

## La capa de servicios

<b>La capa de servicios</b>	<b>1</b>
¿Qué aprenderás?	2
Introducción	2
Capa de servicio o Service	2
Implementar la capa de servicio	4



**¡Comencemos!**

## ¿Qué aprenderás?

- Entender por qué se usa y se debe aplicar la capa de servicio.
- Integrar y aplicar interfaces de servicio en la aplicación.

## Introducción

La capa de servicio cumple un rol no menos importante en el desarrollo de una aplicación, ya que esta es un agregado a la arquitectura del software. Aprenderemos para qué nos sirve la capa de servicio, cuál es su importancia y cómo implementarla.

**¡Vamos con todo!**



## Capa de servicio o Service

La capa de servicio o Service, es un patrón de diseño que se aplica principalmente para separar las implementaciones de cada una de las capas del software que se está desarrollando. En este caso, la capa Service se aplicará para separar la implementación del Modelo con la implementación del controlador de la aplicación, convirtiéndose en el core de comunicación entre el controlador y nuestro modelo de datos.

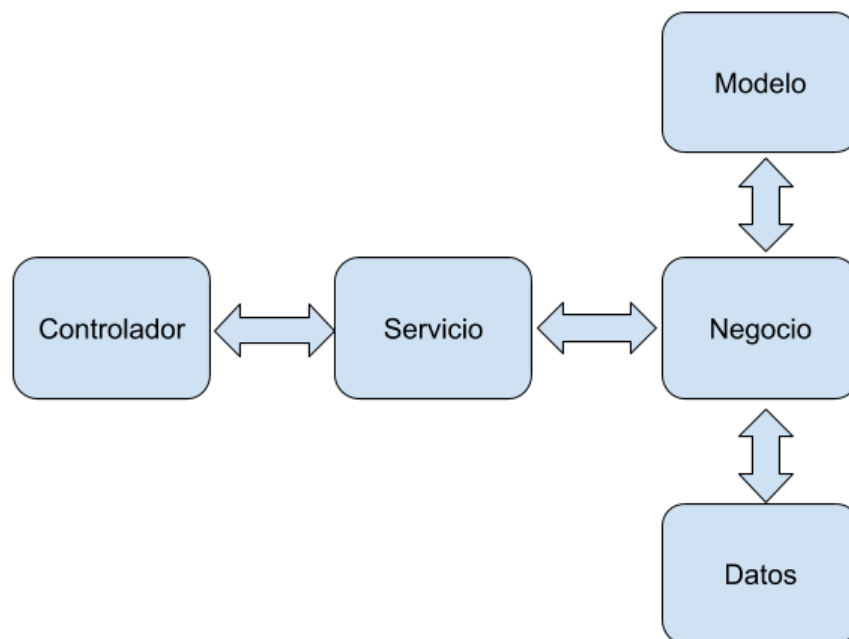


Imagen 1. Integrando la capa de servicio.  
Fuente: Desafío Latam.

En la imagen 1, se denotan todas las capas que debieran tener como mínimo un sistema desarrollado y bien diseñado con MVC. La capa de servicio, se implementa para generar una interfaz entre el controlador y el negocio. El Negocio por su parte, se encarga de ir a buscar los datos y devolver al controlador un objeto del modelo. Por ejemplo, si necesitamos mostrar los datos de una persona, la capa de negocio debería ser capaz de ir a buscar los datos correspondientes a la persona y luego generar el objeto de la clase `Persona.java` correspondiente al modelo. En resumen, el controlador recibirá un objeto `Persona`.

Spring, implementa la anotación `@Service` para aplicar a las implementaciones en la capa de servicio de la aplicación, de tal manera que cuando el servicio se inyecte con `@Autowired` en el controlador, Spring buscará la anotación `@Service` para crear el bean correspondiente. Es decir, lo que hará `@Autowired` es buscar un beans que implemente una determinada interfaz para hacer referencia a él, de tal forma que no sea necesario crear una instancia nueva cada vez que necesitemos usar su funcionalidad.



Imagen 2. Capa servicio.  
Fuente: Desafío Latam.

En la imagen 2 se muestra un primer plano de la capa de servicio. Aquí se ve, que esta se compone por su interfaz Service y por la implementación de la interfaz ServiceImpl. En ServiceImpl es donde se debe implementar toda la lógica de negocio, acceso a datos y creación de los objetos del modelo para que sean devueltos al controlador.

## Implementar la capa de servicio

1. Crear los siguientes package en nuestro proyecto
  - a. `cl.desafiolatam.holamundospringmvc.service`, en este package colocaremos la interfaz del servicio.
  - b. `cl.desafiolatam.holamundospringmvc.service.impl`, en este package colocaremos la implementación de la interfaz.
  - c. `cl.desafiolatam.holamundospringmvc.model`, en este package, colocaremos una clase modelo. El modelo, se tratará con mayor detalle en la siguiente unidad, sin embargo por ahora usaremos esta clase para el ejemplo.
2. En el package `cl.desafiolatam.holamundospringmvc.model`, crearemos la siguiente clase Modelo.

```
package cl.desafiolatam.holamundospringmvc.model;
@Component("mensaje")
public class Mensaje {
    String remitente;
    String mensaje;
    public String getRemitente() {
        return remitente;
    }
    public void setRemitente(String remitente) {
        this.remitente = remitente;
    }
    public String getMensaje() {
        return mensaje;
    }
    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }
}
```

La clase modelo "Mensaje", se ha anotado con `@Component("mensaje")`, para que Spring sepa que esto es un componente de Spring al momento de hacer `@Autowired` a esta clase, de la siguiente manera:

```
@Autowired
private Mensaje mensaje;
```

3. En el package `cl.desafiolatam.holamundospringmvc.service`, crear la interfaz con el siguiente código:

```
package cl.desafiolatam.holamundospringmvc.service;
import java.util.List;
import cl.desafiolatam.holamundospringmvc.model.Mensaje;
public interface MensajeService {
    List<Mensaje> getDataMessageList ();
    void saveDataMessage(Mensaje mensaje);
}
```

4. En el package `cl.desafiolatam.holamundospringmvc.service.impl`, crear la clase que implementará a la interfaz `MensajeService`.

```
package cl.desafiolatam.holamundospringmvc.service.impl;
import java.util.List;
import org.springframework.stereotype.Service;
import cl.desafiolatam.holamundospringmvc.model.Mensaje;
import cl.desafiolatam.holamundospringmvc.service.MensajeService;
@Service("mensajeService")
public class MensajeServiceImpl implements MensajeService{
    @Override
    public List<Mensaje> getDataMessageList() {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public void saveDataMessage(Mensaje mensaje) {
        // TODO Auto-generated method stub
    }
}
```

Esta clase, se ha marcado con la anotación `@Service("mensajeService")`, en donde se le está pasando el nombre del bean que queremos que Spring crea. Este nombre, se usará

próximamente cuando queramos inyectar el servicio con `@Autowired`, de la siguiente manera:

```
@Autowired
private MensajeService mensajeService; //El nombre de la variable
mensajeService corresponde al nombre del bean @Service("mensajeService")
```

Además, notar que `mensajeService` se está declarando mediante la interfaz `MensajeService`, lo cual nos permite separar la implementación.

5. En el controlador `HelloController.java`, crear los siguientes métodos:

```
@RequestMapping(value="/messageList", method=RequestMethod.GET)
public String getDataMessageList() {
    // TODO Auto-generated method stub
    return mensajes;
}
@RequestMapping(value="/saveMessage", method=RequestMethod.POST)
public String saveDataMessage(@ModelAttribute("mensaje") Mensaje
mensaje) {
    // TODO Auto-generated method stub
    return mensajes;
}
```

El método `getDataMeageList`, obtendrá la lista de mensajes que se irán agregando mediante el segundo método llamado `saveDataMessage`. En este último, se está utilizando la anotación `@ModelAttribute`, el cual le está indicando al método del controlador que desde la vista va a recibir un objeto modelo llamado "mensaje".

6. Inyectamos el servicio en el controlador `HelloController`. Colocar en la sección de declaraciones del controlador, lo siguiente:

```
public class HelloController {
    @Autowired
    private MensajeService mensajeService;
```

7. Luego, debemos implementar la lógica de los métodos de la clase `MensajeServiceImpl.java`. Primero, para poder probar rápidamente, implementaremos la siguiente lógica en el método `getDataMessageList()`.

- a. Inyectamos a la clase `MensajeServiceImpl.java`, el modelo `Mensaje.java` y declaramos una lista de mensajes.

```
@Service("mensajeService")
public class MensajeServiceImpl implements MensajeService{
    @Autowired
    private Mensaje mensaje;
    private List<Mensaje> messageList;
```

- b. Creamos el constructor de la clase MensajeServiceImpl.java, para así poder instanciar la lista de mensajes.

```
MensajeServiceImpl(){
    super();
    messageList = new ArrayList<Mensaje>();
}
```

- c. Creamos la siguiente lógica en el método getDataMessageList(), que crea una lista de mensajes de prueba.

```
public List<Mensaje> getDataMessageList() {
    // TODO Auto-generated method stub
    mensaje.setRemitente("Pepe");
    mensaje.setMensaje("Quiere pan");
    messageList.add(mensaje);
    return messageList;
}
```

8. En el controlador HelloController.java, en el método getDataMessageList(), debemos llamar al método getDataMessageList del servicio MensajeService y agregar el resultado (La lista de mensajes) al modelo de la vista.

```
@RequestMapping(value="/messageList", method=RequestMethod.GET)
public String getDataMessageList(ModelMap model) {
    // TODO Auto-generated method stub
    model.addAttribute("dataMessageList",
    mensajeService.getDataMessageList());
    return "mensajes";
}
```

9. Ahora, debemos crear la vista mensajes.jsp, en donde listamos todos los mensajes que contienen el modelo dataMessageList. Crear la vista con el siguiente código:

```
<%@ 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:forEach items="${dataMessageList}" var="mensaje">
        <c:out value="${mensaje.getRemitente()} dice que
        ${mensaje.getMensaje()}" /><br />
    </c:forEach>
</body>
</html>
```

Nótese, que en el `forEach` del código anterior, se está iterando sobre la lista de mensajes "dataMessageList" devuelta desde el controlador.

10. Probamos si la vista muestra correctamente el mensaje por pantalla. Iniciar el servidor e ingresar a la URL.

  - <http://localhost:8080/saludos/messageList>

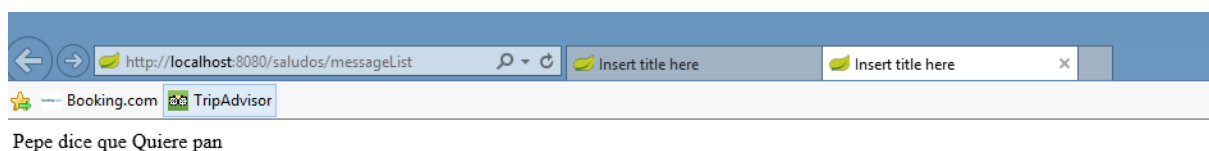


Imagen 3. Vista resultante.  
Fuente: Desafío Latam.

11. Siguiendo con el ejemplo, ahora vamos a crear el formulario que agrega mensajes a la lista. En este caso, también se debe agregar la lógica en la implementación del servicio. Primero, debemos agregar el tag `form` de Spring MVC en nuestra vista `mensajes.jsp`.

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
```

12. Creamos el formulario con el tag `form` agregado en el punto 11, e indicando lo siguiente:



- **Modelo que vamos a enviar:** @ModelAttribute("mensaje") Mensaje mensaje, recordemos que en el método del controlador, se definió el modelo con el que vamos a trabajar. En este caso, este se llama "mensaje".
- Acción que queremos ejecutar, en este caso será "/saludos/saveMessage"

```
<form:form id="frmMensajes" action="/saludos/saveMessage"
modelAttribute="mensaje">
</form:form>
```

En el código anterior, notamos como se declara el modelAttribute="mensaje". Esto es gracias al tag form de Spring MVC.

13. Dentro del formulario creado anteriormente, crearemos:

- 2 cuadros de textos, uno para el remitente y el otro para el mensaje.
- Un botón que envíe el mensaje mediante el action del formulario.

```
<form:form id="frmMensajes" action="/saludos/saveMessage"
modelAttribute="mensaje">
    Remitente: <input type="text" id="txtRemitente" name="remitente" />
<br />
    Mensaje: <input type="text" id="txtMensaje" name="mensaje" /> <br />
    <input type="submit" id="btnEnviar" value="Enviar" />
</form:form>
```

Destacando el código anterior, para que esto funcione, los atributos name de cada cuadro de texto deben tener el mismo nombre que el atributo del modelo mensaje.java.

14. Para comprobar, en esta instancia que todo está funcionando como lo esperamos, en el controlador HelloController, imprimir por consola lo siguiente:

```
System.out.println(mensaje.getRemitente() + " dice que " +
mensaje.getMensaje());
```

15. Iniciamos el servidor y verificamos nuestra aplicación en la siguiente url:

- <http://localhost:8080/saludos/messageList>.
- a. Ingresar datos en los cuadro de textos correspondientes y luego presionar el botón Enviar:

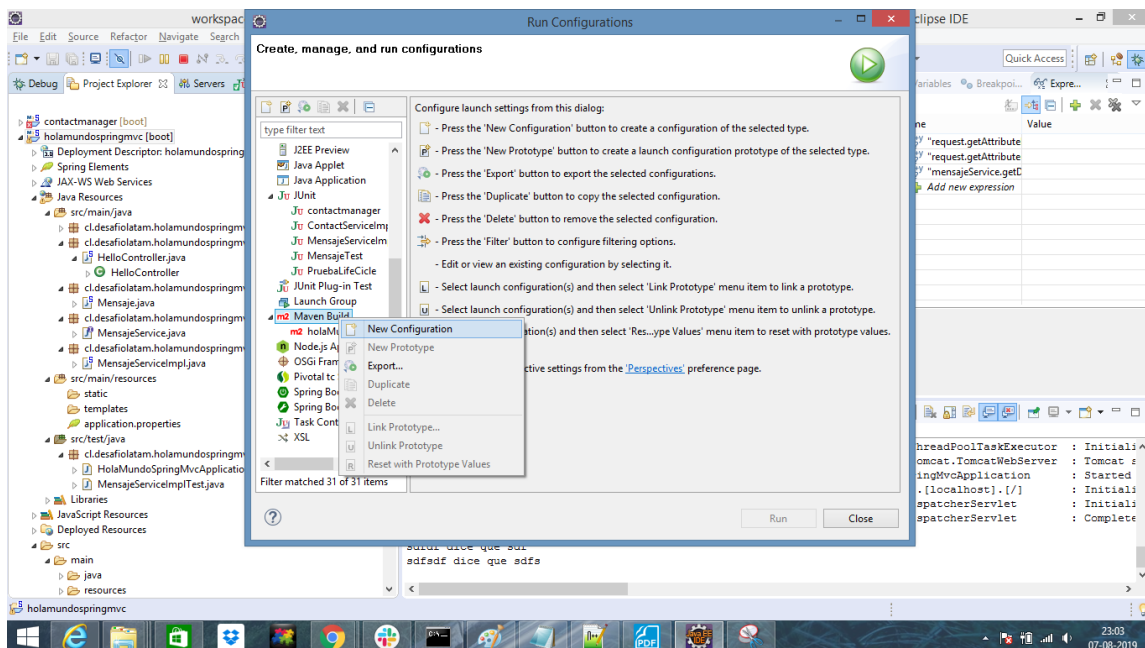


Imagen 4. Enviando el mensaje.

Fuente: Desafío Latam.

#### b. Verificar la consola de Eclipse IDE

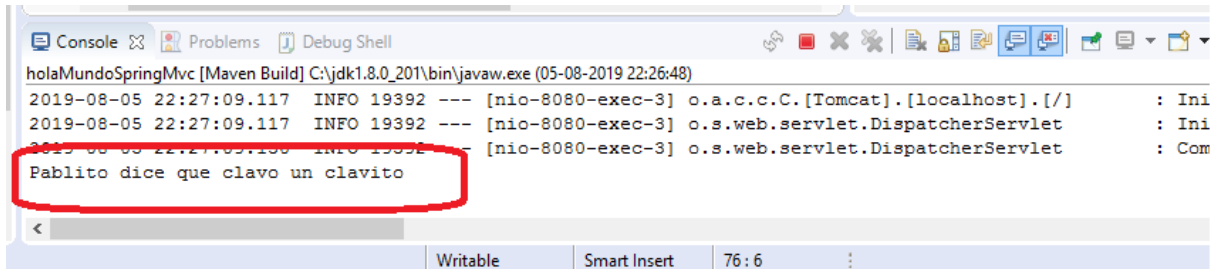


Imagen 5. Verificación en consola.

Fuente: Desafío Latam.

16. Nos queda solo implementar, el agregar mensaje en nuestra clase Service. En la clase `MensajesServiceImpl.java`, en el método `saveDataMessage()`, incluir el siguiente código:

```
messageList.add(mensaje);
```

El método completo, quedaría de la siguiente manera:

```
@Override
public void saveDataMessage(Mensaje mensaje) {
    /* Se evalúa si remitente es nulo, de tal forma que no se agregue
```

```
el item a la lista*/
    if(mensaje.getRemitente() != null) {
        messageList.add(mensaje);
    }
}
```

17. En el constructor de la clase `MensajeServiceImpl.java`, agregar la siguiente línea para inicializar la lista.

```
messageList.clear();
```

El código completo del constructor, quedaría de la siguiente manera:

```
MensajeServiceImpl(){
    super();
    messageList = new ArrayList<Mensaje>();
    messageList.clear();
}
```

18. En la clase controller `HelloController.java`, en el método `saveDataMessage`, debemos agregar las siguientes líneas:

```
/*Enviamos el objeto mensaje, recibido desde la vista, a nuestro
servicio*/
mensajeService.saveDataMessage(mensaje);
/*Obtenemo la lista de mensajes desde nuestro servicio, y la
actualizamos en el modelo*/
model.addAttribute("dataMessageList",
mensajeService.getDataMessageList());
```

El código del método completo, quedaría de la siguiente manera:

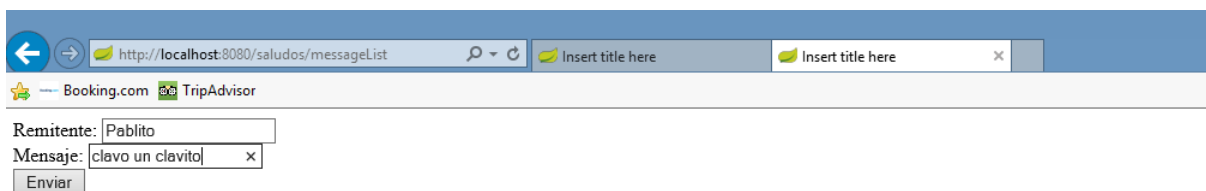
```
@RequestMapping(value="/saveMessage", method=RequestMethod.POST)
public String saveDataMessage(ModelMap model, @ModelAttribute("mensaje")
Mensaje mensaje) {
    // TODO Auto-generated method stub
    System.out.println(mensaje.getRemitente() + " dice que " +
mensaje.getMensaje());
    /*Enviamos el objeto mensaje, recibido desde la vista, a nuestro
servicio*/
    mensajeService.saveDataMessage(mensaje);
```

```
/*Obtenemos la lista de mensajes desde nuestro servicio, y la
actualizamos en el modelo*/
model.addAttribute("dataMessageList",
mensajeService.getDataMessageList());
return "mensajes";
}
```

19. Probamos en la siguiente URI la implementación completa:

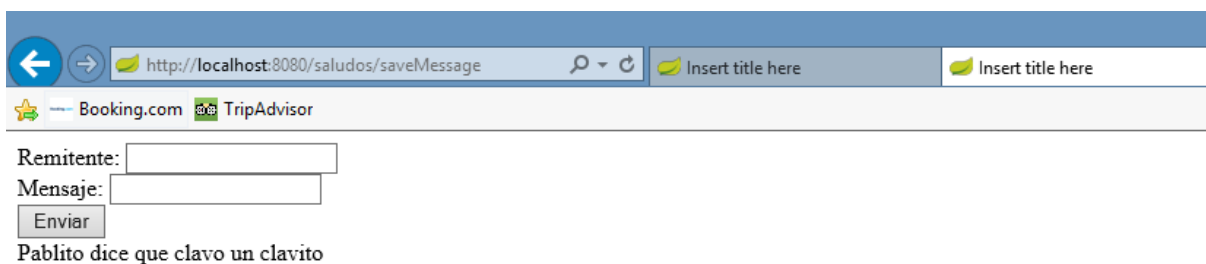
- <http://localhost:8080/saludos/messageList>

a. Ingresamos datos y presionamos el botón Enviar.



The screenshot shows a web browser window with the address bar displaying `http://localhost:8080/saludos/messageList`. The page contains a form with two input fields: "Remitente:" with the value "Pablito" and "Mensaje:" with the value "clavo un clavito". Below the fields is a button labeled "Enviar".

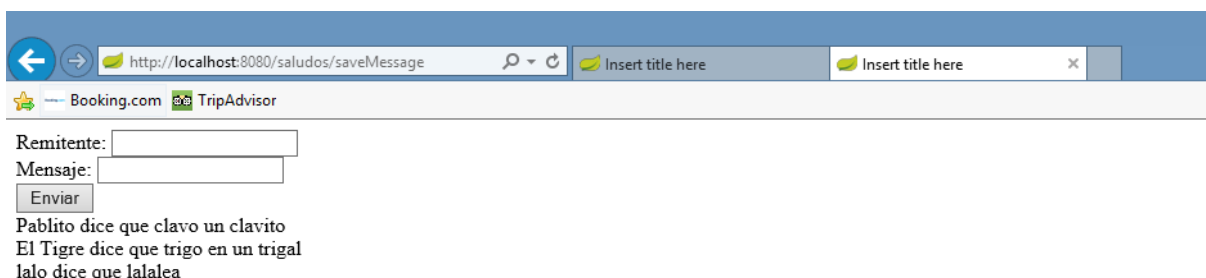
Imagen 6. Enviando datos.



The screenshot shows the same web browser window, but the address bar now displays `http://localhost:8080/saludos/saveMessage`. The form fields are empty. Below the "Enviar" button, the text "Pablito dice que clavo un clavito" is displayed.

Imagen 7. Mensaje enviado.  
Fuente: Desafío Latam.

b. Realizar lo anterior, repetidas veces.



The screenshot shows the same web browser window, but the address bar now displays `http://localhost:8080/saludos/saveMessage`. The form fields are empty. Below the "Enviar" button, the text "Pablito dice que clavo un clavito", "El Tigre dice que trigo en un trigal", and "lalo dice que lalalea" are displayed.

Imagen 8. Varios mensajes enviados.  
Fuente: Desafío Latam.