Creando el controlador

¡Comencemos!

¿Qué aprenderás?

- Implementar y crear el o los controladores dentro de nuestro proyecto, implementando anotaciones tales como @Controller, @RequestMapping, @RequestParam.
- Entender el funcionamiento de la anotación @RequestMapping.
- Enviar datos desde el controlador a la Vista mediante la implementación del objeto ModelMap.
- Mostrar datos en la vista mediante la implementación de JSTL.
- Recibir datos desde la vista al controlador mediante la implementación de Form de spring.
- Probar el proyecto Spring Boot mediante un Navegador Web (Browser).

Introducción

A continuación, finalizamos el proyecto holamundosrpingmvc con la creación del controlador, quien se encargará de recibir la petición y procesarla para así devolver una vista con los datos en el navegador. Por lo cual, aquí se aprenderá a crear el controlador, recibir datos desde la vista en el controlador, enviar datos a la vista desde el controlador, para así completar el flujo y el primer flujo del sistema que estamos desarrollando.

El último paso para crear nuestro proyecto y que este funcione, es crear el controlador, quien se encargará de gestionar y procesar las peticiones de la o las vistas de nuestro programa.

¡Vamos con todo!

Controlador

Antes que todo, cabe destacar que utilizaremos la anotación @controller, la cual nos permitirá decirle a Spring que la clase que estamos creando corresponde a un controlador de Spring. Vamos a la práctica.

1. En \src\main\java\cl\desafiolatam\holamundospringmvc, crear el package llamado controller.

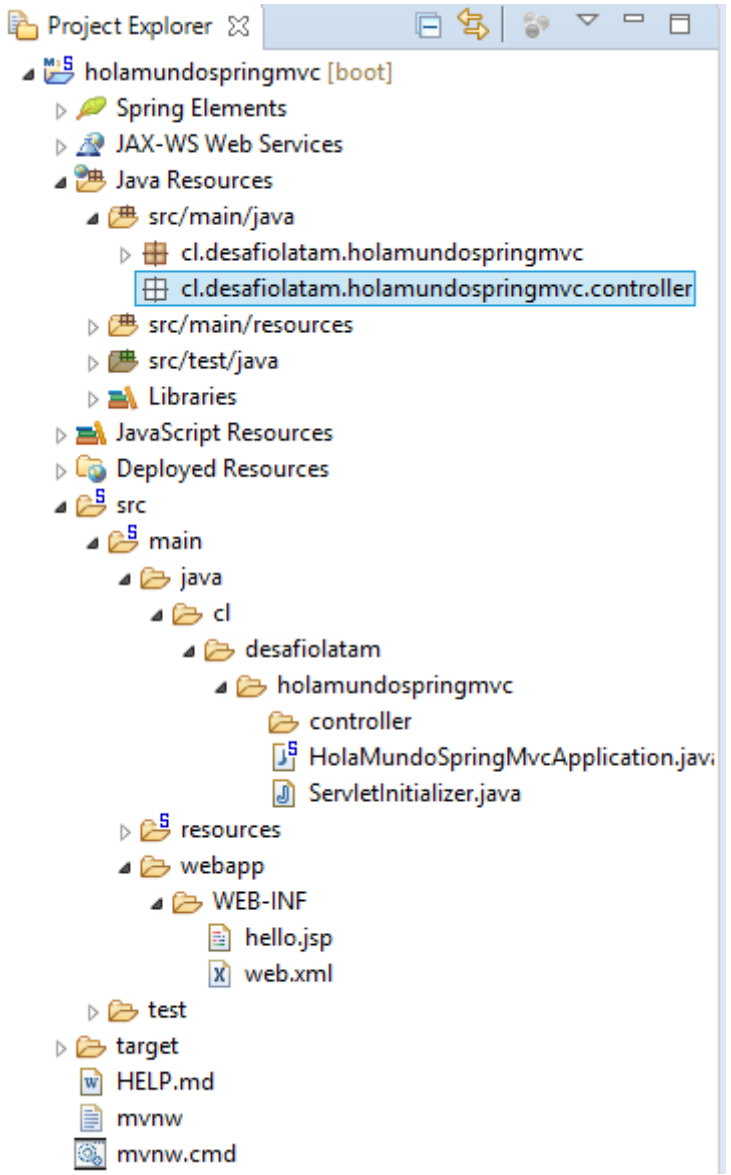

Imagen 1. Creando package “controller”.
Fuente: Desafío Latam.

2. Dentro del package controller, crearemos la clase HelloController.java. Por convención, estándar y buenas prácticas de programación, todas las clases Controller deben terminar con la palabra Controller.

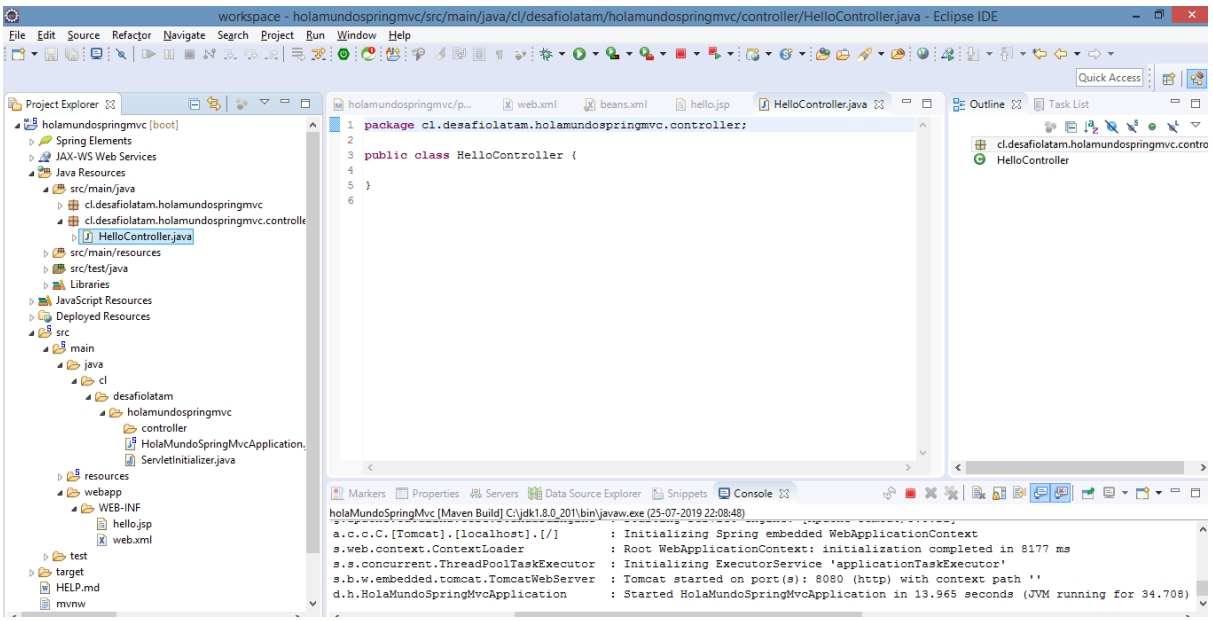

Imagen 2. HelloController.
Fuente: Desafío Latam.

3. Como se mencionó anteriormente, debemos anotar la clase como un controlador, por lo que agregaremos antes de la palabra reservada class, la anotación @Controller.

```
package cl.desafiolatam.holamundospringmvc.controller;
import org.springframework.stereotype.Controller;
@Controller
public class HelloController {

}
```

4. Luego, debemos crear el método dentro del controlador, el cual recibirá la petición de la vista.

```
package cl.desafiolatam.holamundospringmvc.controller;
import org.springframework.stereotype.Controller;
@Controller
public class HelloController {
   public String hello() {
```

```
        return "hello";
    }
}
```

La parte resaltada del código, corresponde al método que recibirá las peticiones desde hello.jsp (Vista). El funcionamiento de Spring, será el siguiente:

a. El dispatcher, intercepta la petición del cliente. Esta petición, se invoca cuando el usuario ingresa en su navegador la siguiente URL

b. El dispatcher, al reconocer en la url que se está solicitando la vista hello, este busca en el o los controladores, el método llamado hello. Cuando lo encuentra, invoca al método

.

c. El controlador hello, comienza a trabajar. Para este caso, lo único que está haciendo el controlador es devolver un String "hello".

d. El dispatcher recibedispatcher, recibe hello desde el controlador, revisa la configuración de Spring en application.properties, y busca la vista hello.jsp en el directorio WEB-INF de la aplicación.

5. Hasta este punto, solo nos falta una configuración más para que todo funcione, por lo cual debemos agregar la anotación @RequestMapping al método hello(). Esta anotación es crucial, ya que le informará al dispatcher, entre otras cosas, de qué manera o bajo qué método está esperando recibir la petición desde el cliente (Navegador o browser). Entonces, agregamos @RequestMapping al método, indicando que, el controlador recibirá la petición mediante el método GET.

```
package cl.desafiolatam.holamundospringmvc.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
@Controller
public class HelloController {
    @RequestMapping(method=RequestMethod.GET)
    public String hello() {
        return "hello";
    }
}
```

6. Levantamos la aplicación como se ha indicado en los apartados anteriores.

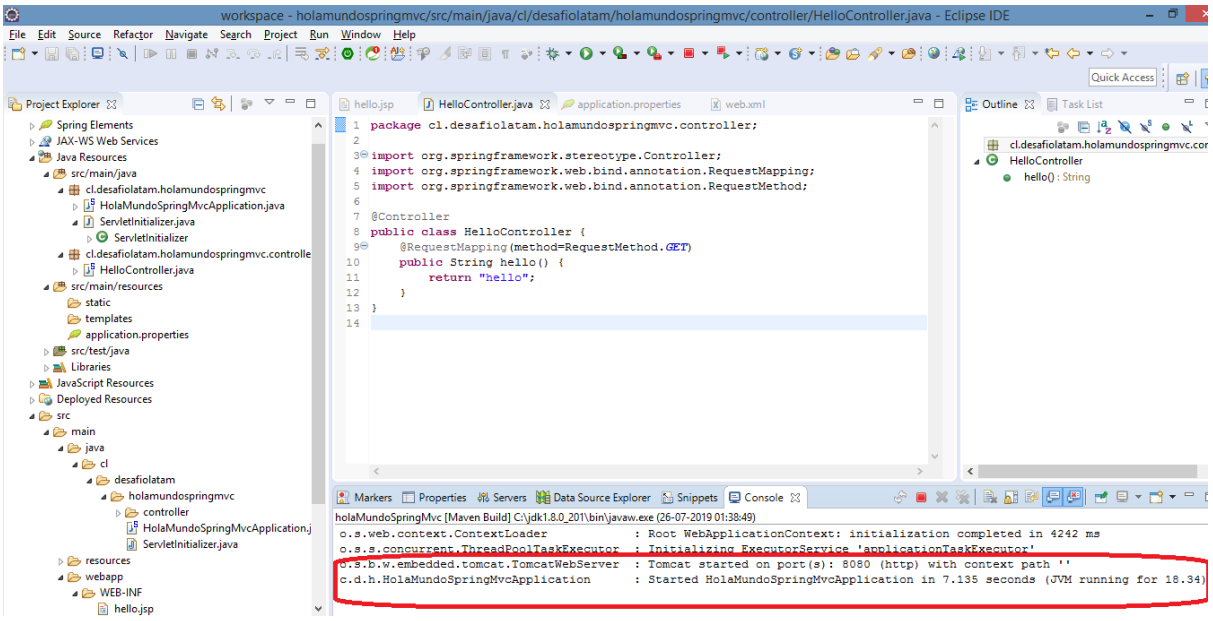

Imagen 3. Levantando la aplicación.
Fuente: Desafío Latam.

7. Probamos nuestra aplicación ejecutando en un navegador la siguiente URL.

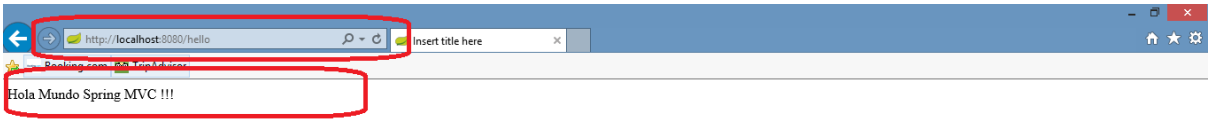

Imagen 4. Ejecutando la aplicación en el navegador.
Fuente: Desafío Latam.

Creando el controlador - anotación @RequestMapping

La anotación RequestMapping es una de las más utilizadas de Spring, la cual nos permite gestionar las peticiones del cliente. Algunos de los parámetros más usados e imprescindibles de RequestMapping, son los siguientes:

- **value:** Este parámetro permite decirle al método del controlador, cuál va a ser el punto final de la url. Por ejemplo:

```
http://localhost:8080/hello
```

Para este caso, hello corresponde al value del RequestMapping, no obstante, en el ejemplo Hola Mundo Spring MVC!!!, no hemos especificado un valor para value debido a que este no es obligatorio, y Spring tomará como value el nombre del método, siempre que existan más de un método en la clase Controller. Vamos a un ejemplo.

1. En nuestro proyecto, ir al controlador HelloControler.java y cambiar el nombre del método hello por sayHello.

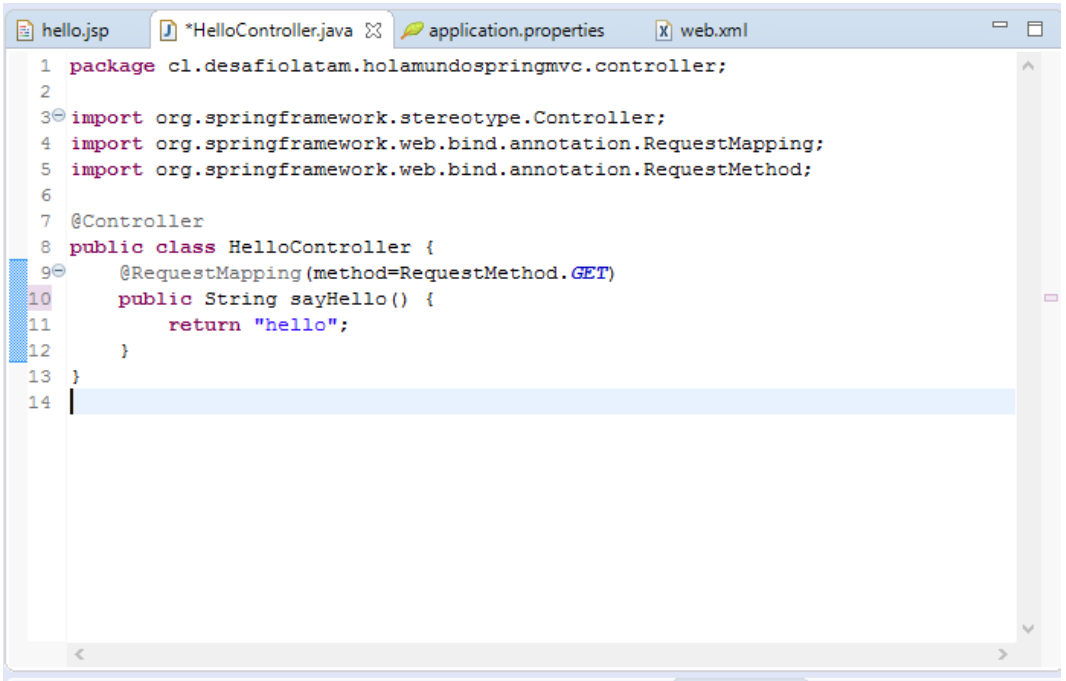

Imagen 5. Cambiando el nombre del método a SayHello().
Fuente: Desafío Latam.

2. Guardar y correr el proyecto en el servidor. Recordar verificar que el servidor se haya iniciado correctamente. (Ver sección encerrada en rojo de la siguiente imagen).

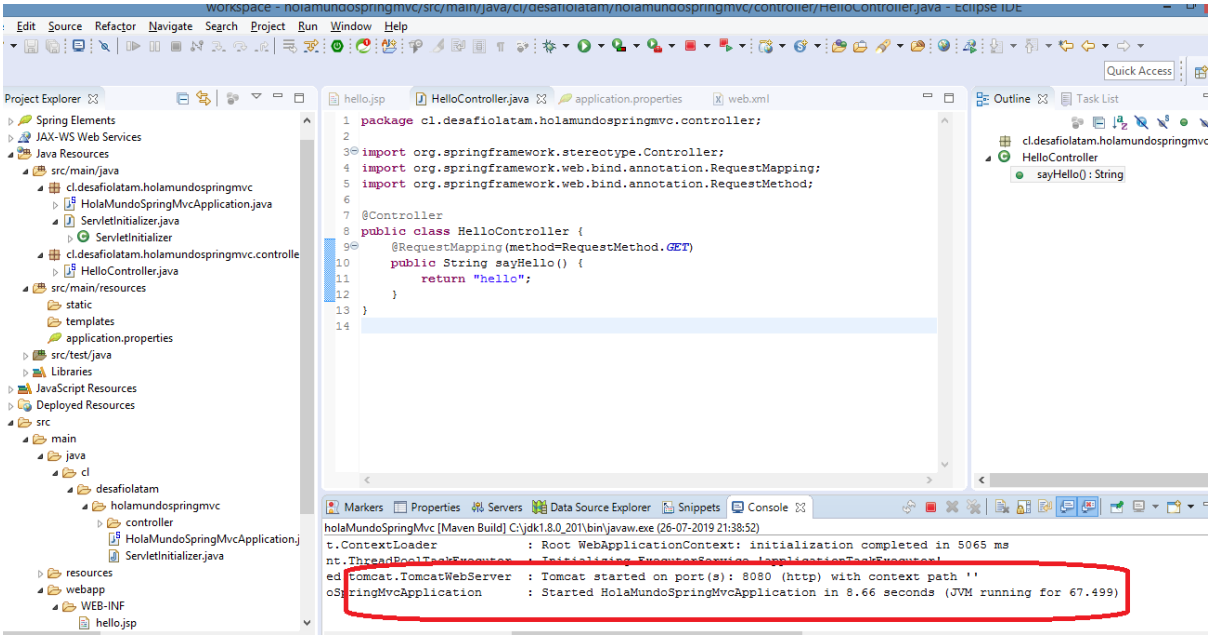

Imagen 6. Verificar el inicio correcto del servidor.
Fuente: Desafío Latam.

3. Ir al navegador, y probar con la url indicada [http://localhost:8080/hello.](http://localhost:8080/hello)

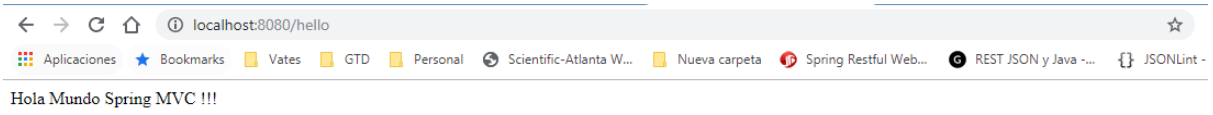

Imagen 7. Probando la aplicación.
Fuente: Desafío Latam.

Nos damos cuenta, que todo sigue funcionando correctamente, debido a que el controlador solo tiene un método, por lo que el dispatcher siempre ejecutará el método para hello.

4. Pues bien, agregamos un nuevo método:

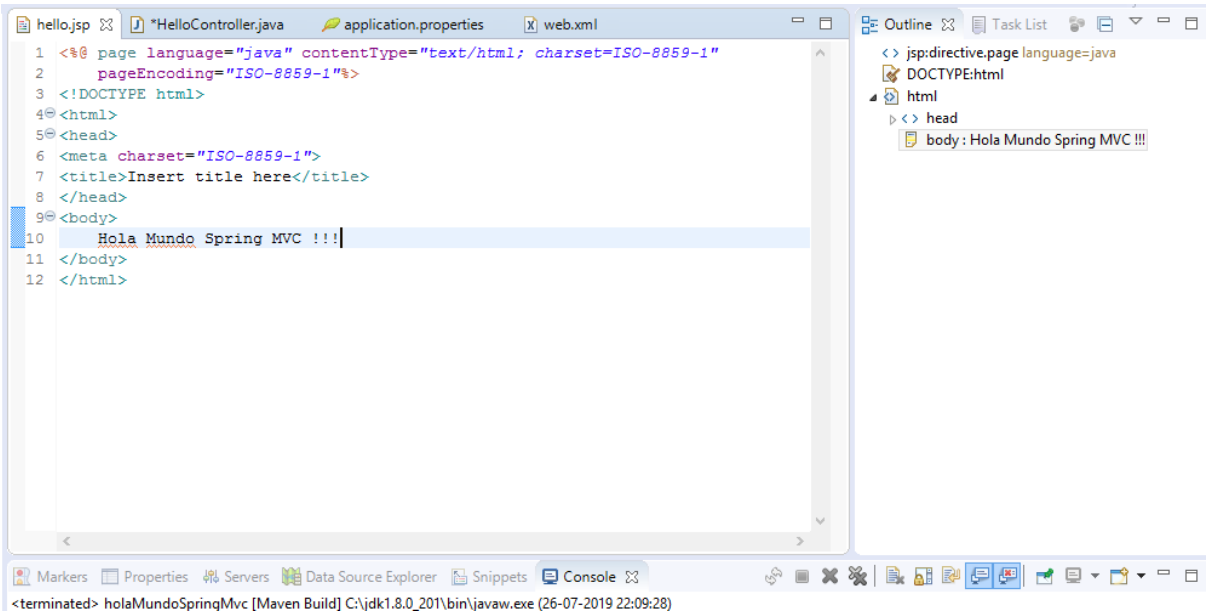

Imagen 8. Modificando hello.jsp.
Fuente: Desafío Latam.

5. Agregar al archivo jsp, lo siguiente:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

Lo que hemos hecho, es agregar a nuestro jsp la referencia a las custom tag o al taglib, que la librería jstl nos ofrece.

Una custom tag, son etiquetas personalizadas y especializadas en realizar ciertas acciones. En este caso, hemos importado los taglibs de la librería jstl y la hemos llamado con el prefijo "c".

ANTES DE SEGUIR, para lo anterior, debemos incorporar la librería o la dependencia de jstl en nuestro proyecto, ya que Spring Boot no la incluye. Por lo tanto, debemos agregar las siguientes dependencias en nuestro archivo pom.xml:

```
<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>jstl</artifactId>
</dependency>
<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>javax.servlet-api</artifactId>
```

```
</dependency>
```

6. Donde, en nuestra vista hemos colocado Hola Mundo Spring MVC !!!, debemos colocar lo siguiente:

```
<c:out value="${saludo}" />
```

Esto, lo que hace es tomar la clave o variable saludo desde el modelo de Spring que utilizamos en cada uno de los métodos del controlador. Recordemos:

```
@RequestMapping(value="/sayhello", method=RequestMethod.GET)
public String sayHello(ModelMap model) {
 model.put("saludo", "Hola Mundo Spring desde mi controlador");
 return "hello";
}
@RequestMapping(value="/saygoodbye", method=RequestMethod.GET)
public String sayGoodbye(ModelMap model) {
 model.put("saludo", "Adios Mundo Spring desde mi controlador");
 return "hello";
}
```

La variable, en nuestra vista jsp del controlador, debe ir encerrada entre ${nombre_variable}, ya que de esta forma se le indica al tag de jstl, que lo que queremos mostrar es una variable de nuestro modelo.

7. Por último, iniciamos el servidor tomcat de Spring Boot, y probamos las siguientes urls:

- http://localhost:8080/saludos/sayhello

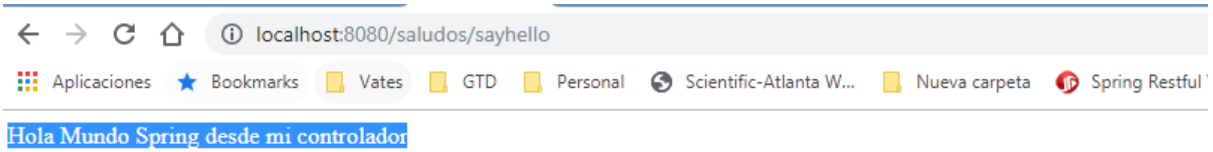

Imagen 9. saludos/sayhello.
Fuente: Desafío Latam.

- http://localhost:8080/saludos/saygoodbye

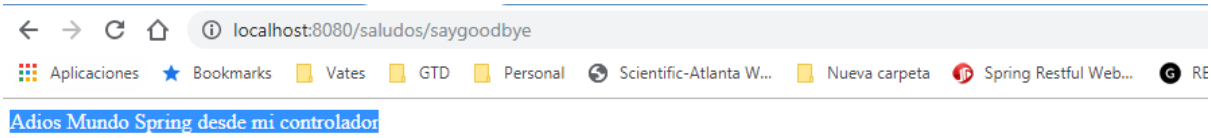

Imagen 10. saludos/sayhello.
Fuente: Desafío Latam.

Archivos modificados

HelloController.java

```
package cl.desafiolatam.holamundospringmvc.controller;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
@Controller
@RequestMapping(value="/saludos", method=RequestMethod.GET)
public class HelloController {
   @RequestMapping(value="/sayhello", method=RequestMethod.GET)
   public String sayHello(ModelMap model) {
      model.put("saludo", "Hola Mundo Spring desde mi controlador");
      return "hello";
   }
   @RequestMapping(value="/saygoodbye", method=RequestMethod.GET)
   public String sayGoodbye(ModelMap model) {
      model.put("saludo", "Adios Mundo Spring desde mi controlador");
      return "hello";
   }
}
```

hello.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
```

```
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
   <c:out value="${saludo}" />
</body>
</html>
```

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.1.6.RELEASE</version>
      <relativePath /> <!-- lookup parent from repository -->
   </parent>
   <groupId>cl.desafiolatam</groupId>
   <artifactId>holamundospringmvc</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>war</packaging>
   <name>HolaMundoSpringMVC</name>
   <description>Demo project for Spring Boot</description>
   <properties>
      <java.version>1.8</java.version>
   </properties>
   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-tomcat</artifactId>
         <scope>provided</scope>
      </dependency>
      <dependency>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
      </dependency>
      <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
      </dependency>
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
      </dependency>
   </dependencies>
   <build>
      <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
   </build>
</project>
```

Recibir datos en el controlador

Un controlador, como se ha explicado a lo largo de este documento, tiene la responsabilidad de orquestar las peticiones de los usuarios o las vistas respectivas, en base a peticiones y respuestas. En el acápite anterior, hablamos de la respuesta desde un controlador a la vista respectiva, por lo que en esta sección se explicará todo acerca de las peticiones o el envío de datos al o los controladores del sistema.

Una petición (request), es aquella provocada por alguna acción de un usuario, sobre la vista, que requiera de un envío de datos hacia un servidor de aplicaciones esperando una respuesta del mismo. Por ejemplo, una petición puede ser enviada al servidor mediante acciones tales como el ingreso de una URL en un navegador, presionar un botón (por

ejemplo para buscar, eliminar, actualizar datos), escribir texto en un cuadro de texto, un click, etc.

Las peticiones o request, se pueden tipificar en dos grupos: Asíncronas y Síncronas. Además, cada una de las peticiones lleva asociado un método, que define la forma en que se está enviando la petición al servidor. Estos métodos pueden ser GET, POST, PUT, DELETE, PATCH ó HEAD.

Peticiones Asíncronas

Estas peticiones, son aquellas en que al enviar el request al servidor, la vista no queda en espera de la respuesta (response) del servidor, si no que puede seguir ejecutándose. En otras palabras, el usuario no queda en espera de dicha respuesta, sino que puede seguir operando con la vista. Estas peticiones, se utilizan principalmente cuando necesitamos que procesos complejos se sigan ejecutando en el servidor, sin que los usuarios se den por enterados. Un ejemplo de esto, son las actualizaciones masivas.

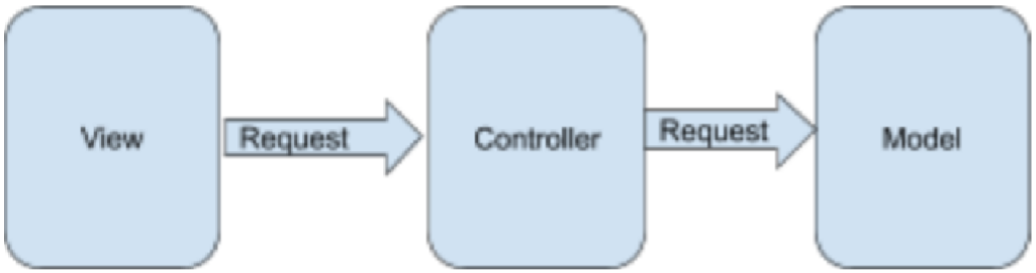

Imagen 11. Peticiones Asíncronas.
Fuente: Desafío Latam.

En una explicación gráfica bajo la arquitectura MVC de Spring, en una petición asíncrona, la vista no queda esperando la respuesta del servidor, por lo que esta sigue su ejecución normal.

Peticiones Síncronas

Son aquellas en que al enviar el request al servidor, la vista queda en espera de la respuesta del mismo. Este tipo de peticiones son más comunes, ya que se utilizan, por ejemplo, al escribir la url en el navegador, para consultar datos al servidor y cargarlos en una tabla o grilla, para actualizar un registro y esperar la confirmación exitosa, entre otras aplicaciones.

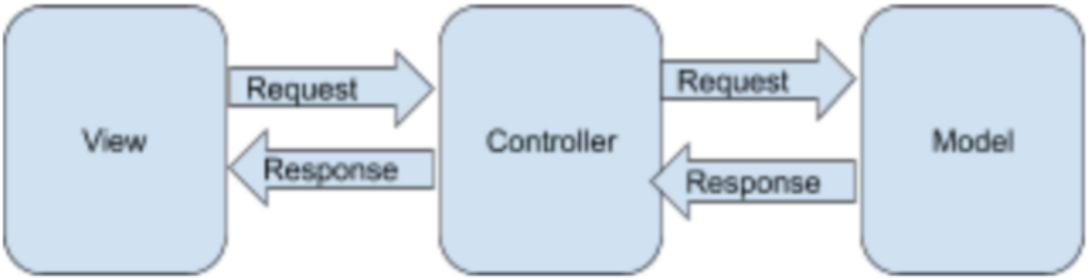

Imagen 12. Peticiones Síncronas.
Fuente: Desafío Latam.

En una explicación gráfica bajo la arquitectura MVC de Spring, en una petición síncrona, la vista queda esperando la respuesta (response) del servidor, por lo que esta sigue su ejecución normal.

Enviando datos al controlador

Ya hemos aprendido a enviar datos desde el controlador a la vista, puesto que nos queda realizar la función inversa, la cual es enviar datos desde la vista a nuestro controlador. Para esto, en este apartado vamos a seguir modificando el proyecto holamundospringmvc, en el cual realizaremos lo siguiente:

- Crearemos un formulario, con un cuadro de texto para ingresar el dato que queremos enviar en la pantalla del navegador o más bien en la vista del sistema. El dato se enviará por método POST.

- Crearemos dos nuevos métodos en el controlador HelloController.java: initSendData y sendData. Primero, iniciará la pantalla (vista addData) mediante una petición GET, para que el usuario pueda ingresar el dato. La segunda, ejecutará la acción al método sendData del controlador, mediante un botón, para que este reciba el dato desde la vista.

- Mostraremos en la consola del servidor (Eclipse IDE) el dato desde el controlador del sistema.

- Además, crearemos un formulario, en una nueva vista (addData), con un cuadro de texto para ingresar el dato que queremos enviar en la pantalla del navegador o más bien en la vista del sistema. El dato se enviará, esta vez, por método GET.

- Se creará una lista, para que almacene cada mensaje enviado al controlador addData.

- Mostraremos la lista actualizada por pantalla, esta vez en el mismo controlador (addData).

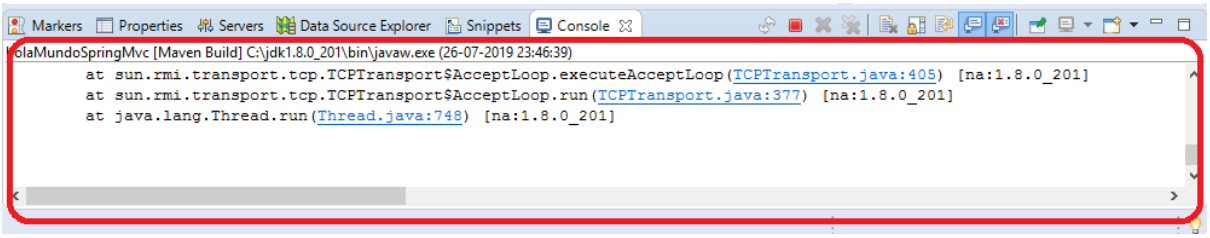

Imagen 13. Sección de Eclipse IDE correspondiente a la consola del servidor (log).
Fuente: Desafío Latam.

1. Crear archivo sendData.jsp bajo el directorio webapp/WEB-INF.

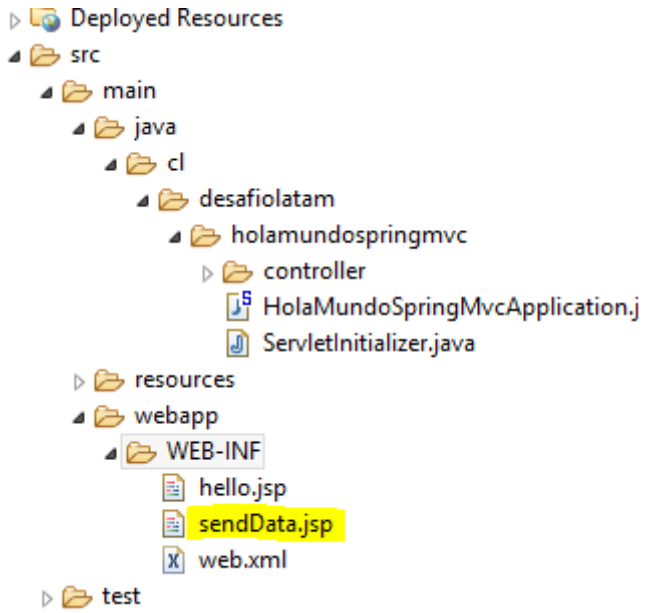

Imagen 14. Agregando sendData.jsp al proyecto.
Fuente: Desafío Latam.

2. Crear el cuadro de texto y el botón. En sendData.jsp, colocar el siguiente código html.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
      <form action="/saludos/sendData" method="post" id="frmSend">
            Dato a enviar: <input type="text" id="txtData" name="data"/>
            <input type="submit"  id="btnSend" value="Enviar dato"/>
      </form>
</body>
</html>
```

En el código anterior, es importante destacar que Spring, en el controlador, recibirá como parámetro el nombre (name) del input text mediante la anotación @RequestParam.

```
<input type="text" id="txtData" name="data"/>
```

El action de la etiqueta form, especifica la acción que se ejecutará en el controlador. La acción, corresponde a toda la parte de la url después del host:puerto. Ejemplo: http://localhost:8080/action.

3. En el controlador HelloController.java, agregar dos métodos:

```
@RequestMapping(value="/initSendData", method=RequestMethod.GET)
public String initSendData(ModelMap model) {
   return "sendData";
}
```

Este método nos permite recibir por método GET, la petición a la vista sendData.jsp. Para así mostrarla por pantalla.

```
@RequestMapping(value="/sendData", method=RequestMethod.POST)
public String sendData(ModelMap model, @RequestParam(value="data",
required = true) String dataReceived) {
```

```
    System.out.println("El dato recibido desde la vista sendData es: " +
dataReceived);
    return "sendData";
}
```

Este método nos permite recibir el dato ingresado en la vista, mediante el input text, y enviado al controlador. En este método, se utiliza la anotación @RequestParam, la cual permite mediante Spring MVC recibir el valor del parámetro especificado en el atributo name del input text de la vista.

El atributo value de la anotación @RequestParam, le dice a Spring que coloque el valor enviado desde la vista, con el nombre data en la variable dataReceived.

El atributo required de la anotación @RequestParam, le dice a Spring que el valor de "data" es obligatorio, en el caso de true, y opcional en el caso que required fuera false.

4. Iniciamos el servidor, y probamos que todo esté funcionando correctamente.

a. Entramos a la siguiente url: http://localhost:8080/saludos/initSendData.

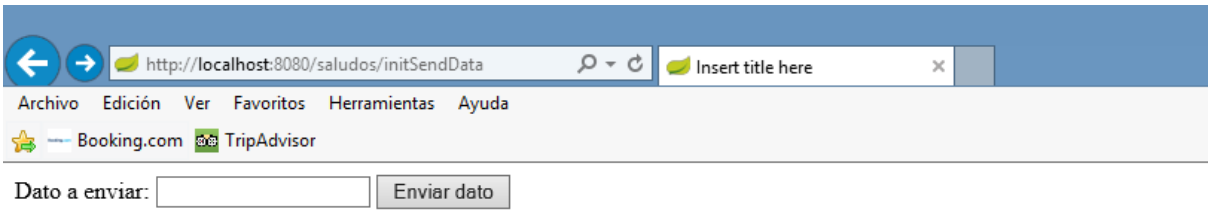

Imagen 15. Vista de datos a enviar.
Fuente: Desafío Latam.

b. Ingresamos un dato en el cuadro de texto y presionamos el botón "Enviar Dato".

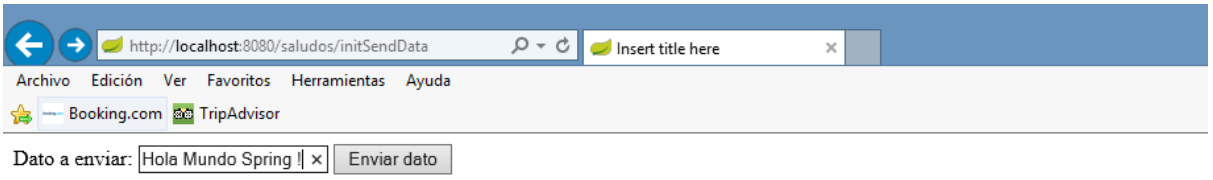

Imagen 16. Ingresando valor a enviar.
Fuente: Desafío Latam.

c. Revisamos la consola de Eclipse IDE.

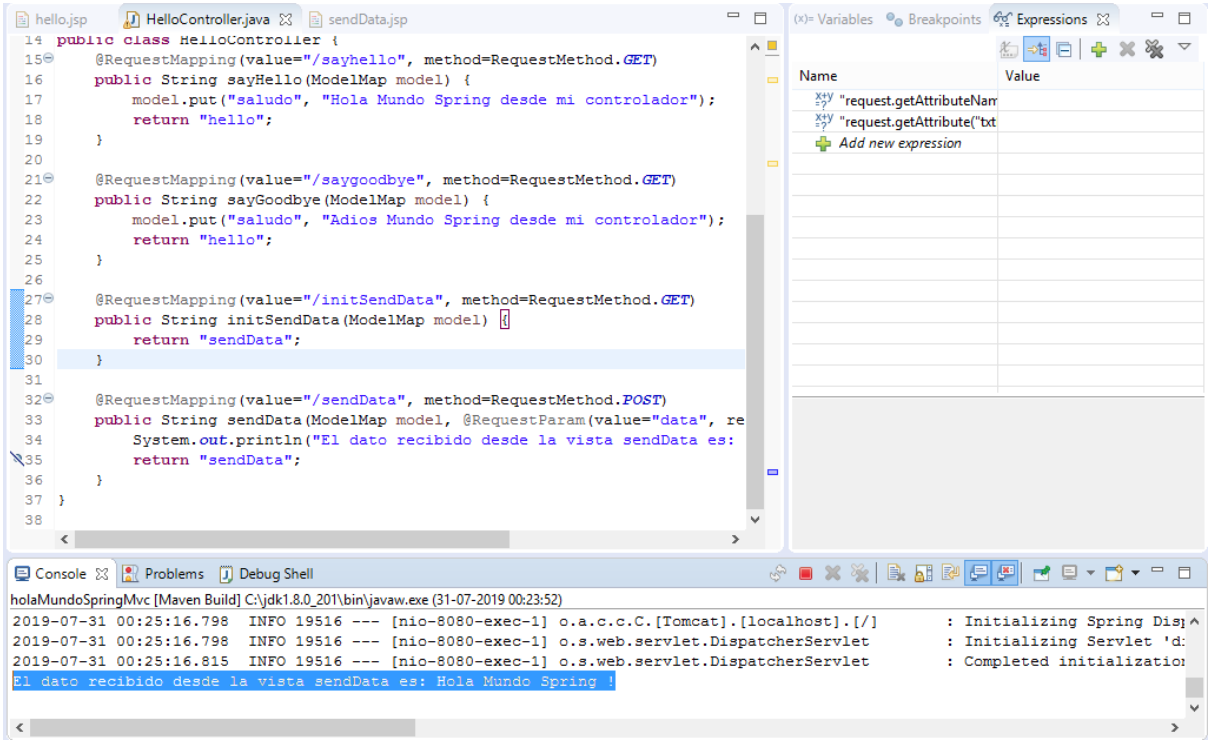

Imagen 17. Revisando la consola de Eclipse.
Fuente: Desafío Latam.

d. Comprobamos, que la url de nuestra aplicación cambió a la del action del request.

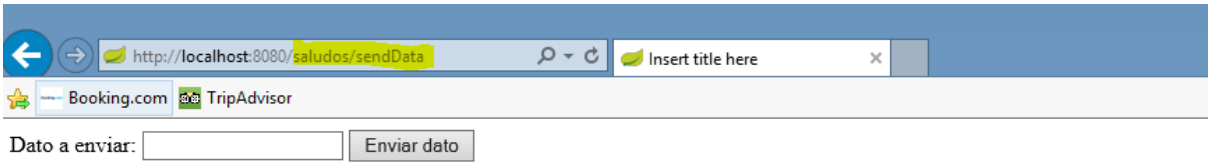

Imagen 18. Revisando el cambio de la URL.
Fuente: Desafío Latam.

5. Creamos una lista que almacenará cada mensaje enviado a la acción addData del controlador HelloController. Esta lista será instanciada en el constructor del controlador:

```
@Controller
@RequestMapping(value="/saludos", method=RequestMethod.GET)
public class HelloController {
   private List<String> listaMensajes;
   public HelloController() {
      // TODO Auto-generated constructor stub
      super();
```

```
        listaMensajes = new ArrayList<String>();
    }
```

6. Creamos el método addData, que almacenará cada mensaje enviado en una lista, devolviendo, en el modelo de Spring, la lista actualizada.

```
@RequestMapping(value="/addData", method=RequestMethod.GET)
public String saveData(ModelMap model, @RequestParam(value="data",
required = false) String dataReceived) {
    if(dataReceived != null) {
        listaMensajes.add(dataReceived);
    }
    System.out.println("El dato recibido desde la vista sendData es: " +
dataReceived);
    model.put("listaMensajes", listaMensajes);
    return "addData";
}
```

En este caso, el RequestParam es opcional, ya que la primera vez que se ejecute la aplicación este dato vendrá nulo.

```
public String saveData(ModelMap model, @RequestParam(value="data",
required = false)
```

Para evitar que la lista agregue el primer dato nulo (null), se evalúa para que solo se agregue el dato a la lista mientras este no sea nulo.

```
if(dataReceived != null) {
        listaMensajes.add(dataReceived);
    }
```

Se agrega la lista al modelo de Spring para después ser utilizada en la vista de nuestro programa.

```
model.put("listaMensajes", listaMensajes);
```

7. Crear la vista en /webapp/WEN-INF/ de nuestro proyecto.

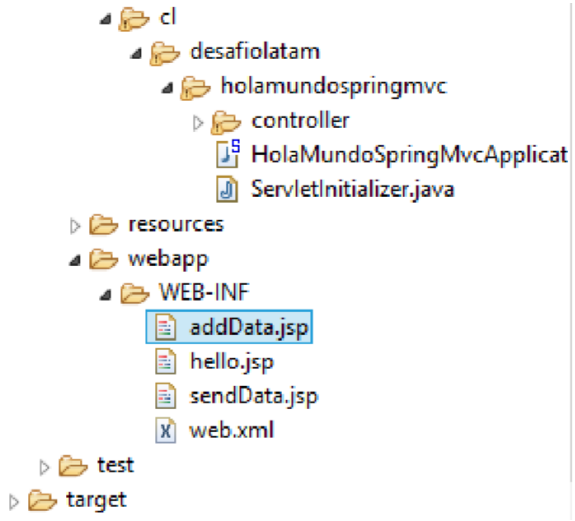

Imagen 19. Estructura de las vistas.
Fuente: Desafío Latam.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
   <form action="/saludos/addData" method="get" id="frmAdd">
      Dato a enviar: <input type="text" id="txtData" name="data"/>
      <input type="submit"  id="btnAdd" value="Add dato"/>
   </form>
   <c:forEach items="${listaMensajes}" var="mensaje">
      <c:out value="${mensaje}" /><br />
   </c:forEach>
</body>
</html>
```

El tag del core de jstl c:forEach, itera sobre una lista y deja el valor de cada iteración en una variable. En este caso, usamos esto para iterar sobre la lista del modelo “listaMensajes” y cada valor de cada ítem lo dejaremos en una variable “mensaje”.

```
<c:forEach items="${listaMensajes}" var="mensaje">
 <c:out value="${mensaje}" /><br />
</c:forEach>
```

Como ya hemos visto antes, con el tag c:out imprimimos la variable “mensaje” por la pantalla del navegador.

```
<c:out value="${mensaje}" /><br />
```

8. Levantamos el servidor, y probamos la aplicación con la siguiente url:

- http://localhost:8080/saludos/addData

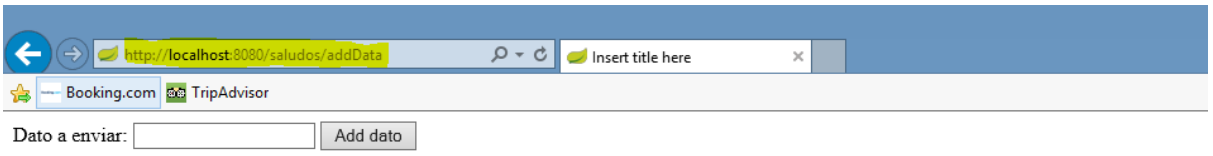

Imagen 20. Levantando el servicio para addData.
Fuente: Desafío Latam.

Ingresamos datos y presionamos el botón repetidas veces.

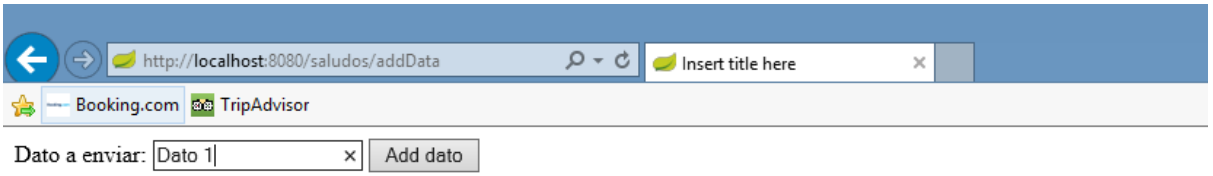

Imagen 21. Ingresamos el dato 1.
Fuente: Desafío Latam.

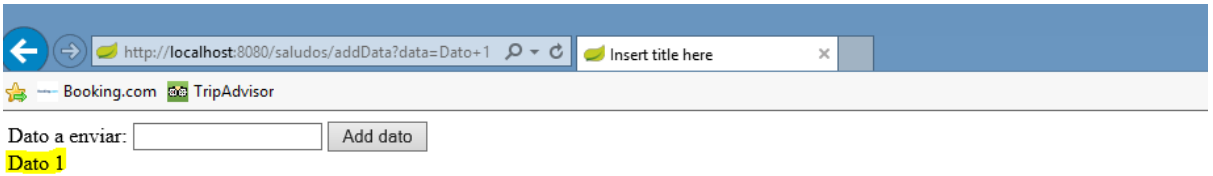

Imagen 22. Dato 1 ingresado.
Fuente: Desafío Latam.

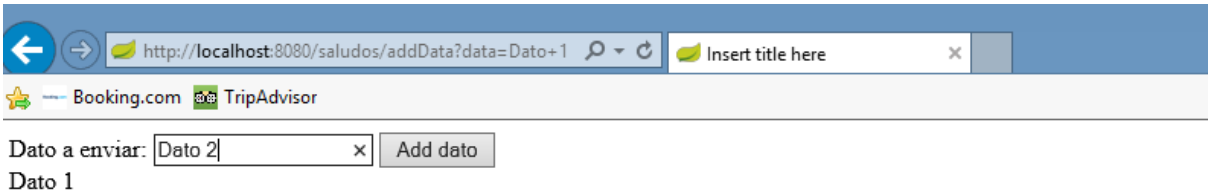

Imagen 23. Ingresamos Dato 2.
Fuente: Desafío Latam.

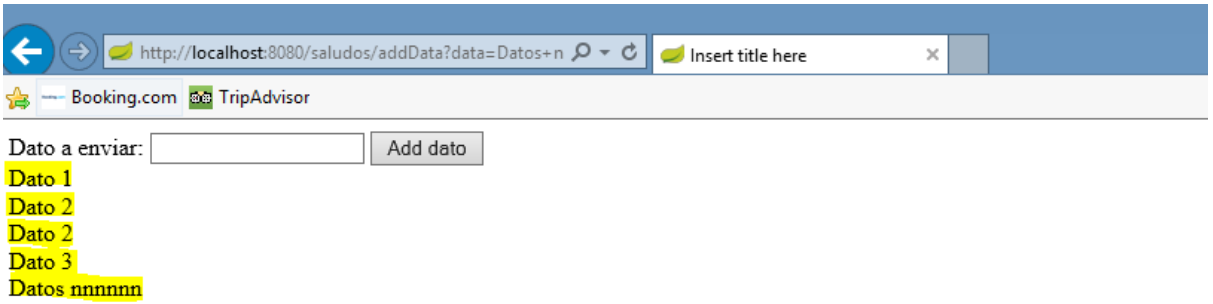

Imagen 24. Varios datos ingresados.
Fuente: Desafío Latam.

Cabe destacar, que el dato se mantiene y no se pierde en cada request (Cada vez que presionamos el botón), debido a que el controlador mantiene su estado en toda la sesión. Esto se puede cambiar mediante la anotación @Scope, indicando a la clase Controller que mantenga su estado, por ejemplo en cada petición o request.