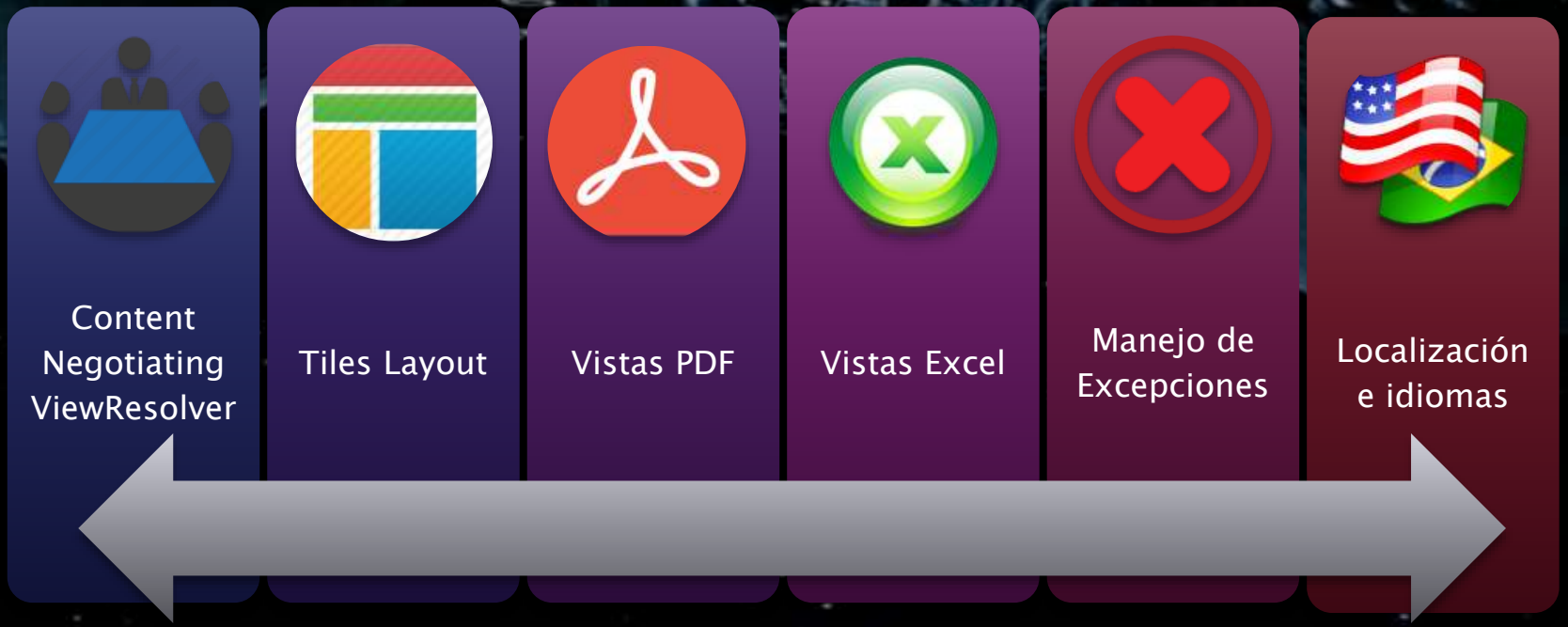
Formación
BDI

Curso Spring Framework

Módulo 5
Tecnologías de Vistas y Localización

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com

¿Qué veremos?


Content Negotiating ViewResolver
Tiles Layout
Vistas PDF
Vistas Excel
Manejo de Excepciones
Localización e idiomas

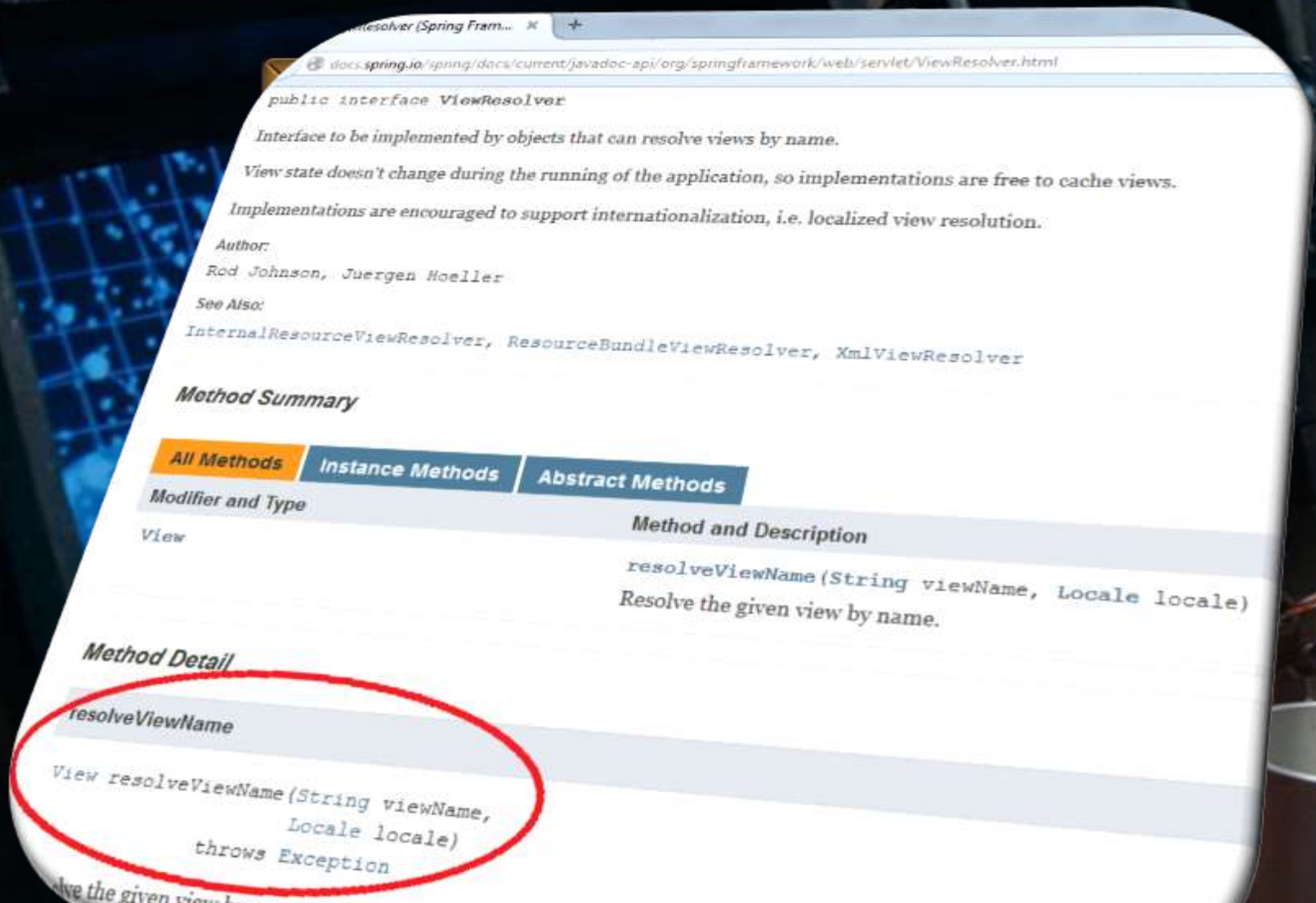
Resolviendo Vistas: ContentNegotiating ViewResolver

ContentNegotiatingViewResolver

El ContentNegotiatingViewResolver no resuelve la vista por si mismo, si no que delega esta tarea a otros view resolvers

Un seleccionado view resolver se encarga de resolver el nombre de la vista lógico para un objeto View

Interfaz ViewResolver



Dos formas para determinar el Tipo de Vista

Forma 1: Usar extensión del archivo

- La URL http://www.imdb.com/actor/BruceLee.**pdf** pide una representación en PDF de Bruce Lee
- La URL http://www.imdb.com/actor/BruceLee.**xml** pide una representación en XML de Bruce Lee

Dos formas para determinar el Tipo de Vista

Forma 2 - Asignar el "Accept HTTP request header" para listar los tipos de archivos "media types" (o Content-Type) soportados

- Para el request http://www.imdb.com/actor/BruceLee con un Accept header **application/pdf** pide una representación en PDF de Bruce Lee
- Para el request http://www.imdb.com/actor/BruceLee con un Accept header **text/xml** pide una representación en XML

Atributo "mediaTypes" de ContentNegotiatingViewResolver

Para soportar la resolución de la vista basado en la extensión del archivo (Forma #1), el bean ContentNegotiatingViewResolver usa el atributo "mediaTypes" para definir un mapeo de las extensiones de archivo hacia los media types

```
<bean
class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="html" value="text/html"/>
      <entry key="json" value="application/json"/>
      <entry key="xml" value="application/xml" />
      <entry key="atom" value="application/atom+xml"/>
      <entry key="pdf" value="application/pdf" />
      <entry key="xsl" value="application/vnd.ms-excel" />
    </map>
  </property>
....
</bean>
```

Ejemplo ContentNegotiatingViewResolver

- Supongamos el siguiente ejemplo:

```
<bean
class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
...
   <property name="viewResolvers">
      <list>
         <bean
          class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
         <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
            <property name="prefix" value="/WEB-INF/jsp/"/>
            <property name="suffix" value=".jsp"/>
         </bean>
      </list>
   </property>
...
</bean>
```

Ejemplo ContentNegotiatingViewResolver

El ContentNegotiatingViewResolver selecciona el View Resolver más apropiado para manejar el request comparando el "media type" (conocido también como Content-Type) con el "media type" soportado por la vista asociada con cada uno de sus ViewResolvers

Si el atributo viewResolvers (lista de view resolvers) del bean ContentNegotiatingViewResolver NO se encuentra definido explícitamente, entonces automáticamente usa cualquier ViewResolvers que tengamos definido en el xml de contexto de Spring

View Resolver por Defecto

```
<bean
class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
...
   <property name="defaultViews">
      <list>
         <bean
class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
      </list>
   </property>
```

- Si la cadena de los View Resolvers no detectan ninguna vista compatible para aplicar, entonces se consultará la lista de vistas por defecto definidas en el atributo defaultViews

Ejemplo retorno nombre lógico de la vista

El la siguiente dispositiva, observaremos un controlador que retorna contenido compatible con formato Atom RSS feed mediante las URLs:
http://localhost/content.atom o
http://localhost/content

Con un Accept header asignado a **application/atom+xml**.

Además tenga en cuenta de que se define el nombre lógico de la vista, pero no hay ningún tipo de código específico al tipo de vista

Ejemplo retorno nombre lógico de la vista

```
@Controller
public class ContentController {

   private List<SampleContent> contentList = new ArrayList<SampleContent>();

   @RequestMapping(value="/content", method=RequestMethod.GET)
   public ModelAndView getContent() {
      ModelAndView mav = new ModelAndView();
      mav.setViewName("content");
      mav.addObject("sampleContentList", contentList);
      return mav;
   }
}
```

Continuando con el ejemplo, dada la siguiente configuración

```
<bean
class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="atom" value="application/atom+xml"/>
      <entry key="html" value="text/html"/>
      <entry key="json" value="application/json"/>
    </map>
  </property>
  <property name="viewResolvers">
    <list>
      <bean
class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
      <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
      </bean>
    </list>
  </property>
  <property name="defaultViews">
    <list>
      <bean
class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
    </list>
  </property>
</bean>
<bean id="content"
class="com.springsource.samples.rest.SampleContentAtomView"/>
```

Explicación del ejemplo de las diapositivas anteriores

Si el request URL es con extensión .html, el view resolver buscará una vista que coincida con el media type text/html.

- Entonces un InternalResourceViewResolver será el encargado de resolver la vista para el content-type text/html, para que la encuentre debería existir una vista JSP con nombre content.jsp.

Explicación del ejemplo de las diapositivas anteriores

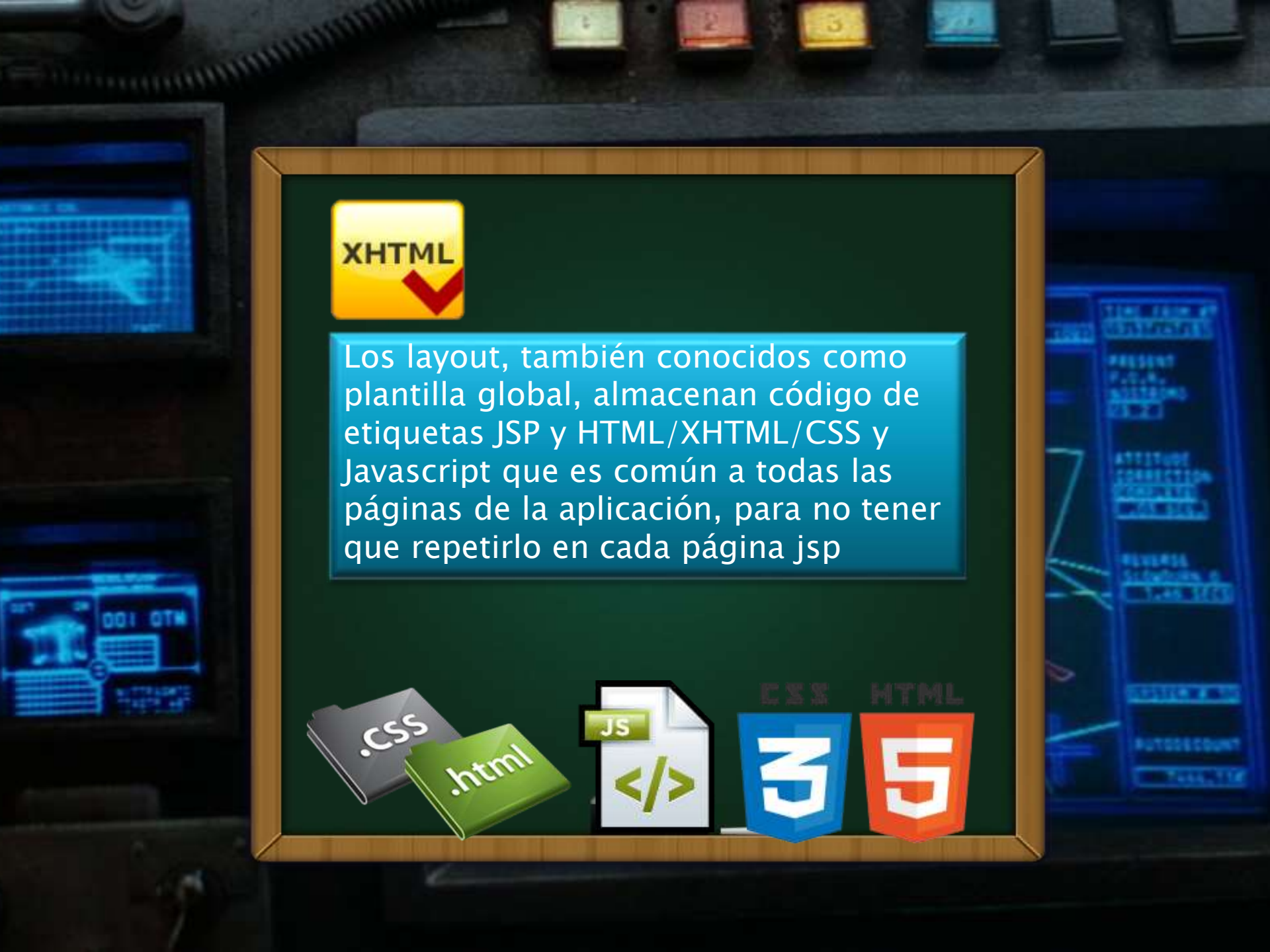
Si el request URL es con extensión .atom, el view resolver buscará una vista que coincida con el media typ application/atom+xml.

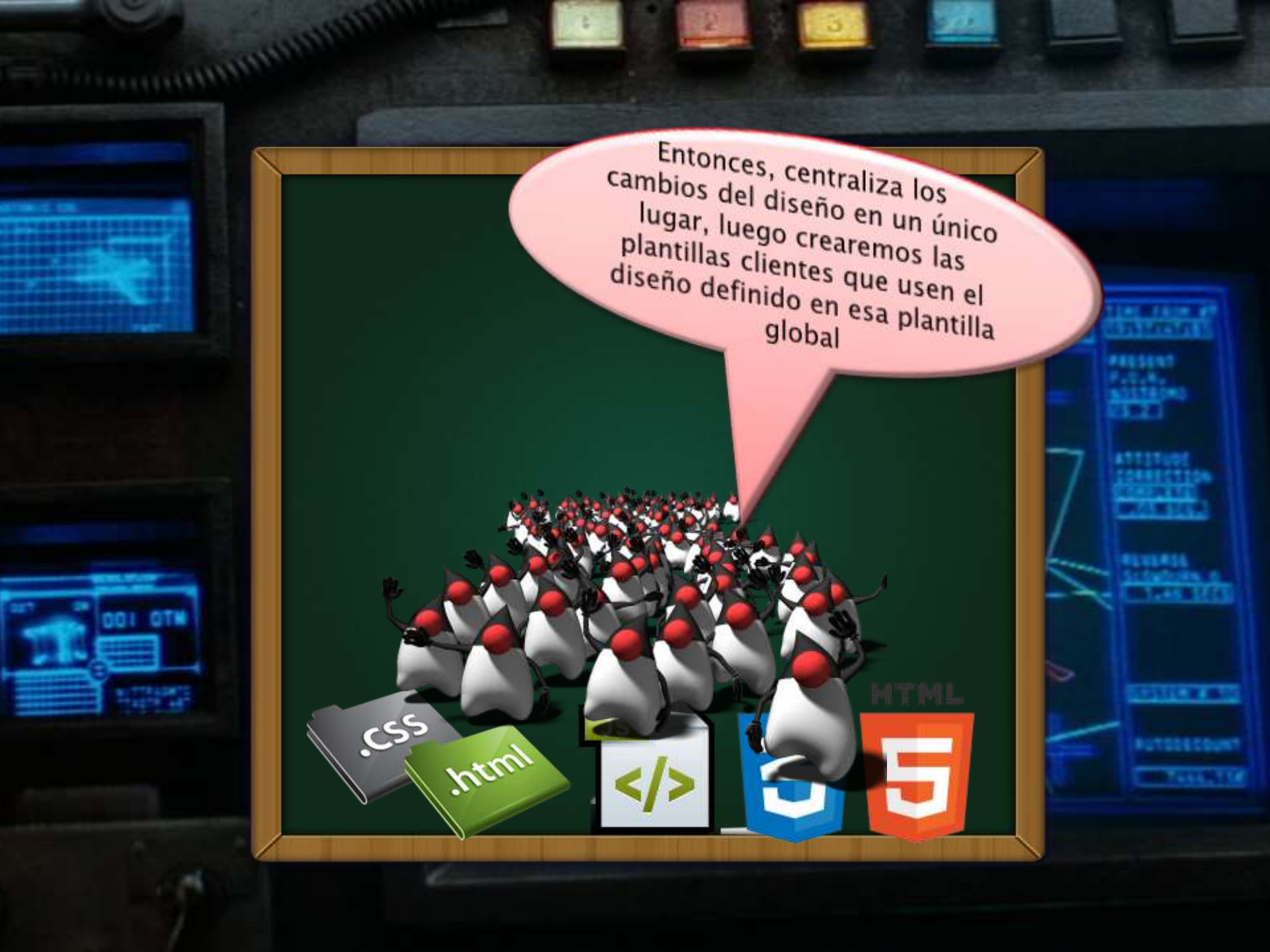
- Esta vista es proveída por el BeanNameViewResolver que mapea hacia el bean view SampleContentAtomView siempre y cuando el nombre de la vista retornada sea "content".

Explicación del ejemplo de las diapositivas anteriores

Si el request URL es con extensión .json, entonces se seleccionará la vista JSON MappingJacksonJsonView de la lista DefaultViews independientemente del nombre de la vista retornada

Layout o Plantillas Tiles


XHTML
Los layout, también conocidos como plantilla global, almacenan código de etiquetas JSP y HTML/XHTML/CSS y Javascript que es común a todas las páginas de la aplicación, para no tener que repetirlo en cada página jsp
.CSS
.html
JS
CSS
HTML


Entonces, centraliza los cambios del diseño en un único lugar, luego crearemos las plantillas clientes que usen el diseño definido en esa plantilla global
.css
.html
</>
HTML

Tiles con Spring MVC

Si, es posible integrar Tiles en aplicaciones Web con Spring Framework MVC – tal como cualquier otra tecnología de vista.

Configurar Tiles con Spring

- Para habilitar el uso de Tiles, tenemos que configurar utilizando archivos XML que contienen las definiciones de Tiles
- En Spring, esto se hace definiendo el bean TilesConfigurer en el XML de contexto Web de spring.

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/WEB-INF/tiles.xml</value>
    </list>
  </property>
</bean>
```

Donde la definición tiles·xml podría ser algo como lo siguiente:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles
Configuration 3.0//EN" "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
    <!-- La definición principal del layout o root -->
   <definition name="definicion.principal" template="/WEB-
INF/tiles/layout.jsp">
      <put-attribute name="header" value="/WEB-INF/tiles/header.jsp" />
      <put-attribute name="body" value="" />
      <put-attribute name="title" value="" />
      <put-attribute name="footer" value="/WEB-INF/tiles/footer.jsp" />
   </definition>

   <!-- Esta es una definición Tiles hija que extiende de la principal
"definicion.principal" sobrescribe los componentes "title" y "body" -->
   <definition name="form" extends="definicion.principal">
      <put-attribute name="title" value="Crear Cliente" />
      <put-attribute name="body" value="/WEB-INF/views/clienteForm.jsp" />
   </definition>
</tiles-definitions>
```

*Donde el atributo **name** de las definiciones tiles hijas corresponde al nombre lógico de la vista retornada en los métodos handler del controlador, ejemplo:*

En el controlador retornamos la vista **estudianteForm**:

```
@RequestMapping(method = RequestMethod.GET)
  public String getCreateForm(ModelMap model) {
     Estudiante estudiante = new Estudiante("Andrés");
     estudiante.setTemas(new String[] { "Matematicas" });
     model.addAttribute("estudianteCommand", estudiante);
     return "estudianteForm";
  }
```

En el XML de tiles hacemos una definición con ese nombre retornado por el controlador **estudianteForm**:

```
<definition name="estudianteForm" extends="definicion.principal">
     <put-attribute name="title" value="Crear Estudiante" />
     <put-attribute name="body" value="/WEB-INF/views/estudianteForm.jsp" />
  </definition>
```

Otro ejemplo de Configurar Tiles

- Veamos otro ejemplo sencillo, donde tiene más de un archivo XML con las definiciones Tiles:

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/WEB-INF/defs/general.xml</value>
      <value>/WEB-INF/defs/widgets.xml</value>
      <value>/WEB-INF/defs/administrator.xml</value>
      <value>/WEB-INF/defs/customer.xml</value>
      <value>/WEB-INF/defs/templates.xml</value>
    </list>
  </property>
</bean>
```

Usando Tiles como Vistas

En el arranque de Web ApplicationContext de Spring, los archivos XML se cargan y se inicializan las definiciones de tiles

Luego, Tiles incluye las definiciones xml que pueden ser usadas como vistas dentro de nuestra aplicación web con Spring

Para poder utilizar las vistas hay que tener configurado “como siempre un ViewResolver” al igual que con cualquier otra tecnología de vista

Configurar Tiles con UrlBasedViewResolver

- El bean view resolver UrlBasedViewResolver instancia los viewClass por cada vista que tenga que resolver

```
<!-- Para resolver vistas Tiles -->
<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
   <property name="viewClass"
        value="org.springframework.web.servlet.view.tiles3.TilesView"/>
</bean>
```

Dependencias Maven

```
<!-- Apache Tiles -->
<dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-api</artifactId>
   <version>3.0.5</version>
</dependency>
<dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-core</artifactId>
   <version>3.0.5</version>
</dependency>
<dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-jsp</artifactId>
   <version>3.0.5</version>
</dependency>
<dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-servlet</artifactId>
   <version>3.0.5</version>
</dependency>
<dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-template</artifactId>
   <version>3.0.5</version>
</dependency>
<dependency>
   <groupId>org.apache.tiles</groupId>
   <artifactId>tiles-el</artifactId>
   <version>3.0.5</version>
</dependency>
```

Vistas PDF

Crea documentos PDF de forma muy simple heredando de la clase AbstractPdfView

Vistas PDF

Podemos implementar una vista PDF mediante una clase que extienda de la clase Abstracta AbstractPdfView e implemente el método buildPdfDocument(…):

```
public class PaginaPDF extends AbstractPdfView {

   protected void buildPdfDocument(Map model, Document doc,
            PdfWriter writer, HttpServletRequest req,
            HttpServletResponse resp) throws Exception {

      String[] palabras= (String[]) model.get("palabrasList");
      for (int i=0; i<palabras.length; i++)
         doc.add( new Paragraph((String) palabras[i]));
   }
}
```

Vistas PDF

Luego el controlador asigna los datos (palabras) a la clase Model view y retorna el nombre de la vista pdf “palabras_pdf”:

```
@RequestMapping(method=RequestMethod.GET)
  public String getPalabrasPDF(Model model) {

    // Agregamos algunos datos de ejemplo
    String[] palabras = { 'Spring', 'MVC', 'PDF' };
    model.addAttribute("palabras", palabras);
    return "palabras_pdf";
  }
```

Finalmente en el XML del contexto web de Spring, configuramos el ContentNegotiatingViewResolver, luego el view resolver BeanNameViewResolver (para resolver la vista por nombre del bean) y la vista PDF palabras_pdf:

```
<bean
class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
   <property name="mediaTypes">
      <map>
         <entry key="html" value="text/html"/>
         <entry key="json" value="application/json"/>
         <entry key="xml" value="application/xml" />
         <entry key="pdf" value="application/pdf" />
         <entry key="xsl" value="application/vnd.ms-excel" />
      </map>
   </property>
....
</bean>

<!-- BeanNameViewResolver -->
<bean id="beanNameViewResolver"
class="org.springframework.web.servlet.view.BeanNameViewResolver">
   <property name="order" value="#{contentNegotiatingResolver.order+1}" />
</bean>

<!-- La vista "palabras_pdf" que es manejada por la clase "PaginaPDF" -->
<bean id="palabras_pdf" class="com.bolsadeideas.ejemplos.views.PaginaPDF" />
```

Dependencia Maven

```
<!-- PDF support -->
<dependency>
   <groupId>com.lowagie</groupId>
   <artifactId>itext</artifactId>
   <version>2.1.7</version>
</dependency>
```

Vistas Excel

También permite crear documentos Excel heredando de AbstractExcelView

Podemos implementar una vista XLS mediante una clase que extienda de la clase Abstracta AbstractXlsxView e implemente el método buildExcelDocument(...):

```
public class PaginaExcel extends AbstractXlsxView {

   protected void buildExcelDocument((Map model, Workbook wb,
            HttpServletRequest req,
            HttpServletResponse resp) throws Exception {
      Sheet sheet = wb.createSheet("Spring")
      Cell cell;

      sheet.setDefaultColumnWidth((short) 12);
      cell = getCell(sheet, 0, 0);
      setText(cell, "Spring-Excel Ejemplo");
      String[] palabras= (String[]) model.get("palabrasList");
      for (int i=0; i<palabras.length; i++){
         cell = getCell(sheet, 2+i, 0);
         setText(cell, (String) (String) palabras[i]);
      }
   }
}
```

Vistas Excel

Luego el controlador asigna los datos (palabras) a la clase Model view y retorna el nombre de la vista Excel “palabras_xls”:

```
@RequestMapping(method=RequestMethod.GET)
  public String getPalabrasXLS(Model model) {

    // Agregamos algunos datos de ejemplo
    String[] palabras = { 'Spring', 'MVC', 'PDF' };
    model.addAttribute("palabras", palabras);
    return "palabras_xls";
  }
```

Finalmente en el XML del contexto web de Spring, configuramos el ContentNegotiatingViewResolver, luego el view resolver BeanNameViewResolver (para resolver la vista por nombre del bean) y la vista Excel palabras_xls:

```
<bean
class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
   <property name="mediaTypes">
      <map>
         <entry key="html" value="text/html"/>
         <entry key="json" value="application/json"/>
         <entry key="xml" value="application/xml" />
         <entry key="pdf" value="application/pdf" />
         <entry key="xsl" value="application/vnd.ms-excel" />
      </map>
   </property>
....
</bean>

<!-- BeanNameViewResolver -->
<bean id="beanNameViewResolver"
class="org.springframework.web.servlet.view.BeanNameViewResolver">
   <property name="order" value="#{contentNegotiatingResolver.order+1}" />
</bean>

<!-- La vista "palabras_xls" que es manejada por la clase "PaginaExcel" -->
<bean id="palabras_xls" class="com.bolsadeideas.ejemplos.views.PaginaExcel" />
```

Dependencia Maven

```
<!-- Excel View support -->
<dependency>
   <groupId>org.apache.poi</groupId>
   <artifactId>poi-ooxml</artifactId>
   <version>3.12</version>
</dependency>

<dependency>
   <groupId>org.apache.poi</groupId>
   <artifactId>poi</artifactId>
   <version>3.12</version>
</dependency>
```

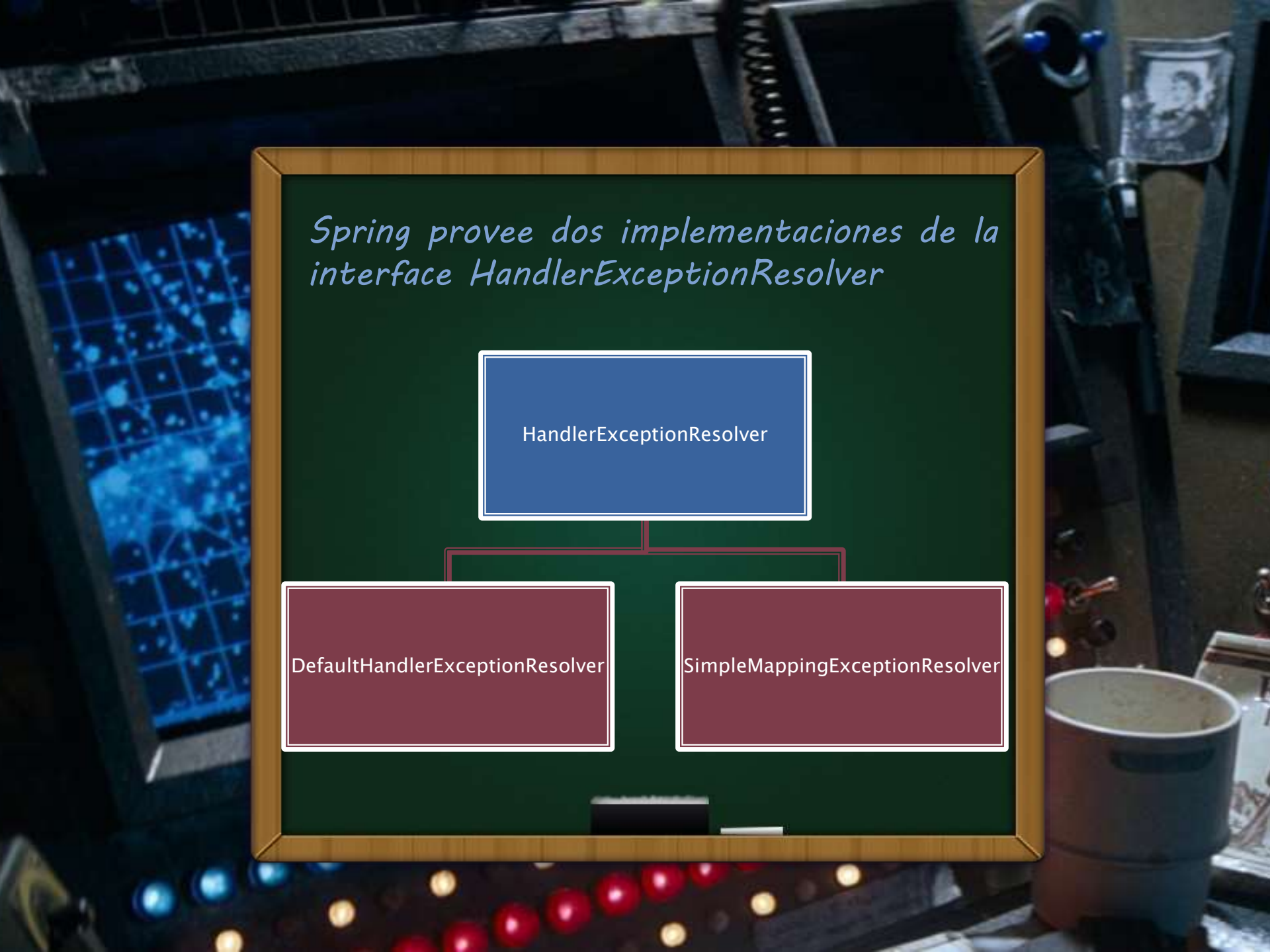
Manejo de Excepciones

Dos formas de manejar Excepciones

Agregando un HandlerExceptionResolver

Usando anotación @ExceptionHandler


Interface HandlerExceptionResolver
Similar al manejo de excepciones que se configuran y mapean en el web.xml. Sin embargo, spring provee una foma más simple y flexible para manejar excepciones
• Proporciona información sobre el controlador donde ocurre la excepción.
• Por otra parte, provee una forma programática para el manejo de excepciones, con más y mejores opciones para responder apropiadamente antes de que el request sea reenviado a otra URL


Spring provee dos implementaciones de la interface HandlerExceptionResolver
HandlerExceptionResolver
DefaultHandlerExceptionResolver
SimpleMappingExceptionResolver

DefaultHandlerExceptionResolver

Por defecto, el DispatcherServlet registra automáticamente el DefaultHandlerExceptionResolver

Esta resolución se encarga de ciertas excepciones estándares en Spring MVC estableciendo un código específico de respuesta (response status code)

- ConversionNotSupportedException-> 500
- HttpMediaTypeNotAcceptableException-> 406
- HttpMediaTypeNotSupportedException-> 415
- NoSuchRequestHandlingMethodException-> 404
- TypeMismatchException-> 400 (Bad Request)
- MissingServletRequestParameterException-> 400

SimpleMappingExceptionResolver

Esta resolución permite tomar el nombre de clase de cualquier excepción que pudiera ser lanzada y mapearla a una vista

"DefaultErrorView" es el atributo donde se define la vista de error por defecto

En el siguiente ejemplo, la excepción RecursoNoEncontradoException resultará en la visualización de la vista "cuentaNotExiste.jsp"

SimpleMappingExceptionResolver

```
<!-- View resolver de las Excepciones-->
<!-- Mapeando las Excepciones -->
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResol
ver">
   <property name="exceptionMappings">
      <props>
         <prop
key="com.bolsadeideas.ejemplos.cuenta.controllers.RecursoNoEncontradoExce
ption">cuentaNotExiste</prop>
         <!-- Definimos todas las clases de excepciones en particular y el property
defaultErrorView (para los errores en general) -->
         <!-- <prop key="java.lang.Exception">error</prop> -->
      </props>
   </property>
   <property name="defaultErrorView" value="error" />
</bean>
```

cuentaNotExiste.jsp es la vista que veremos a continuación

cuentaNotExiste.jsp

Los atributos del objeto Exception pueden ser accedidos desde la vista

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
<head>
<title>La Cuenta NO Existe</title>
</head>
<body>
<h3>La cuenta no existe, está tratando de acceder a una cuenta no registrada en nuestro sistema.</h3>
<p>Si ve esta página es porque el manejo y control de excepciones (errores) está funcionando perfectamente, tal cómo se esperaba</p>
<br/>
El número de cuenta ${exception.recursoId} no existe!
<fmt:formatDate value="${exception.fecha}" pattern="yyyy-MM-dd" />
</body>
</html>
```

Clase Exception

```
public class RecursoNoEncontradoException extends
RuntimeException {

   private Long recursoId;
   private Date fecha;

   public RecursoNoEncontradoException(Long recursoId) {
      this.recursoId = recursoId;
      fecha = new Date();
   }

   public Date getFecha() { return fecha; }

   public void setFecha(Date fecha) { this.fecha = fecha; }

   public Long getRecursoId() { return recursoId; }

}
```

Lanzando una Exception en el @Controller

```
@Controller
@RequestMapping(value="/cuenta")
public class CuentaController {
...
   @RequestMapping(value="{id}", method=RequestMethod.GET)
   public String verDetalle(@PathVariable Long id, Model model) {
      Cuenta cuenta = this.cuentas.get(id);
      if (cuenta == null) {
         throw new RecursoNoEncontradoException(id);
      }
      model.addAttribute("cuenta", cuenta);
      return "cuenta/detalle";
   }

}
```

Vista error por defecto

```
<html>
<head>
<title>Página de Error</title>
</head>

<body>
<p>Ha ocurrido un error. Por favor, póngase en contacto con nuestro administrador para obtener más detalles.</p>
<p>(Estamos en la página o vista error.jsp.)</p>
</body>
</html>
```

Locale en Spring

Seam nos permite contar con un sistema de multidioma para nuestros proyectos web

Soporte Locale en Spring

Si queremos implementar un proyecto web que apunte hacia un mercado globalizado, necesariamente necesitamos contar con una herramienta que nos resuelva el problema

Dos formas de Locale

Agregando un LocaleResolver

Usando Interceptor

A través de LocaleResolver

DispatcherServlet resuelve automáticamente los mensajes que utilizan la configuración regional del cliente a través del bean LocaleResolver

Cuando llega una petición, el DispatcherServlet busca una resolución del locale (configuración regional), y si encuentra uno, trata de usarlo para establecer la configuración regional.

A través de LocaleResolver

Spring ofrece clases de implementación de los siguientesLocaleResolver

- AcceptHeaderLocaleResolver
- CookieLocaleResolver
- SessionLocaleResolver

Utilizando el método RequestContext·getLocale() podemos obtener la configuración regional que fue resuelto por el locale resolver

AcceptHeaderLocaleResolver

- Esta resolución del locale inspecciona el header "accept-language" en el request que fue enviado por el cliente (por ejemplo, un navegador web)
- Por lo general, el atributo "accept-language" contiene la configuración regional del sistema operativo del cliente

CookieLocaleResolver

- Esta resolución del locale inspecciona una cookie que podría existir en el cliente para ver si se encuentra especificado el locale
- En los atributos de este locale resolver, podemos especificar el nombre de la cookie y el tiempo de expiración

Configuracion CookieLocalResolver

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
>
   <property name="cookieName" value="clientlanguage"/>
   <!-- en segundos. Si es -1, la cookie no será persistente (se
eliminan apenas se cierra el navegador) -->
   <property name="cookieMaxAge" value="100000">
</bean>
```

A través de Interceptor

Podemos habilitar el cambio del locales agregando el bean interceptor LocaleChangeInterceptor aplicado a todos los controladores anotados @Controller

Se detectará un parámetro en el request y cambiará la configuración regional o locale, por defecto el nombre del parámetro es "locale", ejemplo ?locale=es

Spring provee una implementación para la clase Locale Interceptor: org.springframework.web.servlet.i18n.LocaleChangeInterceptor

Configuración LocaleChangeInterceptor

```
<!-- Interceptores que son aplicados a todos los controladores anotados
@Controller -->
<mvc:interceptors>
   <!-- Cambiamos el local cuando envia un parametro del request 'locale' e.j.
/?locale=es -->
   <bean
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
      <property name="paramName" value="siteLanguage"/>
   </bean>
</mvc:interceptors>

<!-- Guardamos los cambios del local usando cookie -->
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver" />
```

Entonces mediante un mecanismo integrado vía interfaz de usuario podemos cambiar la configuración regional

El usuario ajusta el locale

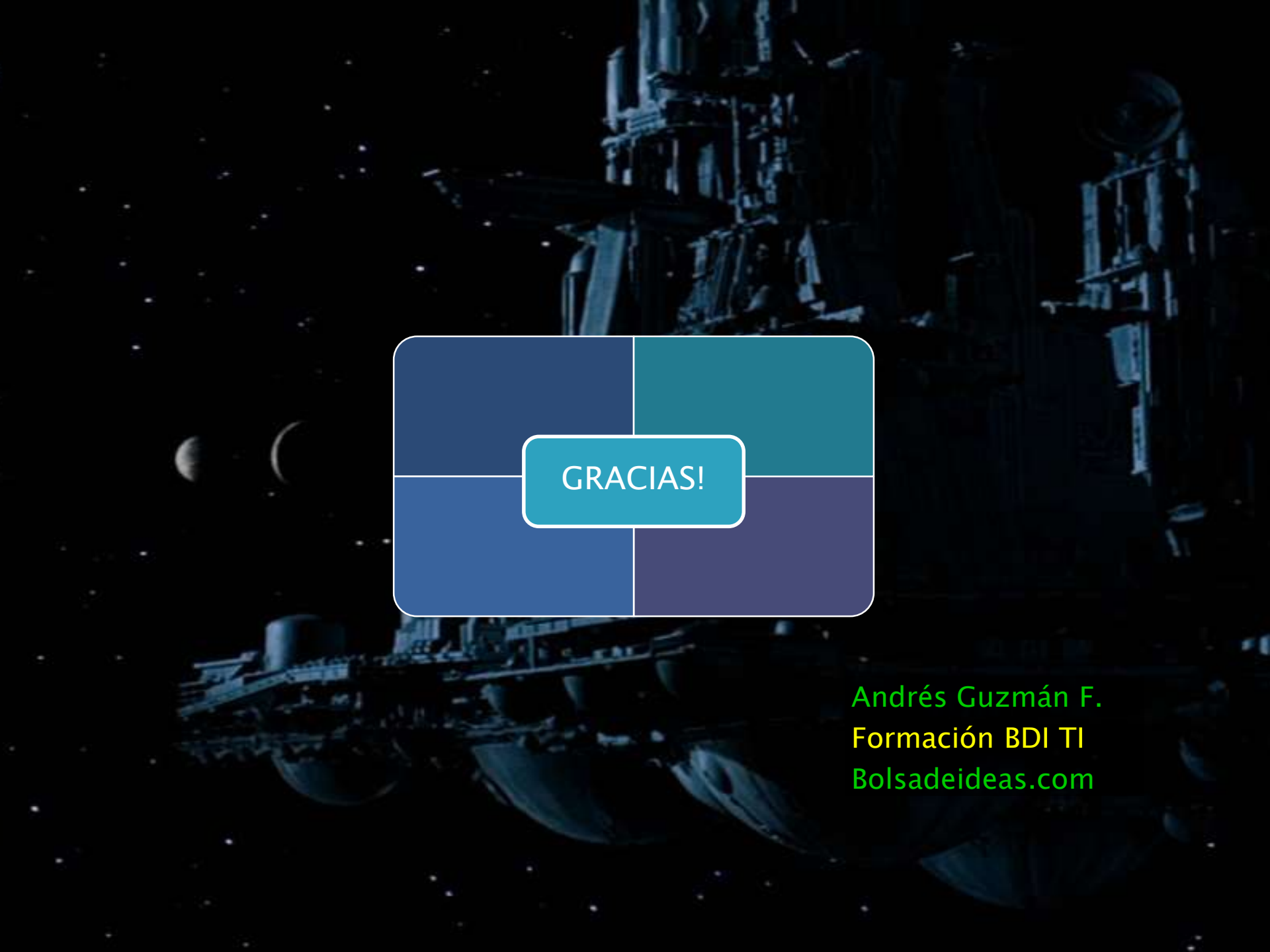
a través de un formulario o url con parámetro

en la aplicación web

LANGUAGE

ENGLISH

DEUTSCH



Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com