Servicios REST

Sesión 4:
HATEOAS. Seguridad

Índice

- **HATEOAS**
- Seguridad

HATEOAS

- HATEOAS: Hypermedia As The Engine Of the Application State
- La idea de HATEOAS es que el formato de los datos proporciona información extra sobre cómo cambiar el estado de nuestra aplicación
- En la Web, los enlaces HTML nos permiten cambiar el estado de nuestro navegador. Por ejemplo cuando estamos leyendo una página web, un enlace nos indica qué posibles documentos (estados) podemos ver a continuación. Cuando hacemos "click" sobre un enlace, el estado del navegador cambia al visitar y mostrar una nueva página web.
- Los formularios HTML, por otra parte, nos proporcionan una forma de cambiar el estado de un recurso específico de nuestro servidor.
- Por último, cuando compramos algo en Internet, por ejemplo, estamos creando dos nuevos recursos en el servicio: una transacción con tarjeta de crédito y una orden de compra.

Enlaces Atom: en el cuerpo de la respuesta

- Mecanismo estándar para incluir enlaces en documentos XML
- Se pueden usar en los datos XML que procesa o devuelve el servicio REST
- Permite indicar a la aplicación cómo seguir la navegación (por ejemplo, la siguiente página de un listado, o el siguiente elemento a mostrar)

```
<clientes>
   <link rel="next"
         href="http://ejemplo.com/clientes?inicio=2&total=2"
        type="application/xml"/>
   <cliente id="123">
      <nombre>Juan Garcia</nombre>
   </cliente>
   <cliente id="332">
      <nombre>Pablo Bozo</nombre>
   </cliente>
</clientes>
```

El atributo `rel` se utiliza para indicar la relación del enlace con el elemento XML en el que anidamos dicho enlace. Es el **nombre lógico** utilizado para referenciar el enlace.

El atributo `href` es la **URL** a la que podemos acceder para obtener nuevai nformación o cambiar el estado de nuestra aplicación

El atributo `type` indica el ***media type*** asociado con el recurso al que apunta la URL

Enlaces en la cabecera de la respuesta

- Se puede utilizar la cabecera Link para indicar los enlaces HATEOAS, en lugar de incluirlos en el cuerpo de la respuesta

Petición

```
GET /pedidos/333
```

Respuesta

```
HTTP/1.1 200 OK
Content-Type: application/xml
Link: <http://ejemplo.com/pedidos/333/cancelado>; rel=cancel,
      <http://ejemplo.com/pedidos/333>; rel=update

<pedido id="333">
   ...
</pedido>
```

- La cabecera `Link` tiene las mismas características que un enlace Atom.
  - La URI está entre los signos `<` y `>` y está seguida por uno o más atributos delimitados por `;`.
  - El atributo `**rel**` es **obligatorio** y tiene el mismo significado que el correspondiente atributo Atom com el mismo nombre.

Ventajas de usar HATEOAS

- Sólo es necesario hacer públicas unas pocas URLs del servicio
- Desacoplamiento de los detalles de la interacción - la propia respuesta indica cómo continuar preguntando
- Reducción de errores de transición de estado

```
<clientes>
   <link rel="next"
         href="http://ejemplo.com/
                clientes?inicio=2&total=2"
        type="application/xml"/>
   <cliente id="123">
      <nombre>Juan Garcia</nombre>
   </cliente>
   <cliente id="332">
      <nombre>Pablo Bozo</nombre>
   </cliente>
</clientes>
```

```
<pedido id="333">
   <cliente id="123">...</cliente>
   <importe>99.99</importe>
   <cancelado>false</cancelado>
   <link rel="cancelar"
        href="http://ejemplo.com/
                pedidos/333/cancelado"/>
   <lineas-pedido>
            ...
   </lineas-pedido>
</pedido>
```

HATEOAS y JAX-RS

- JAX-RS proporciona un conjunto de clases y métodos para construir enlaces
- La clase `UriBuilder` permite construir una URI elemento a elemento:

```
UriBuilder builder = UriBuilder.fromPath("/clientes/{id}");
builder.scheme("http")
       .host("{hostname}")
       .queryParam("param={param}");
//hacemos una copia por si queremos reutilizar las plantillas
UriBuilder clone = builder.clone();
URI uri = clone.build("ejemplo.com", "333", "valor");
```

```
http://{hostname}/clientes/{id}?param={param}
```

```
http://ejemplo.com/clientes/333?param=valor
```

- Otro ejemplo:

```
Map<String, Object> map = new HashMap<String, Object>();
map.put("hostname", "ejemplo2.com");
map.put("id", 444);
map.put("param", "valor2");
UriBuilder clone = builder.clone();
URI uri = clone.buildFromMap(map);
```

```
http://ejemplo2.com/clientes/444?param=valor2
```

HATEOAS y JAX-RS (2)

- Podemos crear URIs a partir de las expresiones `@Path`:

```
@Path("/clientes")
public class ServicioClientes {

   @Path("{id}")
   public Cliente getCliente(@PathParam("id")
                int id) {...}
}
...
```

```
UriBuilder builder = UriBuilder.fromResource(ServicioClientes.class);
builder.host("{hostname}")
builder.path(ServicioClientes.class,"getCliente");
```

```
http://{hostname}/clientes/{id}
```

URIs relativas mediante el uso de UriInfo

- La aplicación se puede desplegar en distinto servidor o en un contexto distinto
- Es conveniente obtener la parte de la URL del host y la del nombre de la aplicación en tiempo de ejecución
- La clase `UriInfo`, como ya hemos visto, proporciona métodos para atacar este problema con sus métodos:

```
public interface UriInfo {
  public URI getRequestUri();
  public UriBuilder getRequestUriBuilder();
  public URI getAbsolutePath();
  public UriBuilder getAbsolutePathBuilder();
  public URI getBaseUri();
  public UriBuilder getBaseUriBuilder();
```

Clase javax.ws.core.Link

- Esta clase nos permite construir los enlaces para incluirlos en la cabecera de respuesta, o en el cuerpo de de la respuesta
  - Representa todos los metadatos relativos a los enlaces (URIS) de un recurso
  - Los métodos *getRel()*, *getType()* y *getUri()*, devuelven los atributos rel, type y href del enlace
  - Para crear instancias de Link podemos utilizar alguno de los métodos estáticos: *fromXXX()*, que devuelven un objeto *Link.Builder*, el cual nos permitirá construir el enlace encadenando sucesivas llamadas a métodos para, finalmente, invocar al método build(), que construirá el enlace

```
Link link = Link.fromUri("http://{host}/raiz/clientes/{id}")
                .rel("update")
                .type("text/plain")
                .build("localhost", "1234");
```

  - Si invocamos a *link.toString()*, obtendremos lo siguiente:

```
 http://localhost/raiz/clientes/1234>; rel="update"; type="text/plain"
```

Ejemplo

Supongamos que tenemos un servicio que permite acceder a Clientes desde una base de datos. En lugar de tener una URI base que devuelva todos los clientes en un único documento, podemos incluir los enlaces "previo" y "siguiente", de forma que podamos "navegar" por los datos.

```
@Path("/clientes")
public class ServicioClientes {
  @GET
  @Produces("application/xml")
  public Clientes getClientes(
            @QueryParam("start") int start,
            @QueryParam("size"), @Context UriInfo uriInfo) {

    //creamos la uri base
    UriBuilder builder = uriInfo.getAbsolutePathBuilder();
    builder.queryParam("start", "{start}");
    builder.queryParam("size", "{size}");

    ArrayList<Cliente> list = new ArrayList<Cliente>();
    ArrayList<Link> links = new ArrayList<Link>();

   //seleccionamos los clientes entre start y (start+size-1)
    int i = 0;
    for (Customer customer : customerDB.values()) {
     if (i >= start && i < start + size) list.add(customer);
     i++;
    }
    // creamos el link "siguiente"
    if (start + size < customerDB.size()) {
      int next = start + size;
      URI nextUri = builder.clone().build(next, size);
      Link nextLink = Link.fromUri(nextUri)
                          .rel("siguiente")
                          .type("application/xml")
                          .build();
      links.add(nextLink);
    }
```

```
http://org.expertojava/jaxrs/clientes?start=5&size=10
```

```
     // creamos el link "previo"
    if (start > 0) {
      int previous = start - size;
      if (previous < 0) previous = 0;
      URI previousUri = builder.clone()
                               .build(previous, size);
      Link previousLink = Link.fromUri(previousUri)
                              .rel("previous")
                              .type("application/xml")
                              .build();
      links.add(previousLink);
    }

    Clientes clientes = new Clientes();
    clientes.setClientes(list);
    clientes.setLinks(links);
    return clientes;
}
```

Ejemplo: incluimos los Links en una clase anotada con JAXB

```
@XmlRootElement(name = "clientes")
@XmlAccessorType(XmlAccessType.FIELD)
public class Clientes{
   protected Collection<Clientes> clientes;
   protected List<Link> links;

   ...

   @XmlElement(name="link")
   @XmlJavaTypeAdapter(Link.JaxbAdapter.class)
   public List<Link> getLinks() {
      return links;
   }

   @XmlTransient
   public URI getNext()
   {
      if (links == null) return null;
      for (Link link : links)
      {
         if ("siguiente".equals(link.getRel())) return link.getUri();
      }
      return null;
   }
```

Utilizamos la anotación XmlJavaTypeAdapter para indicar la clase encargada de serializar/deserializar objetos Link a xml/json

```
   @XmlTransient
   public URI getPrevious()
   {
      if (links == null) return null;
      for (Link link : links)
      {
         if ("anterior".equals(link.getRel())) return link.getUri();
      }
      return null;
   }
}
```

Índice

- HATEOAS
- **Seguridad: autenticación y autorización**

Seguridad BASIC

- Vamos a explicar la **autenticación** BASIC
- Intentamos acceder a una URL protegida y el servidor nos devuelve la indicación

```
HTTP/1.1 401 Unauthorized
WWW-Autenticate: Basic realm="Cliente Realm"
```

Identificador del realm (colección de recursos seguros de un sitio web)

La URL requiere una autenticación de tipo Basic

- Enviamos el login y el password codificados (que no encriptados!) en Base64 y en la cabecera Authorization:

```
GET /clientes/333 HTTP/1.1
Authorization: Basic ZmVsaXBlOmxvY2tpbmc=
```

felipe:locking

Creación de usuarios y roles

- Para poder utilizar la autentificación básica necesitamos tener creados previamente los *realms* en el servidor de aplicaciones Wildfly, y registrar los usuarios que pertenecen a dichos realms. La forma de hacerlo es idéntica a lo que ya habéis visto en la asignatura de Componentes Web
  - Un realm identifica una colección de recursos seguros en un sitio web
  - Para añadir y registrar usuarios usaremos el comando `add-user.sh`
    - Elegimos el ream "ApplicationRealm"
    - Introducimos los datos para cada nuevo usuario que queramos añadir, indicando su login, password, y el grupo (rol) al que queremos que pertenezca dicho usuario.

Definición de la autenticación BASIC en web.xml

- En el fichero `web.xml` definimos el tipo de autenticación BASIC y el patrón de URLs y métodos HTTP a los que se va a aplicar esa autenticación

```
<web-app>
  ...
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>ApplicationRealm</realm-name>
  </login-config>

  <security-constraint>
    <web-resource-collection>
       <web-resource-name>customer creation</web-resource-name>
       <url-pattern>/rest/resources</url-pattern>
       <http-method>POST</http-method>
       <auth-constraint>
          <role-name>admin</role-name>
       </auth-constraint>
    </web-resource-collection>
  </security-constraint>
  <security-role>
    <role-name>admin</role-name>
  </security-role>
</web-app>
```

Encriptación

- Las contraseñas y los logins en BASE64 no están encriptadas
- Para asegurar la encriptación hay que activar el protocolo HTTPS en el servidor y comunicarse con él usando este protocolo
- Lo veremos en el módulo de servidores de aplicaciones y PaaS

Utilización de anotaciones

- En lugar del fichero web.xml podemos restringir el acceso usando anotaciones
- Las mismas que vimos en Componentes Enterprise

```
@Path("/clientes")
@RolesAllowed({"ADMIN", "CLIENTE"})
public class ClienteResource {

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Cliente getCliente(@PathParam("id") int id) {...}

  @RolesAllowed("ADMIN")
  @POST
  @Consumes("application/xml")
  public void crearCliente(Customer cust) {...}

  @PermitAll
  @GET
  @Produces("application/xml")
  public Customer[] getClientes() {}
}
```

Las anotaciones nos proporcionan una mayor flexibilidad frente al uso del fichero de configuración `web.xml`, ya que podemos definir diferentes autorizaciones a nivel de método.

Seguridad programada

- Podemos comprobar en el código restricciones de seguridad usando la interfaz javax.ws.rs.core.SecurityContext:

```
public interface SecurityContext {
    public Principal getUserPrincipal();
    public boolean isUserInRole(String role);
    public boolean isSecure();
   public String getAuthenticationScheme();
}
```

- Ejemplo:

```
@Path("/clientes")
public class CustomerService {
   @GET
   @Produces("application/xml")
   public Cliente[] getClientes(@Context SecurityContext sec) {
      if (sec.isSecure() && !sec.isUserInRole("ADMIN")) {
             logger.log(sec.getUserPrincipal() +
                        " ha accedido a la base de datos de clientes");
      }
      ...
   }
}
```

Login del usuario que ha accedido al método

¿Alguna duda, pregunta…?