Es especialmente útil para simular una aplicación cliente y probar los diferentes endpoints de una API antes de implementar el frontend con tecnologías como React o Angular.

Para utilizar Postman, se crea una colección, que es un grupo de peticiones relacionadas a una API en particular. Dentro de la colección, se agregan las peticiones correspondientes a cada endpoint que se desea probar.

Cada petición en Postman se configura indicando el verbo HTTP, la URL del endpoint y otros parámetros necesarios. Al enviar la petición, Postman muestra la respuesta recibida, incluyendo el código de respuesta, las cabeceras y otra información técnica relevante.

En el caso de endpoints que utilizan el verbo POST, es común que se envíen datos en el cuerpo de la petición, en formato JSON, por ejemplo. Para recibir estos datos en el backend, se utiliza la anotación `@RequestBody` en el parámetro del método correspondiente. Esto permite que los datos enviados en el cuerpo de la petición se mapeen automáticamente a un objeto Java en el backend, facilitando su procesamiento.

En resumen, Postman es una herramienta muy útil para probar y simular peticiones HTTP con diferentes verbos en una API REST. Permite configurar y enviar peticiones de forma sencilla, mostrando las respuestas recibidas. Es especialmente útil cuando se necesitan enviar datos en el cuerpo de la petición, como en el caso de los endpoints que utilizan el verbo POST.

Bruno

bruno

Bruno es un asistente de desarrollo que facilita la creación y gestión de aplicaciones backend. Con Bruno, los desarrolladores pueden configurar rápidamente su entorno de trabajo y manejar diferentes componentes de sus aplicaciones de forma eficiente. Es especialmente útil para optimizar el proceso de desarrollo y ayudar a los equipos a mantenerse organizados durante todo el ciclo de vida del software.

Para utilizar Bruno, los desarrolladores configuran un proyecto que agrupa todas las dependencias y configuraciones necesarias para su aplicación. Dentro de este proyecto, se pueden definir los servicios, controladores y modelos que componen la lógica del backend.

Cada componente en Bruno se configura especificando sus propiedades y relaciones, lo que permite una integración fluida entre los diferentes elementos de la aplicación. Al ejecutar el proyecto, Bruno proporciona retroalimentación instantánea, incluyendo mensajes de error, advertencias y otras informaciones relevantes que ayudan a los desarrolladores a identificar y resolver problemas rápidamente.

En el caso de los servicios que interactúan con bases de datos, es común que se utilicen patrones de diseño como repositorios para manejar la persistencia de datos. Esto permite que los datos se gestionen de forma eficiente y se integren fácilmente con los objetos Java en la aplicación, simplificando así el proceso de desarrollo.

En resumen, Bruno es una herramienta valiosa para desarrolladores que buscan mejorar su flujo de trabajo al crear aplicaciones backend. Permite configurar y gestionar proyectos de manera sencilla, ofreciendo una interfaz amigable y funcionalidades que optimizan el proceso de desarrollo y despliegue.

Swagger

Swagger es una especificación de código abierto y una suite de herramientas que se utilizan para describir, construir y documentar servicios web basados en HTTP. La especificación de Swagger, también conocida como **OpenAPI Specification**, utiliza un formato JSON o YAML para describir la API. Esta descripción incluye información como la versión de la API, los endpoints disponibles, los métodos HTTP permitidos (como GET, POST, PUT, DELETE), los parámetros necesarios y opcionales, los códigos de respuesta esperados y los esquemas de datos utilizados.

Una de las principales ventajas de Swagger es que permite generar automáticamente una documentación interactiva de la API a partir de la especificación. Esta documentación incluye detalles sobre cómo llamar a cada endpoint, qué parámetros se deben proporcionar y qué respuestas se pueden esperar. Además, Swagger también puede generar código cliente y servidor en varios lenguajes de programación a partir de la especificación, lo que facilita la implementación y el consumo de la API.

Otra característica importante de Swagger es su capacidad para realizar pruebas y validaciones de la API. Puedes enviar solicitudes de prueba a través de la interfaz de Swagger y verificar si las respuestas son las esperadas. Esto ayuda a detectar posibles problemas antes de implementar la API en producción.

Algunas de las herramientas más populares asociadas a Swagger incluyen:

- **Swagger UI**: Es una herramienta que genera una documentación interactiva y visualmente atractiva de una API basada en la especificación de Swagger. Permite explorar y probar los endpoints de la API directamente desde el navegador.
- **Swagger Editor**: Es una herramienta basada en navegador que proporciona un entorno de edición en tiempo real para la especificación de Swagger. Permite escribir, validar y previsualizar la documentación de la API en formato JSON o YAML.
- **Swagger Codegen**: Es una utilidad de línea de comandos que genera automáticamente el código de una API basándose en la especificación de Swagger. Puede generar código cliente y servidor en varios lenguajes de programación, lo que agiliza el proceso de implementación.

Estas herramientas, junto con otras disponibles en el ecosistema de Swagger, brindan una solución integral para la descripción, implementación y documentación de APIs. Swagger se ha convertido en un estándar ampliamente adoptado en la industria para la construcción de APIs RESTful.

Swagger Editor

Swagger Editor permite diseñar, describir y documentar una API en línea. Es compatible con múltiples especificaciones de API y formatos de serialización. El editor de Swagger ofrece una manera fácil de comenzar con la especificación OpenAPI (anteriormente conocida como Swagger). Solo es necesario conocer cómo la OpenAPI permite describir las funcionalidades a desarrollar y escribirlas en un formato YML o JSON. La herramienta automáticamente genera visualmente la especificación de la API permitiendo además probarla desde el mismo sitio.

Pasos para documentar con **Swagger Editor**

1. Acceder a https://editor.swagger.io/ en el navegador web.
2. Es posible comenzar con una especificación en blanco o cargar una especificación existente en formato JSON o YAML. Se puede cargar un archivo local o proporcionar una URL que apunte a la ubicación del archivo de especificación.
3. Una vez que se haya cargado o creado la especificación, el editor Swagger se abrirá en la interfaz principal.
4. En el editor Swagger, se puede comenzar a definir la API REST. A continuación, se muestra una descripción general de las acciones comunes que puedes realizar:
    - Definir información general de la API: En la sección "info" se puede proporcionar detalles como el título, la versión, la descripción y la información de contacto de la API.
    - Agregar y describir endpoints: En la sección "paths", se pueden agregar los endpoints y describir cada uno de ellos. Se puede especificar el método HTTP (GET, POST, PUT, DELETE, etc.), los parámetros, las respuestas esperadas y cualquier otra información relevante.
    - Definir modelos de datos: En la sección "components/schemas", se pueden definir los modelos de datos utilizados en la API. Se puede especificar las propiedades, los tipos de datos, las restricciones y las relaciones entre los modelos.
    - Personalizar las configuraciones de Swagger: Se puede ajustar las configuraciones generales de Swagger en la sección "components".
5. A medida que se haya definiendo la API en el editor Swagger, se podrá ver una representación visual y estructurada de la especificación en la parte derecha de la pantalla. Esta vista previa permite verificar cómo se está construyendo la API en tiempo real.
6. Mientras se trabaja en la especificación, el editor Swagger ofrece funciones útiles como la validación en tiempo real para asegurarse de que sigas el formato correcto y no haya errores.
7. Una vez completada la documentación, se pueden utilizar las opciones de exportación para guardar la especificación en formato JSON o YAML en la computadora local. Además de esta opción, el editor Swagger también ofrece otras funcionalidades como la **generación de código**, donde se puede elegir generar código cliente o servidor en diferentes lenguajes de programación a partir de la especificación definida.

Para mayor detalle se este video con un ejemplo de uso de la herramienta.

Otras herramientas

Otras opciones para documentar, describir y probar APIs se enumeran a continuación:

- https://openapi-generator.tech/
- https://stoplight.io/
- Crear una definición a partir de POSTMAN
- Integrar la documentación de la API con el entorno de desarrollo de Spring

Servicios y Repositorios

En el desarrollo de sistemas con cierta complejidad, es recomendable organizar las clases siguiendo algún criterio para mantener una estructura clara y modular. Una arquitectura comúnmente utilizada es la arquitectura en capas.

En esta arquitectura, se divide la responsabilidad en diferentes capas y se organizan las clases en paquetes según su funcionalidad. A continuación, se describen las capas y su propósito principal:

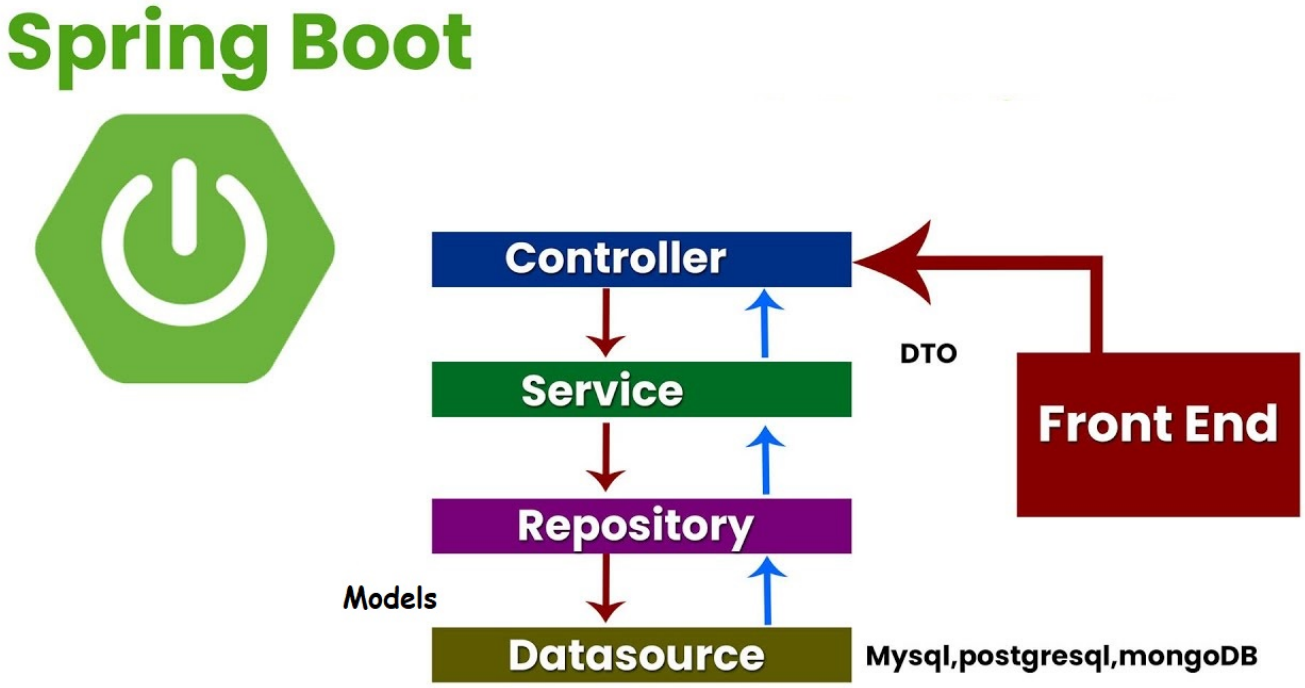
- **Capa de Controllers**: Esta capa se encarga de manejar las solicitudes de la API y de interactuar con los clientes. Aquí se definen los controladores (RestController) que reciben las solicitudes HTTP y las dirigen a las capas correspondientes.

- **Capa de Models**: En esta capa se encuentran las clases que representan los objetos de entidad del sistema. Estas clases contienen propiedades y métodos relacionados con los datos que serán manipulados. Por lo general, estas clases se corresponden con las tablas de la base de datos y se utilizan para almacenar y recuperar datos.

- **Capa de Repositories**: En esta capa se definen los repositorios, que son responsables de interactuar con la base de datos. Aquí se implementan operaciones como guardar, actualizar, buscar y eliminar registros en la base de datos. Los repositorios se comunican con la capa de servicios para proporcionar o recibir datos.

- **Capa de Services**: En esta capa se encuentra la lógica de negocio del sistema. Aquí se definen los servicios que encapsulan la lógica y operaciones complejas. Los servicios utilizan los repositorios para acceder a los datos y realizar las operaciones requeridas. También se pueden aplicar reglas de validación, cálculos y otras operaciones específicas del dominio.

La comunicación entre las capas se realiza de la siguiente manera: los controladores (Controllers) reciben las solicitudes de los clientes y se comunican con los servicios (Services) correspondientes. Los servicios utilizan los repositorios (Repositories) para acceder a los datos y realizar operaciones en la base de datos, tal como se muestra en la siguiente imagen:



Esta estructura en capas permite una separación clara de responsabilidades y facilita el mantenimiento y la evolución del sistema. Además, permite reemplazar o modificar una capa sin afectar el resto del sistema, lo que brinda flexibilidad y escalabilidad.

Cabe mencionar que esta es una arquitectura general y cada desarrollo puede adaptarla según sus necesidades y requisitos específicos. En la imagen la comunicación hacia el frontend es mediante objetos planos de Java llamados **DTOs** que sirven para recibir y transportar datos desde la API. También existen otras arquitecturas y patrones de diseño que pueden ser utilizados en diferentes contextos.

Caso de aplicación: Personas

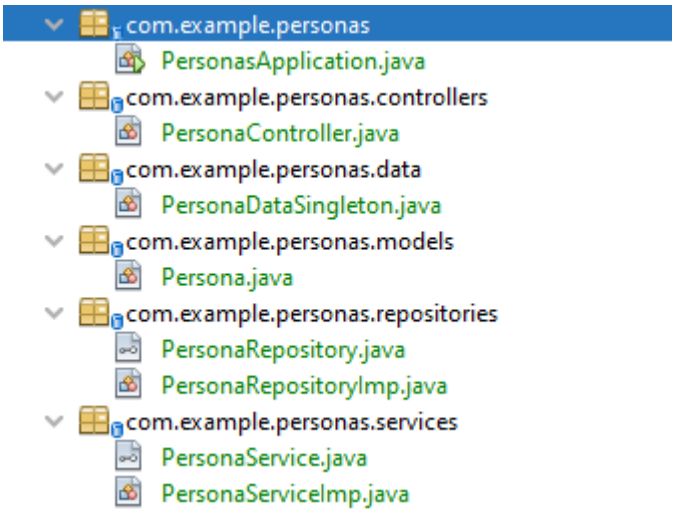
A continuación se presenta el desarrollo de una API REST con Spring boot que permite gestionar el listado de personas registradas a un evento. Para ello será necesario definir:

- un esquema: **Persona** con los siguientes datos: idPersona (Integer), nombre (String), fechaNacimiento (Date) y esExtranjero(Boolean).
- los siguientes endpoints:
    - POST /personas:
    - GET /personas:
    - GET /personas/{id}:

La descripción de la API mediante Swagger se muestra en el siguiente link

En este caso los datos de personas persistirán en memoria utilizando las herramientas y técnicas abordadas en la semana 3.

La estructura del proyecto se muestra en la siguiente imagen:



Cabe mencionar que las personas serán persistidas en una estructura `HashMap<int, Persona>` dentro de la clase `PersonaDataSingleton`. En esta clase se implementa el patrón Singleton para garantizar solo una única instancia de almacenamiento de los datos que será compartida por los componentes de la aplicación.

El código completo del proyecto se encuentra en Spring boot - Personas API.