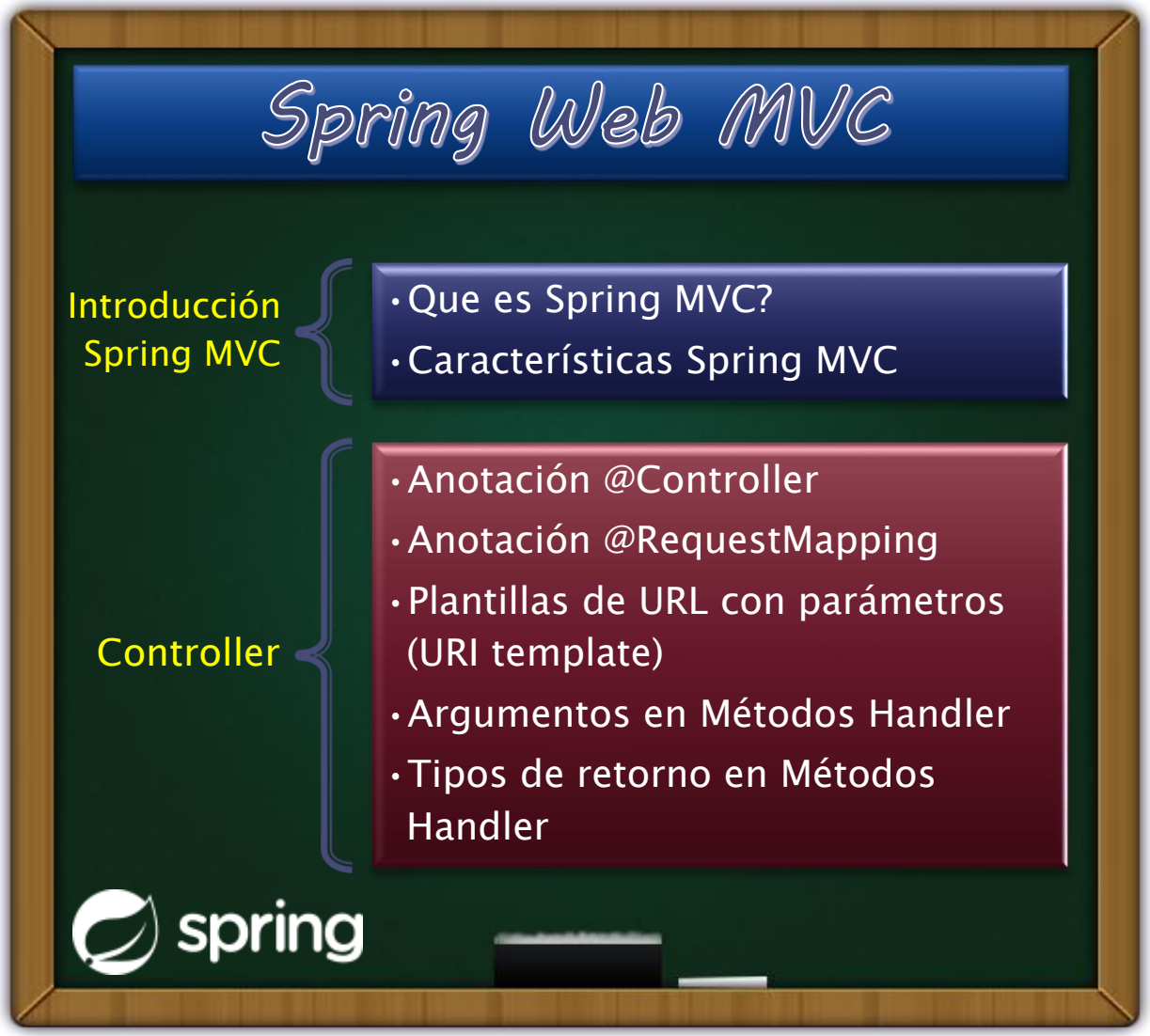

Spring Web MVC
Introducción Spring MVC
Que es Spring MVC?
Características Spring MVC
Controller
Anotación @Controller
Anotación @RequestMapping
Plantillas de URL con parámetros (URI template)
Argumentos en Métodos Handler
Tipos de retorno en Métodos Handler
spring

¿Que es Spring MVC?

¿Que es Spring MVC?

Un Framework de aplicaciones web basado en MVC que toma ventajas de los siguientes principios de diseño

- Inyección de Dependencia
- Orientado al uso de Interfaces
- Extenso uso de clases POJO
- Desarrollo testeable

Características Spring MVC

Características de spring MVC

- Clara separación de funciones
  - ✓ Controller, validator, command object (form object), DispatcherServlet, handler mapping, view resolver, etc.
  - ✓ Llevan a cabo una tarea específica y pueden ser reemplazables sin afectar a los demás
- Configuración Robusta pero sencilla de realizar
  - ✓ Tanto para las clases del framework como las clases propias de la aplicación como JavaBeans

Características de spring MVC

- Adaptabilidad, no intrusivo, y flexibilidad
  - ✓ Diferentes formas de definir los métodos en los controladores, diversas firmas de métodos que necesitemos implementar para un escenario determinado, por ejemplo usando anotaciones en los argumentos (tales como @RequestParam, @RequestHeader, @PathVariable, etc)
- ✓ Conversiones Configurables (binding) y validaciones

Características de spring MVC

- Mapeo URL Configurable de los controladores (handler mapping) y resolución de vista (view resolution)
  - ✓ Diferentes estrategias para la resolución de vista, por ejemplo configuración basada en URL.

- Transferencia del objeto model, desde el controlador a la vista y viceversa
  - ✓ Usando Map (llave/valor) fácil integración con cualquier tecnología de vista

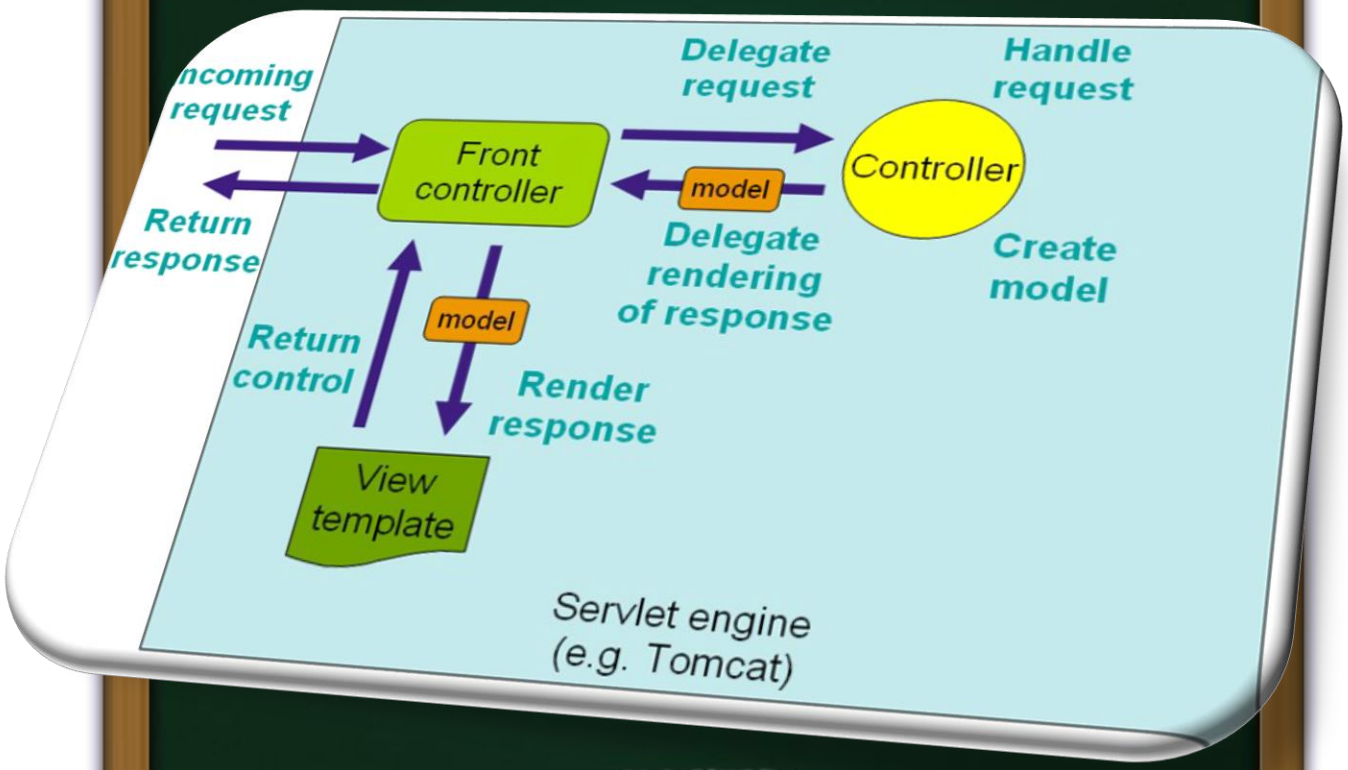
- Personalizable Locale y resolución tema
- Potente librería de etiquetas JSP: Spring tag library

DispatchServlet

¿Qué es DispatcherServlet?

- Juega el papel del Front controller en Spring MVC, similar al filtro Filter Dispatcher de Struts2
- Coordina las peticiones web (HTTP request) y ciclo de vida
- Observa el request y aplica el controller apropiado según la url (handler mapping)
- Configurado internamente por Spring Boot

¿Qué es DispatcherServlet?



Cómo Funciona Spring

El Front Controller DispatchServlet recibe una solicitud HTTP del navegador

El DispatchServlet aplica un Controlador basado en la URL (Handler mapping) y asigna el request al Controller

El Controller se relaciona con componentes de la lógica de negocio y envía datos a la vista usando el objeto Model

El Controller retorna/asigna el nombre de la vista lógica a despachar

Se selecciona un ViewResolver, el cuál aplica un determinado tipo de vista (JSP, Thymeleaf, PDF, Excel, etc.)

Finalmente la vista es mostrada al cliente usando los valores del objeto Model

Controllers
Spring MVC

¿Qué hace un controlador?

- Los Controladores proporcionan acceso a la lógica de negocio
  - Normalmente, un controlador delega el proceso de lógica de negocio a un conjunto de componentes de servicios

    Los servicios a su vez acceden a las bases de datos mediante la interfaz Dao (Objeto de Acceso a Datos)
- Los Controladores reciben parámetros del usuario (input) y lo convierten en un objeto del modelo, poblando en sus atributos los datos enviados, como resultado de la lógica de negocio

@Controller: Controladores con anotaciones

Recién en Spring 2.5 se introducen los controladores en base a anotaciones

- No necesita extender de ninguna clases base controllers ni tampoco implementar la interfaz Controller
- Los controladores en base a anotaciones tampoco tienen directa dependencia de los Servlet o Portlet
- En otras palabras los controladores anotados son clases POJO anotados, no heredan de nada
- Usamos las anotación @Controller para definir nuestros controladores en Spring

Ejemplo Controlador Anotado

- Controlador basado en anotación no tiene que extender de las clases base Spring ni implementar interfaces específicas::

```
@Controller
public class HolaMundoController {

    // No hay reglas ni limitaciones en el nombre y firma del método
    @RequestMapping("/hola")
    public ModelAndView holaMundo() {
        ModelAndView mav = new ModelAndView();
        // asignamos la vista (sólo el nombre)
        mav.setViewName("inicio");
        // agregamos un atributo a la vista
        mav.addObject("mensaje", "Hola Mundo!");
        return mav;
    }
}
```

Anotación @RequestMapping

- @RequestMapping es usada para mapear las URLs hacia métodos handler de una clase Controller

- La anotación @RequestMapping se puede especificar:
  - A nivel de la clase
  - A nivel de método

Anotación @RequestMapping

- "La URL especificada en la anotación @RequestMapping a nivel de clase" se concatena con "la URL especificada en la anotación @RequestMapping a nivel de método vía atributo de value"

  - ✓ Entonces la URL especificada en la anotación @RequestMapping a nivel de método es relativa a la URL especificada en la anotación @RequestMapping a nivel de clase

Anotación @RequestMapping

```
@Controller
@RequestMapping("/agenda") // Usada a nivel de clase
public class AgendaController {

private IAgendaDao agendaDao;

    @Autowired
    public AgendaController(IAgendaDao agendaDao) {
        this.agendaDao = agendaDao;
    }

    // Ejemplo http://locahost:8080/agenda
    @RequestMapping(method = RequestMethod.GET) // Usada a nivel de método
    public Map<String, Agenda> getTodas() {
        return agendaDao.listarTodas();
    }

    // Ejemplo http://locahost:8080/agenda/4 o
    // http://locahost:8080/agenda/5
    @RequestMapping(value="/{day}", method=RequestMethod.GET) //Usada a nivel método
    public Map<String, Agenda> getPorDia (@PathVariable
                                          @DateTimeFormat(iso=ISO.DATE) Date dia,
                                          Model model) {

        return agendaDao.getReunionesPorDia(dia);
    }
```

... Continúa en la siguiente página

Anotación @RequestMapping

... Continuación de la diapositiva anterior

```
    // Ejemplo http://locahost:8080/agenda/nueva
    // Muestra al usuario el formulario en pantalla
    @RequestMapping(value="/nueva", method=RequestMethod.GET)
    public Agenda form() {
        return new Agenda();
    }

    // Ejemplo http://locahost:8080/agenda/nueva
    // Procesa el envío del formulario
    @RequestMapping(value="/nueva", method=RequestMethod.POST)
    public String crear(@Valid Agenda agenda, BindingResult result) {

        if (result.hasErrors()) {
            return "agenda/nueva";
        }
        agendaDao.guardar(agenda);
        return "redirect:/agenda";
    }
}
```

Anotación @RequestMapping sólo a nivel de métodos

"No es requerida la anotación @RequestMapping a nivel de clase. Sin esta, todas las rutas son absolutas, y no relativas

```
@Controller
public class ClinicController {

    private Clinic clinic;

    @Autowired
    public ClinicController(Clinic clinic) {
        this.clinic = clinic;
    }

    // Maneja http://locahost:8080/
    @RequestMapping("/")
    public void welcomeHandler() {}

    // Maneja http://locahost:8080/vets
    @RequestMapping("/vets")
    public ModelMap vetsHandler() {
        return new ModelMap(this.clinic.getVets());
    }
}
```

URI Template (Plantillas URL)

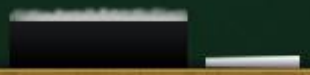
¿Qué es una URI Template?

- Una URI Template es un string URL que contiene uno o más nombres de variables (variables de url)
  - Variables que tienen forma de {nombreDeVariable}

```
@RequestMapping(value="/cliente/{clienteId}"))
```

  - El nombreDeVariable necesita ser pasado como argumento del método handler acompañado de la anotación @PathVariable

```
public String findCliente(@PathVariable String clienteId, Etc ...)
```

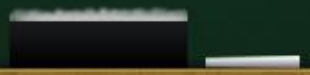


¿Qué es una URI Template?

- Cuando se substituyen los valores de esas variables, el URI template se convierte en una URL concreta

/cliente/3
/cliente/5

- El nombreDeVariable necesita ser pasado como argumento del método handler acompañado de la anotación @PathVariable
- Entonces, el request URI es comparado con alguna plantilla URI (URI Template) y si coincide se invoca el metodo handler correspondiente



¿Qué es una URI Template?

- Supongamos el siguiente request URL

http://locahost:8080/cliente/3

- El valor 3 será capturado en el argumento "clienteId" del tipo String.

```
@RequestMapping(value="/cliente/{clienteId}", method=RequestMethod.GET)
public String findCliente(@PathVariable String clienteId, Model model) {
   // Ahora podemos usar la variable clienteId
   // en nuestra lógica de negocio
   Cliente cliente = clienteDao.findPorId(clienteId);
   model.addAttribute("cliente", cliente);
   return "cliente/detalle";
}
```

¿Qué es una URI Template?

- Podemos usar múltiples anotaciones @PathVariable para capturar múltiples variable URI Template

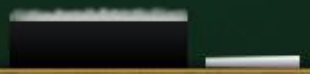
- Supongamos el siguiente request URL

http://locahost:8080/cliente/3/factura/5

  - El valor 3 será capturado en el argumento "clienteId" del tipo String
  - El valor 5 será capturado en el argumento "facturaId" del tipo String

¿Qué es una URI Template?

```
@RequestMapping(value="/cliente/{clienteId}/factura/{facturaId}",
method=RequestMethod.GET)
public String findFactura(@PathVariable String clienteId,
                          @PathVariable String facturaId,
                          Model model) {

   Cliente cliente = clienteDao.findPorId(clienteId);
   Factura factura = cliente.getFactura(facturaId);
   model.addAttribute("factura", factura);
   return "cliente/factura";

}
```



Argumentos en Métodos Handler

Objetos auto-creados por Spring

Podemos usar cualquiera de estos objetos como argumentos del los métodos handler del controlador. Estos serán creados automáticamente por Spring y pasados al método:

- HttpServletRequest o HttpServletResponse
  - Objeto Request o response (Servlet API)
- HttpSession
  - Objeto Session (Servlet API)
- java.util.Locale (Configuración Regional)
  - Para obtener el objeto locale actual del request
- java.security.Principal
  - Usuario autenticado

@PathVariable y @RequestParam

@PathVariable

- Parámetros anotados para acceder a variables de las plantillas URL o URI template
- Extrae los datos desde el request URI
- http://host/catalogo/items/123
- Los valores de los parámetros son convertidos y pasados como argumentos en los métodos handler usando la anotación @PathVariable

@PathVariable y @RequestParam

@RequestParam("name")

- Parámetros anotados para acceder a específicos parámetros del Servlet request.
- Extrae los datos desde request query parameters URL
- http://host/catalogo/items/?name=abc
- Los valores de los parámetros son convertidos y pasados como argumentos en los métodos handler usando la anotación @RequestParam

@PathVariable valores URI Path

```
// Usamos la anotación @PathVariable para poblar valores
// en parámetros de métodos del controlador
@Controller
@RequestMapping("/catalogo")
public class CatalogoController {

   // Ejemplo ../catalogo/4 o ../catalogo/10
   // 4 y 10 son convertidos a tipos int por Spring.
   @RequestMapping(value="/{prodId}")
   public String getData(@PathVariable int prodId,
                         ModelMap model) {
      Producto producto = productoDao.getProductoPorId(prodId);
      model.addAttribute("producto", producto);
      return "form";
   }
// ...
```

@RequestParam para valores Query Parameters

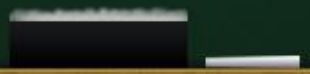
```
// Usamos la anotación @RequestParam para poblar
// valores query request en parámetros de métodos del controlador
@Controller
@RequestMapping("/catalogo")
public class CatalogoController {

   // Ejemplo ../catalogo?pordId=4 or ../catalogo?pordId=10.
   // 4 y 10 son convertidos a tipos int por Spring.
   public String getData(@RequestParam("prodId") int prodId,
                         ModelMap model) {
      Producto producto = productoDao.getProductoPorId(prodId);
      model.addAttribute("producto", producto);
      return "form";
   }
// ...
```

Request Header y Body

@RequestHeader("name")	@RequestBody
• Parámetros anotados para acceder a específicas cabeceras HTTP (request HTTP headers)	• Parámetros anotados para acceder al cuerpo HTTP (request HTTP body) • Los valores de los parámetros son convertidos y pasados como argumentos en los métodos handler usando HttpMessageConverter

@RequestBody

```
// La anotación @RequestBody indica que un argumento del método
// deberá ser poblado con el valor del HTTP request body
@RequestMapping(value = "/algunaurl", method =
RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer)
throws IOException {
    writer.write(body);
}
```



Tipos de objeto Model como argumentos en Métodos Handler

Implícitos tipos de Models como argumentos Handler

Un objeto Model es creado por Spring y es pasado como argumento en los métodos handler del controlador

- Podemos asignar atributos a la vista/form vía llave/valor

Los objetos Models son expuesto en las vistas (ej. archivo jsp o html)

- Vista puede acceder a estos modelos (atributos del objeto model) usando lengiaje de expreciones (EL)

Implícitos Models como argumentos Handler

Tipos de Model soportados:

- java.util.Map: el más genérico y típico de todos
- org.springframework.ui.Model: contenedor de atributos para la vista
- org.springframework.ui.ModelMap: Soporta invocación de métodos en cadena y auto-generación de nombres de atributos del modelo

Ejemplo objeto Implícito Model como argumento Handler

```
import org.springframework.ui.Model;

@RequestMapping(value = "/buscar-hoteles", method =
RequestMethod.GET)
public String listar(SearchCriteria criteria, Model model) {

   List<Hotel> hotels = bookingService.findHotels(criteria);
   // Ageregamos un atributo al objeto model. La vista podrá acceder
   // al atributo "hotelList" vía ${hotelList}
   model.addAttribute("hotelList", hotels);
   // Retornamos el nombre lógico de la vista "hotels/list",
   // lo que resulta el despliegue de la vista "hotels/list.jsp".
   return "hotels/list";
}
```

Vista (JSP) que accede al objeto Model

```
<table class="summary">
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Dirección</th>
      <th>Ciudad, Estado</th>
      <th>Detalle</th>
    </tr>
  </thead>
  <tbody>
   <c:forEach var="hotel" items="${hotelList}">
     <tr>
       <td>${hotel.name}</td>
       <td>${hotel.address}</td>
       <td>${hotel.city}, ${hotel.state}, ${hotel.country}</td>
       <td><a href="hotels/${hotel.id}">Ver Hotel</a></td>
     </tr>
   </c:forEach>
  <c:if test="${empty hotelList}">
    <tr>
      <td colspan="5">No se han encontrado hoteles</td>
    </tr>
  </c:if>
  </tbody>
</table>
```

Ejemplo objeto ModelMap como argumento Handler

```
import org.springframework.ui.ModelMap;

@RequestMapping("/deposito")
public String deposito(
                @RequestParam("numeroCuenta") int numeroCuenta,
                @RequestParam("monto") double monto,
                ModelMap modelMap) {

   cuentaDao.depositar(numeroCuenta, monto);
   // Objeto ModelMap permite llamada en cadena
   modelMap.addAttribute("numeroCuenta", numeroCuenta)
                .addAttribute("balance",
                                cuentaDao.getBalance(numeroCuenta));
   return "success";
}
```

Tipos de retorno en Métodos Handler (para Selección de Vista)

Objeto ModelAndView (como tipo de retorno)

```
@Controller
@RequestMapping("/carro")
public class CarroDeComprasController{

   @RequestMapping(value="/ver")
   public ModelAndView verCarro() {
      List itemsCarro = // Obtenemos la lista de items del carro de compras

      //Creamos el objeto ModelAndView y asignamos el nombre de la vista
      //lo que resulta el despliegue de la vista "catalogo/verCarro.jsp".
      ModelAndView mav = new ModelAndView("catalogo/verCarro");

      // Agregamos atributos al objeto ModelAndView
      // La vista podrá acceder al atributo "itemsCarro" vía ${itemsCarro}
      mav.addObject("itemsCarro", itemsCarro);

      // Retornamos el objeto ModelAndView
      return mav;
   }
}
```

Objeto String (como tipo de retorno)

Retornamos un String que es interpretado como el nombre lógico de la vista

Es la forma más comúnmente utilizada

```
@Controller
@RequestMapping("/carro")
public class CarroDeComprasController{

   @RequestMapping(value="/ver")
   public String verCarro(Model model) {
      List itemsCarro = // Obtenemos la lista de items del carro de compras

      // Agregamos atributos al objeto Model pasado por argumento
      // La vista podrá acceder al atributo "itemsCarro" vía ${itemsCarro}
      model.addAttribute("itemsCarro", itemsCarro);

      // Retornamos el objeto String con el nombre de la vista
      // lo que resulta el despliegue de la vista "catalogo/verCarro.jsp".
      return "catalogo/verCarro";
   }
}
```

void
(como tipo de retorno)

Cuando el nombre de la vista es implícita, es decir no se define en ninguna parte en el método handler, es determinada mediante RequestToViewNameTranslator, vía Mapping URL (Request Mapping)

```
// El nombre de la vista se establece implícitamente a "nombreVista123"
@RequestMapping(value="/nombreVista123", method=RequestMethod.GET)
public void usandoRequestToViewNameTranslator(Model model) {
   model.addAttribute("foo", "bar");
   model.addAttribute("fruit", "apple");
}
```

Tipos de retorno en Métodos Handler (para crear Respuesta)

Anotación de método @ResponseBody

Si el método es anotado con @ResponseBody, se declara el tipo de retorno como String y su contenido es almacenado en la respuesta dentro del cuerpo HTTP (response HTTP body)

No hay selección de vista

Comúnmente usado para peticiones del tipo AJAX y RESTful (XML, JSON)

```
@RequestMapping(value="/response/annotation", method=RequestMethod.GET)
public @ResponseBody String responseBody() {
  return "Un String ResponseBody";
}
```

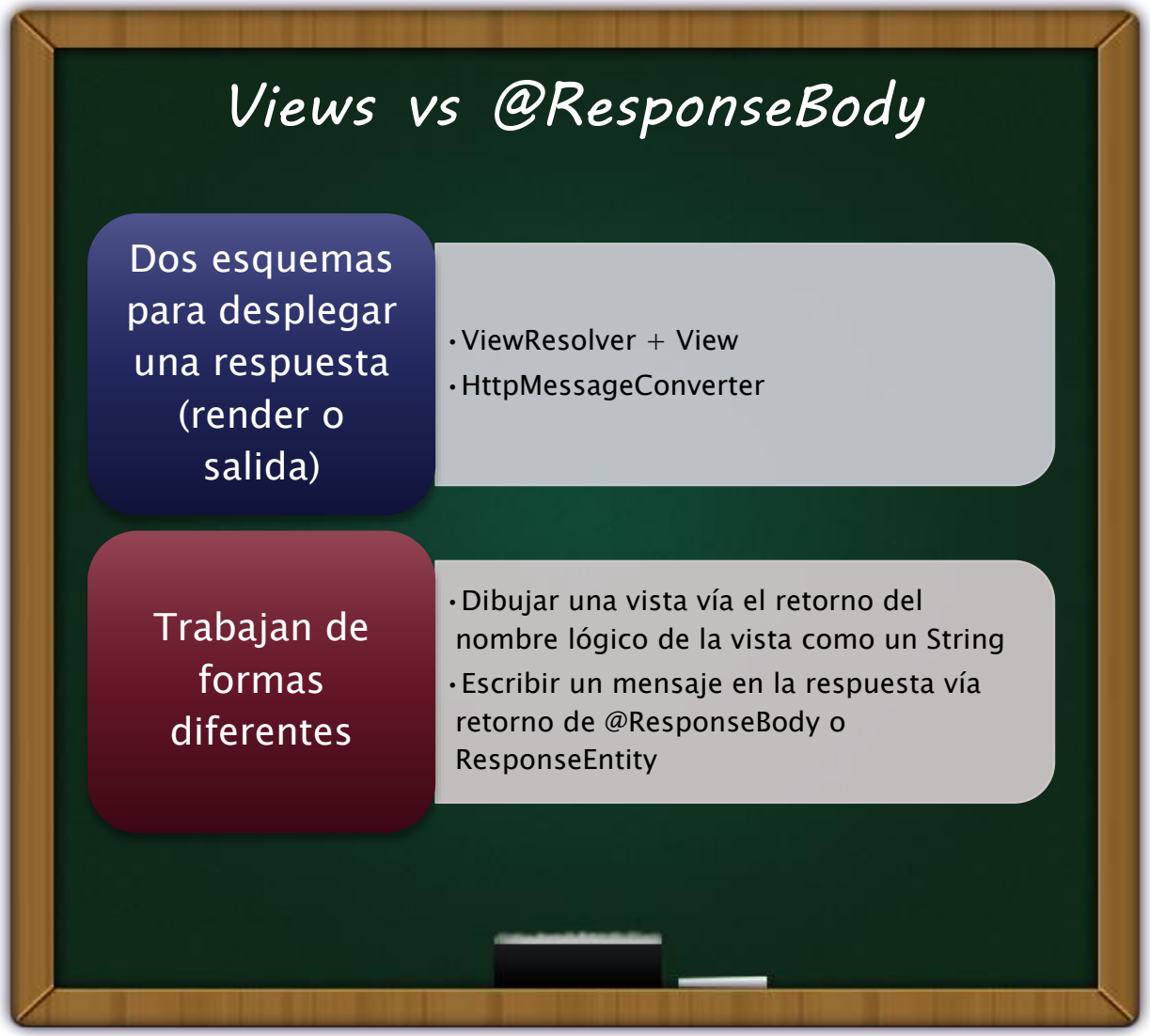
Anotación de método @ResponseBody

Provee acceso hacia las cabeceras HTTP de la Respuesta (Servlet Reponse HTTP Headers)

El entity body será convertido hacia el stream de salida (response stream) usando HttpMessageConverter

```
@RequestMapping(value="/response/entity/headers", method=RequestMethod.GET)
public ResponseEntity<String> responseEntityCustomHeaders() {
   HttpHeaders headers = new HttpHeaders();
   headers.setContentType(MediaType.TEXT_PLAIN);
   return new ResponseEntity<String>("Un String ResponseBody con cabecera
                                      personalizada Content-Type=text/plain",
                                      headers, HttpStatus.OK);
}
```

Views vs @ResponseBody


Views vs @ResponseBody
Dos esquemas para desplegar una respuesta (render o salida)
• ViewResolver + View
• HttpMessageConverter
Trabajan de formas diferentes
• Dibujar una vista vía el retorno del nombre lógico de la vista como un String
• Escribir un mensaje en la respuesta vía retorno de @ResponseBody o ResponseEntity

Views vs @ResponseBody

Cuál usar?

- Usamos ViewResolver + View para generar documentos en el navegador, ejemplo HTML, PDF, XLS etc
- Usamos @ResponseBody para el intercambio de datos con web servicios y Ajax, ejemplo JSON, XML, etc

Anotación @SessionAttributes

Anotación @SessionAttributes

Define atributos de sesión utilizados por un controlador

Contiene una lista de nombres de atributos del objeto view model, cuyos valores u objetos son almacenados transparentemente en la sesión HTTP, típicamente beans asociados a un formulario (objeto comando o de formulario) para que sean persistentes y accesibles entre request (solicitudes posteriores)

Ideal para formularios

```
@Controller
@RequestMapping("/editarUsuario")
@SessionAttributes("usuario")
public class EditarUsuarioController {
   // ...
}
```