

➤ La interoperabilidad entre los sistemas

Plan formativo: Desarrollo de Aplicaciones Full Stack Java Trainee V2.0

HOJA DE RUTA

¿Cuáles **skill** conforman el programa?



REPASO CLASE ANTERIOR

En la clase anterior trabajamos :

- ✓ La notación *JSON* para el traspaso de información

LEARNING PATHWAY

Nº de la unidad . ¿Sobre qué temas trabajaremos?

6.5

Start! 🏁

**La interoperabilidad
entre los sistemas**

RestTemplate

RestTemplate

Consumiendo un servicio REST con Spring y RestTemplate
Principios de diseño de una API de REST

OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?



Aprender el proceso para consumir un servicio REST con Spring y RestTemplate



Conocer y aplicar los principios de diseño de una API de REST



› RestTemplate



RestTemplate



Spring Boot proporciona una forma conveniente de realizar solicitudes HTTP mediante el uso de la clase **RestTemplate**. Esta clase es una herramienta poderosa para realizar solicitudes a servicios web RESTful y se puede usar para solicitudes sincrónicas y asincrónicas.





RestTemplate



Un servicio **RESTful** expone recursos a través de **URIs**. Cada recurso puede tener múltiples representaciones, siendo **JSON** y **XML** las más comunes. Las operaciones **HTTP** definen las acciones sobre los recursos: **GET** para recuperar, **POST** para crear, **PUT** para actualizar y **DELETE** para eliminar.



```
GET /usuarios/1 // Obtener información sobre el usuario con ID 1
POST /usuarios // Crear un nuevo usuario
PUT /usuarios/1 // Actualizar información del usuario con ID 1
DELETE /usuarios/1 // Eliminar al usuario con ID 1
```





RestTemplate

RestTemplate es una clase de Spring que simplifica la interacción con servicios RESTful. Proporciona métodos para realizar operaciones HTTP y gestionar la serialización y deserialización de datos en formato JSON o XML.

Para poder utilizarla es necesario tener la dependencia spring-web e importar la clase necesaria:

```
import org.springframework.web.client.RestTemplate;
```

```
@Autowired  
RestTemplate restTemplate;
```





RestTemplate



Mejores Prácticas para consumir servicios RESTful

- **Manejo de Excepciones:** Implementar manejo adecuado de excepciones para gestionar errores de red, problemas de autenticación y otros escenarios inesperados.
- **Paginación y Filtrado:** Utilizar paginación y filtrado en las solicitudes para optimizar el rendimiento y reducir el tamaño de las respuestas.
- **Caching:** Aplicar estrategias de caching para evitar solicitudes redundantes y mejorar la eficiencia.





RestTemplate



Consideraciones de Seguridad a la hora de consumir servicios RESTful

- **Autenticación y Autorización:** Implementar mecanismos de autenticación (como tokens) y autorización para proteger el acceso a los recursos.
- **Conexiones Seguras (HTTPS):** Utilizar conexiones seguras para proteger la confidencialidad y la integridad de los datos transmitidos.
- **Validación de Entradas:** Validar y sanitizar las entradas para prevenir ataques comunes como inyecciones SQL o XSS.





RestTemplate

Los **principales métodos** que provee **RestTemplate** en Spring **para consumir servicios REST** son:

- `getForObject()`
- `getForEntity()`
- `postForObject()`
- `postForEntity()`
- `put()``delete()`
- `exchange()`



Estos métodos permiten de forma sencilla:

- Realizar operaciones con los principales verbos HTTP (GET, POST, PUT, DELETE).
- Enviar objetos Java en el cuerpo y recibir objetos mapeados desde la respuesta.
- Obtener metadatos de la respuesta en `ResponseEntity`.
- Intercambiar datos en formatos como JSON.
- Manejar errores y excepciones de forma transparente.



RestTemplate

Realizar solicitudes GET



Para realizar una solicitud GET, puede utilizar los métodos `getForObject()` o `getForEntity()`.

El método `getForObject()` devuelve el cuerpo de la respuesta como un objeto, mientras que el método `getForEntity()` devuelve la respuesta completa, incluidos los encabezados y el código de estado.



```
// Realiza una solicitud GET y devuelve el cuerpo de la respuesta como un objeto Usuario
Usuario usuario = restTemplate.getForObject( "https://api.example.com/usuarios/{id}" , Usuario.class,1);

// Realiza una solicitud GET y devuelve la respuesta completa
ResponseEntity<Usuario> usuario2 = restTemplate.getForEntity("https://api.example.com/usuarios",Usuario.class);
```



RestTemplate

Realizar solicitudes POST



Para realizar una solicitud POST, puede utilizar los métodos `postForObject()` o `postForEntity()`.

Estos métodos funcionan de manera similar a los métodos GET, donde el método `postForObject()` devuelve el cuerpo de la respuesta como un objeto y el método `postForEntity()` devuelve la respuesta completa.



```
// Realiza una solicitud POST de creación de Usuario y devuelve el cuerpo de la respuesta como un objeto Usuario
Usuario usuario = restTemplate.postForObject("https://api.example.com/usuarios", new Usuario(), Usuario.class);

// Realiza una solicitud POST y devuelve la respuesta completa
ResponseEntity<Usuario> respuesta1 = restTemplate.postForEntity("https://api.example.com/usuarios", new Usuario(), Usuario.class);
```



RestTemplate

Realizar solicitudes PUT y DELETE



Para realizar solicitudes PUT y DELETE, puede utilizar los métodos `put()` y `delete()` respectivamente.

En este ejemplo, estamos realizando una solicitud **PUT** a la URL que representa el recurso del usuario con ID 1. Estamos enviando el objeto `usuarioActualizado` como parte del cuerpo de la solicitud, y RestTemplate se encarga de serializarlo automáticamente en el formato correcto (por ejemplo, JSON o XML) antes de enviar la solicitud al servidor.



```
RestTemplate restTemplate = new RestTemplate();
String url = "https://api.example.com/usuarios/{id}";
Usuario usuarioActualizado = new Usuario("UsuarioActualizado", 30);

restTemplate.put(url, usuarioActualizado, 1);
```





RestTemplate

Realizar solicitudes PUT y DELETE



La operación HTTP DELETE se utiliza para eliminar recursos en el servidor. RestTemplate ofrece el método delete para realizar solicitudes DELETE.

En este ejemplo, estamos realizando una solicitud **DELETE** a la URL que representa el recurso del usuario con ID 1. RestTemplate se encarga de construir y enviar la solicitud DELETE al servidor.



```
RestTemplate restTemplate = new RestTemplate();  
String url = "https://api.example.com/usuarios/{id}";  
  
restTemplate.delete(url, 1);
```





RestTemplate



Casos de uso:

El RestTemplate típicamente se inyecta y utiliza en los **controllers** de Spring MVC. De esta forma, encapsulamos las llamadas a servicios externos en los controladores, manteniendo la lógica del negocio separada. El RestTemplate nos resulta muy útil en aplicaciones que consumen múltiples APIs.



```
@RestController
@RequestMapping("/api")
public class MyController {

    @Autowired
    RestTemplate restTemplate;

    @GetMapping("/users")
```





RestTemplate

Con **getForObject** enviamos una petición GET y recibimos el cuerpo de la respuesta directamente mapeado a una instancia de la clase **User**.
En el controlador agregaremos el objeto modelo y la vista a devolver en un **ModelAndView**.



```
@RestController
@RequestMapping("/api")
public class MyController {
    @Autowired
    RestTemplate restTemplate;

    @GetMapping("/users")
    public ModelAndView getUsers() {

        // Llamada a API externa
        List<User> users = restTemplate.getForObject("https://api.example.com/users", List.class);

        // Agregar al modelo
        ModelAndView model = new ModelAndView("users");
        model.addObject("users", users);

        return model;
    }
}
```





RestTemplate

Luego, en la vista `users.jsp` accedemos al modelo



```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
  <title>Users</title>
</head>
<body>
  <h1>Users</h1>

  <ul>
    <c:forEach items="${users}" var="user">
      <li>${user.name} - ${user.email}</li>
    </c:forEach>
  </ul>

</body>
</html>
```





LIVE CODING

Ejemplo en vivo

Controlador Rest: *Vamos a configurar los controladores para que respondan al protocolo REST e implemente RestTemplate. Primero debemos anotar las clases controlador como @RestController*

*1- Crear un controlador “buscarUsuario(id)” que implemente el método **getForEntity()** y devuelva la respuesta completa (ResponseEntity).*

  **Tiempo: 15 minutos**



Momento: ✚

Time-out!

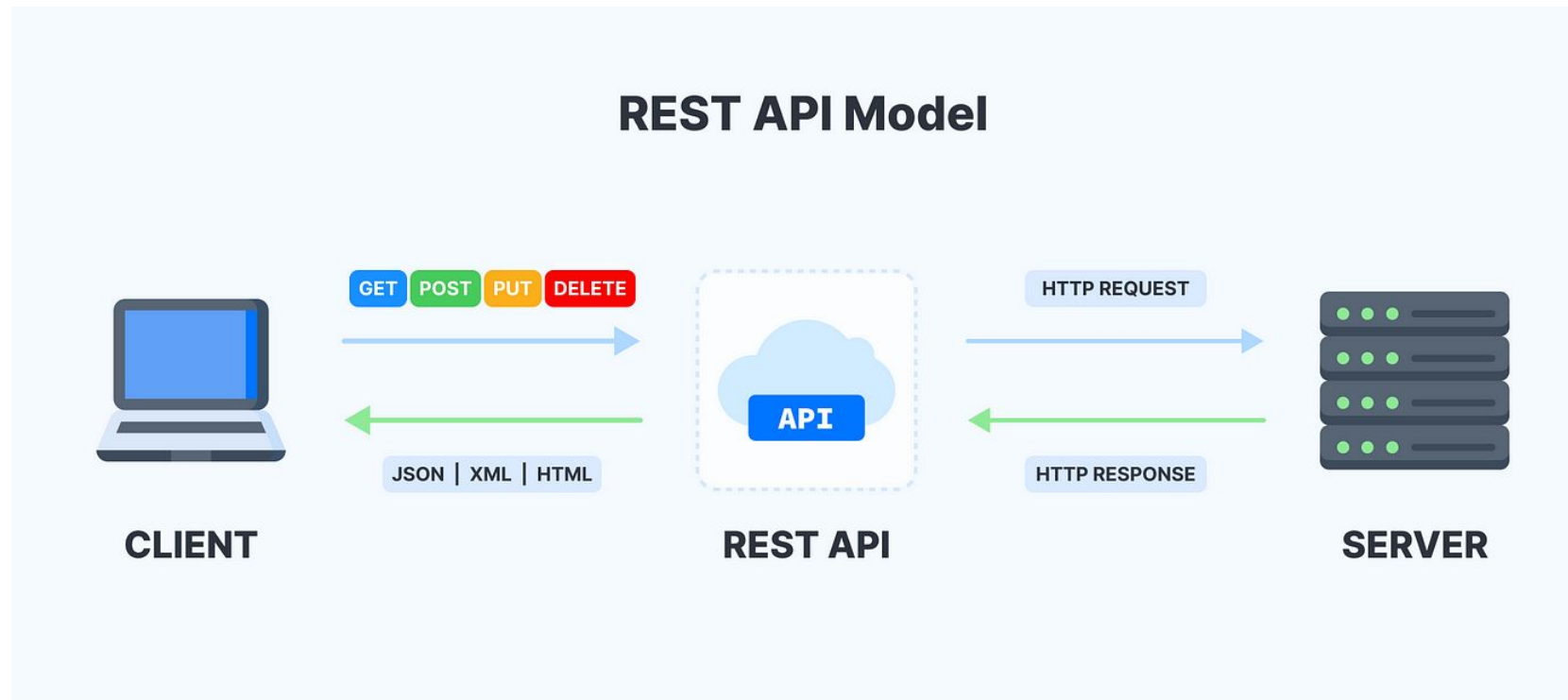
🕒 10 min.



› Principios de Diseño de una API REST

Principios de Diseño de una API REST

Una **API REST** es un conjunto de reglas y convenciones que **permite que dos aplicaciones se comuniquen entre sí a través de HTTP** de manera eficiente y escalable. Se basa en los principios de la arquitectura REST, que se centra en la simplicidad, la escalabilidad y la transferencia de representaciones de estado.





Principios de Diseño de una API REST



Identificación y Nomenclatura de Recursos: Uno de los principios fundamentales de una API REST es la identificación clara y consistente de recursos. Los recursos son entidades o conceptos que pueden ser accedidos o manipulados a través de la API. La nomenclatura de los recursos debe ser significativa y seguir convenciones para mejorar la comprensión y la usabilidad.



- GET /usuarios** : Obtener la lista de usuarios
- GET /usuarios/{id}** : Obtener información sobre un usuario específico
- POST /usuarios** : Crear un nuevo usuario
- PUT /usuarios/{id}** : Actualizar información de un usuario
- DELETE /usuarios/{id}** : Eliminar un usuario





Principios de Diseño de una API REST



Uso Apropiado de Métodos HTTP: Los métodos HTTP (GET, POST, PUT, DELETE, etc.) deben ser utilizados de acuerdo con su semántica específica. Por ejemplo, GET se utiliza para obtener información, POST para crear recursos, PUT para actualizar recursos y DELETE para eliminar recursos. Esta alineación con la semántica HTTP mejora la comprensión y previsibilidad de la API.





Principios de Diseño de una API REST



Uso de Formatos de Datos Estandarizados: El uso de formatos de datos estandarizados, como JSON o XML, facilita la interoperabilidad y el entendimiento. JSON ha ganado popularidad debido a su simplicidad y ligereza.



Documentación Clara y Completa: Una documentación clara y completa es esencial para que los desarrolladores comprendan y utilicen la API de manera efectiva. Herramientas como Swagger o OpenAPI pueden facilitar la generación de documentación a partir del código fuente.





Principios de Diseño de una API REST



Paginación y Filtrado para Listas de Recursos: Cuando se devuelven listas de recursos, se deben proporcionar mecanismos de paginación y filtrado para limitar la cantidad de datos transferidos y mejorar el rendimiento.



- `GET /usuarios?page=1&size=10` : Obtener la primera página de 10 usuarios
- `GET /usuarios?estado=activo` : Filtrar usuarios por estado activo





Principios de Diseño de una API REST

Respuestas HTTP Apropriadas y Códigos de Estado: Las respuestas HTTP deben proporcionar códigos de estado apropiados y significativos para indicar el resultado de la solicitud. Por ejemplo, **200 OK** para respuestas exitosas, **201 Created** para la creación de recursos, **204 No Content** para operaciones sin contenido, y códigos de error específicos para situaciones de error.



●	1XX	Códigos informativos	El servidor ha recibido la petición y procederá con ella.
●	2XX	Códigos de éxito	El servidor ha recibido, entendido y procesado la solicitud correctamente.
●	3XX	Códigos de redirección	El servidor ha recibido la solicitud, pero hay una redirección a alguna otra parte (o, en raras ocasiones, alguna acción adicional que debe completarse).
●	4XX	Códigos de error de cliente	El servidor no puede encontrar (o alcanzar) la página o la web. Se trata de un error del lado de la web.
●	5XX	Códigos de error de servidor	El cliente ha realizado una solicitud válida, pero el servidor ha fallado al completarla.

semrush.com





Principios de Diseño de una API REST



HATEOAS (Hypertext As The Engine Of Application State): HATEOAS es un principio que sugiere que la aplicación del cliente debe ser guiada dinámicamente por enlaces incluidos en las respuestas de hipertexto de las aplicaciones servidoras. Esto facilita la navegación a través de la API sin la necesidad de entender previamente todas las posibles interacciones.

Webhooks y Event-Driven Architecture: La implementación de webhooks permite que la API notifique eventos a los clientes interesados en tiempo real. Esto se alinea con la arquitectura orientada a eventos, proporcionando una forma eficiente de manejar actualizaciones y cambios.





Ejercicio **RestTemplate**



RestTemplate

Contexto: 🙌

Vamos a trabajar sobre un controlador (puede ser en el proyecto en el que vienes practicando).

Consigna: 🖋️ Respondan levantando la mano!

1- Implementa el uso de RestTemplate y de su método postForEntity() en un controlador de inicio de sesión.

2- Reescribe las URIs de tus métodos para que respeten los principios de diseño de una API Rest

Tiempo 🕒: *puedes terminarlo de manera asíncrona*

○

¿Alguna consulta?

+



RESUMEN

¿Qué logramos en esta clase?

- ✓ **Comprender el concepto y las características de *RestTemplate***
- ✓ **Conocer los Principios de diseño de una API de REST**



#WorkingTime

Continuemos ejercitando

¡Antes de cerrar la clase! Te invitamos a: 📌 📌 📌

1. Repasar nuevamente la grabación de esta clase
2. Revisar el material compartido en la plataforma de Moodle (lo que se vio en clase y algún ejercicio adicional)
 - a. *Lectura Modulo 6, Lección 4: páginas 5 - 9*
3. Traer al próximo encuentro, todas tus dudas y consultas para verlas antes de iniciar nuevo tema.

¡Muchas Gracias!

Nos vemos en la próxima clase 🙌



Momento: ✦

Time-out!

🕒 15 min.

