# Spring

Sesión 4

Aplicaciones AJAX y REST

# Puntos a tratar

- AJAX con Spring MVC
  - Enviar fragmentos de HTML/objetos al cliente
  - Recibir objetos del cliente
- Servicios web REST
  - Repaso de la filosofía REST
  - Obtener recursos (GET)
  - Crear o modificar recursos (POST/PUT)
  - Eliminar recursos (DELETE)
  - Parte del cliente
- Tratamiento de errores en AJAX y REST

# Serialización de objetos en la petición/respuesta HTTP

- Ya hemos visto que la conversión de parámetros HTTP a objeto Java es automática
- Los encargados de serializar/deserializar de HTTP en Spring son los `HttpMessageConverter`
  - Por defecto el de XML ya está implementado en Spring, solo sería necesario anotar el objeto con JAXB
  - Simplemente incluyendo en el *classpath* la librería *open source* Jackson (http://jackson.codehaus.org), podemos convertir de/a JSON
- `@RequestBody` anotando un parámetro de un método indica que queremos deserializar el cuerpo de la petición HTTP y convertirlo al tipo del parámetro
- `@ResponseBody` anotando el valor de retorno de un método indica que queremos serializar el valor en el cuerpo de la respuesta HTTP
  -

# AJAX: enviar fragmentos de HTML al cliente

```
@RequestMapping("/loginDisponible.do")
    public @ResponseBody String loginDisponible(@RequestParam("login")
                                                String login) {
        if (ubo.getUsuario(login)==null)
            return "login disponible";
        else
            return "login <strong>no</strong> disponible";
    }
```

```
<form id="registro" action="#">
    Login: <input type="text" name="login" id="campo_login">
           <span id="mensaje"></span><br>
    Password: <input type="password" name="password"> <br>
    Nombre y apellidos: <input type="text" name="nombre"> <br>
    <input type="submit" value="registrar">
</form>
<script type="text/javascript">
    $('#campo_login').blur(
        function() {
            $('#mensaje').load('loginDisponible.do',
                               "login="+$('#campo_login').val())
        }
    )
</script>
```

# AJAX (II): enviar objetos al cliente

```
public @ResponseBody InfoLogin loginDisponible(@RequestParam("login") String login) {
        if (ubo.getUsuario(login)==null)
            //Si está disponible, no hacen falta sugerencias
            return new InfoLogin(true, null);
        else
            //si no lo está, generamos las sugerencias con la ayuda del IUsuarioBO
            return new InfoLogin(false, ubo.generarSugerencias(login));
}
```

- Para enviar JSON, simplemente incluir la dependencia de Jackson
- Para enviar XML, anotar la clase `InfoLogin` con JAXB

```
@XmlRootElement
public class InfoLogin {
    private boolean disponible;
    private List<String> sugerencias;

    @XmlElement
    public boolean isDisponible() {
        return disponible;
    }
    ...
}
```

# AJAX (III): Recibir objetos del cliente

```
@RequestMapping("/altaUsuario.do")
public void altaUsuario(@RequestBody Usuario usuario) {
    ...
}
```

```
<form id="registro" action="#">
    Login: <input type="text" name="login" id="login"> <span id="mensaje"></span><br>
    Password: <input type="password" name="password" id="password"> <br>
    Nombre y apellidos: <input type="text" name="nombre" id="nombre"> <br>
    <input type="submit" value="registrar">
</form>
<script type="text/javascript">
    $('#registro').submit(function(evento) {
        $.ajax({
            url: 'altaUsuario.do',
            type: 'POST',
            data: JSON.stringify({login: $('#login').val(),
                password: $('#password').val(), nombre: $('#nombre').val()}),
            processData: false,
            contentType: "application/json"
        })
        evento.preventDefault();
    });
</script>
```

# Servicios Web REST: repaso

- Cada URI identifica un recurso (entidad, objeto)
  - Se suelen organizar de manera jerárquica

```
/hoteles/excelsiorMad/ofertas/15
/hoteles/ambassador03/ofertas/15
/hoteles/ambassador03/ofertas/    (todas las del hotel)
```

- El método HTTP indica qué operación queremos realizar sobre el recurso

```
GET /hoteles/excelsiorMad/ofertas/15    ver la oferta
GET /hoteles/excelsiorMad/ofertas/      ver todas las ofertas del hotel
POST /hoteles/excelsiorMad/ofertas/15   crear la oferta
PUT /hoteles/excelsiorMad/ofertas/15    actualizar la oferta
DELETE /hoteles/excelsiorMad/ofertas/15 eliminar la oferta
```

- La respuesta del servidor se acompaña de un código de estado HTTP
  - 404 recurso inexistente
  - 400 petición incorrecta

# URIs y *PathVariables* (@PathParam en JAX-RS)

- El diseño de las URIs REST hace que hayan partes fijas y partes variables. Nos interesa saber el valor de esas partes variables

```
/hoteles/excelsiorMad/ofertas/15

/hoteles/{idHotel}/ofertas/{idOferta}
```

```
@Controller
public class OfertaRESTController {
    @Autowired
    IOfertasBO obo;

    @RequestMapping(value="hoteles/{idHotel}/ofertas/{idOferta}",
                    method=RequestMethod.GET)
    @ResponseBody
    public Oferta mostrar(@PathVariable String idHotel,
                          @PathVariable int idOferta) {
        Oferta oferta =  obo.getOferta(idHotel, idOferta);
        return oferta;
    }
}
```

# Obtener recursos: GET

- `ResponseEntity<Clase>` representa una respuesta HTTP en la que se serializa un objeto de la clase `Clase`. Para no serializar nada, `ResponseEntity<Void>`
- También se podría usar `@ResponseBody`, pero la convención en Spring es usar esto para REST

```
@Controller
@RequestMapping("/hoteles/{idHotel}/ofertas")
public class OfertasController {
    @Autowired
    IOfertasBO obo;


    @RequestMapping(method=RequestMethod.GET,
                    value="{idOferta}",
                    produces=MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Oferta> obtenerOferta(@PathVariable String idHotel,
                                                @PathVariable int idOferta)
                                                  throws OfertaInexistenteException {
        Oferta oferta = obo.getOferta(idHotel, idOferta);
        return new ResponseEntity<Oferta>(oferta, HttpStatus.OK);
    }
}
```

# Crear/modificar recursos (POST/PUT)

```
@RequestMapping("/hoteles/{idHotel}/ofertas")
public class OfertasController {
    @Autowired
    IOfertasBO obo;

    //Ya no se muestra el código de obtenerOferta
    //...

    @RequestMapping(method=RequestMethod.POST, value="")
    public ResponseEntity<Void> insertarOferta(@PathVariable idHotel,
                                               @RequestBody Oferta oferta,
                                               HttpServletRequest peticion) {
        int idOFerta = obo.crearOferta(idHotel, oferta);
        HttpHeaders cabeceras = new HttpHeaders();
        try {
            cabeceras.setLocation(new URI(peticion.getRequestURL()+
                                          Integer.toString(idOFerta)));
        } catch (URISyntaxException e) {
            e.printStackTrace();
        }
        return new ResponseEntity<Void>(cabeceras, HttpStatus.CREATED);
    }
}
```

# Eliminar recursos (DELETE)

```
@RequestMapping(method=RequestMethod.DELETE, value="{idOferta}")
public ResponseEntity<Void> borrarOferta(@PathVariable idHotel,
                                         @PathVariable idOferta) {
    obo.eliminarOferta(idHotel, idOferta);
    return new ResponseEntity<Void>(HttpStatus.OK);
}
```

# Clientes REST

- En principio se puede programar un cliente REST con JavaSE estándar, pero es tedioso
  - Hay librerías como Jakarta Commons HttpClient, que facilitan la tarea, además de implementar autentificación BASIC y otras características
  - Spring proporciona `RestTemplate`

```
RestTemplate template = new RestTemplate();
String uri = "http://localhost:8080/ServidorREST/hoteles/{idHotel}/ofertas/{idOferta}";
Oferta oferta = template.getForObject(uri, Oferta.class, "ambassador", 1);
System.out.println(oferta.getPrecio() + "," + oferta.getFin());
```

```
RestTemplate template = new RestTemplate();
String uri = "http://localhost:8080/TestRESTSpring3/hoteles/{idHotel}/ofertas";
//aquí le daríamos el valor deseado a los campos del objeto
Oferta oferta = new Oferta( ..., ..., ...);
URI result = template.postForLocation(uri, oferta, "ambassador");
```

# Tratamiento de errores en AJAX y REST

- Si un método de un Controller lanza una excepción, por defecto se le envía al cliente en la respuesta HTTP, lo que no es muy apropiado
- Tampoco es apropiado para un cliente AJAX o REST saltar a una página de error JSP o HTML (típico en operaciones no AJAX)
- Podemos convertir ciertas excepciones en códigos de estado HTTP con un método del controller anotado con `@ExceptionHandler`
  - De hecho, Spring ya lo hace automáticamente con ciertas excepciones, por ejemplo, al producirse un error de validación se genera una respuesta con status 400 (Bad Request)

```
@ExceptionHandler({OfertaInexistenteException.class,
HotelInexistenteException.class})
public ResponseEntity<String> gestionarNoExistentes(Exception e) {
    return new ResponseEntity<String>(e.getMessage(), HttpStatus.NOT_FOUND);
}
```

# ¿Preguntas...?