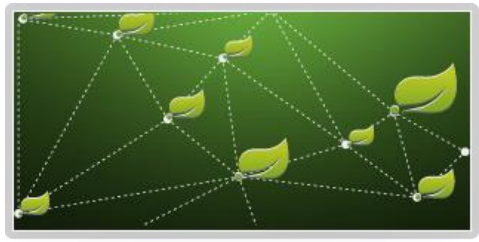
Formación
BDI

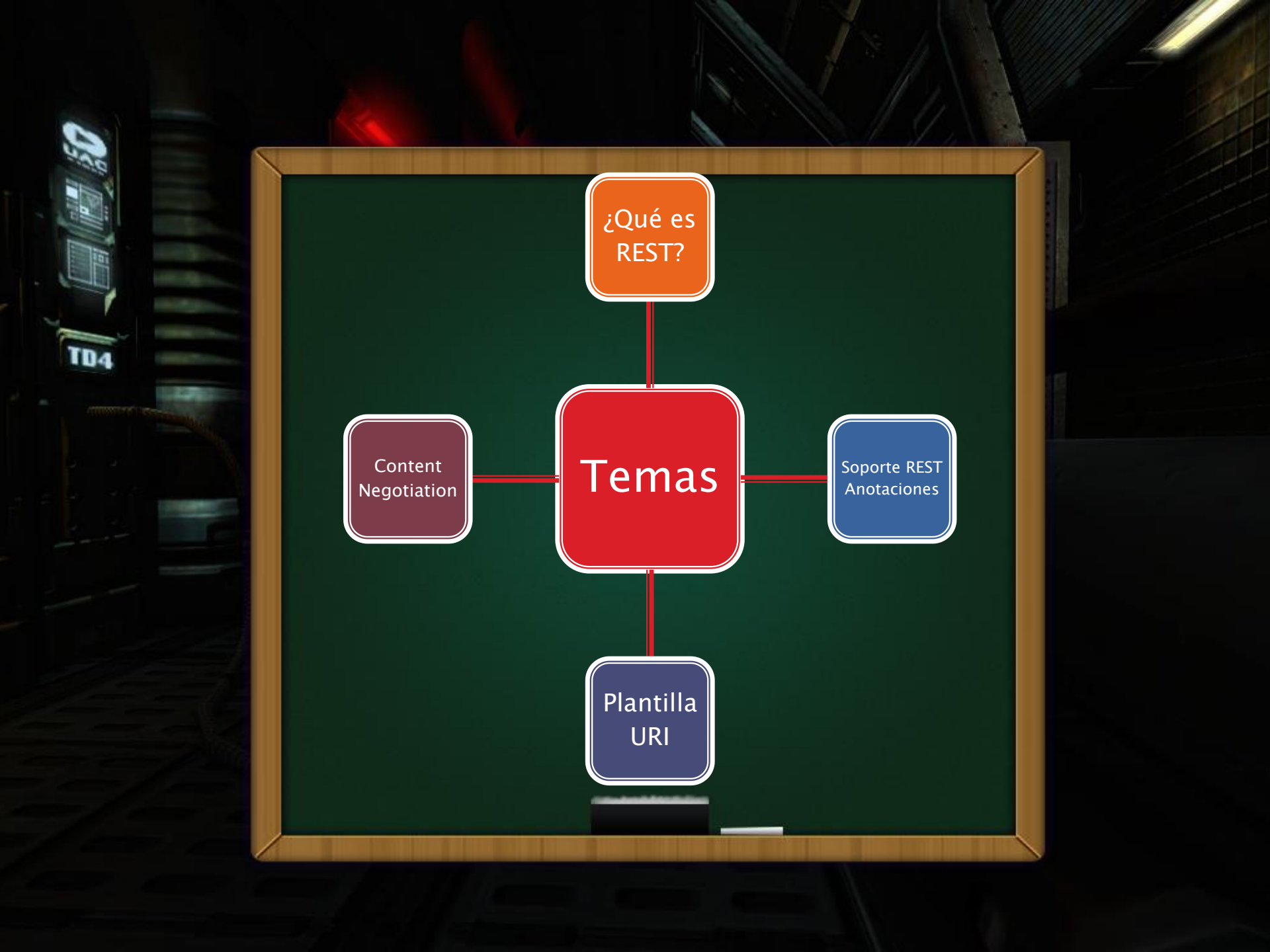


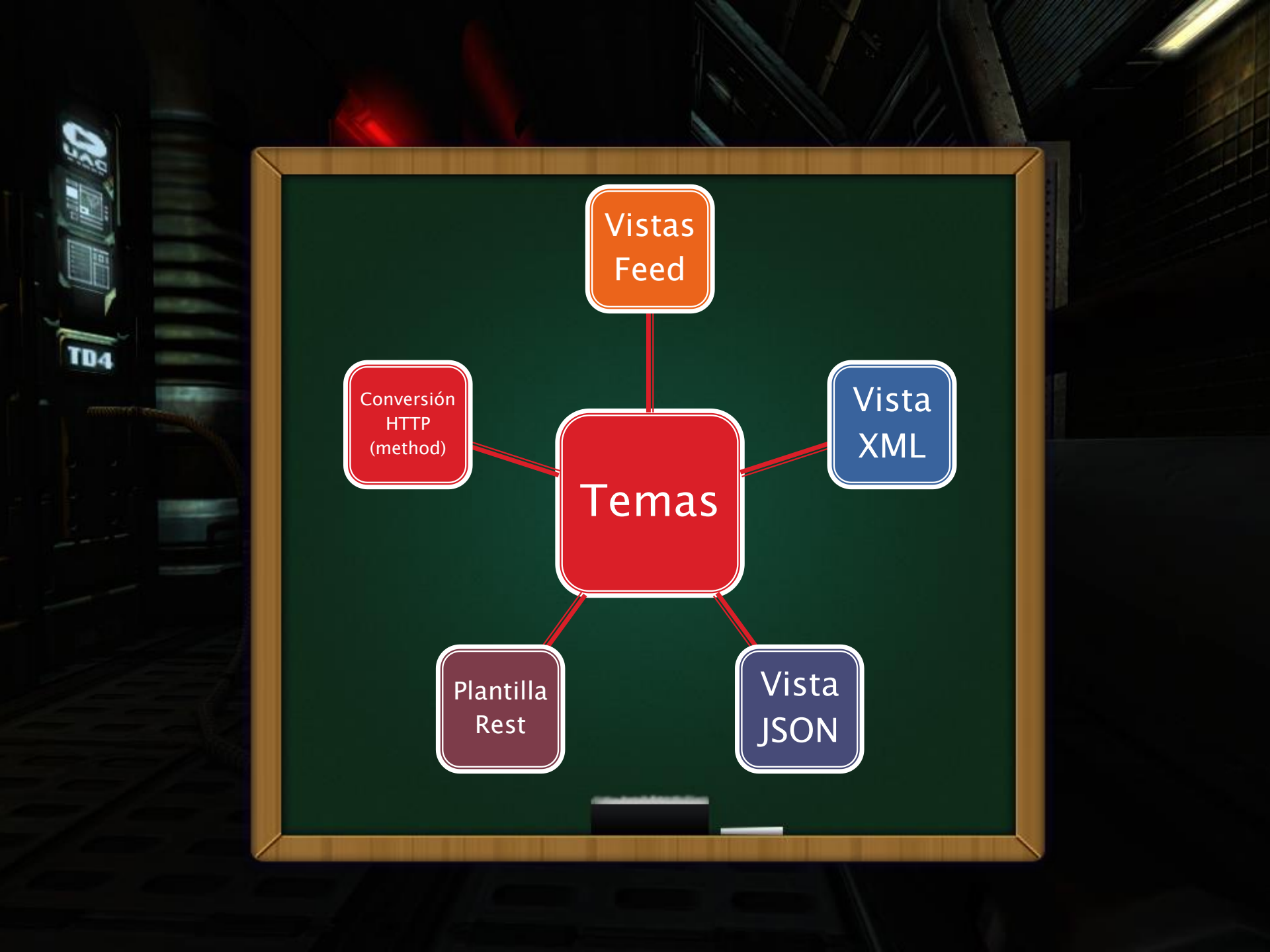
Curso Spring Framework

Módulo 8

Spring MVC REST

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com

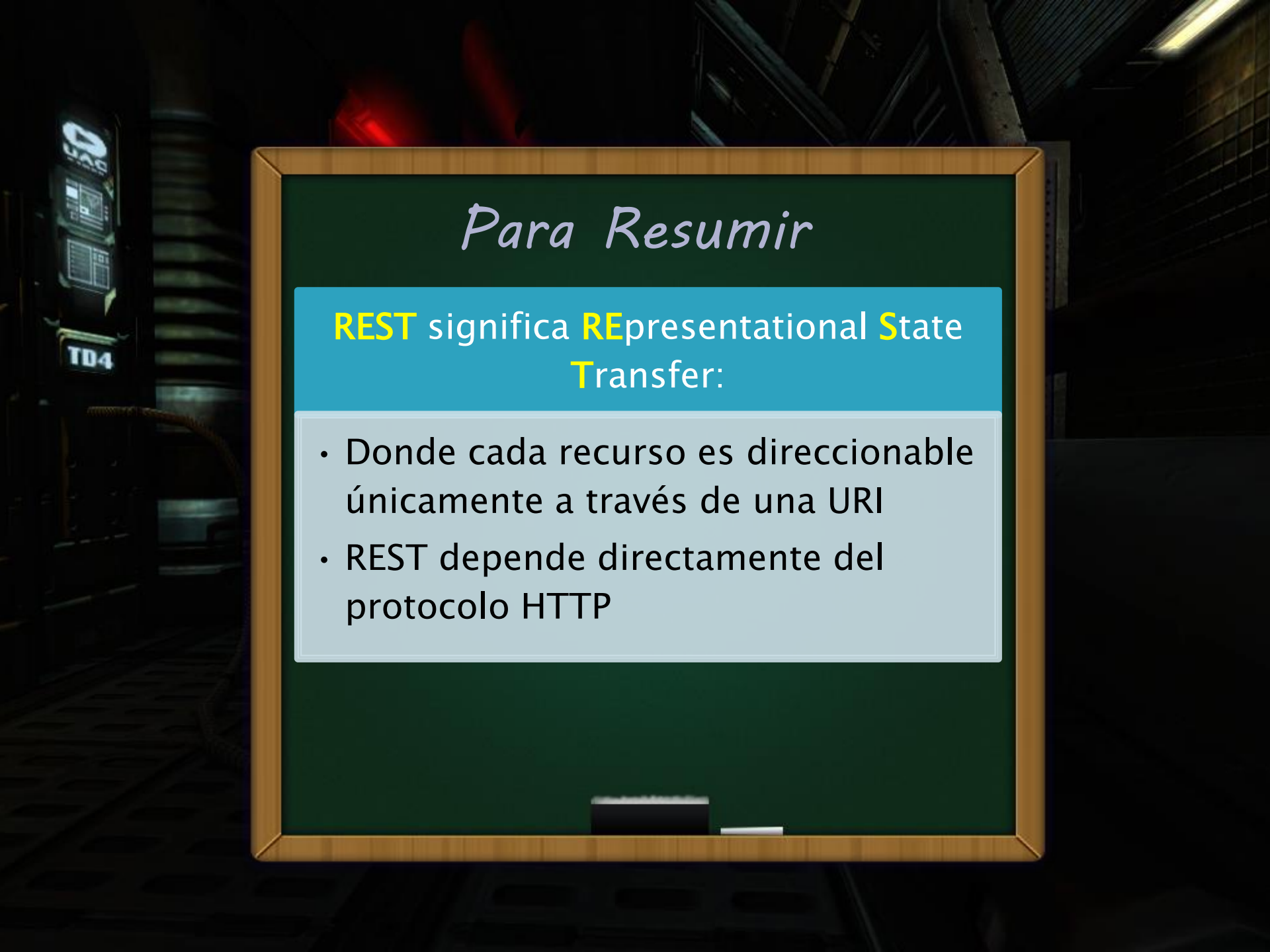

¿Qué es REST?
Content Negotiation
Temas
Soporte REST Anotaciones
Plantilla URI

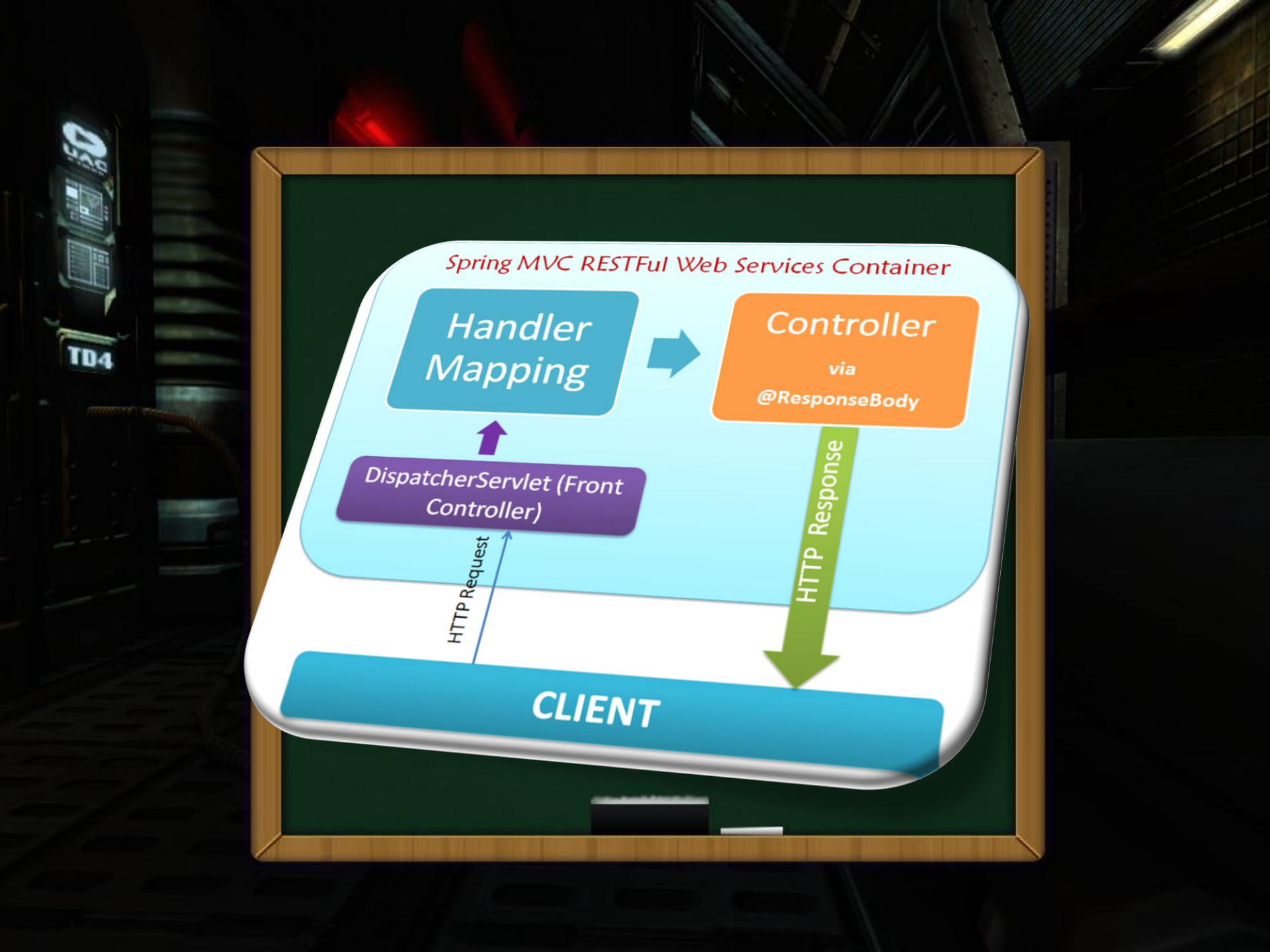

Vistas
Feed
Conversión
HTTP
(method)
Temas
Vista
XML
Plantilla
Rest
Vista
JSON

¿Qué es REST?

¿Qué es REST?

- Simple, Ligero y de Alto rendimiento
- Basado en el protocolo HTTP
- REST utiliza especificación XML y JSON
- Basados en nomenclatura URI, recursos direccionables
- Comunicación sin estado (Stateless)


Para Resumir
REST significa REpresentational State Transfer:
• Donde cada recurso es direccionable únicamente a través de una URI
• REST depende directamente del protocolo HTTP


Spring MVC RESTFul Web Services Container
Handler Mapping
Controller
via
@ResponseBody
DispatcherServlet (Front Controller)
HTTP Request
HTTP Response
CLIENT

Recursos Direccionables

- URI para cada recurso en el sistema
- Recursos accesible por ID único

http://ejemplo.com/rest/actualizar/100

- Permite pasar parámetros mediante la URI para filtrar en consultas:
  - GET Param (Query string)
  - PathParam (URI path)
- Hace que sea posible enlaces de links (linkeables)
- Permite que las aplicaciones interactúen y compartan recursos y servicios

Operaciones CRUD

Protocol method == operación	Una arquitectura basada en 4 métodos
• 4 métodos HTTP: GET, PUT, DELETE, POST	• SQL (SELECT, INSERT, UPDATE, DELETE)

HTTP method	Path	Function
GET	{path}/{id}	Read
POST	{path}	Create
PUT	{path}/{id}	Update
DELETE	{path}/{id}	Delete

Métodos HTTP

GET: leer datos, idempotente y seguro

PUT: inserción o actualización de datos, idempotente

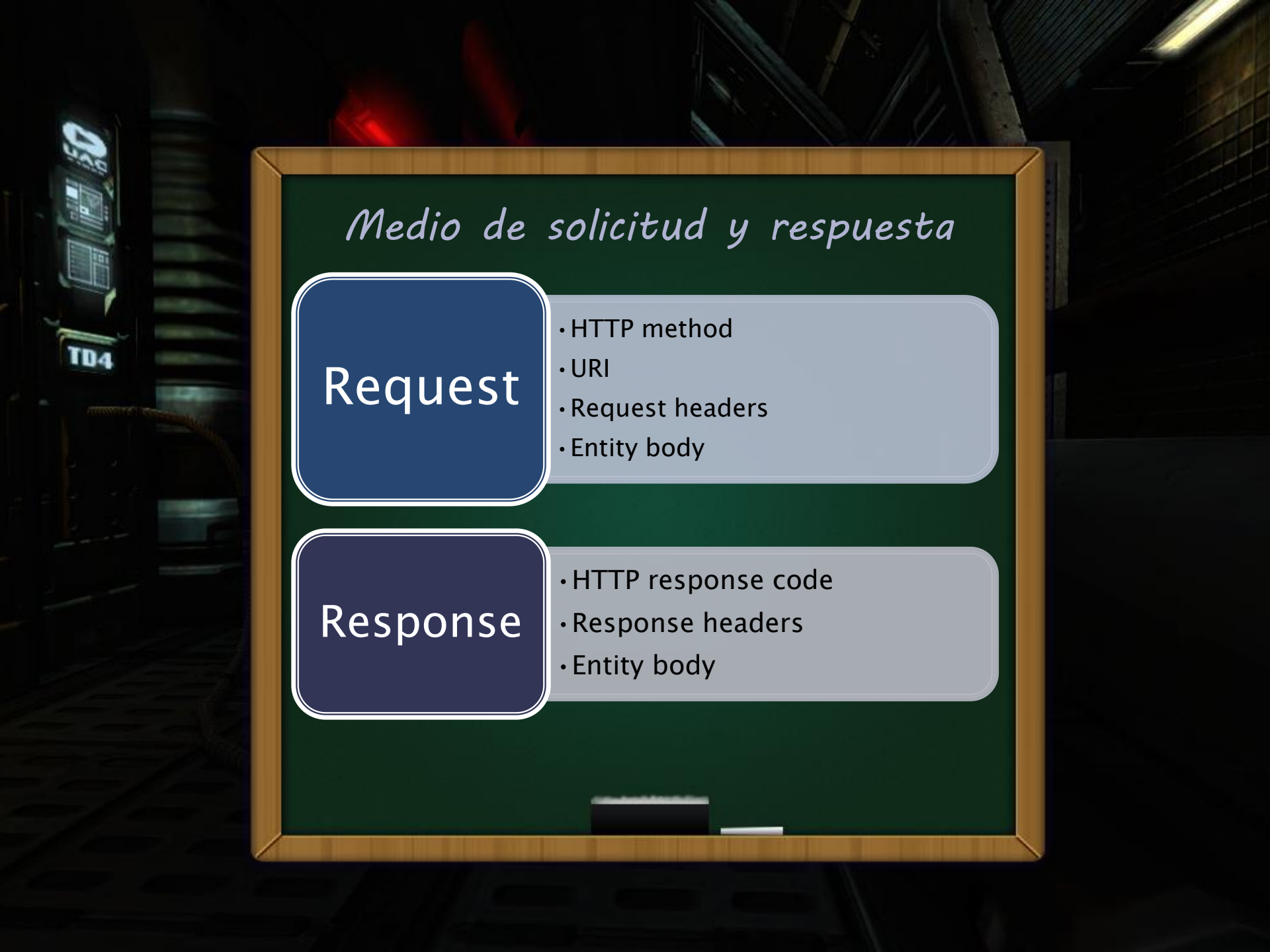
DELETE: eliminar datos, idempotente

POST: NO es idempotente, ni inseguro, más orientado a la web para inserción y actualización

Orientada a comunicación cliente/servidor

Los datos tienen una representación:

- Negociación entre cliente y servidor
- HTTP fue diseñado para este propósito
- Cliente "Yo preferiría ..."
  - ✓ Accept (MIME type)
  - ✓ Accept-Language
  - ✓ Accept-Encoding
- Servidor - "Esto es lo que te voy a dar ..."
  - ✓ Content-type header (MIME type)


Medio de solicitud y respuesta
Request
HTTP method
URI
Request headers
Entity body
Response
HTTP response code
Response headers
Entity body

¿Qué necesitamos para REST?

- Cliente HTTP (navegador, aplicaciones clientes, teléfono inteligente)
- Servidor HTTP con soporte REST
  - ✓ JAX-RS

Soporte REST basado en Anotaciones

Soporte REST en Spring

Trata directamente con el modelo de anotaciones de Spring MVC

En general trata de lo siguiente:

- Plantillas URI (URI templates)
- Content negotiation
- Conversión HTTP (Request method)

Plantilla URI

Plantilla URI

Trabaja igual que en Spring MVC

- Una Plantilla URI (URI template) como tipo String, contiene una o más nombres de variables. Cuando estas variables son remplazadas por valores, la plantilla se convierte en una URI
- El uso de las Plantillas URI es mediante la anotación @PathVariable

```
@RequestMapping("/hoteles/{hotelId}")
public String getHotel(@PathVariable String hotelId, Model model) {
    Hotel hotel = hotelService.getHotel(hotelId);
    model.addAttribute("hotel", hotel);
    return "hotelDetail";
}
```

Ejemplo URI Template

Puedes manejar rutas encadenadas, por ejemplo, /hoteles/2/reservas/4 o /hoteles/1/reservas/3:

```
@RequestMapping(value="/hoteles/{hotel}/reservas/{reserva}",
                        method=RequestMethod.GET)
public String getReserva(@PathVariable("hotel") long hotelId,
                         @PathVariable("reserva") long reservaId,
                                Model model) {
    Hotel hotel = hotelService.getHotel(hotelId);
    Reserva reserva = hotel.getReserva(reservaId);
    model.addAttribute("reserva", reserva);
    return "reserva";
}
```

Ejemplo URI Template

Puedes manejar un estilo de rutas wildcard *, por ejemplo:

```
@RequestMapping(value="/hoteles/*/reservas/{reserva}",
                        method=RequestMethod.GET)
public String getReserva(@PathVariable("reserva") long reservaId,
                                Model model)  {
    ...
}
```

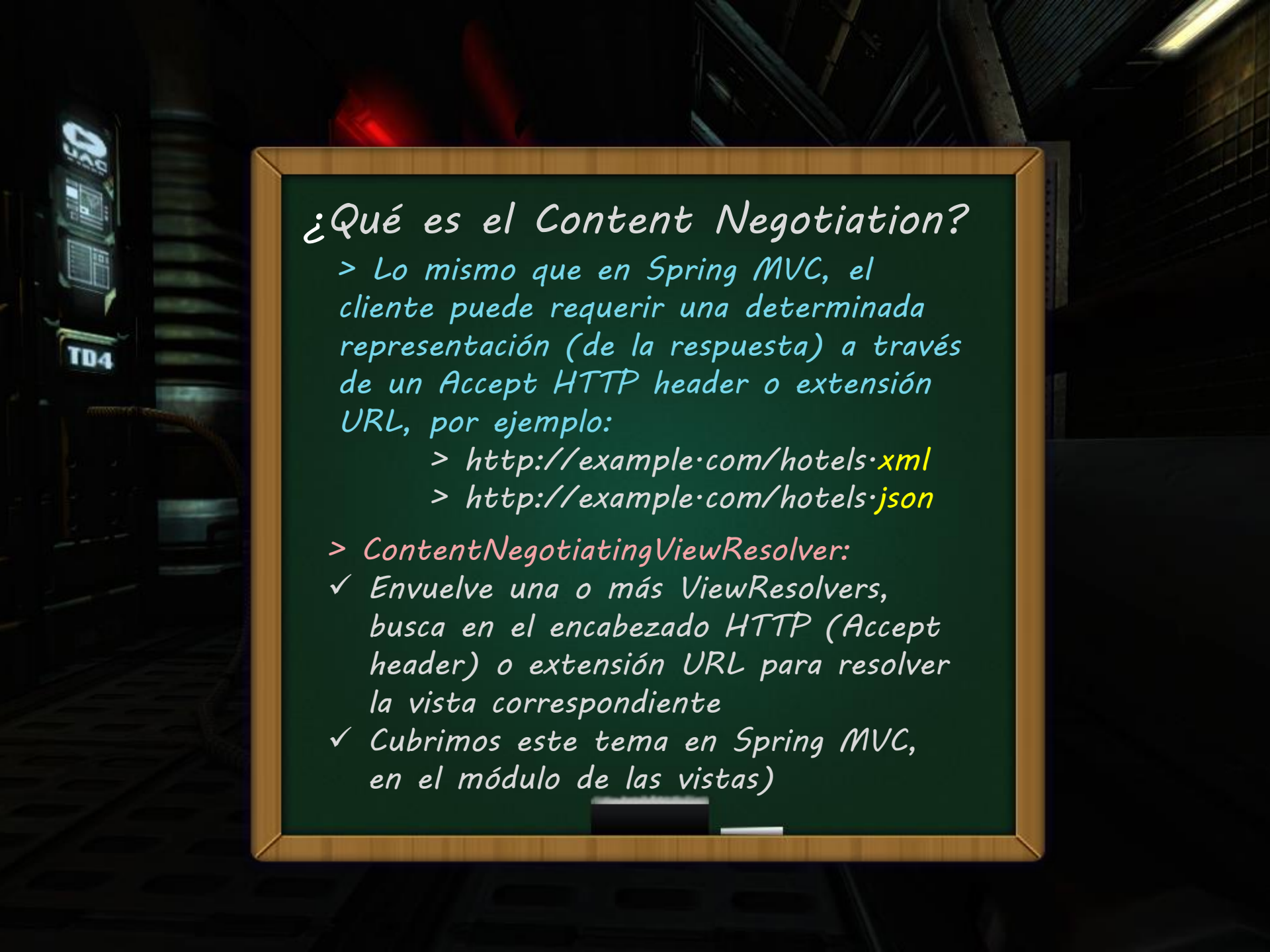
Ejemplo URI Template

Puedes usar data binding:

```
@InitBinder
public void initBinder(WebDataBinder binder) {
   SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
   binder.registerCustomEditor(Date.class,
                                       new CustomDateEditor(dateFormat, false));
}

@RequestMapping("/hoteles/{hotel}/fechas/{fecha}")
public void fecha(@PathVariable("hotel") String hotel,
                  @PathVariable Date fecha) {
   ...
}
```

Content Negotiation


¿Qué es el Content Negotiation?
> Lo mismo que en Spring MVC, el cliente puede requerir una determinada representación (de la respuesta) a través de un Accept HTTP header o extensión URL, por ejemplo:
> http://example.com/hotels.xml
> http://example.com/hotels.json
> ContentNegotiatingViewResolver:
✓ Envuelve una o más ViewResolvers, busca en el encabezado HTTP (Accept header) o extensión URL para resolver la vista correspondiente
✓ Cubrimos este tema en Spring MVC, en el módulo de las vistas)

Vistas Feed

Vistas Feed

Tanto las clases AbstractAtomFeedView y AbstractRssFeedView heredan de la clase abtsracta AbstractFeedView y son usadas para proveer vistas Atom y RSS Feed respectivamente.

Creando un Atom

```
// AbstractAtomFeedView requiere implementar el método
// buildFeedEntries() y opcionalmente sobrescribir
// el método buildFeedMetadata()
public class EjemploContenidoAtomView extends AbstractAtomFeedView {

   @Override
   protected void buildFeedMetadata(Map<String, Object> model, Feed feed,
                                    HttpServletRequest request) {
      // ...
   }

   @Override
   protected List<Entry> buildFeedEntries(Map<String, Object> model,
                                          HttpServletRequest request,
                                          HttpServletResponse response)
                                          throws Exception {
      // ...
   }
}
```

Creando un Rss

```
public class EjemploContenidoRssView extends AbstractRssFeedView {

   @Override
   protected void buildFeedMetadata(Map<String, Object> model, Channel
feed,
                        HttpServletRequest request) {
      // ...
   }

   @Override
   protected List<Item> buildFeedItems(Map<String, Object> model,
                        HttpServletRequest request,
                        HttpServletResponse response)
                        throws Exception {
      // ...
   }
}
```

Dependencia Maven

```
<!-- Rome RSS -->
<dependency>
  <groupId>com.rometools</groupId>
  <artifactId>rome</artifactId>
  <version>1.5.1</version>
</dependency>
```

Vista XML

Vista XML Marshalling

El MarshallingView utiliza un Marshaller XML definido en el paquete org.springframework.oxm para hacer que el contenido de la respuesta sea en formato XML

El objeto marshalled se puede establecer explícitamente en el atributo modelKey de la clase bean MarshallingView

JAXB2 Marshaller

```
<bean id="estudianteView"
class="org.springframework.web.servlet.view.xml.MarshallingView">
   <constructor-arg ref="jaxbMarshaller" />
</bean>

<!-- JAXB2 marshaller. Atomaticamente convierte los beans en xml -->
<bean id="jaxbMarshaller"
class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
   <property name="classesToBeBound">
      <list>
         <value>com.formacionbdi.ejemplo.Curso</value>
         <value>com.formacionbdi.ejemplo.Estudiante</value>
      </list>
   </property>
</bean>
```

Xstream Marshaller

```
<!-- Usamos BeanName como view resolver -->
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />

<!-- Vista lógica "libroXmlView" es desplegada como XML -->
<bean id="libroXmlView"
class="org.springframework.web.servlet.view.xml.MarshallingView">
   <constructor-arg>
      <bean class="org.springframework.oxm.xstream.XStreamMarshaller"/>
   </constructor-arg>
</bean>
```

Dependencia Maven

```
<!– Spring OXM XML View (con JAXB2)-->
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-oxm</artifactId>
   <version>4.2.0.RELEASE</version>
</dependency>
```

Vista JSON

Vista JSON MappingJackson

El bean MappingJacksonJsonView usa el API Jackson ObjectMapper para dibujar la respeusta en formato JSON.

Por defecto, todo el contenido/atributos guardado en el objeto model map (datos de la vista) será codificado como JSON.

Para el caso en que el contenido del model map necesite ser filtrado, podemos especificar un set específico de atributos del objeto model para codificar mediante el atributo RenderedAttributes.

Configuración MappingJacksonJsonView

```
<!-- Usamos BeanName como view resolver -->
<bean id="beanNameViewResolver"
class="org.springframework.web.servlet.view.BeanNameViewResolver" />

<!-- Vista lógica "cuentas" manejada por la clase "MappingJackson2JsonView" y
los datos son desplegados como Json  -->
<bean id="cuentas"
class="org.springframework.web.servlet.view.json.MappingJackson2JsonView" />
```

Dependencia Maven

```
<!-- Jackson JSON Mapper -->
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>2.6.1</version>
</dependency>
```

Plantilla Rest

¿Por qué usar RestTemplate?

- Invocar un servicio RESTful desde un cliente en Java típicamente se hace con una clase helper tal como Jakarta Commons HttpClient. Para las típicas operaciones REST este enfoque es de muy bajo nivel, como se muestra en el siguiente ejemplo:

```
String uri = "http://miapp.com/hoteles/1/reservas";
PostMethod post = new PostMethod(uri);
String request = // request de la reserva
post.setRequestEntity(new StringRequestEntity(request));
httpClient.executeMethod(post);
if (HttpStatus.SC_CREATED == post.getStatusCode()) {
   Header location = post.getRequestHeader("Location");
   if (location != null) {
      System.out.println("Se ha creado una nueva reserva:" + location.getValue());
   }
}
```

RestTemplate

El componente RestTemplate de Spring provee métodos de alto nivel que corresponden a cada uno de los principales tipos de métodos HTTP (HTTP methods), capaces de realizar varias invocaciones a servicios RESTful en una sola línea

Métodos RestTemplate

- getForObject(), getForEntity() para HTTP GET
- postForLocation(String url, ...) y postForObject(String url, ...) para HTTP POST
- delete() para HTTP DELETE
- put(String url, ...) para HTTP PUT
- headForHeaders(String url, ..) para HTTP HEAD
- optionsForAllow(String url, ..) para HTTP OPTIONS

Convención de nombres en Métodos

Por ejemplo el método getForObject() invocará un GET, convirtiendo la respuesta HTTP en un objeto del tipo que elijamos, y retornará este objeto con los datos

postForLocation() hará un POST, convirtiendo un determinado objeto (a elección) en una petición HTTP y retornará la respuesta que contiene la cabecera HTTP Location donde se puede ubicar el nuevo objeto creado en el POST

Argumentos URI Template

Cada método toma argumentos URI template en dos formas:

- String varargs (argumentos variables)

```
String result = restTemplate.getForObject(
        "http://miapp.com/hoteles/{hotel}/reservas/{reserva}",
        String.class,"7", "3");
```

- Map<String, String>

```
Map<String, String> vars = new HashMap<String, String>(2);
vars.put("hotel", "7");
vars.put("reserva", "3");
String result = restTemplate.getForObject(
    "http://miapp.com/hoteles/{hotel}/reservas/{reserva}",
    String.class, vars);
```

Conversión HTTP (method)

Conversión Método HTTP

Otro principio clave de REST es el uso de la interfaz Uniform

Todos los recursos (URL) pueden ser manipular utilizando los mismos métodos HTTP: GET, PUT, POST y DELETE

Mientras que HTTP define varios tipos de métodos, HTML sólo admite dos: GET y POST

Conversión Método HTTP

Dos soluciones

- Forma1: Usar JavaScript para hacer un PUT o DELETE
- Forma 2: Hacer un POST con el método "real" como un parámetro adicional (como un campo input hidden en el formulario HTML)

Forma2: Hacer un Post

- Configuración web.xml

```
<!-- Filter that converts posted method parameters into HTTP methods, retrievable
via HttpServletRequest.getMethod(). Since browsers currently only support GET
and POST, a common technique - used by the Prototype library, for instance -
is to use a normal POST with an additional hidden form field (_method) to
pass the "real" HTTP method along. This filter reads that parameter and
changes the HttpServletRequestWrapper.getMethod() return value accordingly.
-->
<filter>
   <filter-name>httpMethodFilter</filter-name>
   <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>httpMethodFilter</filter-name>
   <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

Conversión Método en etiquetas form Spring MVC

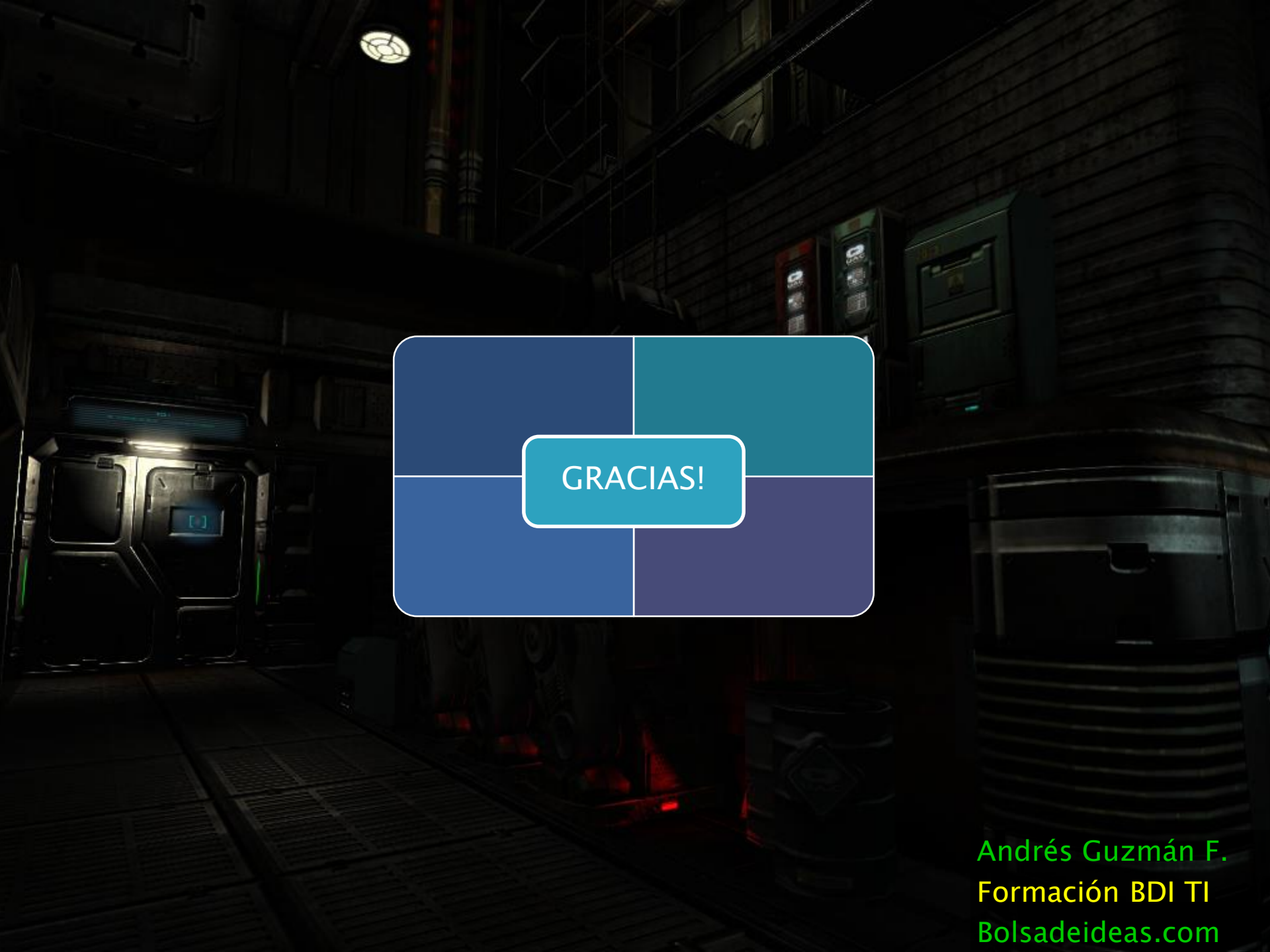
En realidad ejecuta y envía un HTTP POST, con el "real" método DELETE escondido detrás de un parámetro de la petición, para ser recogido y procesado por el filtro HiddenHttpMethodFilter:

```
<form:form method="delete">
  <p class="submit">
    <input type="submit" value="Eliminar Reserva"/>
  </p>
</form:form>
```

Código por el lado del servidor

El código del Controller maneja HTTP DELETE:

```
@RequestMapping(method = RequestMethod.DELETE)
public String deleteReserva(@PathVariable int hotelId,
                            @PathVariable int reservaId) {
   this.dao.deleteReserva(reservaId);
   return "redirect:/hoteles/" + hotelId;
}
```



Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com