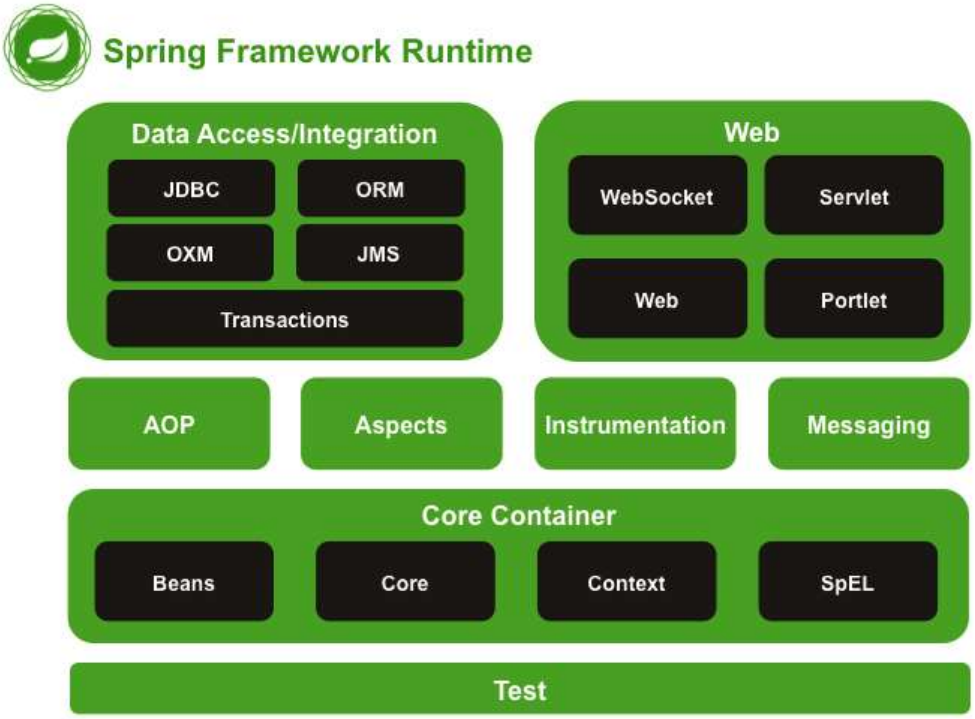
DESARROLLO WEB
CON SPRING BOOT



UNIDAD 04
SPRING BOOT MVC

Eric Gustavo Coronel Castillo
gcoronelc.github.io
I N S T R U C T O R

CONTENIDO

FUNDAMENTOS **4**

ARQUITECTURA MVC 4
MODELO AMPLIADO 5
EL ENFOQUE DE SPRING MVC 6

ARQUITECTURA Y CONFIGURACIÓN **7**

ARQUITECTURA 7
CONFIGURACIÓN 9

ARCHIVO DE PROPIEDADES **10**

EL PROBLEMA 10
ARCHIVO DE PROPIEDADES 10
UTILIZAR LAS VARIABLES QUE DEFINIMOS 11
Clase Environment *11*
ANOTACIÓN @VALUE 12

MVC CON PAGINAS JSP **13**

CONFIGURACIÓN 13
PROGRAMACIÓN 14
RECURSOS ESTÁTICOS 16

PARAMETROS DEL SERVLET **17**

HTTPSERVLETREQUEST 17
HTTPSERVLETRESPONSE 17
EJEMPLO ILUSTRATIVO 18

MODEL Y MODELANDVIEW **19**

MODEL Y VIEW 19
INTERFACE MODEL 20
CLASE MODELANDVIEW 20

MAPEO DE PETICIONES **21**

@REQUESTMAPPING 21
@REQUESTPARAM 22
@MODELATTRIBUTE 23
CASO 1 23
CASO 2 23

RETORNAR JSON **24**

RETORNAR UN VEAN 24
RETORNAR UNA COLECCIÓN 24

CURSOS VIRTUALES **25**

CUPONES 25

FUNDAMENTOS DE PROGRAMACIÓN 25
JAVA ORIENTADO A OBJETOS 26
PROGRAMACIÓN CON JAVA JDBC 27
PROGRAMACIÓN CON ORACLE PL/SQL..... 28

FUNDAMENTOS

ARQUITECTURA MVC

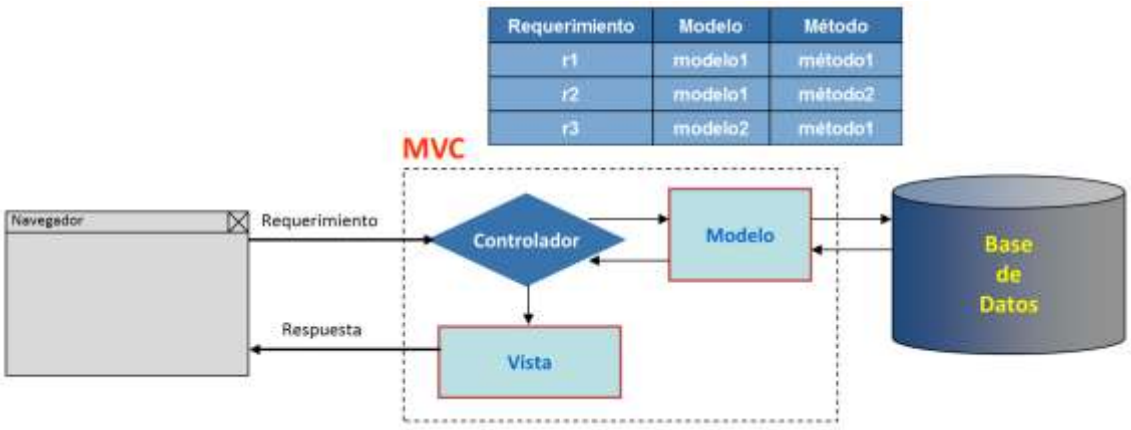


Figura 1

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

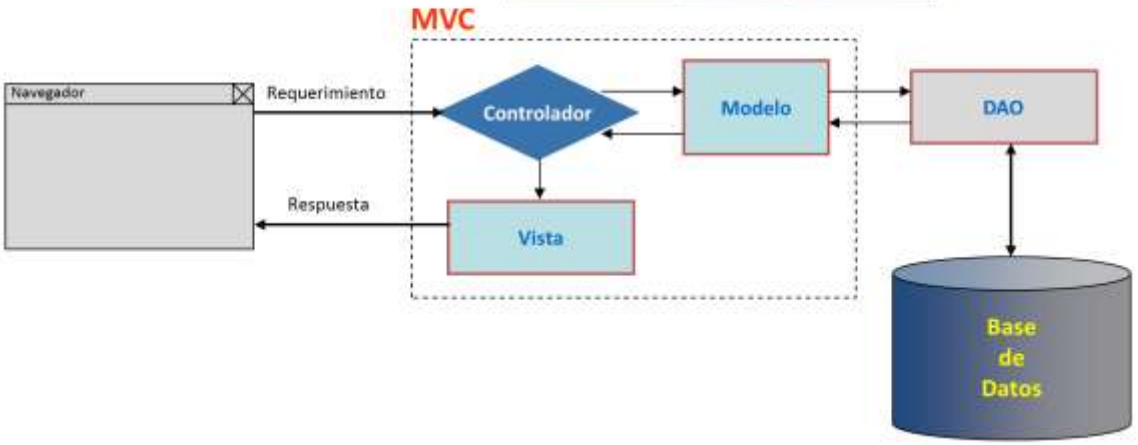


Figura 2

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

EL ENFOQUE DE SPRING MVC

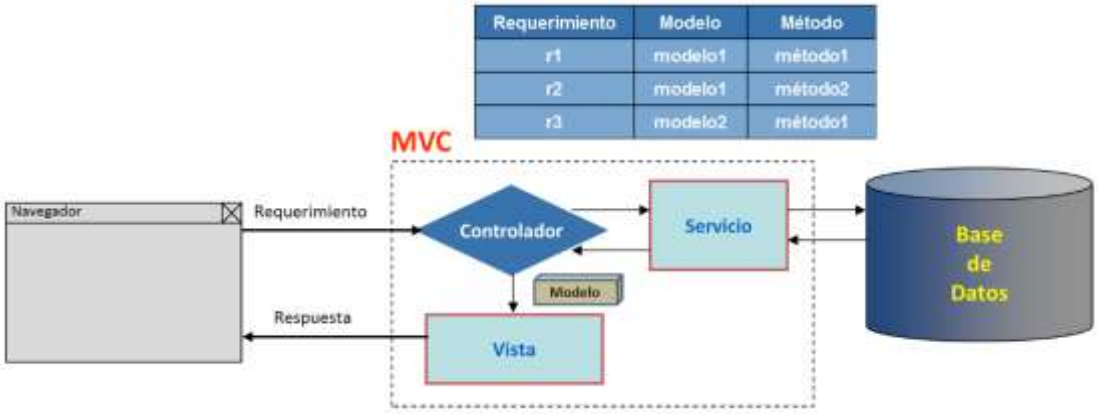


Figura 3

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

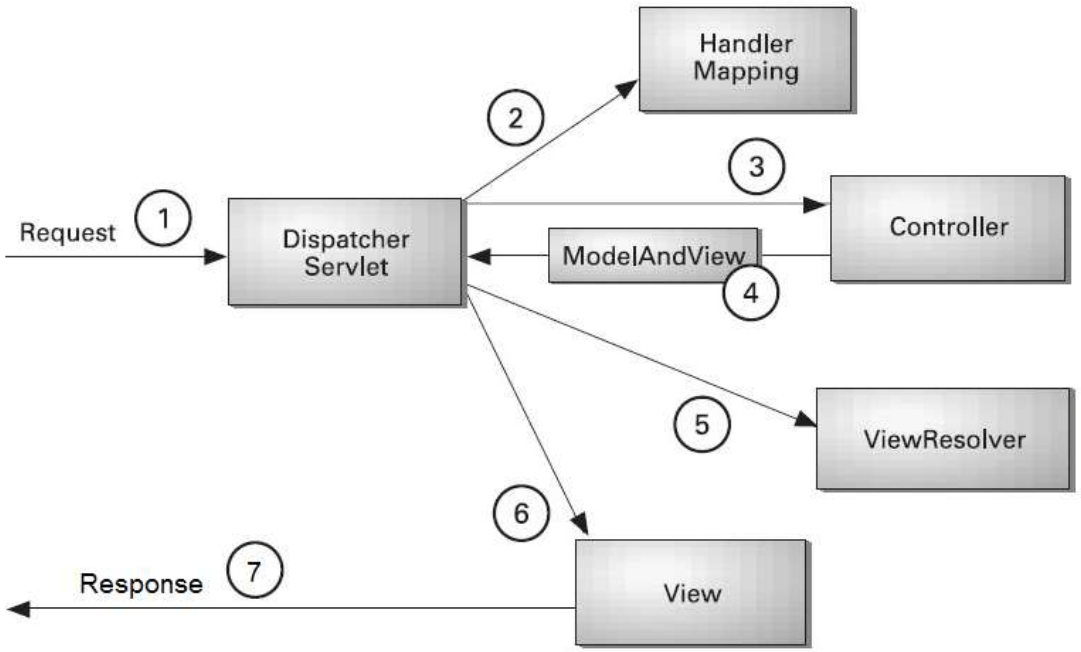


Figura 4

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.

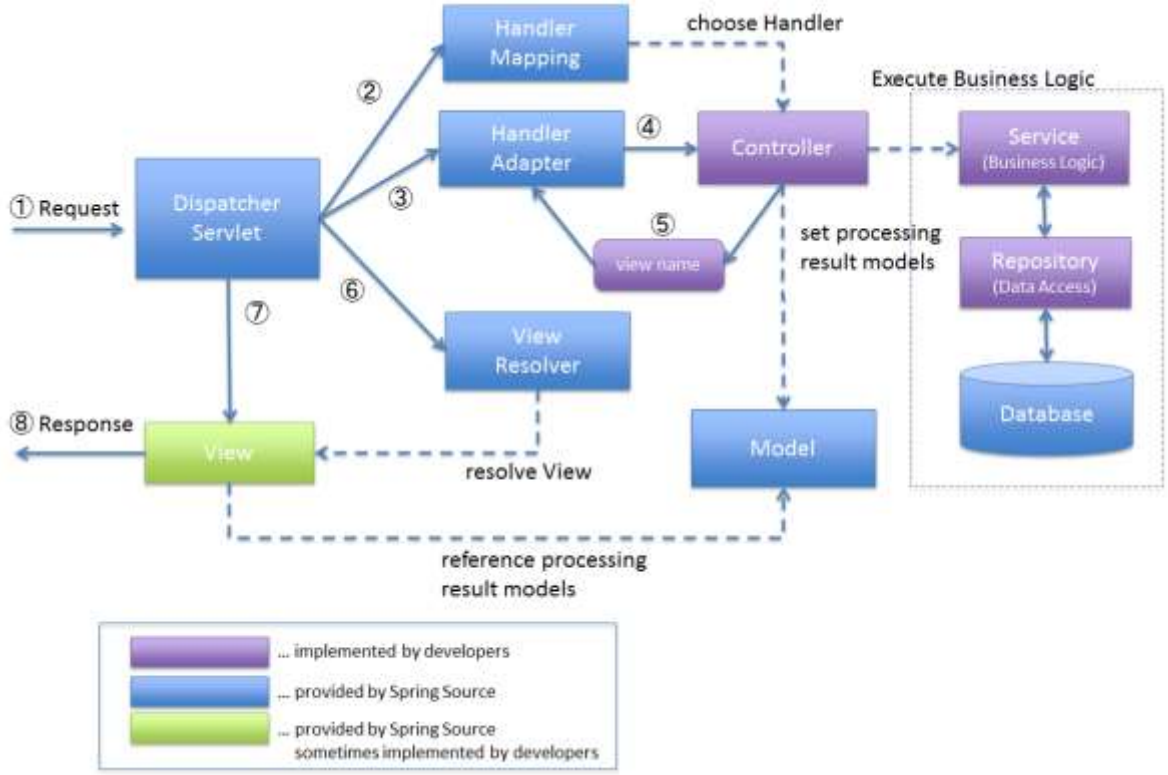
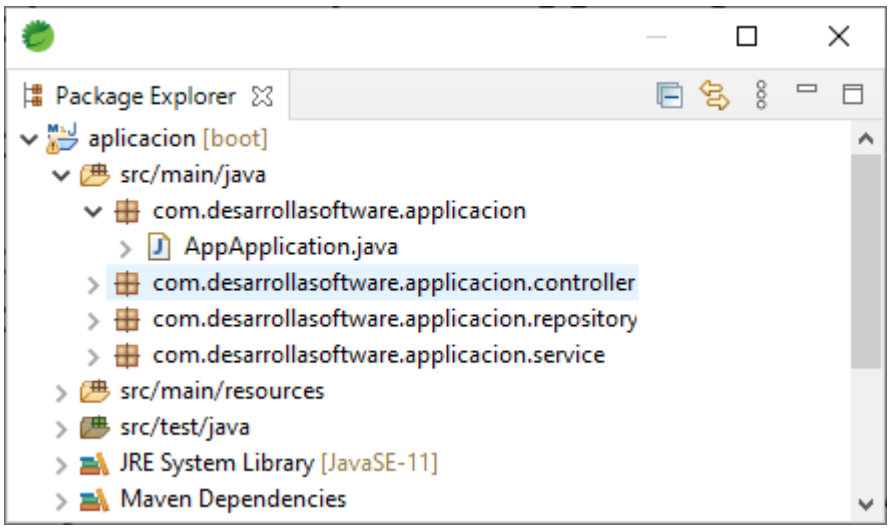


Figura 5

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.

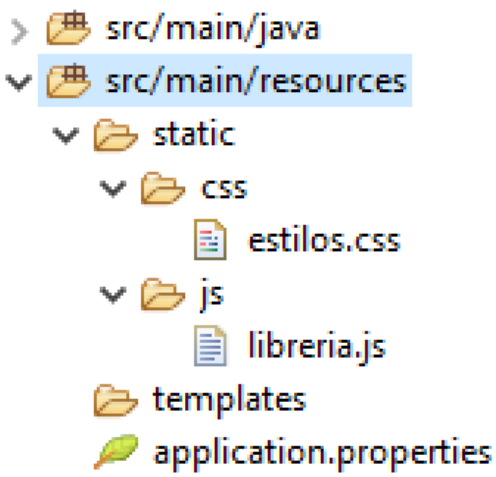


Figura 6

En el Script 3, tienes un ejemplo de como 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

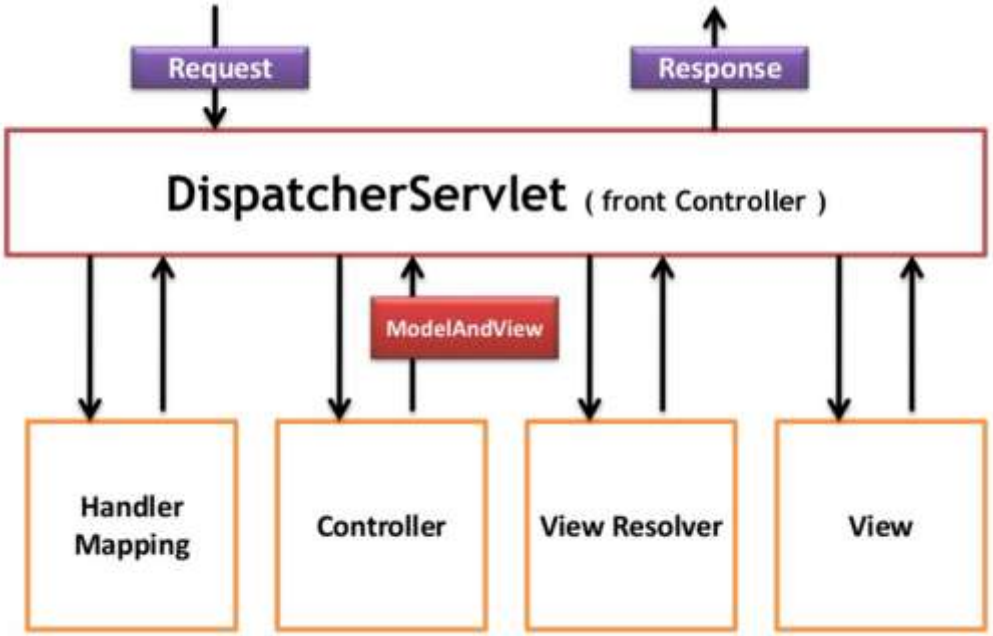


Figura 7

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 PETICIONES

@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, como se ilustra en el Script 10.

Script 10

```
  @RequestMapping(value="/consulta.htm")
  public String handler(@RequestParam("recetaId") Long recetaId) {
     ...
  }
```

Para una petición de la forma:

```
/consulta.htm?recetaId=123
```

El argumento `recetaId` del método tomaría el valor `123` directamente del parámetro de la URL. Por otra parte, si el parámetro no fuera obligatorio, se podría indicar con `required=false`.

El Script 11 ilustra el uso de `required`, en este caso el argumento `recetaId` del método toma valor `null`, por lo que no puede ser de tipo primitivo.

Script 11

```
  @RequestMapping(value="/consulta")
  public String handler(@RequestParam(value="recetaId", required=false) Long
recetaId) {
     ...
  }
```

Es necesario tener en cuenta que durante el proceso se hace necesario verificar si el argumento `recetaId` ha tomado valor `null` mediante una estructura `if`.

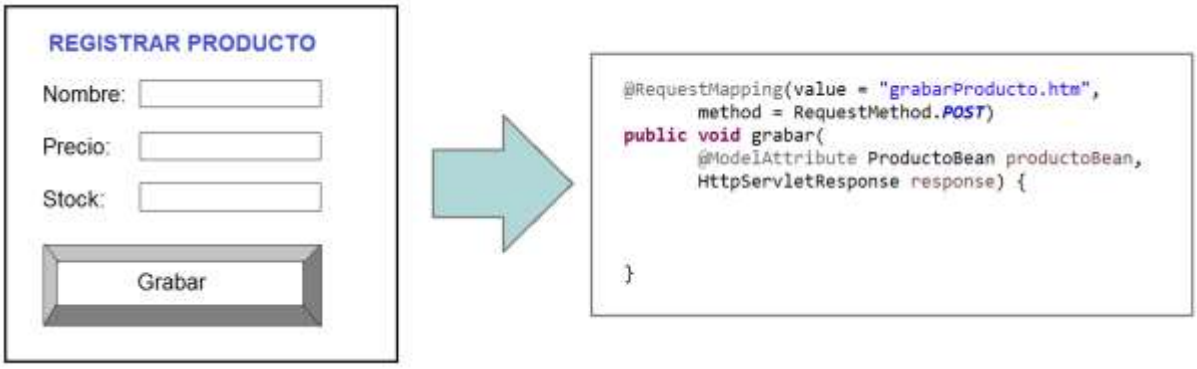
@ModelAttribute

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



RETORNAR JSON

Retornar un vean

El Script 12 es un ejemplo ilustrativo donde puedes ver la configuración para retornar un bean en formato JSON.

Script 12

```
@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 13 es un ejemplo ilustrativo donde puedes ver la configuración para retornar una colección en formato JSON

Script 13

```
@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;
}
```

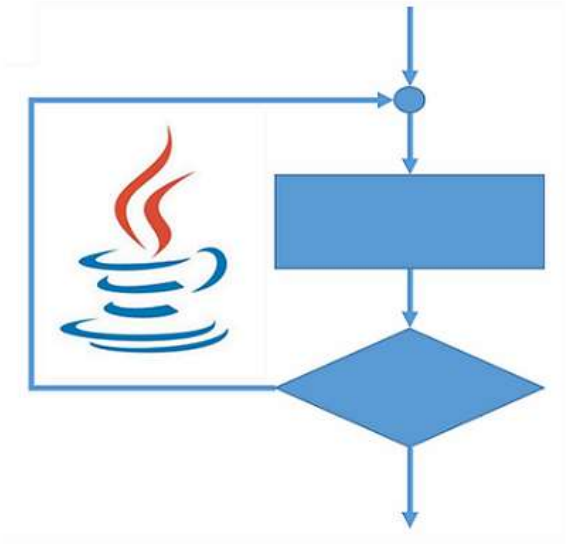
CURSOS VIRTUALES

CUPONES

En esta URL se publican cupones de descuento:

http://gcoronelc.github.io

FUNDAMENTOS DE PROGRAMACIÓN



Tener bases sólidas de programación muchas veces no es fácil, creo que es principalmente por que en algún momento de tu aprendizaje mezclas la entrada de datos con el proceso de los mismos, o mezclas el proceso con la salida o reporte, esto te lleva a utilizar malas prácticas de programación que luego te serán muy difíciles de superar.

En este curso aprenderás las mejores practicas de programación para que te inicies con éxito en este competitivo mundo del desarrollo de software.

URL del Curso: **https://www.udemy.com/course/fund-java**

Avance del curso: **https://n9.cl/gcoronelc-fp-avance**

Cupones de descuento: **http://gcoronelc.github.io**

JAVA ORIENTADO A OBJETOS

En este curso aprenderás a crear software aplicando la Orientación a objetos, la programación en capas, el uso de patrones de software y swing.

Cada tema está desarrollado con ejemplos que demuestran los conceptos teóricos y finalizan con un proyecto aplicativo.

URL del Curso: **https://bit.ly/2B3ixUW**

Avance del curso: **https://bit.ly/2RYGXIt**

Cupones de descuento: **http://gcoronelc.github.io**

PROGRAMACIÓN CON JAVA JDBC

En este curso aprenderás a programas bases de datos Oracle con JDBC utilizando los objetos Statement, PreparedStatement, CallableStatement y a programar transacciones correctamente teniendo en cuenta su rendimiento y concurrencia.

Al final del curso se integra todo lo desarrollado en una aplicación de escritorio.

URL del Curso: **https://bit.ly/31apy0O**

Avance del curso: **https://bit.ly/2vatZOT**

Cupones de descuento: **http://gcoronelc.github.io**

PROGRAMACIÓN CON ORACLE PL/SQL

En este curso aprenderás a programas las bases de datos ORACLE con PL/SQL, de esta manera estarás aprovechando las ventas que brinda este motor de base de datos y mejoraras el rendimiento de tus consultas, transacciones y la concurrencia.

Los procedimientos almacenados que desarrolles con PL/SQL se pueden ejecutarlo de Java, C#, PHP y otros lenguajes de programación.

URL del Curso: **https://bit.ly/2YZjfxT**

Avance del curso: **https://bit.ly/3bciqYb**

Cupones de descuento: **http://gcoronelc.github.io**