AMS Bootcamp Fullstack Developer.

**REST y SpringREST**



Índice

[1. Personal Notebook 3](#_Toc157634064)

[1.1 Datos alumno 3](#_Toc157634065)

[1.2 Personal Notebook 3](#_Toc157634066)

[2. REST, Spring REST, Spring Boot 4](#_Toc157634067)

[2.1 Objetivos de aprendizaje 4](#_Toc157634068)

[2.2 ¿Qué es MVC? ¿Cómo funciona (teórico)? 5](#_Toc157634069)

[2.2.1 Objetivos 5](#_Toc157634070)

[2.2.2 Recursos formativos 5](#_Toc157634071)

[2.2.3 Posibles preguntas 5](#_Toc157634072)

[2.2.4 Tareas 5](#_Toc157634073)

[2.2.5 Dudas 5](#_Toc157634074)

[2.3 REST 6](#_Toc157634075)

[2.3.1 Objetivos 6](#_Toc157634076)

[2.3.2 Recursos formativos 6](#_Toc157634077)

[2.3.3 Posibles preguntas 6](#_Toc157634078)

[2.3.4 Tareas 7](#_Toc157634079)

[2.3.5 Dudas 7](#_Toc157634080)

[2.4 Spring Boot 8](#_Toc157634081)

[2.4.1 Objetivos 8](#_Toc157634082)

[2.4.2 Recursos formativos 8](#_Toc157634083)

[2.4.3 Posibles preguntas 8](#_Toc157634084)

[2.4.4 Tareas 8](#_Toc157634085)

[2.4.5 Dudas 8](#_Toc157634086)

[2.5 Spring REST 9](#_Toc157634087)

[2.5.1 Objetivos 9](#_Toc157634088)

[2.5.2 Recursos formativos 9](#_Toc157634089)

[2.5.3 Posibles preguntas 9](#_Toc157634090)

[2.5.4 Tareas 9](#_Toc157634091)

[2.5.5 Dudas 9](#_Toc157634092)

# Personal Notebook

## Datos alumno

* DAS: A926220
* Nombre Apellidos: Sandra Moriana Herrera
* Email: Sandra.moriana@eviden.com
* URL de GitHub corporativo

## Personal Notebook

Este es tu cuaderno de trabajo. En él tendrás que registrar tu aprendizaje, tus avances y tus dudas. ¡Vas a generar tu propio material!

Puede ser una gran herramienta si la utilizas correctamente.

El cuaderno estará compartido con los tutores y el resto de colaboradores para ver tu progreso y ayudarte.

Al final del Bootcamp tendrás que entregarlo como evidencia de tu aprendizaje.



Instrucciones de uso:

El fichero está ubicado en tu carpeta del espacio que tenemos en Teams.

Siempre deberá ser editado en modo online y/o con guardado automático en la nube.

Responde a las preguntas con la información que necesites para dar una respuesta cualificada al formador o a tus compañeros. No es necesario copiar texto solo por “rellenar”. Cita las Fuentes en caso de utilizar otros recursos.

# REST, Spring REST, Spring Boot

## Objetivos de aprendizaje

* Objetivo:
  + **Comprender el funcionamiento de las aplicaciones basadas en REST, las diferencias con arquitecturas web clásicas, los principales métodos y herramientas para trabajar con servicios REST en Spring.**
  + **Conocer las ventajas de usar Spring Boot y su configuración inicial**
* Conceptos principales a adquirir:
  + Arquitectura MCV y REST
  + Spring Boot
  + Spring REST

Lecciones:

1. MVC (teoría)
2. REST (teoría)
3. Sprint Boot
4. Spring REST

## ¿Qué es MVC? ¿Cómo funciona (teórico)?

### Objetivos

1. **Comprender el flujo de las peticiones, desde que sale de un navegador hasta la bbdd.**
2. **Explicar el modelo de capas**

### Recursos formativos

<https://www.youtube.com/watch?v=T_DOdKInZNk&list=PLU8oAlHdN5Blq85GIxtKjIXdfHPksV_Hm&index=26> (26-30)

<https://www.youtube.com/watch?v=z6WppAQ3LUg>

VIP: recuerda que tienes que responder a estas preguntas de forma que te ayude a explicárselas al formador o a los compañeros!!

<https://www.youtube.com/watch?v=-0b6VtyU5S0>

### Posibles preguntas

**¿Cómo funciona una petición http?**

Una petición HTTP (Hypertext Transfer Protocol) es un mensaje o petición que se envía desde un cliente hacia un servidor para solicitar datos, recursos o documentos. Por ejemplo, al indicar un URL en un navegador web para acceder al contenido de una determinada web se está realizando una petición HTTP a un determinado servidor.

En el contexto de una aplicación web desarrollada con el patrón MVC, el proceso sería el siguiente:

1. El cliente inicia la petición a través de una vista (por ejemplo, HTML en un navegador web), enviando la solicitud a través de botones o los mecanismos establecidos en la vista.
2. La petición se envía al servidor. Cuando la recibe, la dirige al controlador correspondiente de la aplicación web, en función de la URL solicitada y los mecanismos de enrutamiento configurados en la aplicación.
3. El controlador recibe la solicitud, la analiza y la procesa, realizando las operaciones que se requieran para consultar o actualizar el modelo donde se almacenan los datos (generalmente, una base de datos).
4. El modelo se actualiza y el controlador genera una respuesta HTTP que envía de vuelta a la vista del usuario, recibiendo este la información requerida en su petición.

**¿Qué es MVC?**

Se trata del Modelo-Vista-Controlador. Es un patrón de diseño muy generalizado en el desarrollo de aplicaciones web. Este patrón se basa en la división de la aplicación en tres componentes principales:

* **Controlador:** Recoge las peticiones del usuario, las procesa y realiza las consultas al modelo. Cuando el modelo le devuelve la información requerida, el controlador envía una respuesta hacia una vista para que finalmente el usuario reciba la información. En definitiva, es un intermediario entre el modelo y la vista, manejando las solicitudes y las respuestas.
* **Modelo:** Almacena los datos, gestiona el acceso y las operaciones que se pueden realizar con los mismos. En la mayoría de los casos se trata de una Base de Datos. El modelo se encarga de proporcionar al controlador la información que éste le requiere.
* **Vista:** Es la interfaz de usuario que muestra el resultado de la petición que ha realizado el usuario. Es decir, es donde el controlador envía la respuesta a su petición. La vista simplemente muestra la información, sin participar en ninguna lógica ni manipulación de datos. En una aplicación web suele ser una página HTML.

**¿Cuáles son los componentes por los que pasa la petición? ¿Para qué sirven las 3 capas? ¿Ventajas?**

Como se ha explicado en la respuesta anterior, los componentes son:

* + **Controlador**: Es el primer componente que interactúa con la petición HTTP, procesa la solicitud del cliente. Decide qué acción tomar en función de la solicitud recibida y puede interactuar con el modelo para recuperar, actualizar o eliminar datos. Actúa como un intermediario entre la petición HTTP y el resto de la aplicación.
  + **Modelo**: Interactúa con el controlador. Puede realizar operaciones como recuperar datos de una base de datos, validar información, realizar cálculos y actualizar el estado de la aplicación.
  + **Vista**: Presenta la información al usuario en un formato comprensible. No realiza ninguna lógica ni manipula datos, simplemente muestra la información enviada por el controlador. Puede ser una página HTML generada dinámicamente, una interfaz de usuario en una aplicación de escritorio o cualquier otra representación visual de los datos.

Dividir o modularizar las aplicaciones de esta forma, aporta las siguientes ventajas:

* Favorece el desarrollo integral de la aplicación, puesto que cada componente puede desarrollarse por separado.
* Favorece el mantenimiento, por similar razón: Al estar dividida en componentes, en partes más pequeñas y simples, es más fácil realizar las operaciones de mantenimiento.
* Favorece la depuración: dado que los componentes son independientes, es más fácil identificar y corregir los errores, así como la introducción de correcciones sin afectar al resto de la aplicación.
  + - Favorece la escalabilidad: Favorece la ampliación de la aplicación ya que de igual forma se puede trabajar sore los componentes de forma paralela.

**¿Hay otros modelos y/o arquitecturas? (MVVM, MVW, MVP...) ¿Cuándo usar uno u otro? ¿Cuales?**

Sí, entre otras:

* **MVVM (Modelo-Vista-Modelo de Vista)**: En este caso, el modelo y la vista son componentes similares a los del patrón MVC, siendo el modelo de vista el intermediario entre ambos, manejando la lógica de presentación. Es útil cuando se quiere establecer una clara separación entre el modelo de datos (lógica de negocio), la interfaz de usuario y la lógica de presentación.
* **MVP (Modelo-Vista-Presentador)**: En este caso, se mantienen el modelo y la vista, existiendo un presentador que se encarga de la lógica de presentación. Puede ser útil, por ejemplo, en una aplicación de correo electrónico en la que el modelo representa los correos y la lógica asociada a los mismos, la vista muestra la interfaz de la bandeja de entrada y el presentador maneja la interacción del usuario (como la selección de correos o el envío de nuevos mensajes).
* **MVW (Modelo-Vista-Whatever):** En este caso, se mantienen el modelo y la vista, siendo el tercero cualquier otro componente que facilite la comunicación entre ambos (ya sea un controlador, un modelo de vista o un presentador). Útil cuando se desarrolla una aplicación que deba ser flexible a la hora de gestionar cómo se muestran los datos al usuario.

En resumen, los modelos MVVM, MVW y MVP comparten el mismo objetivo general de separar las responsabilidades de la aplicación en capas distintas, pero se distinguen en la forma en que se estructuran y comunican estos componentes. MVVM se centra en la vinculación de datos bidireccional entre la Vista y el Modelo de Vista, MVW permite flexibilidad en la elección de componentes adicionales para la comunicación entre la Vista y el Modelo, mientras que MVP se enfoca en la coordinación de las interacciones del usuario a través del Presentador. La elección entre estos modelos dependerá de las necesidades y requisitos específicos del proyecto.

**¿Qué es el modelo por capas?**

El modelo por capas divide la aplicación en componentes independientes, lo que facilita el desarrollo, el mantenimiento y la escalabilidad del software. Además, permite que cada capa sea desarrollada, probada y mantenida de forma independiente, lo que mejora la modularidad y la reutilización del código.

Todos los modelos explicados anteriormente son modelos por capas, ya que las aplicaciones se dividen en componentes que representan una responsabilidad o un nivel lógico específico. Independientemente de como llamemos al componente (vista, modelo, controlador…) las capas que representan son las siguientes:

* **Capa de Presentación (o Interfaz de Usuario)**: Se encarga de interactuar con el usuario final. Su función principal es mostrar información al usuario y manejar sus interacciones. En el MVC, esta capa se materializa en la Vista.
* **Capa de Lógica de Aplicación (o Capa de Negocio)**: Contiene la lógica de negocio o la lógica de aplicación de la aplicación. Aquí se implementan las, los algoritmos y las operaciones que realizan la funcionalidad principal de la aplicación. En el modelo MVC, esta capa se materializa en el Controlador.
* **Capa de Acceso a Datos**: Esta capa se encarga de interactuar con el almacenamiento de datos, como una base de datos, un sistema de archivos o servicios web externos. Aquí se realizan operaciones de lectura y escritura de datos. Esta capa abstracta la forma en que se accede a los datos, lo que facilita el cambio de la fuente de datos sin afectar a otras capas de la aplicación.

### Tareas

NA

### Dudas

¿?

¿?

VIP: registra las dudas que tengas para la sesión con el formador!!

¿?

¿?

¿?

¿?

## REST

### Objetivos

1. **Comprender el funcionamiento de las aplicaciones basadas en REST,**
2. **Comprender las diferencias con arquitecturas web clásicas,**
3. **Conocer los principales métodos y herramientas para trabajar con servicios REST**

### Recursos formativos

* Píldoras video
  + Intro a REST: <https://www.youtube.com/watch?v=aoOKxUOhg18> (8 min)
  + 6 consejos para que diseñes bien tu API REST <https://www.youtube.com/watch?v=bUmy7Nvsh4s&t=413s> (16 min)
  + Aprende A Documentar Tu API Con Swagger
* Long videos
  + Diseño de APIs REST <https://www.youtube.com/watch?v=3EceTpFV4Ys> (70 min)
  + API Rest. Introducción. Buenas prácticas <https://www.youtube.com/watch?v=t_L88F3lTdQ> (42 min)
* Lecturas
  + <https://www.arquitecturajava.com/servicios-rest/>
  + <https://bravedeveloper.com/2022/06/06/estilos-de-arquitectura-para-construir-apis/>
  + <https://desarrolloweb.com/articulos/que-es-rest-caracteristicas-sistemas.html>
  + <https://www.baeldung.com/spring-rest-openapi-documentation>

### Posibles preguntas

* **¿Qué significa REST? ¿Para qué se usa? ¿En qué tipo de aplicaciones?**

REST son las siglas de Representational State Transfer. Es un estilo de arquitectura de software que permite realizar una comunicación cliente-servidor; se trata de una interfaz que permite la comunicación entre un servicio web y otros sistemas o aplicaciones diferentes basándose en el protocolo HTTP. Los mensajes que se envían o reciben pueden estar en formatos como XML o JSON. JSON es comúnmente utilizado por defecto.

Se basa en los siguientes principios:

* **Interfaz uniforme**: Rest está basado en recursos, que pueden ser objetos, datos o cualquier otra cosa que tenga una entidad o identificación. Se opera sobre recursos y no sobre servicios o métodos.
* Cada recurso se maneja mediante su **URL**, a la que se puede aplicar las operaciones CRUD (Create, Read, Update, Delete).
* Emplea los métodos clásicos del **protocolo HTTP** (GET, POST, PUT, PATCH, DELETE…).
* **Stateless**: Rest no mantiene ningún tipo de estado entre peticiones distintas de forma que cada una de ellas es totalmente independiente de la siguiente. Entre dos llamadas cualesquiera, el servicio pierde todos los datos, no los “recuerda” en la siguiente petición..
* **Operaciones específicas**: Cada operación tiene un único propósito sin existir multifuncionalidad dentro de la misma operación.
* **Cliente-Servidor**: El servidor es quien realiza el procesamiento de la API REST y expone los recursos a los Clientes. Cliente y Servidor son totalmente independientes y la comunicación entre ellos se realiza a través de la API.

Se utiliza para desarrollar gran variedad de aplicaciones y servicios web, como pueden ser, entre otras:

* Aplicaciones de redes sociales que permiten a los usuarios acceder y manipular su información de perfil, publicaciones, amigos, etc.
* Aplicaciones móviles que se comunican con servidores para obtener y enviar datos.
* Aplicaciones web de comercio electrónico que permiten a los usuarios buscar productos, realizar pedidos y realizar pagos en línea.
* Servicios de backend para aplicaciones de IoT (Internet de las cosas) que recopilan y procesan datos de dispositivos conectados.
* **Explicar la arquitectura de una aplicación Front – Back**

Es lo que también se conoce como arquitectura Cliente-Servidor. Dicha arquitectura permite separar, en responsabilidad y funcionalidad, la parte del frontend (asociada al Cliente) y la parte del backend (asociada al servidor). Cada parte se caracteriza por lo siguiente:

* **Frontend (Cliente)**
* Es la presentación de los datos, la parte de una aplicación con la que interactúan los usuarios finales. Se compone de todos los elementos que componen la interfaz de usuario y la lógica del lado del cliente.
* Las tecnologías comúnmente utilizadas para desarrollar la parte frontend son: HTML, CSS y JavaScript.
* **Backend (Servidor)**
* Es la parte de una aplicación que contiene la fuente de datos y maneja la lógica de la aplicación (o de negocios). Se ejecuta en el servidor, incluyendo el servidor web, las bases de datos y todas las aplicaciones o servicios web que proporcionan funcionalidades al frontend.
* También se encara de toda la parte de seguridad de la aplicación (autenticación, autorización, protección contra ataques…).
* Las tecnologías comúnmente utilizadas para desarrollar la parte backend son lenguajes de programación como Java, Pythom, Ruby…

El frontend puede comunicarse con el backend a través de solicitudes HTTP para obtener o enviar datos o realizar otras operaciones. El backend se encarga de procesar dichas solicitudes HTTP y devolver una respuesta al frontend, en formato JSON o XML generalmente. Esta comunicación entre ambas partes se realiza a través de una API, como puede ser una API REST, de forma que ambas partes (frontedn y backend) pueden interactuar entre sí con independencia de la tecnología o lenguaje con la que han sido desarrolladas y sin necesidad de conocer cómo funciona cada cual.

* **¿Cómo funcionan las peticiones? ¿Qué formatos de datos puede manejar?**

Las peticiones, en este contexto, se realizan mediante el protocolo HTTP, de forma que un Cliente envía dicha petición al Servidor sobre un recurso determinado. La solicitud incluye un método HTTP (GET, POST, PUT, DELETE…) que especifica la acción que se pretende realizar sobre el recurso en cuestión, identificado mediante su URL. Una vez el Servidor recibe la solicitud, la procesa y ejecuta la acción correspondiente, devolviendo una respuesta al Cliente.

Tanto las peticiones del lado del Cliente como las respuestas del lado del Servidor pueden manejar gran variedad de formatos, pero en este contexto, principalmente deben manejar los formatos:

* **JSON**: Lenguaje de marcas o formato para el intercambio de datos de forma estructurada, ligera y fácilmente interpretable.
* **XML**: Otro lenguaje de marcas o formato, similar a JSON, difiriendo en su sintaxis.
* **¿Qué es API Rest? ¿Cuál es la diferencia con SOAP?**

**API Rest** es una API que implementa la arquitectura REST, explicada en la primera pregunta. Una **API** es una Interfaz de Programación de Aplicaciones: un conjunto de reglas, funcionalidades o recursos que permite a diferentes aplicaciones externas interactuar con una determinada aplicación o servicio web. Define como se pueden solicitar y compartir los datos y funcionalidades entre diferentes sistemas de software.

Por tanto, API REST es la interfaz a través de la cual se comunica la parte del frontend (Cliente) con la parte del backend (Servidor), independientemente del lenguaje o tecnología con que esté desarrollada cada una de ellas. Se basa en los principios REST (en resumen):

* Recursos identificables: Cada recurso se identifica mediante una URI única.
* Operaciones uniformes: Las operaciones CRUD se realizan mediante métodos HTTP estándar: GET, POST, PUT, DELETE…
* Sin estado: Las solicitudes al servidor no mantienen estado entre ellas, lo que significa que cada solicitud contiene toda la información necesaria para ser procesada.

**SOAP**, por su parte, son las siglas de Simple Object Acces Protocol. Se trata de otro estilo de arquitectura de software, alternativo a REST, que igualmente define cómo dos sistemas pueden comunicarse entre sí. Se trata de un estilo de mayor complejidad y rigidez que REST.

Las diferencias clave entre REST y SOAP son:

* REST utiliza HTTP como protocolo de transporte, mientras que SOAP puede utilizar varios protocolos.
* REST suele usar JSON o XML para intercambiar datos, mientras que SOAP usa XML exclusivamente.
* REST tiende a ser más simple y flexible que SOAP debido a su uso de HTTP y formatos de datos livianos como JSON.
* Las API REST suelen ser más fáciles de entender y utilizar debido a su interfaz uniforme y a la naturaleza autodescriptiva de HTTP. SOAP, por otro lado, requiere un mayor nivel de conocimiento técnico y una documentación más detallada.
* **¿Qué es el concepto Stateless?**

Stateless significa “sin estado”. En el contexto del desarrollo de aplicaciones, este concepto se refiere a que no se mantiene ningún tipo de estado entre peticiones distintas de forma que cada una de ellas es totalmente independiente de la siguiente. Ello implica que el servidor no almacena la información de una solicitud o sobre el estado de una sesión de cara a otras solicitudes. Es la propia solicitud la que debe proporcionar el estado o la información completa requerida para la acción que se pretende. De esta forma, el servidor puede manejar cualquier solicitud sin necesidad de coordinarse o sincronizarse con otras instancias o sesiones.

* **¿Cómo puedo trabajar con WebService con Java?**

Se puede hacer utilizando la API de Java que implementa RESTful (JAX-RS, Java API for RESTful Web Services). De esta forma, en Java, mediane las anotaciones @Path pueden definirse métodos para manejar las peticiones HTTP. También puede desplegarse la aplicación en un contenedor web, como el servidor Apache Tomcat y usar la clase HttpClient para realizar peticiones HTTP al servicio REST.

* **¿Qué es el principio HATEOAS?**

HATEOAS son las siglas de Hypermedia As The Engine Of Application State. Este principio establece que los recursos estén interconectados entre sí. Ante una petición, se devuelven enlaces (hipervínculos) a otro tipo de recursos con las que está relacionado el recurso de la petición, de forma que, para el cliente o usuario, la aplicación es auto descubrible, ya que puede ir explorándola y descubriéndola sin necesidad de realizar peticiones específicas para ello.

El principio HATEOAS establece que una API, a partir de un recurso raiz, pueda ser recorrida sin necesidad de ninguna documentación.

Ventajas:

* Aumenta la interoperabilidad, facilitando la implementación de la API con clientes de diferente naturaleza.
* En teoría, se puede ahorrar documentación, al ser auto descubrible no es necesario extender tanto el “manual”.
* Ayuda a reducir la cantidad de datos que devuelve el servidor, ya que una petición que requiere gran volumen de datos puede devolverse los principales y enlaces al resto de datos.
* **En el contexto de Rest: ¿Qué es un Resource?**

Resource es un recurso, es cualquier objeto u entidad que pueda identificarse y que pueda manipularse para proporcionar datos o información. Por ejemplo, en una aplicación, un recurso puede ser un artículo en un blog de noticias, un usuario registrado, un producto de una tienda online, una operación bancaria...

En el contexto de REST, puede operarse sobre un recurso a través de su URI y de los verbos HTTP estándar (GET, POST, PUT y DELETE).

* **¿Cuáles son los niveles de madurez de REST?**

Establecen unos criterios o baremos que evaluar en qué medida una API cumple con los principios REST y sus reglas de “buen diseño”, de forma que el nivel 0 sería el cumplimiento más básico (poca madurez de laAPI) y el nivel 3 sería un cumplimiento ideal (API madura).

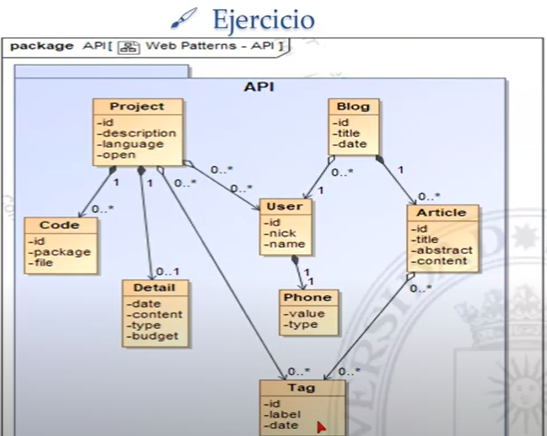
* **Nivel 0**: HTTP como transporte, es decir, se utiliza el protocolo HTTP como canal para enviar peticiones y devolver respuestas.
* **Nivel 1**: El diseño se orienta a recursos y no a acciones o métodos.
* **Nivel 2**: Uso correcto de verbos HTTP, eliminando otras sintaxis de acción incluidas en la URL.
* **Nivel 3**: Uso de controles hipermedia, de acuerdo al principio HATEOAS, generando una aplicación autodescubrible.

Para un correcto diseño de una API REST, debería alcanzarse al menos el nivel 2

### Tareas

* Plantear distintos Endpoints para el siguiente modelo de datos
* Documentarlo con OpenAPI (Swagger)

Nota: No hay que diseñar la BBDD, son los endpoinds para esos modelos de datos.



**Enlace a repositorio:**

https://github.com/Sanmohe/EjercicioEndpointsSwagger.git

### Dudas

¿?

¿?

VIP: registra las dudas que tengas para la sesión con el formador!!

¿?

¿?

¿?

## Spring Boot

### Objetivos

* **Conocer que es Spring boot, que ventajas aporta y sus configuraciones básicas**

### Recursos formativos

* Píldoras videos
  + <https://www.youtube.com/watch?v=rnyObUvNEdU> (7 min)
* Lecturas:
  + https://spring.io/guides/gs/spring-boot/
  + <https://www.arquitecturajava.com/que-es-spring-boot/>
  + <https://www.arquitecturajava.com/spring-boot-starter-un-concepto-fundamental/>
  + <https://www.arquitecturajava.com/spring-boot-devtools-y-recarga-de-aplicaciones/>
  + <https://www.baeldung.com/spring-boot>
  + https://www.baeldung.com/spring-vs-spring-boot
* Ejemplos
  + <https://github.com/spring-projects/spring-petclinic> (<https://www.baeldung.com/spring-boot-package-structure>)

### Posibles preguntas

* **¿Qué es Spring Boot? ¿Cómo se usa/configura? ¿Qué ventajas aporta?**

**SpringBoot** es una extensión de Spring Framework (teniendo en cuenta que Spring Framework proporciona una infraestructura de trabajo, una serie de funcionalidades para inyectar dependencias). Spring Boot es un conjunto de herramientas que proporcionan unos modelos o plantillas, unas autoconfiguraciones para empezar a desarrollar una aplicación basada en Spring sin tener que perder tiempo en la configuración base de la aplicación.

Proporciona una serie de “paquetes” de dependencias predefinidos en función del tipo de aplicación que se desea desarrollar, por lo que no habría que seleccionar las dependencias o librerías JAR una a una.

Para **configurar** Spring Boot es necesario crear un nuevo proyecto en Java y agregar las dependencias de SpringBoot en el fichero pom.xml (Maven) o build.gradle (Gradle). Con ello, ya se implementan en el proyecto unas configuraciones predeterminadas, que pueden ser personalizadas por el desarrollador mediante propiedades de configuración o anotaciones. A partir de ahí, se puede desarrollar la aplicación utilizando anotaciones de Spring: Spring Bot escaneará automáticamente los componentes y los configurará.

Las ventajas principales son:

* Se simplifica la configuración inicial de la aplicación Spring ya que se parte de una base predefinida. Se ahorran tiempos en este aspecto.
* Ofrece una configuración para muchos aspectos “estándar” de una aplicación, por lo que no es necesario configurar uno a uno cada aspecto. Se traduce nuevamente en un aumento de la productividad y en una reducción de tiempos.
* Facilidad de despliegue, ya que Spring Boot empaqueta la aplicación con todas sus dependencias.
* **¿Es cierto que solo sirve para crear aplicaciones web?**

No es cierto. Spring Boot está muy generalizado para el desarrollo de aplicaciones web en Java, pero no está limitado únicamente a este tipo de aplicaciones. Es una herramienta muy versátil y desplegable por lo que puede usarse en muchos desarrrollos. Entre otros:

* Servicios RESTful
* Aplicaciones de microservicios
* Procesamiento de lotes
* Aplicaciones de Internet de las Cosas
* **¿Cuál es la diferencia con Spring? ¿Si uso Spring estoy usando Spring Boot?**

Spring es el marco de trabajo principal que proporciona la base para el desarrollo: herramientas para la Inversión de Control (IoC), la inyección de dependencias, la programación orientada a aspectos...

Spring Boot es una extensión de Spring, un paso adelante: ofrece un conjunto de configuraciones predeterminadas de Spring que permiten empezar a desarrollar directamente la aplicación, incluyendo un servidor embebido (como Tomcat).

Es decir, se puede utilizar Spring sin utilizar Spring Boot, ya que las configuraciones se pueden realizar desde cero y manualmente a través de Spring Framework.

* **¿Para qué sirve Spring initializer?**

Es el asistente de Spring Boot: una herramienta que guía al desarrollador en la creación de un nuevo proyecto de Spring Boot. Proporciona una interfaz en la que se pueden especificar las dependencias, configuraciones y opciones del proyecto. Una vez ejecutados los pasos del asistente (Spring Initializer), se genera un proyecto de Spring Boot listo para empezar a desarrollar. En definitiva, es una herramienta de “inicio rápido” para un proyecto en Spring Boot.

* **¿Por qué se asocia tanto Spring Boot a los Microservicios?**

Porque es una herramienta rápida y de fácil configuración (como ya se ha explicado, carga configuraciones por defecto) que permite ahorrar tiempos en el desarrollo de aplicaciones sencillas o de funcionalidad específica como pueden ser los microservicios. La flexibilidad y modularidad que ofrece Spring Boot permite también una alta integración con otras herramientas específicas para el desarrollo de microservicios.

### Tareas

### Dudas

## Spring REST

### Objetivos

* **Conocer las opciones que proporciona Spring para crear servicios REST.**
* **Adquirir conocimiento sobre buenas prácticas en diseño de servicios REST**

### Recursos formativos

Lecturas

* + <https://www.arquitecturajava.com/responseentity-spring-y-rest/>
  + <https://www.baeldung.com/spring-rest-openapi-documentation>
  + <https://spring.io/guides/gs/rest-service/>

Curso

* + Cómo crear una API REST (<https://www.youtube.com/watch?v=wT-hIeYyxBg>)
  + Seguridad en Spring Boot <https://www.youtube.com/watch?v=nwqQYCM4YT8>

Antiguos, pero sirven para afianzar conceptos:

* + Seguridad en Spring boot [Protege tu API usando JWT y Spring Security (2022) - YouTube](https://www.youtube.com/watch?v=_p-Odh3MZJc)
  + Curso práctico de APIs REST en Spring Boot con JWT , Spring Security , MySQL y Spring Data JPA <https://www.youtube.com/watch?v=dJaY43Butm8> (complete 5hrs)

### Posibles preguntas

* **¿Cuáles son los componentes de Spring Rest?**

En el contexto del desarrollo de servicios web REST, Spring tiene los siguientes componentes:

* **Controladores (Controllers)**: Son clases de Java anotadas con @RestController o @Controller, que manejan las solicitudes HTTP y generan las respuestas correspondientes. Los métodos en estos controladores están mapeados a las URLs de la API y responden a diferentes métodos HTTP (GET, POST, PUT, DELETE, etc.) utilizando anotaciones como @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, etc.
* **Anotaciones (Annotations)**: Para configurar y mapear controladores y sus métodos a endpoints REST. Algunas de las anotaciones comunes son @RestController, @Controller, @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, etc.
* **Conversiones de mensajes (Message Converters)**: Esto permite que las respuestas y las solicitudes HTTP sean convertidas automáticamente a objetos Java y viceversa. Spring proporciona convertidores de mensajes por defecto para los formatos de datos más comunes (JSON,XML,TAML...) y también permite la configuración de convertidores personalizados.
* **Validación de datos (Data Validation)**: Spring REST permite la validación de datos recibidos en las solicitudes HTTP utilizando las anotaciones de validación de javax.validation. Esto permite garantizar que los datos enviados por el cliente cumplan con ciertos criterios antes de ser procesados por la aplicación.
* **Manejo de excepciones (Exception Handling)**: Spring REST proporciona mecanismos para manejar excepciones y errores en las solicitudes HTTP de forma adecuada. Esto incluye el manejo de excepciones a través de anotaciones (@ExceptionHandler) o mediante clases que implementan la interfaz HandlerExceptionResolver.
* **Documentación de la API (API Documentation)**: Spring REST se puede integrar con herramientas de documentación de API como Swagger o Springfox para generar documentación automática de la API a partir de los controladores y las anotaciones utilizadas en el código.
* **¿Qué ocurre cuando uso la anotación @RestController? ¿Es distinto a @Controler?**

La anotación **@RestController** se utiliza para marcar una clase como un controlador REST en Spring. Indica que todos los métodos en esa clase devolverán directamente objetos que se convertirán automáticamente en formatos JSON o XML y serán escritos directamente al cuerpo de la respuesta HTTP. Es decir, @RestController marca la clase que tendrá la función de controlador al mismo tiempo que indica que el valor devuelto por los métodos se incorpore directamente al cuerpo de la respuesta HTTP.

La anotación **@Controller** se utiliza simplemente para marcar una clase como un controlador y devuelver una vista como respuesta.

Por tanto, **@RestController** combina la funcionalidad de @Controller y @ResponseBody en una sola anotación.

* **¿Cómo podemos gestionar la seguridad en el backend?**

Con el componente Spring Security: Proporciona herramientas de seguridad, autenticación, autorización, protección contra ataques comunes, gestión de sesiones, cifrado de contraseñas, etc. Spring Security proporciona anotaciones como @Secured, @PreAuthorize y @PostAuthorize que puedes aplicar a métodos de controlador para restringir el acceso a los mismos basándose en roles de usuario o expresiones de seguridad.

* **¿Para qué sirve RestTemplate?**

Es una clase de Spring Framework que simplifica la comunicación, desde la parte cliente, con servicios RESTful, proporcionando métodos convenientes para realizar solicitudes HTTP, gestionar respuestas, gestionar errores y excepciones, y más.

Por ejemplo:

* Permite realizar solicitudes HTTP a servicios RESTful utilizando métodos como getForObject(), postForObject(), exchange(), etc. Estos métodos permiten realizar las acciones HTTP, como GET, POST, PUT, DELETE, etc.
* Facilita la conversión de objetos Java a formato JSON o XML para incluirlos en el cuerpo de una solicitud HTTP.
* Permite agregar encabezados HTTP personalizados y parámetros de consulta a tus solicitudes HTTP utilizando métodos como addHeader(), addParameter(), etc. Esto te permite enviar información adicional junto con tu solicitud.
* **¿para qué nos sirve un Header?**

El encabezado (header) proporciona información adicional en una petición o respuesta HTTP. Como el principio de REST es “sin estado” (no se almacena información entre peticiones) cada petición debe proporcionar la información necesaria para que pueda procesarse correctamente. En ese sentido, los headers permiten que los clientes y servidores comuniquen detalles importantes o tokens sobre la solicitud o la respuesta, como el tipo de contenido, la codificación, la autenticación, las autorizaciones...

* **¿Qué podemos hacer con el ResponseEntity?**

Es una clase en Spring que controla todos los aspectos de una la respuesta HTTP, incluyendo el cuerpo, los headers y el código de estado. COmo tal, permite controlar el cuerpo de la respuesta devolviendo un objeto, como un fichero JSON o XML, establece los headers y el código en sí que manipula la respuesta HTTP.

* **¿Qué es JWT?**

Es un estándar que define la forma que debe tener un fichero JSON para transmitir información de forma segura entre dos partes, en este caso, cliente y servidor.

Un JWT se compone de tres partes:

* **Header**: Contiene metadatos sobre el tipo de token y el algoritmo de firma utilizado.
* **Payload**: Es donde se almacenan los datos que se transmiten.
* **Signature**: Es la firma que se utiliza para verificar la integridad del token y asegurarse de que no haya sido manipulado.

JWT se usa, principalmente, para la autenticación y la seguridad en aplicaciones web y servicios API.

* **¿Cuál crees que son las ventajas y desventajas de usar Spring Rest?**

Las **ventajas**:

* Simplifica el desarrollo de servicios web RESTful al proporcionar un conjunto de herramientas que facilitan la creación de endpoints. Esto permite a los desarrolladores centrarse en la lógica de negocio en lugar de preocuparse por los detalles de la infraestructura web.
* Es altamente flexible y configurable, lo que permite a los desarrolladores adaptar los servicios a los requisitos específicos de su aplicación.
* Se integra perfectamente con otros proyectos y módulos de Spring, como Spring Security, Spring Data, Spring Boot, etc. Esto facilita la construcción de aplicaciones web completas y escalables utilizando tecnologías de Spring.

Las **desventajas**:

* La configuración inicial puede ser compleja y requerir un conocimiento detallado de los conceptos de Spring y las mejores prácticas de diseño.
* Un proyecto de Spring REST puede tener un tamaño considerable debido a la inclusión de bibliotecas y dependencias de Spring, lo que puede aumentar el tiempo de compilación y el tamaño de implementación.

Aunque en general y en mi opinión, son mucho más valorables las ventajas que las desventajas. En general facilita enormemente la tarea del desarrollador y disminuye la complejidad de las aplicaciones.

### Tareas

### Dudas

¿?