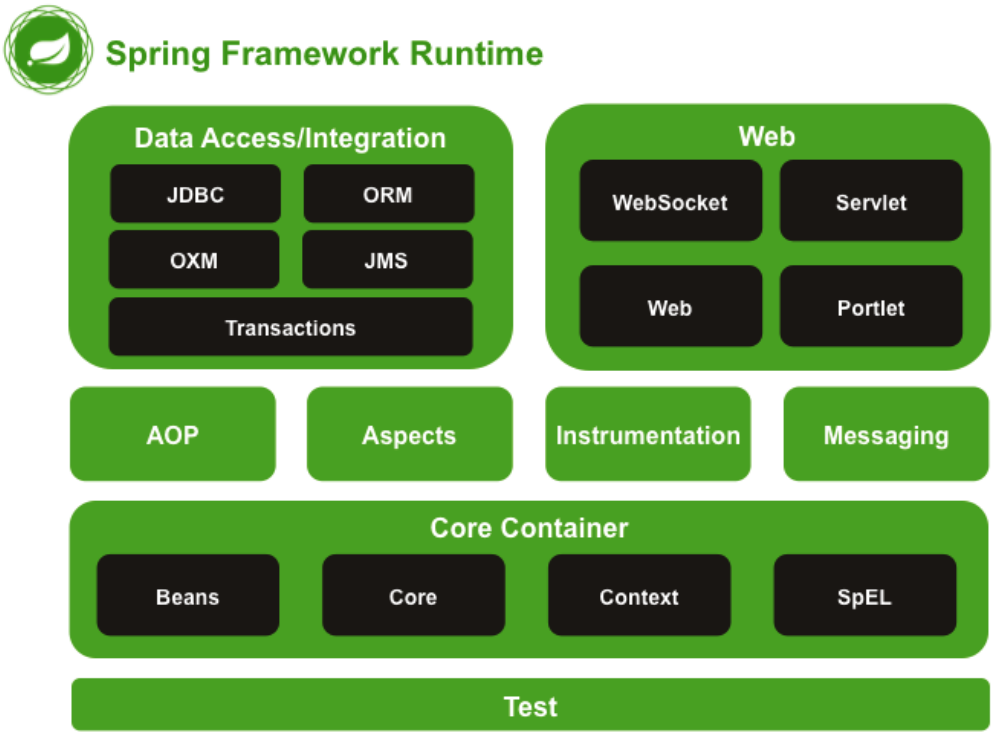
GUSTAVO CORONEL
DESARROLLA SOFTWARE

DESARROLLO WEB

CON SPRING BOOT



UNIDAD 04

SPRING MVC

Eric Gustavo Coronel Castillo
www.desarrollasoftware.com
gcoronelc@gmail.com
I N S T R U C T O R

CONTENIDO

FUNDAMENTOS

ARQUITECTURA MVC

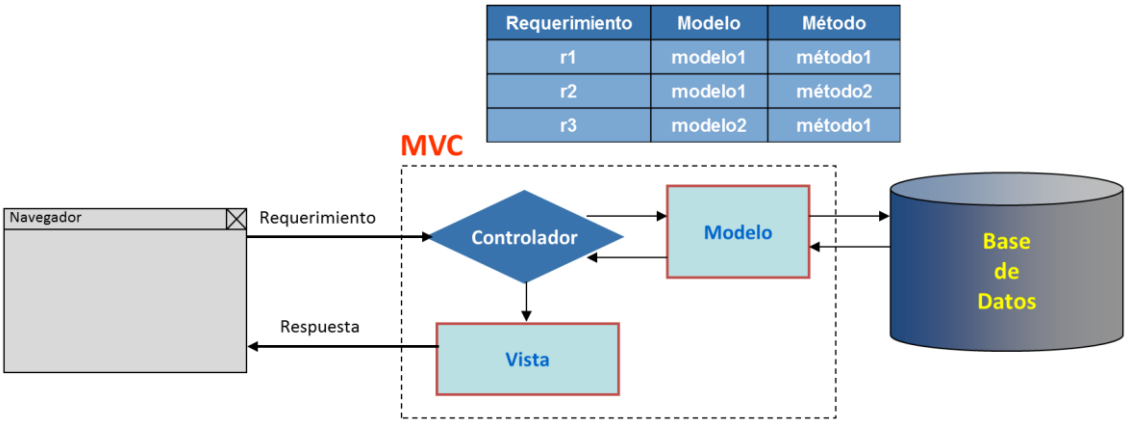


Figura 1

El patrón MVC (Figura 1) divide la aplicación en tres tipos de componentes:

- **Vista:** Se encarga de generar el código HTML que se envía al navegador, normalmente se implementa con paginas JSP y JSTL
- **Modelo:** Se encarga de resolver la lógica de negocio y si es necesario accede a la fuente de datos. Normalmente se implementa con POJOs.
- **Controlador:** Se encarga de recibir los requerimientos del cliente, elige un modelo y método (servicio) que resuelve el requerimiento, obtiene la respuesta y la envía al view para general el HTML que se envía al browser. El controlador se implementa normalmente con Servlets.

MODELO AMPLIADO

Requerimiento	Modelo	Método
r1	modelo1	método1
r2	modelo1	método2
r3	modelo2	método1

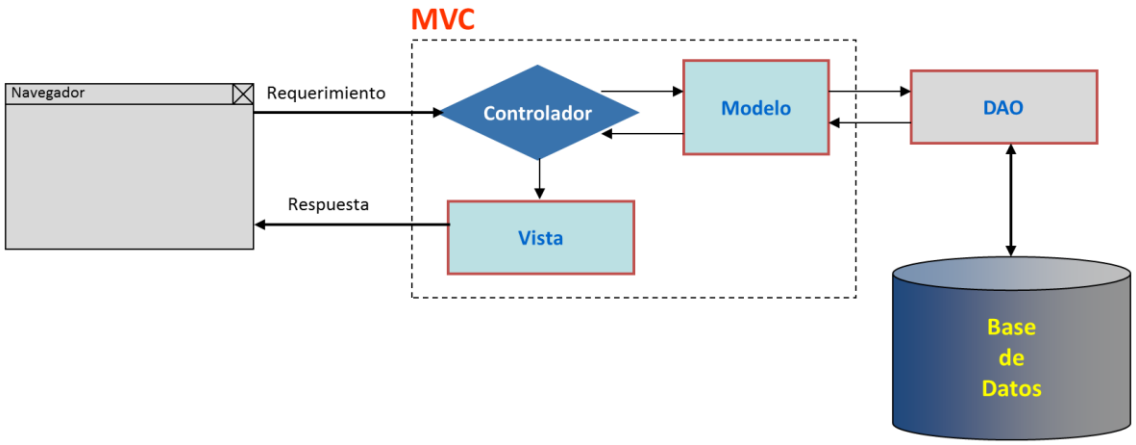


Figura 2

En este modelo ampliado (Figura 2) se agrega el patrón DAO para que implemente la lógica de persistencia.

EL ENFOQUE DE SPRING MVC

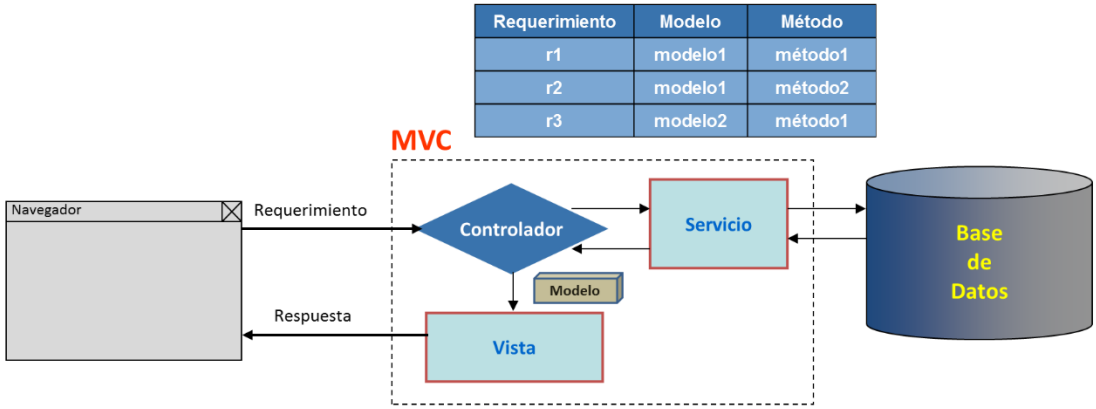


Figura 3

En el enfoque de Spring se tiene los siguientes cambios:

- El Modelo (Componente Model del MVC) se convierte en un componente de servicios (Service).
- El Modelo (Model) representa el componente que encapsula los datos que se deben comunicar entre la Vista y el Controlador. La librería Spring MVC ya implementa una clase Model para este fin.

ARQUITECTURA Y CONFIGURACIÓN

Arquitectura

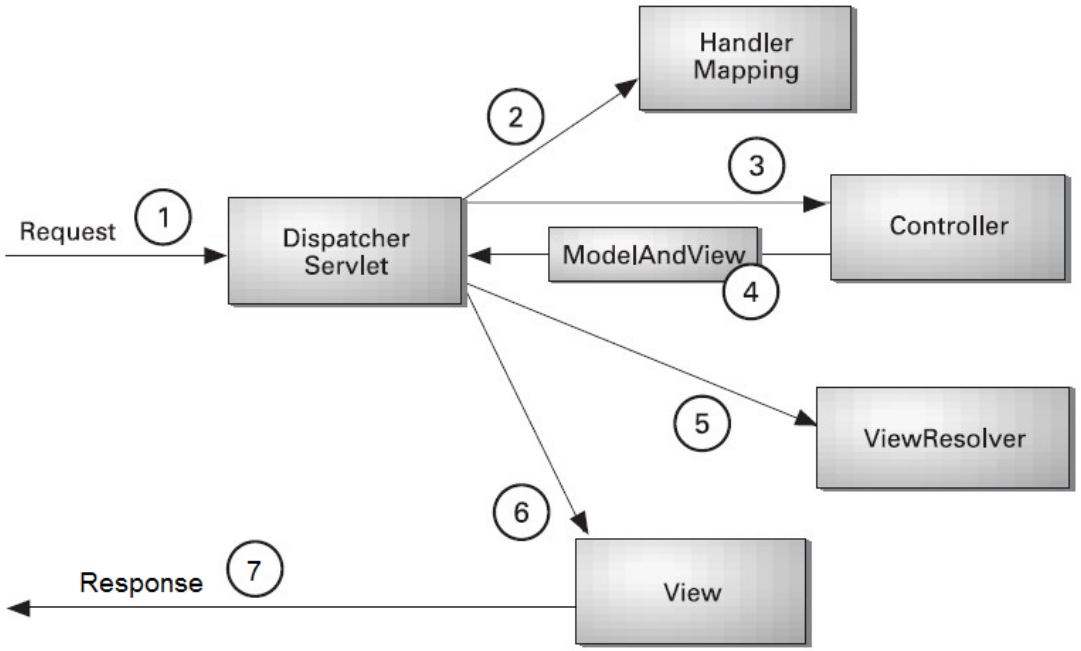


Figura 4

En la Figura 4 se tiene la arquitectura del funcionamiento de una aplicación con Spring MVC.

Se puede identificar claramente que en Spring MVC, el servlet **DispatcherServlet** funciona bajo el patrón **Front Controller**. El patrón front controller proporciona un punto de entrada único; de manera que todos los request son procesados por un mismo Servlet, en el caso de Spring MVC, se trata de **DispatcherServlet**. Este servlet se va a encargar de gestionar toda la lógica en la aplicación.

El flujo básico en una aplicación bajo Spring MVC es el siguiente:

1. El request llega al **DispatcherServlet** (1)
2. El DispatcherServlet tendrá que encontrar el controlador que va a tratar el request. Para ello el DispatcherServlet tiene que encontrar el manejador asociado a la URL del request. Todo esto se realiza en la fase de **HandlerMapping** (2).
3. Una vez encontrado el Controller, el DispatcherServlet le dejará gestionar a éste el request (3). En el controlador se deberá realizar toda la lógica de negocio correspondiente al request, es decir, aquí se llamará a la capa de servicios. El controlador devolverá al Dispatcher un objeto de tipo **ModelAndView**. El **Model**

representa los valores que se obtienen de la capa de servicio y **View** será el nombre de la vista en la que se debe mostrar la información que va contenida dentro de ese Model.

4. Una vez pasado el objeto **ModelAndView** al **DispatcherServlet**, será éste el que tendrá que asociar el nombre de la vista retornada por el controlador a una vista concreta, en este caso una página **JSP**. Este proceso es resuelto por el ViewResolver (4).
5. Finalmente, y una vez resuelta la vista, el DispatcherServlet tendrá que pasar los valores del **Model** a la vista concreta, en este caso la página **JSP**, View (5).

En la Figura 5, tienes una imagen ampliada de la arquitectura de una aplicación con Spring MVC.

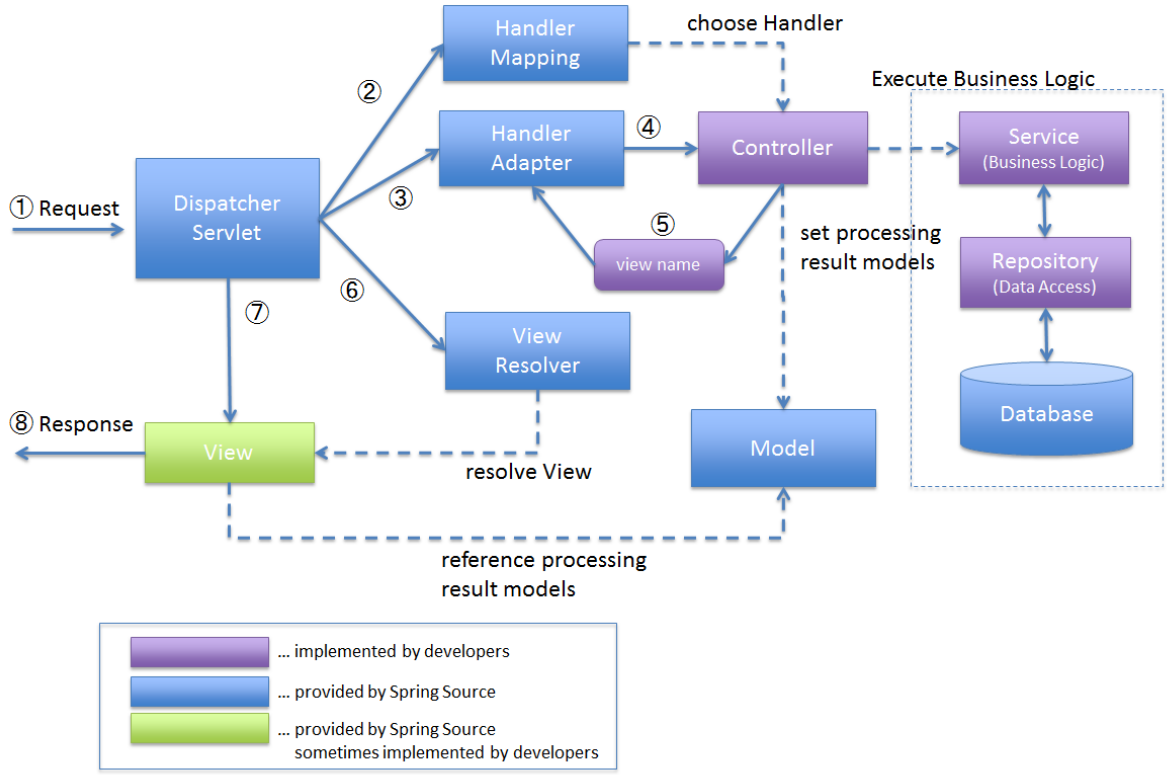
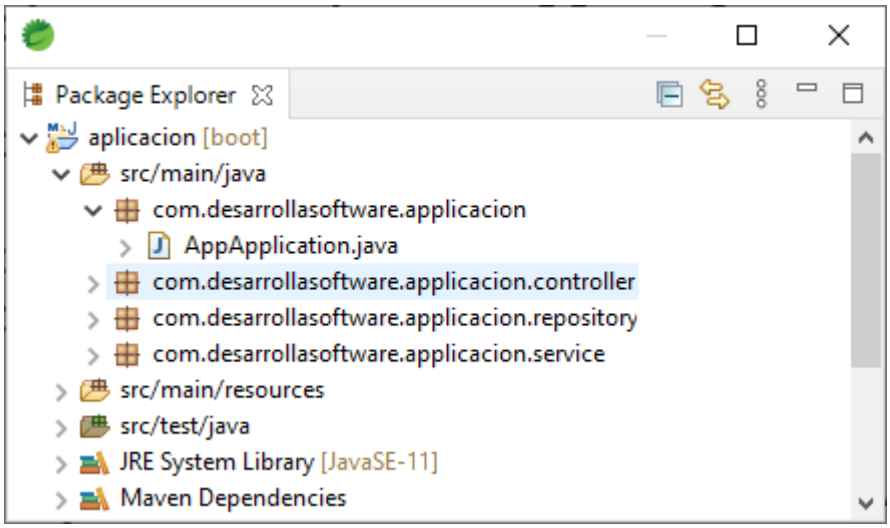


Figura 5

Configuración



El código de la clase principal es:

```
@SpringBootApplication
public class AppApplication {

   public static void main(String[] args) {
      SpringApplication.run(AppApplication.class, args);
   }

}
```

ARCHIVO DE PROPIEDADES

El problema

Necesitamos pasarle ciertos valores a nuestra aplicación o API para que funcione como esperamos

Archivo de propiedades

Una aplicación con Spring Boot cuenta con un archivo de propiedades de nombre **application.properties**, es en este archivo, donde se le pasa todos los datos para que se ejecute correctamente.

A continuación, tiene un ejemplo de lo que podría ser un archivo de propiedades de una determinada aplicación:

```
spring.datasource.url= jdbc:oracle:thin:@datacenter:1521:ORCL
spring.datasource.username=chavo
spring.datasource.password=ocho
spring.datasource.driver.class=oracle.jdbc.OracleDriver
lista.correos = gcoronel@uni.edu.pe,gcoronelc@gmail.com

app.saludo=Bienvenido Gustavo

spring.jpa.hibernate.ddl-auto = validate
server.port=8010
```

Utilizar las variables que definimos

Clase Environment

```
@SpringBootApplication
@RestController
public class AppApplication {

   @Autowired
   private Environment environment;

   public static void main(String[] args) {
      SpringApplication.run(AppApplication.class, args);
   }

   @RequestMapping("/saludo")
   public String saludo() {
      return environment.getProperty("app.saludo");
   }

}
```

También puedes acceder a variables del sistema, por ejemplo:

```
String path = environment.getProperty("path");
```

Permite acceder al contenido de la variable path.

Anotación @Value

Esta anotación permite acceder a las variables definidas en el archivo de propiedades.

Por ejemplo, el siguiente código permite acceder a la variable **app.saludo**:

```
@Value("${app.saludo}")
private String mensaje;
```

El siguiente ejemplo, permite tener acceso a los correos como una lista:

```
@Value("#{'${lista.correos}'.split(',')}")
private List<String> correos;
```

MVC CON PAGINAS JSP

Configuración

En primer lugar, el proyecto debe empaquetar en archivos **WAR**.

Luego, debes crear el folder para los archivos JSP:

Debes incluir la dependencia para que interprete y compile las paginas JSP:

```
<!-- Interpreta y compila JSP -->
<dependency>
   <groupId>org.apache.tomcat.embed</groupId>
   <artifactId>tomcat-embed-jasper</artifactId>
   <scope>provided</scope>
</dependency>
<!-- Interpreta y compila JSP -->
```

Si utilizamos páginas JSP también se necesita la librería para JSTL:

```
<dependency>
   <groupId>jstl</groupId>
   <artifactId>jstl</artifactId>
   <version>1.2</version>
</dependency>
```

En el archivo de propiedades se debe configurar el prefijo y sufijo de las vistas:

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

Programación

En el Script 1 tienes un ejemplo de una clase controladora.

En el objeto de tipo **Model** se tienen los datos que se enviaran al **view**, en este caso se envía un saludo.

El método **home()** retorna el nombre del **view**, en este caso **"home"**, esto quiere decir que en la carpeta "/WEB-INF/jsp" debe existir el archivo **home.jsp**.

Script 1

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {

   @RequestMapping(value = "/", method = RequestMethod.GET)
   public String home( Model model ) {
      model.addAttribute("mensaje", "Hola GUSTAVO CORONEL." );
      return "home";
   }

}
```

Las view son generalmente archivos JSP, para el caso del Script 1, sería por ejemplo un archivo JSP de nombre **home.jsp**, el Script 2 muestra un ejemplo de lo que podría ser la codificación de esta vista.

Script 2

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
   <title>Home</title>
</head>
<body>
   <h1>SALUDO</h1>
   <p>${mensaje}</p>
</body>
</html>
```

Recursos estáticos

Los recursos estáticos los debes ubicar en la carpeta static, tal como lo puedes observar en la Figura 6.

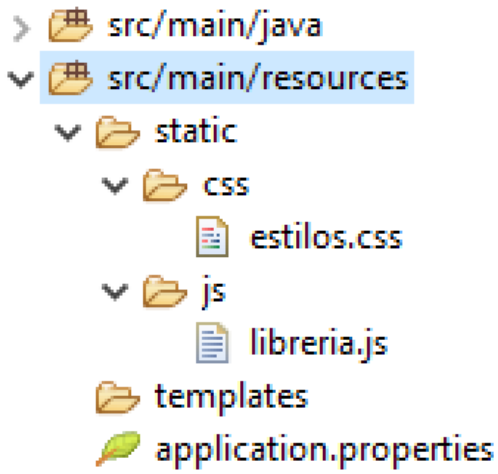


Figura 6

En el Script 3, tienes un ejemplo de cómo hacer referencia a los recursos estáticos.

Script 3

```
<link href="/css/estilos.css" rel="stylesheet">
<script src="/js/libreria.js"></script>
```

PARAMETROS DEL SERVLET

Desde un controlador de Spring se puede tener acceso a los parámetros del servlet.

HttpServletRequest

Un controlador de Spring MVC soporta como parámetro **HttpServletRequest**, de esta manera tienes acceso a por ejemplo a los parámetros que recibe, similar a como se hace en un Servlet.

Script 4

```
@RequestMapping(value="procesarFactura.htm", method=RequestMethod.POST)
public String sumar(HttpServletRequest request, Model model){


}
```

En el Script 4 se tiene un ejemplo de lo que podría ser un controlador que accede al objeto **HttpServletRequest** para acceder a los parámetros.

HttpServletResponse

Un controlador de Spring MVC soporta como parámetro **HttpServletResponse**, de esta manera tienes acceso a generar una salida de manera directa hacia el navegador.

Script 5

```
@RequestMapping(value="procesarFactura.htm", method=RequestMethod.POST)
public String sumar(HttpServletRequest request, HttpServletResponse response){


}
```

En el Script 5 se tiene un ejemplo de lo que podría ser un controlador que accede al objeto **HttpServletRequest** para acceder a los parámetros y HttpServletResponse para generar una salida directa al navegador.

Ejemplo Ilustrativo

Script 6

```
@RequestMapping(value = "venta.htm", method = RequestMethod.GET)
public void venta(HttpServletRequest request, HttpServletResponse response)
throws IOException {

  // Datos
  double precio = Double.parseDouble(request.getParameter("precio"));
  int cant = Integer.parseInt(request.getParameter("cant"));

  // Proceso
  double importe = precio * cant;

  // Reporte
  PrintWriter out = response.getWriter();
  response.setContentType("text/html");
  out.println("<h1>VENTA</h1>");
  out.println("<p>Precio: " + precio + "</p>");
  out.println("<p>Cant: " + cant + "</p>");
  out.println("<p>Importe: " + importe + "</p>");

}
```

En el Script 6 se tiene un ejemplo ilustrativo de cómo utilizar parámetros **HttpServletRequest** y **HttpServletResponse** en un controlador.

MODEL Y MODELANDVIEW

Model y View

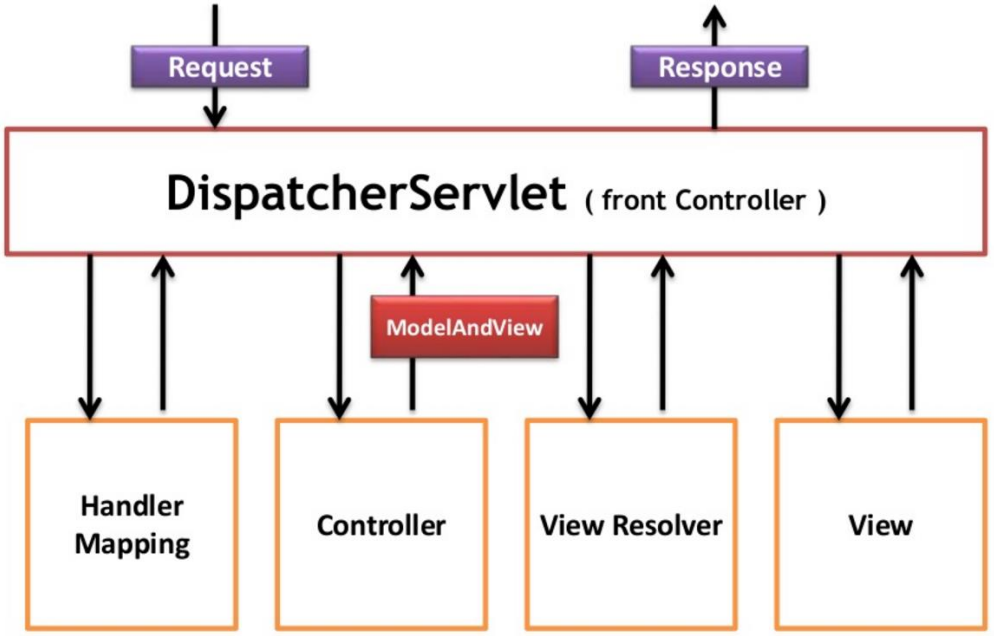


Figura 7

El **Model** y **View** son los elementos básicos con los que trabaja un controlador, tal como se puede apreciar en la Figura 7. El modelo almacena los datos del proceso y la vista decide su representación.

Un modelo en Spring es simplemente un array asociativo. Es decir, una colección de pares <clave,valor>. Lo que en Java corresponde con el tipo Map. Esto permite una gran flexibilidad, ya que dicho modelo se puede convertir fácilmente a cualquier otra clase según la tecnología que se quiera utilizar, como por ejemplo al formato de atributos que espera una página JSP.

La vista en Spring es una cadena de texto con un nombre. El mecanismo de resolución de dicho nombre es totalmente configurable, y la salida puede ser el resultado de aplicar una plantilla JSP, o una salida personalizada utilizando XML, JSON, o cualquier otro tipo de formato.

Interface Model

Normalmente, esta interfaz se utiliza como parámetro de un controlador para comunicar el modelo de datos que se debe enviar a la vista.

Script 7

```
@RequestMapping(value = "procesar.htm", method = RequestMethod.POST)
public String sumar(HttpServletRequest request, Model model) {

  . . .

  return "nombreVista";
}
```

El Script 7 ilustra un caso de cómo se puede usarse esta interfaz:

- A través del parámetro **request** recibe los parámetros.
- A través del parámetro **model** retorna los datos para la vista.
- Con la sentencia **return** retorna el nombre de la vista.

Clase ModelAndView

Esta clase se debe utilizar cuando se quiere retornar el nombre del view y el modelo de datos como un solo objeto.

Script 8

```
@RequestMapping(value = "procesar.htm", method = RequestMethod.POST)
public ModelAndView sumar(HttpServletRequest request) {

  ModelAndView mav = new ModelAndView("nombreVista");

  . . .

  return mav;
}
```

El Script 8 ilustra un caso de cómo se puede usarse esta clase:

- A través del parámetro **request** recibe los parámetros.
- A través de un objeto de tipo **ModelAndView** comunica el nombre de la vista y el modelo de datos.

MAPEO DE REQUERIMIENTOS

Definiendo Estándares

CLIENTES	EMPLEADOS	PRODUCTOS
/clientes	/empleados	/productos
/listado /traerPorId /nuevo /editar /grabar /eliminar	/listado /traerPorId /nuevo /editar /grabar /eliminar	/listado /traerPorId /nuevo /editar /grabar /eliminar
/clientes/listado /clientes/traerPorId	/empleados/listado /empleados/traerPorId	/productos/listado /productos/traerPorId

Al momento de diseñar los mapeos de los requerimientos, es recomendable tener formas estándares para facilitar su desarrollo y la integración con las vistas.

@RequestMapping

Esta anotación se utiliza para configurar la URL a la que tiene que atender una clase o un método. Si se aplica a una clase, entonces las URLs de sus métodos son relativas a la indicada en la clase. Un método que tenga esta anotación no tiene que seguir ningún patrón específico, un ejemplo ilustrativo se tiene en el Script 9.

Script 9

```
@Controller
@RequestMapping(value = "/facturas")
public class FacturaController {

   @RequestMapping(method = RequestMethod.GET)
   public void listado(HttpServletResponse response) throws IOException {
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      out.println("<h1>LISTADO DE FACTURAS</h1>");
   }

   @RequestMapping(value = "/nueva", method = RequestMethod.GET)
   public void nueva(HttpServletResponse response) throws IOException {
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      out.println("<h1>NUEVA FACTURA</h1>");
   }

}
```

Como se observa en el Script 9, la anotación permite utilizar algunos parámetros adicionales, como el método HTTP concreto. Sólo si la petición HTTP es del tipo indicado se llamará al método.

Otros parámetros de la anotación permiten indicar el formato aceptado según el contenido de la cabecera Content-Type (`consumes="application/json"`), el formato generado según la cebecera Accept (`produces="text/plain"`), los parámetros presentes en la URL (`params="mode=online"`), o la cabecera HTTP (`headers="cabecera=personalizada"`).

Una característica interesante es que la expresión de los atributos también se pueden negar para excluir condiciones en vez de incluirlas (`consumes="!text/plain"`).

@RequestParam

Esta anotación permite acceder a los parámetros de una petición HTTP, puede ser un requerimiento tipo GET o POST.

Sintaxis

```
@RequestParam[(name="NombreParámetro", ...)] TipoDato NombreVariable
```

Por defecto, los parámetros son de tipo String, pero Spring MVC cuenta con conversores que convierten el dato de tipo String al tipo de dato de la variable (int, doubles, etc.).

Mapeo simple

Para ilustrar cómo funciona puedes revisar el mapeo del Script 10:

Script 10

```
@GetMapping(value = "/clientes/leer")
public String leerClientes(@RequestParam String departamento, Model mode) {

   . . .

   return "verClientes";
}
```

En este tipo de mapeo, necesariamente debe existir un parámetro de nombre **departamento**.

A continuación, tienes un ejemplo de un requerimiento que estaría invocando a **leerClientes**:

```
http://localhost:8080/clientes/leer?departamento=Contabilidad
```

Es importante que tomes en cuenta que este este caso, tanto el nombre del parámetro y el nombre de la variable debes ser iguales.

Especificando el nombre del parámetro

En este caso, el nombre del parámetro y el nombre de la variable son diferentes, por lo que es necesario especificar el nombre del parámetro.

Para ilustrar cómo funciona puedes revisar el Script 11.

Script 11

```
@GetMapping(value = "/clientes/leer")
public String leerClientes
(@RequestParam(value = "tipo") Integer tipoCliente, Model model) {

   . . .

   return "verClientes";
}
```

Como puedes observar, en este caso, el nombre del parámetro y el nombre de la variable son diferentes, también debes considerar que el tipo de dato de la variable no es String, por lo que Spring MVC debe hacer la conversión respectiva.

A continuación, tienes un ejemplo de un requerimiento que estaría invocando a **leerClientes**:

```
http://localhost:8080/clientes/leer?tipo=2
```

Para especificar el nombre del parámetro, puedes utilizar `@RequestParam(value="tipo")` o simplemente `@RequestParam("tipo")`.

Parámetros opcionales

Los parámetros del método anotados con `@RequestParam` son obligatorios de forma predeterminada.

Esto significa que, si el parámetro no está presente en el requerimiento, obtendrás un error similar al que se muestra a continuación:

```
There was an unexpected error (type=Bad Request, status=400).
Required Integer parameter 'tipo' is not present
```

Este mensaje esta informando que se espera un parámetro de nombre `tipo` y debe ser de tipo `Integer`.

Sin embargo, puedes configurar la anotación @RequestParam para que el parámetro sea opcional, con el atributo `required`, tal como lo puedes observar en el Script 12:

Script 12

```
@GetMapping(value = "/clientes/leer")
public String leerClientes
(@RequestParam(value = "tipo", required = false) Integer tipoCliente,
Model model) {

   . . .

   return "verClientes";
}
```

Cuando no se especifica el parámetro `id` en el requerimiento, el valor de la variable `tipoCliente` será `null`.

Usando Java 8 Optional

Alternativamente, puedes envolver el parámetro en una clase `Optional`, tal como lo puedes ver en el Script 13.

Script 13

```
@GetMapping(value = "/clientes/leer")
public String leerClientes
(@RequestParam(value = "tipo") Optional<Integer> tipoCliente, Model model) {

   Integer tipo = tipoCliente.orElse(0);
   . . .

   return "verClientes";
}
```

En este caso, no necesitamos especificar el atributo requerido, no es necesario el atributo `required`.

En caso de que no se proporcione el valor del parámetro tipo, el valor que se está asumiendo es cero (0); también puedes especificar un valor por predeterminado.

Valor predeterminado

La anotación `@RequestParam` te permite establece un valor predeterminado para tu parámetro en el requerimiento, esto para los casos que no se incluya en el mismo.

En el Script 14 tienes el ejemplo de cómo utilizar al atributo `defaultValue` para especificar el valor predeterminado de un parámetro.

Script 14

```
@GetMapping(value = "/clientes/leer")
public String leerClientes
(@RequestParam(value = "tipo", defaultValue = "0") Integer tipoCliente,
Model model) {

   . . .

   return "verClientes";
}
```

En este caso, la variable `tipoCliente` tomará el valor cero (0) en caso no se especifique el valor del parámetro `tipo` en el requerimiento.

Mapeando todos los parámetros

También puedes obtener todos los parámetros en una sola variable de tipo `Map`.

En el Script 15 tienes un ejemplo donde se ilustra como recibir múltiples parámetros en una variable de tipo `Map`.

Script 15

```
@GetMapping(value = "/clientes/leer")
public String leerClientes
(@RequestParam Map<String,String> datos, Model model) {

   . . .

   return "verClientes";
}
```

En este caso, la variable `datos` recibe todos los parámetros, donde el nombre del parámetro sería la clave de la colección de tipo `Map`.

Mapeo de parámetros de múltiples valores

Se trata de parámetros que pueden tener varios valores.

En el Script 16 tienes un ejemplo donde se ilustra como recibir un parámetro con múltiples valores en una variable de tipo `List`.

Script 16

```
@GetMapping(value = "/clientes/leer")
public String leerClientes
(@RequestParam("codigos") List<String> codigos, Model model) {

   . . .

   return "verClientes";
}
```

A continuación, tienes un ejemplo de un requerimiento con múltiples valores a un parámetro:

```
http://localhost:8080/clientes/leer?codigos=20,50,80
```

Como puedes observar, los valores se separan por comas.

@PathVariable

Contexto

La anotación `@PathVariable` la puedes utilizarla para acceder a variables de plantilla en la URI del requerimiento y usarlas como parámetros del método.

Para que puedas crear estas variables de plantilla se utiliza las llaves, como se ilustra a continuación:

```
@GetMapping(value = "/clientes/leer/{id}")
```

En este caso el nombre de la variable es `id`.

Mapeo simple

Un caso de mapeo simple de la anotación `@PathVariable` sería un punto final que identifica una entidad mediante su clave primaria.

En el Script 17 tienes un ejemplo de cómo utilizar la anotación `@PathVariable` para acceder a la variable `id`.

Script 17

```
@GetMapping(value = "/clientes/leer/{id}")
public String leerCliente(@PathVariable Integer id, Model model) {

   . . .

   return "verCliente";
}
```

En este ejemplo, se está utilizando la anotación @PathVariable para extraer de la plantilla URI la parte representada por la variable `{id}`. El nombre de la variable en la plantilla y el nombre de la variable en el método de ser iguales.

A continuación, tienes un requerimiento simple de tipo GET que invoca al método `leerCliente`.

```
http://localhost:8080/clientes/leer/500
```

El valor que estaría tomando la variable `id` en este caso es 500.

Especificando el nombre de la variable

En caso que el nombre de la variable en la ruta y el nombre de la variable en el método sean diferentes, debes especificar en la anotación `@PathVariable` en nombre de la variable en la ruta.

En el Script 18 tienes un ejemplo de cómo utilizar la anotación `@PathVariable` para acceder a la variable en la plantilla de la ruta.

Script 18

```
@GetMapping(value = "/clientes/leer/{id}")
public String leerCliente
(@PathVariable(value = "id") Integer idCliente, Model model) {

   . . .

   return "verCliente";
}
```

En este caso, la variable en la ruta es `id` y la variable en el método es `idCliente`.

A continuación, tienes un requerimiento simple de tipo GET que invoca al método `leerCliente`.

```
http://localhost:8080/clientes/leer/1000
```

El valor que estaría tomando la variable `idCliente` en este caso es 1000.

Tú puedes especificar el nombre de variable de ruta de dos formas, puedes utilizar `@PathVariable(valor="id")` o `@PathVariable("id")`.

Múltiples variables en la ruta

Dependiendo del caso que estés resolviendo, puedes tener más de una variable de ruta en la URI del requerimiento para un método del controlador, que también debe tener múltiples parámetros en el método.

En el Script 19 tienes un ejemplo de cómo utilizar la anotación `@PathVariable` para acceder a múltiples variables en la plantilla de la ruta.

Script 19

```
@GetMapping(value = "/clientes/buscar/{id}/{nombre}")
public String listarClientes
(@PathVariable Integer id, @PathVariable String nombre, Model model) {

   . . .

   return "listarClientes";
}
```

En este caso, las variables en la ruta son `id` y `nombre`, que tienen el mismo nombre en los parámetros de método.

A continuación, tienes un requerimiento simple de tipo GET que invoca al método `listarClientes`.

```
http://localhost:8080/clientes/buscar/600/CORONEL
```

El valor que estaría tomando las variables `id` y `nombre` son 600 y CORONEL respectivamente.

También puedes manejar más de un parámetro `@PathVariable` usando un parámetro en el método tipo `java.util.Map<String,String>`.

En el Script 20 tienes un ejemplo de cómo utilizar la anotación `@PathVariable` para acceder a múltiples variables en la plantilla de la ruta utilizando una variable en el método de tipo `java.util.Map<String,String>`.

Script 20

```
@GetMapping(value = "/clientes/buscar/{id}/{nombre}")
public String listarClientes
(@PathVariable Map<String,String> parametros, Model model) {

   . . .

   return "listarClientes";
}
```

En este caso, `id` y `nombre` serian claves en la colección de tipo `Map`.

A continuación, tienes un requerimiento simple de tipo GET que invoca al método `listarClientes`.

```
http://localhost:8080/clientes/buscar/600/CORONEL
```

El valor que estaría tomando las variables `id` y `nombre` son 600 y CORONEL respectivamente.

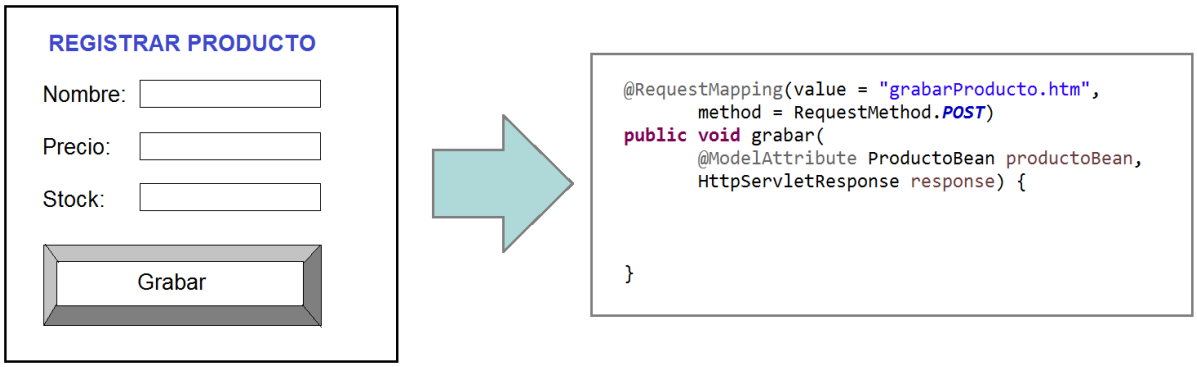
@ModelAttribute

Caso 1

```
@RequestMapping(value = "verProducto.htm", method = RequestMethod.GET)
public String verProducto(@ModelAttribute("producto") ProductoBean productoBean) {
  productoBean.setNombre("Televisor HD");
  productoBean.setPrecio(2500.00);
  productoBean.setStock(500);
  return "verProducto";
}
```

```
<body>
  <h1>PRODUCTO</h1>
  <p>Nombre: ${producto.nombre}</p>
  <p>Precio: ${producto.precio}</p>
  <p>Stock: ${producto.stock}</p>
</body>
```

Caso 2




```
@RequestMapping(value = "grabarProducto.htm",
        method = RequestMethod.POST)
public void grabar(
        @ModelAttribute ProductoBean productoBean,
        HttpServletResponse response) {


}
```

RETORNAR JSON

Retornar un bean

El Script 21 es un ejemplo ilustrativo donde puedes ver la configuración para retornar un bean en formato JSON.

Script 21

```
@RequestMapping(value = "/verProducto", method = RequestMethod.GET,
                  produces = "application/json; charset=UTF-8")
@ResponseBody
public ProductoDto verProducto() {
   ProductoDto dto = new ProductoDto();
   dto.setId(1000L);
   dto.setNombre("Televisor HD");
   dto.setPrecio(4567.89);
   dto.setStock(560L);
   return dto;
}
```

Retornar una colección

El Script 22 es un ejemplo ilustrativo donde puedes ver la configuración para retornar una colección en formato JSON

Script 22

```
@RequestMapping(value = "/verProductos", method = RequestMethod.GET,
                  produces = "application/json; charset=UTF-8")
@ResponseBody
public List<ProductoDto> verProductos() {
   List<ProductoDto> lista = new ArrayList<>();
   lista.add(new ProductoDto(1001L, "Producto 1", 345.67, 657L));
   lista.add(new ProductoDto(1002L, "Producto 2", 357.23, 435L));
   lista.add(new ProductoDto(1003L, "Producto 3", 768.76, 912L));
   return lista;
}
```