5. Servicios Web.

1. Introducción.

Los servicios web consisten en una serie de tecnologías dedicadas a compartir datos entre diferentes aplicaciones a través del protocolo HTTP, usando como lenguaje de intercambio un lenguaje basado en texto como XML o JSON.

Anteriormente existían otras tecnologías como CORBA (Common Object Request Broker Architecture), RPC (Remote Procedure Call), Java RMI (Remote Method Invocation), etc, que tenían una finalidad similar, pero con el auge de Internet y WWW se han popularizado los servicios web.

Algunas de las ventajas de los Servicios Web son;

- Independencia de las plataformas. Permite intercambiar datos entre plataformas heterogéneas.

- Interfaz homogénea y bien definida. Las operaciones siguen un esquema común y bien definido.

- Uso del protocolo HTTP, lo que facilita el acceso a través de las redes (normalmente, los puertos HTTP y HTTPS están abiertos en todos los firewalls y no hay necesidad de realizar configuraciones específicas).

En Java existen dos estándares para Servicios Web, una basada en XML, conocida como JAX-WS (Java API for XML Web Services) y otra basada en servicios REST , JAX-RS (Java API for RESTful Web Services).

La primera de ellas usa XML como lenguaje de intercambio de objetos y WSDL (Web Service Description Language) como protocolo de descripción de las clases (también basado en XML). En esta tecnología se define un conjunto de métodos de intercambio que se comparten con el cliente mediante una especificación WSDL. El cliente, una vez conocidos los métodos disponibles puede invocarlos y la propia API se encarga del intercambio de datos a través de la red, la codificación y decodificación, etc. Esta tecnología es la más compleja y costosa en términos de recursos.

La tecnología REST (REpresentational State Transfer), fue introducida por Roy Fielding en su tesis doctoral, en el año 2000, como un sistema sencillo y versátil para acceder a recursos a través de una API sencilla y bien definida y con el uso de pocos recursos. Es una tecnología que está tomando un gran auge en los últimos años, ya que permite, por ejemplo, que aplicaciones móviles accedan con facilidad a los datos de un servidor.

En esta introducción nos centraremos en la tecnología Java para Servicios Web REST, es decir, en JAX-RS.

Normalmente, llamamos API REST a un conjunto de servicios REST relacionados, ya que proporcionan una interfaz para acceder a un conjunto de recursos del servidor. Por ejemplo, a un conjunto de Servicios REST que permiten acceder a una Base de Datos de películas le llamamos API REST.

2. Estructura de un servicio REST Java.

Un Servicio REST Java debe tener una clase de configuración, que debe heredar de javax.ws.rs.core.Application y debe estar decorada con la anotación @ApplicationPath, donde indicaremos el path principal del Servicio REST dentro de la aplicación. La clase de configuración puede tener cualquier nombre.

```
import javax.ws.rs.core.Application;


@javax.ws.rs.ApplicationPath("servicioweb")
public class ApplicationConfig extends Application {
}
```

Además habrá una clase para cada servicio REST, con la anotación @Path para indicar el path de dicho servicio dentro del path principal especificado en @ApplicationPath. En esta clase colocaremos los métodos de acceso al servicio. La clase podrá ir precedida por anotaciones @Consumes y/o @Produces, para indicar los tipos de contenidos que recibe y produce el servicio, aunque alternativamente podemos poner estas anotaciones en cada método del servicio.

```
@Path("holaMundo")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class HolaMundoRest {

    @GET
    public Response hola() {
        return Response.ok("Hola, Mundo desde el API
REST",MediaType.APPLICATION_JSON).build();
    }
}
```

3. Métodos HTTP.

Los servicios REST definen un interfaz homogéneo a través de los paths y los métodos HTTP, de tal forma que un determinado path indica un recurso y el método usado en la petición indica la operación a realizar sobre ese recurso.

De esta forma el path de un recurso podría ser, por ejemplo servicio/producto, y el método indicaría la acción a realizar, por ejemplo, GET para consultar, POST para crear, etc.

De esta forma se establece la siguiente relación entre operaciones (CRUD) y métodos HTTP:

Método HTTP	Operación	CRUD
POST	Inserción	Create
GET	Consulta	Read
PUT	Actualización	Update
DELETE	Borrado	Delete

Cada método del Servicio REST irá precedido con una anotación que indicará le método HTTP asociado, y que por tanto, determinará la acción a realizar. Las anotaciones pueden ser @GET, @POST, @PUT, etc.

4. Parámetros.

En los Servicios REST los parámetros se pasan normalmente como un elemento más en el path. De esta forma, mientras un GET a servicio/producto nos devolvería la información de todos los productos, un GET a servicio/producto/1 nos devolvería la información del producto con id = 1.

En los métodos de la clase que implementa el servicio, esto se lleva a cabo de la siguiente forma:

```
@Produces(MediaType.APPLICATION_JSON)
@GET
@Path("/producto/{id}")
public Response findProductById(@PathParam("id") long producto) {
    return modelo.findProductById(producto);
}

@Produces(MediaType.APPLICATION_JSON)
@GET
@Path("/producto")
public Response findAllProducts() {
    return modelo.findAllProducts();
}
```

También se pueden recuperar parámetros de la petición (los que aparecen en la URL, después de ?) mediante la anotación @QueryParam o parámetros de un formulario mediante la anotación @FormParam en combinación con @Consumes(MediaType.APPLICATION_FORM_URLENCODED).

También es posible asignar un valor por defecto a un parámetro mediante la anotación @DefaultValue en el propio parámetro. De esta forma, el parámetro sería opcional.

```
@Produces(MediaType.APPLICATION_JSON)
@GET
@Path("/producto")
public Response findAllProducts(@DefaultValue(“1”) @QueryParam int
                                                              pagina) {
    return modelo.findAllProducts(pagina);
}
```

5. Tipos de Contenido.

Mediante la anotación @Consumes podemos especificar el tipo de contenido que es capaz de recibir nuestro servicio. Dicha anotación se puede especificar a nivel de clase o de método y contendrá uno o más tipos mime especificados mediante constantes de la clase MediaType.

De esta forma, nuestro método recibirá como parámetro una entidad (un objeto) obtenido de la decodificación del contenido recibido, que ha sido decodificado mediante un EntityProvider adecuado. Todo esto ocurre de forma transparente, ya que se encarga la propia API JAX-RS.

De la misma forma podemos especificar un tipo mime para el contenido generado mediante la anotación @Produces, que también se puede especificar a nivel de clase o de método. Así cuando devolvamos un objeto o entidad, este será traducido automáticamente al tipo mime especificado.

6. Respuestas personalizadas.

En los Servicios REST se usan ampliamente los códigos de respuesta HTTP para indicar distintas situaciones y errores. Algunos de los más usados son los siguientes:

Código numérico	Mensaje
200	OK
201	Created. (Se ha creado la entidad)
204	No Content. (Petición con éxito que no genera contenido)
400	Bad Request (Petición con sintaxis incorrecta)
401	Unauthorized (Fallo de autenticación)
403	Forbidden (El usuario no tiene privilegios para realizar la petición)

404	Not Found. (Recurso no encontrado)
405	Method not allowed. (No se permite acceder con ese método HTTP)
415	Unsupported Media Type. (El servicio no soporta el tipo de contenido)
500	Internal Server Error (Error en el Servidor)
503	Service Unavailable (Servicio no disponible en ese momento)

Para añadir ciertos metadatos a nuestra respuesta, tales como el código de estado, podemos hacer que nuestros métodos devuelvan un objeto de tipo Response, que actúa como un envoltorio de la entidad a devolver (si la hay), añadiéndole información adicional como el código de respuesta o una redirección.

La forma más común de hacer esto es encadenar algunos métodos estáticos de Response que añaden cierta información y devuelven un ResponseBuilder y al final llamar al método build() para obtener la respuesta completa:

```
@Produces(MediaType.APPLICATION_JSON)
@GET
@Path("/producto/{id}")
public Response getProduct(@PathParam(“id”) long id) {
    Producto p = modelo.findProduct(id);
    return
Response.ok().type(MediaType.APPLICATION_JSON).entity(p).build();
}

@Consumes(MediaType.APPLICATION_JSON)
@POST
@Path("/producto")
public Response createProduct(@Context UriInfo uriInfo, Producto p) {
      long id = modelo.create(p);
      String uri = uriInfo.getPath + “/” + id;
      return Response.created(uri).build();

}
```

Para ver más información sobre los métodos de la clase Response, consultar la referencia de la API javax.ws.rs.core.Response.

7. CORS.

CORS (Cross Origin Resource Sharing, Compartición de Recursos con Orígenes Cruzados) es una especificación de seguridad que siguen los navegadores para evitar que un recurso como una página web haga referencia a otros recursos procedentes de un dominio distinto al suyo, a no ser que se cumplan ciertas características, como por ejemplo, que el recurso solicitado sea una imagen o un fichero css.

Además, en la especificación CORS, se restringen también los accesos para métodos HTTP y tipos de contenidos distintos de los habituales en las páginas web y para ciertas cabeceras.

Para que un recurso de origen cruzado sea accesible, el servidor debe añadir a su respuesta cabeceras de tipo Access-Control-Allow-Origin, Access-Control-Allow-Methods o Access-Control-Allow-Headers.

Como los Servicios REST usan tipos de contenidos y métodos HTTP distintos de los usados en las páginas web, será necesario, en general, que añadamos dichas cabeceras. Además, hay que tener en cuenta que los Servicios REST pueden ser accedidos desde peticiones AJAX, u otros tipos de clientes.

Una forma común de configurar una API REST para que añada las cabeceras de control de acceso, es mediante un servlet filter, que en una configuración sencilla añadirá las cabeceras a todas las respuestas, aunque también es posible controlar el acceso de forma más selectiva.

En el siguiente ejemplo se muestra una configuración sencilla que permite el acceso de forma universal:

```
@Provider
public class MyCrossOriginResourceSharingFilter implements
ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext,
ContainerResponseContext response) {
        response.getHeaders().putSingle("Access-Control-Allow-Origin",
"*");
        response.getHeaders().putSingle("Access-Control-Allow-
Methods", "OPTIONS, GET, POST, PUT, DELETE");
        response.getHeaders().putSingle("Access-Control-Allow-
Headers", "Content-Type");
    }

}
```

8. Seguridad en Servicios REST Java.

Para implementar la auntenticación y el control de acceso a los recursos a nivel de usuarios, se pueden usar varias técnicas. Una de ellas es autenticar al usuario a través de la sesión. Para ello podemos usar un servlet filter y centralizar así todo el control de acceso a la API REST. Para ello el usuario se habrá tenido que registrar previamente a través de una página de login o algo similar.

Otra forma muy común de autenticar a los usuarios es a través de un token (una cadena aleatoria) que se le habrá asignado previamente al usuario y se le habrá hecho llegar por otro medio (email, por ejemplo). El usuario tendrá que enviar dicho token en cada petición para autenticarse.

9. Parche para Glassfish 4.

En el directorio Glassfish_install/glassfish/modules hay que reemplazar el fichero org.eclipse.persistence.moxy.jar por org.eclipse.persistence.moxy-2.6.1.jar para solucionar el problema del parse de JSON.