Desarrollo Basado en Componente y Servicios

Servicios Web de tipo REST

Material adicional: Cliente Rest - AngularJs

Introducción

El objetivo de esta parte es doble. Por un lado probar de manera práctica un cliente REST con tecnología completamente ajena a Java, mostrando como al usar servicios Web en general, y de tipo REST en particular, podemos mezclar distintas tecnologías sin problema. Por otro lado, queremos introducir AngularJs, que es una de las bibliotecas Javascript más utilizadas actualmente para la creación de páginas web dinámicas.

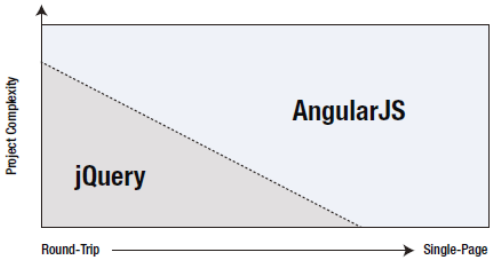
En este documento vamos a realizar una introducción a su utilización, aplicándola al consumo de servicios web de tipo REST. No se pretende hacer una descripción profunda de sus funcionalidades, ya que esto cae fuera de los objetivos de la asignatura. El objetivo aquí, es familiarizar al alumno con una de las bibliotecas javascript más importantes actualmente.

1. Qué es AngularJs

AngularJs es una biblioteca javascript que permite crear páginas web dinámicas. Hay dos enfoques extremos a la hora de crear páginas web dinámicas:

- "Round-trip model": la creación de las páginas recae enteramente en el servidor. Cualquier petición (request) del usuario implica una llamada al servidor, que genera y devuelve una nueva página al navegador. Es la forma en que se ha operado al crear aplicaciones web basadas en JEE. Es también la forma de trabajar más tradicionalmente usada.
- "Single-page application": el servidor sirve la primera página y a partir de ahí todo el procesamiento dinámico (creación de nuevas vistas) se realiza en el cliente.

La mayoría de las aplicaciones web se mueven en enfoques intermedios, con la inclusión de pequeños programas javaescript. AngularJs está pensado para enfoques basados en la opción "single-page" y para proyectos de alta complejidad. Para enfoques intermedios y/o proyectos web simples, hay otras opciones javascript como JQuery (ver figura).



Aunque nuestro proyecto no va a ser complejo, vamos a usar AngularJs por motivos formativos.

2. AngularJs vs Otras Alternativas

Lo primero es diferenciar entre AngularJs y Angular (también llamada a veces, erróneamente, Angular 2). Compartan nombre por el origen (grupo que las creó) y porque Angular 2 debiera haber sido una evolución de Angular 1 (AngularJs). Sin embargo, los desarrolladores decidieron seguir un enfoque completamente diferente. Por esta razón, a las versiones posteriores a Angular 1, se las conoce como Angular 2, aunque actualmente Angular está por la versión 4. Se deja al alumno, profundizar en esas diferencias. Aquí sólo comentaremos que el número de aplicaciones con AngularJs es mayor actualmente, sobre todo debido a la relativamente reciente aparición de Angular. Sin embargo, el futuro parece ir en la dirección de ir cambiando poco a poco a Angular.

Si se buscan alternativas o comparativas entre bibliotecas similares a Angular ("detrás" está Google), quizás la más referenciada sea React (mantenida por Facebook y usada en Instagram). No son incompatibles, ya que se pueden usar las ventajas de Angular en cuanto a construcción de la aplicación, junto con las de React para la construcción de la interfaz o las vistas.

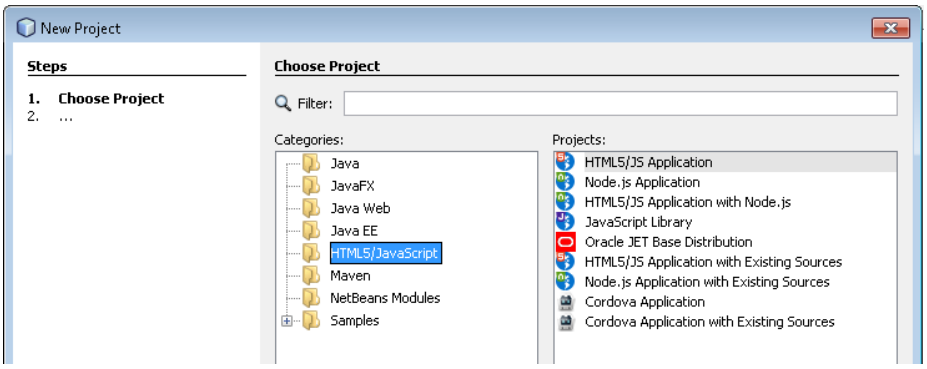
Si se buscar documentación acerca de ambas, será fácil encontrar otras alternativas también interesantes (se deja para el alumno). Es un "mercado" en continua y rápida evolución. Aquí, nos vamos a centrar sólo en Angular.

3. Instalación y Desarrollo

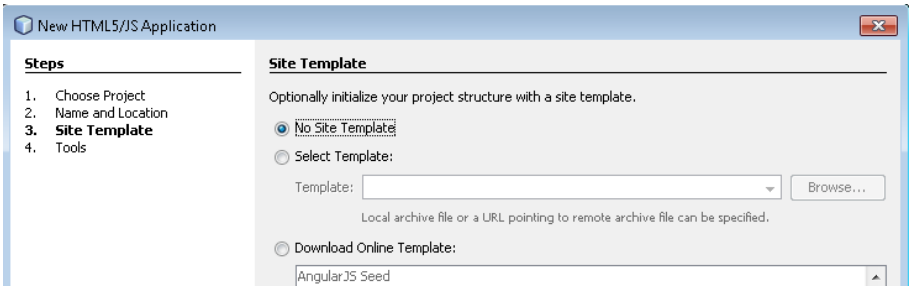
Como con cualquier biblioteca Javascript, debemos incluirla en nuestro html. Podemos acceder a ella descargándola en local e incluir su ubicación en la cabecera del html o podemos hacerlo mediante un enlace a una ubicación web a la que el navegador pueda acceder. Lo que se prefiera.

En cuanto al entorno de desarrollo, hay múltiples opciones. Para los ejemplos simples se pueden crear los ficheros HTML con un editor y ejecutarlos directamente en un navegador. Para aplicaciones más complejas, lo más sencillo es usar Netbeans, ya que lo conocemos. Además, nos va a permitir ejecutar nuestro proyecto, y conectar con el navegador en modo depuración, visualizando peticiones y respuestas. Aunque se puede usar cualquier otro entorno de desarrollo, en lo siguiente nos vamos a centrar en Netbeans.

Debido a la simplicidad de nuestro proyecto, la mejor opción es crear un proyecto de tipo HTML5/Javascript->HTML5/JS Application:



Luego crearle sin ningún tipo de plantilla, ni herramientas adicionales, ya que dada la simplicidad del proyecto a abordar, esto sólo va a complicar innecesariamente el proyecto. Ya iremos creando los ficheros que necesitemos. Con el index.html inicial, nos vale:



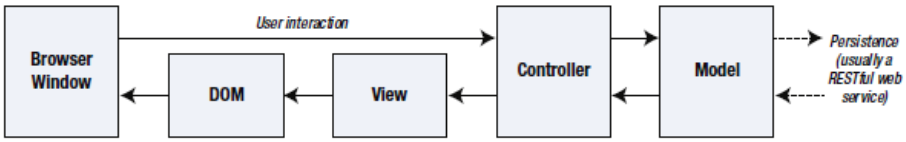
A la hora de probar nuestra aplicación desde Netbeans, vamos a tener que añadir dos complementos al navegador (en las pruebas con Chrome se han tenido que añadir):

- Extensión para Netbeans. De esta forma nuestro IDE se comunica con el navegador para ejecutar el proyecto, al tiempo que permite depurar la ejecución. Esto último puede ser útil para analizar posibles errores, aunque lo mismo lo podemos hacer desde la consola para desarrolladores del navegador.

- Extensión "Allow-Control-Allow-Origin". Al intentar accede al servicio REST y/o cargar otras páginas del disco local, aparece un problema de seguridad debido al Intercambio de Recursos de Origen Cruzado (CORS): el nuevo recurso no está en el mismo dominio que la página que lo invoca. La política de mismo origen impide leer el recurso remoto, por motivos de seguridad, bloqueando el acceso. La extensión que se indica permite desactivar esta opción de seguridad.

4. Elementos Básicos de una Aplicación AngularJs

Las aplicaciones AngularJs siguen el patrón MVC, implementado en el lado del cliente según el esquema:



Como se ve, la persistencia residirá en un servidor externo, al que se accede normalmente vía servicios web de tipo REST, como será en nuestro caso. Según se indica en la bibliografía, hay dos tipos de modelos:

- **Modelos de Vista**. Representan o modelan los datos que son intercambiados entre el controlador y las vistas. Estos serán implementados en la aplicación AngularJs
- **Modelos de Dominio**. Contienen los datos del modelo de negocio, así como las operaciones de transformación y las reglas para realizar las operaciones CRUD. Estos residen en el servidor.

Pasamos a ver los elementos fundamentales y las directivas más importantes en una aplicación AngularJs:

- Las aplicaciones AngularJs son modulares. El **módulo** es el elemento de más alto nivel para estructurar nuestras aplicaciones. Sus funciones básicas son: ayudar a organizar el código, asociar AngularJs (código javascript) con secciones del HTML, actuar de pasarela a las funcionalidades del entorno angular (el resto de elementos que veremos se asocian a un determinado módulo) y empaquetar funcionalidades reutilizables (de manera similar a un componente).

  Toda aplicación Angular contendrá al menos un módulo. Son creados mediante el método *angular.module("nombre_modulo", [ ])*. El segundo argumento es un array, en principio vacío, y que contendrá, si existen, otros módulos y servicios creados en nuestra aplicación. Una vez creado hay que asociarle en el HTML a la zona donde queramos que opere, mediante la directiva *ng-app*. El método *angular.module()* retorna un objeto *Module* que provee el acceso a las herramientas o propiedades principales de AngularJs. A continuación veremos las más básicas

- **Controladores**. Son una de las piezas básicas dentro de las aplicaciones AngularJs. Se definen mediante el método *Module.controller(nombre, constructor)*. En el constructor se inyectaran recursos y servicios mediante Dependency Injection (DI). Esto permite, como se comentó con JEE, acceder a recursos de la aplicación y del entorno mediante su nombre, sin tener que conocer su ubicación. La directiva *ng-controller* nos permitirá, en el HTML, asociar controladores a vistas (zonas del HTML).

  Son el enlace entre el modelo y las vistas. Proveen de datos y servicios a éstas, y definen la lógica de negocia para trasladar acciones del usuario al modelo. Se puede usar un controlador único, aunque se aconseja crear un controlador por vista importante en aplicaciones complejas.

Veamos lo explicado con un ejemplo simple (a continuación), pero usando dos vistas, un controlador por vista y varias opciones adicionales que explicaremos más adelante. Aquí, como es la primera aplicación, vamos a poner todo el contenido en un único fichero. Más adelante se verá cómo estructurar una aplicación de manera más adecuada. Se resalta en negrita todas las partes relacionadas con AngularJs, tanto en el script como en el HTML.

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
  <title>Hola Mundo App</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/angular.min.js"></script>
  <script>
    // Modelo
    var model = {
      ofertaDia: "manzanas a 1 Euro",
      frutas: [{ fruta: "Manzana", precio: 1 },
                  { fruta: "Naranja", precio: 1.5 },
                  { fruta: "Platanos", precio: 2.4 },
                  { fruta: "Papaya", precio: 4.6 }]
      };
var todoApp = angular.module("todoApp", []);  // Creación del módulo
todoApp.controller("productos", function ($scope) {  // Primer controlador
  $scope.productos = model.frutas;
  });
todoApp.controller("oferta", function () {   // Segundo controlador (con y sin $scope)
  //$scope.resaltar = true;
  //$scope.oferta = model.ofertaDia;
  this.resaltar = false;
  this.oferta = model.ofertaDia;
  });
</script>
</head>
<body>

  <div class="productos" ng-controller="productos"> <!-- Vista 1 -->
    <table class="table table-striped">
    <thead>
      <tr>
        <th>Producto</th>
        <th>Precio</th>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="item in productos">
        <td>{{item.fruta}}</td>
        <td>{{item.precio}} &euro;</td>
      </tr>
    </tbody>
    </table>
  </div>

  <div class="oferta" ng-controller="oferta as Vw2Ctrl"> <!-- Vista 2 -->
    <h1 data-ng-attr-style="{{Vw2Ctrl.resaltar ? 'color:red' : 'color:black'}}">
       Oferta del dia </h1>
    <p>{{Vw2Ctrl.oferta}}</p>
  </div>
</body>
</html>
```

Análisis del código:

- Aunque no es necesario, por la simplicidad del ejemplo, se muestra cómo definir y usar varios controladores por cuestiones formativas. Cada uno afecta a una parte del HTML. En caso de haber un único controlador, como será el caso del ejemplo REST de más adelante, normalmente éste se añade a la etiqueta <body> para que su efecto se extienda a toda la página.
- ***$scope***. Objeto global que hace accesible al controlador y a las vistas propiedades (variables) y métodos. Es la forma más sencilla de compartir el modelo entre el controlador y las vistas. Es pasado como argumento al constructor (función), inyectando, así, el recurso en el controlador.
- Controlador sin *$scope*. Como recurso global, determinadas guías de estilo AngularJs aconsejan evitar su uso siempre que se pueda. Se reduce, también, el uso de recursos. Por esta razón, en el segundo controlador se ha puesto un ejemplo de cómo intercambiar propiedades entre controlador y vistas sin usar ese objeto. Se deja comentadas, las líneas a incluir si que quiere usar (recordar que en este caso hay que poner *$scope* como argumento a *function()*). Mirar el código tanto en el script, como la modificación a realizar en la directiva *ng-controller*. Es más sencillo usar $scope, por esa razón en lo siguiente vamos a usar siempre esa opción.
- En el ejemplo se han usado otras directivas relacionadas con AngularJs dentro del HTML. Es especialmente interesante la *ng-repeat*, ya que nos permite crear un bucle dentro de las vistas.
- Para acceder desde las vistas a los valores de las propiedades se usa *{{propiedad}}*. Esto es sustituido por el valor de la variable correspondiente.

Antes de continuar, vamos a ver un segundo ejemplo añadiendo interacción con el usuario. Se muestra, además, como compartir métodos (funciones) entre controlador y vistas usando $scope. En este caso usamos sólo esta opción. Se añade, también, otra forma de modificar dinámicamente el estilo de elementos del DOM. Se resalta, como antes, en negrita todo lo relacionado con AngularJs

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
<title>Hola Mundo App</title>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/angular.min.js"></script>
<script>
var model = {
  ofertaDia: "manzanas a 1 Euro",
  resaltar: true,
  frutas: [{ fruta: "Manzana", precio: 1 },
           { fruta: "Naranja", precio: 1.5 },
           { fruta: "Platanos", precio: 2.4 },
           { fruta: "Papaya", precio: 4.6 }]
  };

var todoApp = angular.module("todoApp", []);

todoApp.controller("productos", function ($scope) {
  $scope.productos = model.frutas;
  $scope.addNuevaFruta = function(elem1,elem2) {
    $scope.productos.push({fruta: elem1, precio: elem2});
    }
  });

todoApp.controller("oferta", function ($scope) {
  $scope.oferta = model.ofertaDia;
    $scope.resaltar= function() {
      return model.resaltar ? {"color": "red"} : {"color" : "black"};
      //return {"color": "red"};
    }
```

```
  });
</script>
</head>

<body>
  <div class="productos" ng-controller="productos">
    <table class="table table-striped">
    <thead>
      <tr>
        <th>Producto</th>
        <th>Precio</th>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="item in productos">
        <td>{{item.fruta}}</td>
        <td>{{item.precio}} &euro;</td>
      </tr>
    </tbody>
    </table>
  Nueva Fruta: <input type="text" name="fruta" ng-model="fruta"><br>
  Precio: <input type="text" name="precio" ng-model="precio"><br>
  <button ng-click="addNuevaFruta(fruta,precio)">Add</button>
  </div>

  <div class="oferta" ng-controller="oferta">
    <h1 ng-style="resaltar()"> Oferta del dia </h1>
    <p>{{oferta}}</p>
  </div>
</body>
</html>
```

En este código, además de lo indicado, resaltar las directivas *ng-model* que nos permite crear propiedades que son dinámicamente añadidos al entorno, scope, del controlador y la *ng-click* que permite ejecutar una función al hacer clic sobre un botón.

Aunque para la aplicación que vamos a construir vamos a usar, básicamente, los elementos AngularJs vistos, vamos a continuar describiendo algunos básicos más.

- **Directivas**. Son un elemento muy potente de AngularJs, ya que permiten extender y mejorar el HTML para crear aplicaciones web muy potentes. Hemos visto algunas de ellas en los ejemplos anteriores (ej., ng-app, ng-controller, ng-repeat, etc). Esas y muchas más son incluidas en la biblioteca estándar. AngularJs permite extenderlas creando directivas personalizadas con el método *Module.directive()*. Su estudio cae fuera del alcance de la asignatura y se deja para el alumno.

- **Filtros**. Se usan en las vistas para dar formato a los datos mostrados al usuario. Igual que con el elemento anterior, existen filtros incluidos con la biblioteca y se pueden crear nuevos con el método *Module.fliter()*. Como en el caso anterior, se deja para el alumno profundizar en esta parte

- **Servicios**. Son objetos *singleton* (instanciados una vez por aplicación) que proporcionan funcionalidades que se quieren usar durante toda la aplicación. Incluyen aquellas funcionalidades que no encajan en ningún elemento del patrón MVC (ej., logging, seguridad, etc). Dentro de los servicios incluidos en la biblioteca estándar nos va a interesar, especialmente, el relacionado con la generación de peticiones (request) HTTP. Será el servicio *$http*, que se apoya en Ajax y que usaremos para la creación del cliente RESTful. Se pueden crear nuevos servicios con el método *Module.service()*. Otro método que nos permiten crear servicios es *Module.value()*, sirve para crear valores y objetos constantes en la aplicación. Es similar a *Module.constant()*, con la diferencia de que *Module.value()* no está disponible durante la fase de configuración y *Module.constant()* coge el primer valor que se le asigne, mientras *Module.value()* si lo volvemos a asignar, la propiedad cambia de valor.

Los servicios son inyectados en el controlador, incluyéndolos como argumentos al constructor. Otros elementos como, por ejemplo, $scope son pasados al controlador de la misma forma, como hemos visto en ejemplos anteriores. Veremos cómo se hace con el servicio $http. En el siguiente ejemplo vemos, de manera simplificada, como inyectar un servicio de tipo *Module.value()* (de igual manera de define e inyecta el *Module.constant()*):

```
...
<script>
  var myApp = angular.module("exampleApp", []);
  . . .
  var now = new Date();
  myApp.value("nowValue", now);
  . . .
  myApp.controller("daysCtrl", function ($scope, nowValue) {  // Inyección
    $scope.today = nowValue.getDay();
    . . .
    });
</script>

...
```

5. Estructura de una Aplicación AngularJs

Para aplicaciones sencillas, como los ejemplos anteriores, se puede poner todo en un único fichero HTML. Sin embargo, esto no es conveniente y más, como se ha comentado, cuando AngularJs está pensado para aplicaciones complejas, con muchos elementos tales como módulos, controladores, directivas, filtros, etc. Una aplicación real, por lo tanto, estará estructurada en múltiples ficheros y directorios.

No hay una forma única de organizar esos ficheros y directorios. Existen plantillas (Netbeans crea aplicaciones AngularJs con alguna de ellas) y documentos de "buenas prácticas" o de guías de estilo (ej., https://www.sitepoint.com/introduction-angularjs-style-guides/ o https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md). La idea, al final, es organizar nuestra aplicación de manera que tenga sentido.

Una aproximación muy común es incluir todos los elementos de un determinado tipo juntos (ej., controladores en un fichero, filtros en otro, etc.) y poner todos los elementos relacionados con una determinada parte de la aplicación juntos, es decir, en distintos ficheros y directorios. Esta estructura al final nos la va a dirigir la organización en módulos que hayamos hecho en nuestra aplicación. Recordemos que el módulo es el elemento básico que nos permite estructurar nuestro proyecto AngularJs. Un enfoque habitual es estructurar nuestros módulos (y elementos asociados) de acuerdo a las principales vistas que tenga nuestra aplicación:

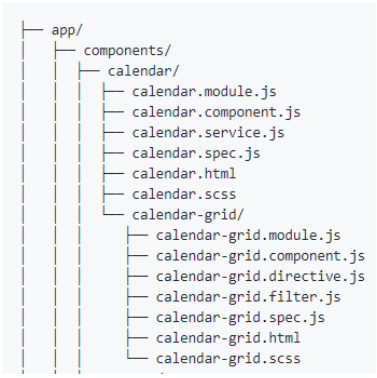


Figura 1. Ejemplo de estructura en ficheros de https://github.com/toddmotto/angularjs-styleguide/blob/master/README.md

Es habitual, también, organizar la vista en distintos ficheros HTML. En el ejemplo RESTful emplearemos esta opción y veremos la directiva AngularJs relacionada.

La forma más simple de organizar la aplicación en módulos es creando uno único. Tiene la ventaja de que es fácil el intercambio y comunicación entre los distintos elementos de la aplicación. Esta opción puede ser válida para aplicaciones sencillas, como será nuestro caso. Sin embargo, aplicaciones complejas se deben estructurar en distintos módulos. Esto permite, además, reutilizar componentes. En este caso, suele existir una dependencia entre módulos, de manera que unos módulos usan funcionalidades de otros (como en el caso de los componentes vistos en la parte anterior). Esto se resuelve usando el segundo argumento de *angular.module()*. Ej.:

Se crea un servicio relacionado con el control de acceso y lo añadimos al módulo "cutomServices" que se incluye en el fichero services.js

```
angular.module("customServices", [])
    .factory("logService", function () {
        var messageCount = 0;
        return {
            log: function (msg) {
                console.log("(LOG + " + messageCount++ + ") " + msg);
            }
        };
    });
```

Añadimos este módulo (componente) al módulo principal de nuestra aplicación, inyectando, además, el nuevo servicio en el controlador. Este código estaría en el fichero indexModule.js (no importa que el código no tenga sentido, ya que faltan partes, lo importante es ver cómo se soluciona la dependencia entre módulos y la inyección de recursos, resaltando esto en negrita):

```
angular.module("exampleApp", ["customServices"])  // Inclusión del modulo anterior
  .controller("defaultCtrl", function ($scope, logService) { // Inyección servicio
      $scope.data = {
          cities: ["London", "New York", "Paris"],
           totalClicks: 0
       };

       $scope.$watch('data.totalClicks', function (newVal) {
           logService.log("Total click count: " + newVal);
       });
```

Para acabar, recordar que los ficheros HTML deberán incluir en su cabecera todos los ficheros javascript que utilice. Esta inclusión debe extenderse a los elementos relacionados, es decir, en el ejemplo anterior, si se incluye el fichero *indexModule.js*, se debe incluir también el *services,js*, ya que están relacionados.

6. Accediendo a Servicios Web RESTful - Ejemplo Básico

Para el acceso a servicios web de tipo REST, aquí vamos a usar el servicio Ajax *$http* que viene incluido en la biblioteca estándar de AngularJs. Se puede decir que $http es la opción de más bajo nivel que proporciona AngularJs. Existen bibliotecas adicionales de más alto nivel (son capas por encima de $http) como *ngResource* para operaciones con servicios RESTful, pero no las vamos a usar aquí. La biblioteca estándar es sencilla de usar y suficiente para lo que vamos a hacer.

La mejor forma de ver cómo crear un cliente, es mediante un ejemplo y eso es lo que vamos a hacer. Vamos a crear un ejemplo sencillo, con las operaciones básicas, operando, también, de la manera más básica posible. Posteriormente añadiremos cómo mejorar el acceso a los servicios mediante la gestión de errores, acceso a la respuesta y personalización de *response* y *request*. Esto lo dejaremos indicado, para que cada alumno profundice en esas partes por su cuenta. Vamos a dividir el ejemplo en pasos, ya que para poder probar el cliente hay que crear un servicio y también lo vamos a hacer.

1.- **Creando la base de datos**. Vamos a crear una sola tabla. Vamos a usar un ejemplo de una tienda de vinos. Las sentencias sql para crear la tabla son (se ha cambiado el nombre de la tabla, para que si se tiene una base de datos con la tabla “vino” anterior, se pueda añadir ésta a esa misma base de datos, así ya tenemos el recurso en el servidor):

```
CREATE TABLE "VINO_PARA_ANGULAR" (
"ID" INTEGER NOT NULL primary key
        GENERATED ALWAYS AS IDENTITY
        (START WITH 1, INCREMENT BY 1),
"NOMBRECOMERCIAL" VARCHAR(50) DEFAULT NULL,
"COMENTARIO" VARCHAR(200) DEFAULT NULL,
"BODEGA" VARCHAR(50) DEFAULT NULL,
"DENOMINACION" VARCHAR(50) DEFAULT NULL,
"CATEGORIA" VARCHAR(50) DEFAULT NULL
);

INSERT INTO VINO_PARA_ANGULAR (NOMBRECOMERCIAL,COMENTARIO,BODEGA,DENOMINACION,CATEGORIA)
VALUES
    ('Nombre1','buena relacion calidad-precio','bodega1','rivera','joven'),
    ('Nombre2','un poco caro, pero buena calidad','bodega2','rivera','joven'),
    ('Nombre3','comentario a nombre 3','bodega2','rioja','crianza'),
    ('Nombre4','comentario a nombre4','bodega3','rueda','reserva'),
    ('Nombre5','comentario a nombre5','bodega4','cigales','joven');
```

2.- **Creando el servicio web RESTful**. Lo vamos a crear usando JEE y como servidor Glassfish. De esta manera podemos probar todas las operaciones y ver cómo mezclamos tecnologías sin problema.

Crear una nueva aplicación web. En el ejemplo la hemos llamado servicioVino2, pero se puede poner el nombre que se quiera. Lo único que habrá que hacer es cambiar el nombre correspondiente en la ruta al servicio en el cliente. Crear el paquete *Dominio* y añadir la Entity *VinoParaAngular*. Crear el paquete *Persistencia* y añadir el DAO (EJB *VinoParaAngularFacade*) para la gestión de la Entity anterior. Ahora crear un nuevo paquete para añadir la clase que implementará el servicio web (en el ejemplo le hemos llamado *REST*). Crear esta clase como se indicó en los tutoriales anteriores y añadir el siguiente código (no se incluyen los “import”, ya que se ha usado clases de la biblioteca estándar de Java, por lo tanto son sencillos de incluir; se incluye la llamada al EJB de la capa de persistencia, como ejemplo, habrá que borrarlo e incluir la llamada al creado por cada alumno):

```
@Path("paraAngular")
public class restfulWsParaAngular {

    VinoParaAngularFacadeLocal vinoFacade = lookupVinoParaAngularFacadeLocal();

    public restfulWsParaAngular() {
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public VinoParaAngular[] getVinos() {
        List<VinoParaAngular> vinos = vinoFacade.findAll();
        VinoParaAngular[] arrayVino = new VinoParaAngular[vinos.size()];
        for (int i = 0; i < arrayVino.length; i++) {
            arrayVino[i] = vinos.get(i);
        }
        return arrayVino;
    }
```

```
    @Path("{nombre}")
    @DELETE
    public void borraVino(@PathParam("nombre") String nombre) {
        vinoFacade.borraPorNombre(nombre);
    }

    //   En este ejemplo se recibe un Json desde el cliente y se accede a el
    // mediante la clase Java JsonObject. Luego lo convertimos a una Entity “a mano”

    //    Se deja comentado el código para hacer esa conversión mediante ObjectMapper.
    // No es una biblioteca estándar, por lo tanto habría que incluirla en el proyecto.
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public void anadeVinoJson(JsonObject vino) {
        //System.out.println("Objeto Json recibido: " + vino.toString());
        VinoParaAngular newVino = new VinoParaAngular();
        newVino.setNombrecomercial(vino.getString("nombrecomercial"));
        newVino.setComentario(vino.getString("comentario"));
        newVino.setCategoria(vino.getString("categoria"));
        vinoFacade.create(newVino);
//        String vinoJsonStr = vino.toString();
//        ObjectMapper mapper = new ObjectMapper();
//        VinoParaAngular vino = mapper.readValue(vinoJsonStr, VinoParaAngular.class);
//        vinoFacade.create(newVino);
    }

    @Path("{nombre}/{comentario}")
    @PUT
    public      void      cambiaComentario(@PathParam("nombre")      String      nombre,
@PathParam("comentario") String comentario) {
        vinoFacade.cambiaNombre(nombre, comentario);
    }

    private VinoParaAngularFacadeLocal lookupVinoParaAngularFacadeLocal() {
        try {
            javax.naming.Context c = new InitialContext();
            return                                          (VinoParaAngularFacadeLocal)
c.lookup("java:global/ServicioVino2/VinoParaAngularFacade!EJBs.VinoParaAngularFacadeLocal"
);
        } catch (NamingException ne) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE,  "exception  caught",
ne);
            throw new RuntimeException(ne);
        }
    }
}
```

Antes de nada compilar y desplegar y ver que funciona. Para ello basta con poner la URL asociada a la llamada GET y ver que en el navegador nos aparece la lista de vinos.

2.- **Creando el cliente**. En Netbeans crear un nuevo proyecto HTML5/JavaScript->HTML5/JS Application. Recordar que no hay que añadir ninguna plantilla ni ninguna herramienta.

Aunque es simple, vamos a organizar la aplicación en distintos ficheros .js y .html. Veamos los ficheros a crear y su contenido

2.1 Fichero *index.html* (la inclusión de la biblioteca bootstrap es opcional) (destacar el uso de las directivas *ng-include* y *ng.show* para gestionar las vistas incluidas en ficheros HTML aparte):

```
<!DOCTYPE html>
<html ng-app="mainMod">
    <head>
        <title>Ejemplo Angular</title>
        <!-- biblioteca angular -->
        <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/angular.min.js"></script>
        <!-- Bootstrap -->
        <link                                                    rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
        <!-- jQuery library -->
        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
        <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
        <!-- Nuestro fichero Angular -->
        <script src="indexModule.js"></script>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body ng-controller="restFulCtrl">
       <div class="panel panel-primary">
         <h3 class="panel-heading">Products</h3>
         <ng-include src="'tableView.html'" ng-show="displayMode == 'list'"></ng-include>
         <ng-include src="'editorView.html'" ng-show="displayMode == 'edit'"></ng-include>
       </div>
    </body>
</html>
```

2.2 Fichero *tableView.html*:

```
<!DOCTYPE html>
<html>
  <head>
    <title>table View</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div class="panel-body">
      <table class="table table-striped table-bordered">
        <thead>
          <tr>
            <th>categoria</th>
            <th>comentario</th>
            <th class="text-right">nombrecomercial</th>
            <th></th>
          </tr>
        </thead>
          <tbody>
          <tr ng-repeat="item in products">
            <td>{{item.categoria}}</td>
            <td>{{item.comentario}}</td>
```

```
            <td>{{item.nombrecomercial}}</td>
            <td class="text-center">
            <button class="btn btn-xs btn-primary"
              ng-click="deleteProduct(item)">Delete</button>
            <button class="btn btn-xs btn-primary"
              ng-click="editOrCreateProduct(item)">Edit</button>
            </td>
          </tr>
        </tbody>
      </table>
    <div>
      <button class="btn btn-primary" ng-click="listProducts()">Refresh</button>
      <button class="btn btn-primary" ng-click="editOrCreateProduct()">New</button>
    </div>
  </div>
</body>
</html>
```

2.3 Fichero *editorView.html*:

```
<!DOCTYPE html>
<!--
Por simplicidad, solo damos valor a 3 campos del nuevo vino.
Añadir mas es inmediato.
-->
<html>
<head>
  <title>editor View</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
  <div class="panel-body">
    <div class="form-group">
      <label>Nombre Comercial:</label>
      <input class="form-control" ng-model="currentProduct.nombrecomercial" />
    </div>
    <div class="form-group">
      <label>Categoria:</label>
      <input class="form-control" ng-model="currentProduct.categoria" />
    </div>
    <div class="form-group">
      <label>Comentario:</label>
      <input class="form-control" ng-model="currentProduct.comentario" />
    </div>
    <button class="btn btn-primary" ng-click="saveEdit(currentProduct)">Save</button>
    <button class="btn btn-primary" ng-click="cancelEdit()">Cancel</button>
  </div>
</body>
</html>
```

2.4 Fichero indexModule.js (se dejan comentados algunos comandos para seguir la traza…). Usamos una copia de los datos del servidor, en un modelo local.

```
angular.module("mainMod", [])
  .constant("baseUrl", "http://localhost:8080/ServicioVino2/webresources/paraAngular")
  .controller("restFulCtrl", function ($scope, $http, baseUrl) { // Inyectamos recursos

    $scope.displayMode = "list"; // Variable que controla la vista
    $scope.listProducts = function () {  // Consume operacion GET en SW
      $http.get(baseUrl).success(function (data) { //
      //console.log(data);
      $scope.products = data; // Modelo local de la aplicación. Mediante $scope es
                              //pasado del controlador a las vistas
      });
    }

    $scope.deleteProduct = function (product) { // Ejecuta operacion DELETE en SW
      $http({
        method: "DELETE",
        url: baseUrl + "/" + product.nombrecomercial
      }).success(function () {
        $scope.products.splice($scope.products.indexOf(product), 1);
      });
    }

    $scope.createProduct = function (product) { // Ejecuta operacion POST en SW
      $http.post(baseUrl, product).success(function () {
        $scope.products.push(product); // Se añade al modelo local
        $scope.displayMode = "list"; // Csambiamos vista
      });
    }

    $scope.updateProduct = function (product) {  // Ejecuta operacion PUT en SW
      $http({
        url: baseUrl + "/" + product.nombrecomercial + "/" +
          product.comentario,
        method: "PUT"
      }).success(function () {
      for (var i = 0; i < $scope.products.length; i++) {  // actualizamos modelo local
        if ($scope.products[i].id == product.id) {
          $scope.products[i] = product;
          break;
        }
      }
      $scope.displayMode = "list";
      });
    }

    $scope.editOrCreateProduct = function (product) {
      $scope.currentProduct = product ? angular.copy(product) : {};
      //console.log("Producto: " + $scope.currentProduct);
      $scope.displayMode = "edit";
      }

    $scope.saveEdit = function (product) {
      if (angular.isDefined(product.id)) {
```

```
        $scope.updateProduct(product);
      } else {
        $scope.createProduct(product);
      }
    }

    $scope.cancelEdit = function () {
      $scope.currentProduct = {};
      $scope.displayMode = "list";
      }

    $scope.listProducts();
  });
```

Una vez creada, ejecutarla mediante Netbeans. Si no se han instalado los complementos indicados anteriormente dará errores durante la ejecución.

7. Accediendo a Servicios Web RESTful - Conceptos Avanzados

En este apartado se van a añadir elementos que permiten una gestión más completa del cliente, en cuanto al acceso a un servicio RESTful. Al igual que se hizo con Java, vamos a ver cómo saber el resultado de la petición y actuar en consecuencia, cómo acceder a la respuesta y cómo configurar tanto la petición como la respuesta. No se va a entrar en detalles, dejando para el alumno un estudio más profundo de esta parte.

- **Accediendo a la respuesta de Ajax (de $http)**. El servicio $http retorna un objeto de tipo *promise*. Una promesa es un patrón de operación que permite ejecutar peticiones a un servidor (*request* en nuestro caso) y que se procesen de manera asíncrona, es decir, de modo que la petición no bloquee al cliente en lo que se espera la respuesta. La petición se queda en segundo plano esperando, mientras el cliente puede sigue operando. Cuando el resultado de la petición llega, entonces se continúa con la operación asociada a la petición. El objeto *promise* retornado por $http tiene 3 métodos:
  - success(fn): invocado cuando la petición ha sido completada con éxito, ejecutando la función asociada. A esta función se le pasan los datos enviados por el servidor (cuerpo de la respuesta). Es el método que hemos usado únicamente en el ejemplo anterior.
  - error(fn): invocado cuando la petición no ha sido completada con éxito, ejecutando la función asociada. A esta función se le pasa el string que describe el problema ocurrido.
  - then(fn,fn): registra una función asociada a operación correcta y otra asociada a operación incorrecta. **A estas funciones se les pasa la respuesta del servidor**, por lo tanto, nos permite acceder a toda la información contenida en *Response*: valor de status, y cabeceras, así como contenido del cuerpo. En el siguiente punto veremos cómo acceder a esta información
- **Accediendo a la respuesta del servidor (*Response*)**. Hay que usar el método *then* del objeto *promise* devuelto por *$http*. El objeto que es pasado a las funciones de ese método tiene las siguientes propiedades:
  - data: retorna el cuerpo de la respuesta
  - status: contiene el valor del código del status HTTP devuelto por el servidor
  - headers: retorna una función que se usa para acceder a las cabeceras de la respuesta
  - config: retorna la configuración usada para realizar la petición. Veremos un poco más adelante que se puede configurar la petición (request) realizada.

Un ejemplo de uso sería (se muestra, además, otra forma de especificar la operación GET en $http) (se usa un servicio REST externo, ir a https://httpbin.org/ para más información) (el ejemplo se puede ejecutar directamente en el navegador, pero deberá tener el complemento "Allow-Control-Allow-Origin"):

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.3/angular.min.js"></script>
<script>
  var todoApp = angular.module("todoApp", []);
  todoApp.controller("prueba", function ($scope,$http) {
    $http({
        method : "GET",
        //url : https://httpbin.org/get/error  // Error en peticion
        url : "https://httpbin.org/get"    // Peticion correcta. Devuelve Json
    }).then(function (response) {
        //console.log("exito: " + response.status);
        $scope.respuesta = response.data;
    }, function (response) {
        //console.log("error: " + response.status);
        $scope.respuesta = response.statusText;
    });
  });
</script>
</head>
<body ng-controller="prueba">
  <p>Mensaje http: {{respuesta}} </p>
</body>
</html>
```

- **Configurando la petición (request) y la respuesta (response)**. Todos los métodos de $http admiten un argumento opcional, que será un objeto conteniendo ajustes de configuración. En principio, la configuración por defecto de Ajax para las operaciones HTTP funciona perfectamente (por cierto, AngularJs también permite cambiar los valores de esta configuración por defecto). Esta opción se emplea en casos especiales. Permite, entre otras cosas, fijar datos en el cuerpo de la petición, fijar cabeceras personalizadas, fijar el método HTTP, fijar los parámetros de la URL y transformar tanto la petición (transformRequest) como la respuesta (trasnformResponse). Se deja para el alumno profundizar en el tema.

8. Bibliografía

- Freeman, Adam, "Pro AngularJs", Ed. Apress, 2014