Servicios REST

Sesión 4:
Procesamiento JSON. HATEOAS. Escalabilidad y Seguridad

Índice

- Negociación del contenido
- Representaciones XML y JSON
- Respuestas

JSON

- JSON (JavaScript Object Notation) es un formato muy popular para el intercambio de datos basado en texto y bastante menos verboso que la representación XML
- Permite representar objetos con propiedades con datos, con otros objetos o con arrays de otros datos
- El API for JSON processing (*JSR 353*) es un muy reciente, de 2013

```
{ "nombre": "John",
  "apellidos": "Smith",
  "edad": 25,
  "direccion": { "calle": "21 2nd Street",
                 "ciudad": "New York",
                 "codPostal": "10021"
               },
  "telefonos": [
         { "tipo": "fijo",
           "numero": "212 555-1234"
         },
         {
           "tipo": "movil",
           "numero": "646 555-4567"
         }
  ]
}
```

Procesamiento de JSON

- El API de procesamiento de JSON permite:
  - Crear modelos de objetos de forma programativa desde el código de la aplicación
  - Procesar objetos leídos, navegando por el modelo de objetos
  - Leer los objetos y escribirlos desde/en un stream
- Consultar los apuntes y el API

HATEOAS

- HATEOAS: Hypermedia As The Engine Of the Aplication 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

- 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>
```

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}");
UriBuilder clone = builder.clone();
URI uri = clone.build("ejemplo.com", "333", "valor");
```

- Otro ejemplo:

```
Map<String, Object> map = new HashMap<String, Object>();
map.put("hostname", "ejemplo.com");
map.put("id", 333);
map.put("param", "valor");
UriBuilder clone = builder.clone();
URI uri = clone.buildFromMap(map);
```

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, "getCustomer");
```

```
@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,
                "getCustomer");
```

URIs relativas mediante el uso de UriInfo

- La aplicación se puede desplegar en distinto servidor o en 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` permite 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();
```

Ejemplo

```
@Path("/clientes")
public class ServicioClientes {
  @GET
  @Produces("application/xml")
  public String getCustomers(@Context UriInfo uriInfo) {
     UriBuilder nextLinkBuilder = uriInfo.getAbsolutePathBuilder();
     nextLinkBuilder.queryParam("inicio", 5);
     nextLinkBuilder.queryParam("total", 10);
     URI next = nextLinkBuilder.build();
      //... rellenar el resto del documento ...
  }
 ...
}
```

Cacheado de páginas en el navegador

- La especificación JAX-RS proporciona `javax.ws.rs.core.CacheControl`, que es una clase que representa la cabecera `Cache-Control`

```
public class CacheControl {
  public CacheControl() {...}
  public static CacheControl valueOf(String value)
                              throws IllegalArgumentException {...}
  public boolean isMustRevalidate() {...}
  ...
  public boolean isProxyRevalidate() {...}
  ...
  public int getMaxAge() {...}
  public void setMaxAge(int maxAge) {...}
  public int getSMaxAge() {...}
  public void setSMaxAge(int sMaxAge) {...}
  ...
  public void setNoCache(boolean noCache) {...}
  public boolean isNoCache() {...}
  public boolean isPrivate() {...}
  public List<String> getPrivateFields() {...}
  public void setPrivate(boolean private) {...}
  public boolean isNoTransform() {...}
  public void setNoTransform(boolean noTransform) {...}
  public boolean isNoStore() {...}
  public void setNoStore(boolean noStore) {...}
  ...
}
```

```
@Path("/clientes")
public class ClienteRecurso {
@Path("{id}")
@GET
@Produces("application/xml")
public Response getCliente(@PathParam("id") int id) {
    Cliente cli = buscarCliente(id);
    CacheControl cc = new CacheControl();
    cc.setMaxAge(300);
    cc.setPrivate(true);
    cc.setNoStore(true);
    ResponseBuilder builder = Response.ok(cli,
                  "application/xml");
    builder.cacheControl(cc);
    return builder.build();
}
```

Filtros e interceptores

- Tanto los filtros como los interceptores son objetos que se "interponen" entre el procesamiento de las peticiones, tanto del servidor como del cliente.
- Este comportamiento está relacionado con código referente a la infraestructura o acciones de protocolo que no queremos "entremezclar" con la lógica de negocio.
- Aunque la mayoría de características del API de JAX-RS se usan por los desarrolladores de aplicaciones, los filtros e interceptores suelen utilizarse más por los desarrolladores de middleware.
- Filtros de petición y de respuesta
- Consultar la forma de definirlos en los apuntes

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, si podemos acceder a una URL con un identificador, podremos acceder a todas las demás con ese mismo identificador

- 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

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 getClienter(@PathParam("id") int id) {...}

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

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

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