Implementación de servicios Web REST

- Integración Rest con Spring Boot
- Captura de parámetros
- Mapeo de objetos
- Devolución y recepción de objetos
- Devolución de URLs
- Códigos de repuesta y excepciones
- Clientes Rest con Spring

JAX-RS

- Es el API estándar manejo de servicios Web REST en Java
- Existen varias implementaciones:
  - Jersey (jersey.java.net) → implementación de referencia
  - Apache CXF (cxf.apache.org) → funciona con servicios Web SOAP o Rest
  - RESTeasy (www.jboss.org/resteasy)
- Spring MVC permite implementar servicios REST **pero no es una implementación de JAX-RS** (el enfoque es similar)

REST con Spring Boot

- Como vimos en tema anteriores, Spring Boot realiza la configuración de forma automática de una aplicación web con Tomcat embebido, aunque también soporta otros servidores más ligeros
- Es posible también anular el servidor embebido e instalar el servicio en un Tomcat ya instalado y configurado
- Para implementar un servicio web restful basta con añadir el artifact y declarar al menos un bean con *@RestController*

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

REST con Spring MVC

- Spring MVC está inicialmente pensado para desarrollo de aplicaciones web pero su diseño soporta de manera directa la implementación de servicios REST

Declarar Bean como controlador Rest

Path base del recurso

- Path y parámetro a capturar
- Método HTTP
- Tipo de contenido producido

```
@RestController
@RequestMapping("/clinica")
public class RecursoClinica {

    @Autowired
    Clinica clinica;

    @RequestMapping(value="/asegurados/{numSeg}", method=GET, produces="application/json")
    public Asegurado obtenerAsegurado(@PathVariable int numSeg)
    {
        Asegurado asegurado = clinica.obtenerAsegurado(numSeg);
        return asegurado;
    }

    @RequestMapping(method=GET, produces="application/json")
    public Clinica obtenerClinica() {
        return clinica;
    }
    …
```

REST con Spring MVC

- Las versiones modernas de Spring permiten usar una anotación simplificada para la captura de paths por distintos métodos (*@GetMapping*, *@PostMapping*, *@PutMapping*, etc.)

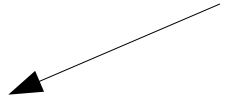
```
@RestController
@RequestMapping("/clinica")
public class RecursoClinica {

    @Autowired
    Clinica clinica;

    @GetMapping("/asegurados/{numSeg}")
    public Asegurado obtenerAsegurado(@PathVariable int numSeg)
    {
        Asegurado asegurado = clinica.obtenerAsegurado(numSeg);
        return asegurado;
    }

    @GetMapping
    public Clinica obtenerClinica() {
        return clinica;
    }
    …
```

Equivalente a:
@RequestMapping(value="/asegurados/{numSeg}", method=GET)



Captura de parámetros

- Hay dos tipos de parámetros que podemos capturar del URL:
  - **/hojasclinicas/2** → parámetro o variable de path (*@PathVariable*)
  - **/hojasclinicas?abiertas=true** → parámetro de petición (*@RequestParam*)

Parámetro de Path

Los parámetros de petición no se indican en la expresión de path

```
@GetMapping("/asegurados/{num}/hojasclinicas")
public List<HojaClinica> obtenerHojasAsegurado(
      @PathVariable("num") numAsegurado,
      @RequestParam(defaultValue="true") bool abiertas) {

    Asegurado asegurado = clinica.obtenerAsegurado(numAsegurado);

    return asegurado.obtenerHojasClinicas(abiertas);
}
…
```

Parámetro de consulta. Opciones:

- value = "<nombre>" si el nombre del parámetro de consulta no coincide con el del argumento
- required = true (por defecto) / false
- defaultValue = "<valor>". Valor por defecto si no se indica el parámetro (implica required=false)

Devolución de objetos

- Por defecto, la biblioteca de conversión desde/a JSON que incorpora Jersey (*Jackson JSON*) es capaz de convertir los POJOs a JSON directamente

```
@GetMapping("/asegurados/{numSeg}")
public Asegurado obtenerAsegurado(@PathVariable(numSeg) int numSeg)
                                  throws AseguradoInexistente {
    Asegurado asegurado = clinica.obtenerAsegurado(numSeg);
    if (asegurado == null) throw new AseguradoInexistente();
    return asegurado;
}
```

Jackson JSON es capaz de hacer la conversión directamente a JSON (por defecto) o XML

Recepción de objetos

- De igual forma, se puede recibir un objeto como un parámetro más, marcándolo con la anotación *@RequestBody*
- Jackson se encarga del mapeado de JSON (siempre que sea posible)

Para obtener un objeto enviado en la petición como JSON usar @RequestBody

```
@PostMapping("/asegurados/{numSeg}")
public void registrarAsegurado(
    @PathVariable String numSeg,
    @RequestBody Asegurado asegurado) throws AseguradoIncorrecto, AseguradoExistente {

    if (asegurado == null) {
        throw new AseguradoIncorrecto();
    }

    if (clinica.obtenerAsegurado(numSeg) != null) {
        throw new AseguradoExistente();
    }

    clinica.nuevoAsegurado(asegurado);
}
```

Devolución de Links

- Cuando queramos introducir HATEOAS, podremos generar links (clase Link) mediante las operaciones de *ControllerLinkBuilder*
- Requiere la siguiente dependencia:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

- La operación *linkTo()* junto a la operación *methodOn()* permite construir un enlace asociado al path definido para una operación junto a los parámetros indicados

Devolución de Links

La siguiente operación devuelve una lista de enlaces a las hojas clínicas de un asegurado

```
@GetMapping("/asegurados/{num}/hojasclinicas")
public List<Link> obtenerHojasAsegurado(
      @PathVariable("num") numAsegurado, @RequestParam(defaultValue="true") bool abiertas) {

    Asegurado asegurado = clinica.obtenerAsegurado(numAsegurado);
    List<HojaClinica> listaHojas = asegurado.obtenerHojasClinicas(abiertas);
    List<Link> listaLinks = new Array<Link>();

    for (HojaClinica hoja: listaHojas) {
        listaURIs.add(ControllerLinkBuilder.linkTo(
                        ControllerLinkBuilder.methodOn(RecursoClinica.class).
                                        obtenerHojaClinica(hoja.getNum()))
                        ).withSelfRel()
                    );
    }

    return listaLinks;
}
```

Códigos de respuesta

- Se puede indicar un código de respuesta por defecto mediante @ResponseStatus
- Spring permite también mapear excepciones a códigos de error

```
@ResponseStatus(HttpStatus.NOT_ACCEPTABLE)
@ExceptionHandler({AseguradoIncorrecto.class})
public void handlerParametroIncorrecto() {}

@ResponseStatus(HttpStatus.CONFLICT)
@ExceptionHandler({AseguradoExistente.class})
public void handlerRecursoExistente() {}

@PostMapping("/asegurados/{numSeg}")
@ResponseStatus(HttpStatus.CREATED)
public void registrarAsegurado(@PathVariable String numSeg,
    @RequestBody Asegurado asegurado) throws AseguradoIncorrecto, AseguradoExistente {
    if (asegurado == null) {
        throw new AseguradoIncorrecto();
    }

    if (clinica.obtenerAsegurado(numSeg) != null) {
        throw new AseguradoExistente();
    }

    clinica.nuevoAsegurado(asegurado);
}
```

Handlers para mapear excepciones en códigos HTTP

Código de respuesta por defecto

Las excepciones se lanzan directamente

Códigos de respuesta

- Una forma alternativa de devolver respuestas con múltiples códigos es usando ResourceEntity<T>
- Permite construir respuestas con un código sólo o con un objeto y un código de respuesta

```
@GetMapping("/asegurados/{numSeg}")
public ResponseEntity<Asegurado> obtenerAsegurado(@PathVariable int numSeg)
{
    Asegurado asegurado = clinica.obtenerAsegurado(numSeg);
    if (asegurado == null) {
       return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    return new ResponseEntity<>(asegurado, HttpStatus.OK);
}
```

Prueba de servicios REST

- Para probar los clientes es útil usar *curl (http://curl.haxx.se)*

```
$ curl -v -X GET http://localhost:8080/rest/clinica
* About to connect() to localhost port 8080 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 8080 (#0)
> GET /rest/clinica HTTP/1.1
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4 OpenSSL/0.9.8y zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Content-Type: application/json;charset=utf-8
< Transfer-Encoding: chunked
< Date: Sun, 24 Nov 2013 22:52:18 GMT
<
* Connection #0 to host localhost left intact
* Closing connection #0
{"cif":"268009201","nombre":"Clinica La Paz"}
$
```

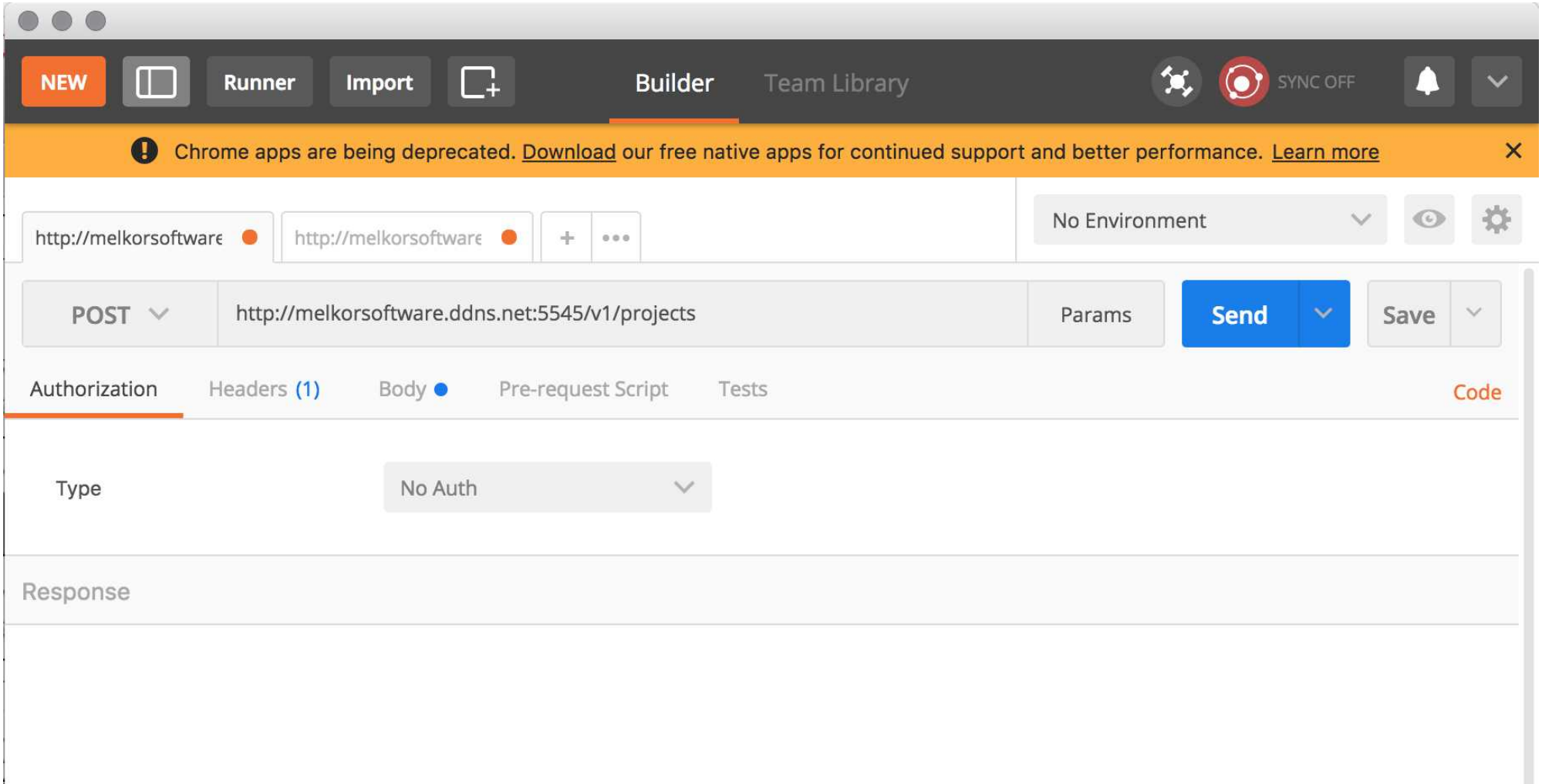
Ejecución de GET en modo "verbose"

Ejecución de PUT con envío de información

```
curl -X PUT -H "Content-Type: application/json" -d '{"numSeg": 2230, "nombre": "Miguel Pérez"}'
http://localhost:8080/rest/clinica/asegurados/2230
```

Prueba de servicios REST

•Existen también extensiones para Chrome que permiten probar un servicio REST cómodamente, como **Postman**:



Implementación de clientes

- Al ser estándar, hay decenas de bibliotecas para trabajar con un API Rest, en cualquier lenguaje de programación
- En Java destacan *Unirest* o *OkHttp*. Spring proporciona *RestTemplate* para la implementación de clientes
- RestTemplate requiere estas dependencias, que incluyen la biblioteca Jackson para el mapeo JSON de POJOs

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>
</dependencies>
```

Implementación de clientes

- RestTemplate incluye operaciones para la ejecución de get, post, put y delete, incluyendo facilidades para subir objetos u obtener un objeto como resultado
- Jackson se encarga de la conversión de los objetos desde/a JSON

```
Clinica clinica = new RestTemplate().getForObject("http://localhost:8080/clinica",
                                                  Clinica.class);

System.out.println(clinica.getNombre());
```

```
new RestTemplate().put("http://localhost:8080/clinica/asegurados/{num}",
                       asegurado, asegurado.getNumSeg());
```

Objeto a enviar

Sustitución de parámetro "num" en URL

Implementación de clientes

- Las operaciones xxxxForObject devuelven objetos tras la operación

```
Asegurado asegurado = new RestTemplate().getForObject(
                                                "http://localhost:8080/rest/clinica/asegurados/{num}",
                                                 Asegurado.class,
                                                 asegurado.getNumSeg()
                                            );
```

- Las operaciones xxxxForEntity son más flexibles: devuelven una respuesta con un código de respuesta más un objeto

```
ResponseEntity<Asegurado> respuesta = new RestTemplate().getForEntity(
                                                "http://localhost:8080/rest/clinica/asegurados/{num}",
                                                 Asegurado.class,
                                                 asegurado.getNumSeg()
                                             );
Asegurado asegurado = respuesta.getBody();
HttpStatus codigoRespuesta = respuesta.getStatusCode();
```

Iniciación de RestTemplate

- Aunque los *RestTemplate* pueden usarse directamente como hemos visto en los ejemplos anteriores, se recomienda la configuración como Bean a partir de un *RestTemplateBuilder*, y la inyección donde sea necesario
- Spring Boot proporciona un RestTemplateBuilder por defecto, al que puede añadirse la configuración adicional que sea necesaria (p. ej. definir la URL base)

```
@SpringBootApplication
public class ServidorClinica {

      @Bean
      public RestTemplate restTemplate(RestTemplateBuilder builder) {
            builder.rootUri("http://localhost:8080/rest/clinica");
            return builder.build();
      }
…
```

Definir la URL base del servicio para evitar repeticiones innecesarias

Más ejemplos con RestTemplate

Obtener hojas abiertas

```
String url = "http://localhost:8080/clinica/hojas?abiertas=true";
ResponseEntity<Hoja[]> response = restTemplate.getForEntity(url, Hoja[].class);
```

Obtener las hojas del asegurado 3

```
String url = "http://localhost:8080/clinica/asegurados/{num}/hojas";
ResponseEntity<Hoja[]> response = restTemplate.getForEntity(url, Hoja[].class, 3);
```

Crear el asegurado con código 7

```
String url = "http://localhost:8080/clinica/asegurados/3";
ResponseEntity<Asegurado> response = restTemplate.postForEntity(url, asegurado, asegurado.class);

if (response.getStatusCode() != HttpStatus.OK) { // Error!!! }
```

Crear una hoja clínica sin indicar código

```
String url = "http://localhost:8080/clinica/hojasClinicas";
ResponseEntity<Hoja> response = restTemplate.postForEntity(url, hoja, hoja.class);

if (response.getStatusCode() != HttpStatus.OK) { // Error!!! }

System.out.println(respuesta.getBody().verCodigo()); // Código asignado tras la inserción
```