



Vista-Controlador_

Parte II

{desafío}
latam_







Inicio

{desafío}
latam_



15 Minutos

*/*Configurar eclipse correctamente para empaquetar una aplicación con maven.*/**

*/*Generar el empaquetado de la aplicación para ser instalado en un servidor de aplicaciones.*/**

Objetivo



Desarrollo

{desafío}
latam_

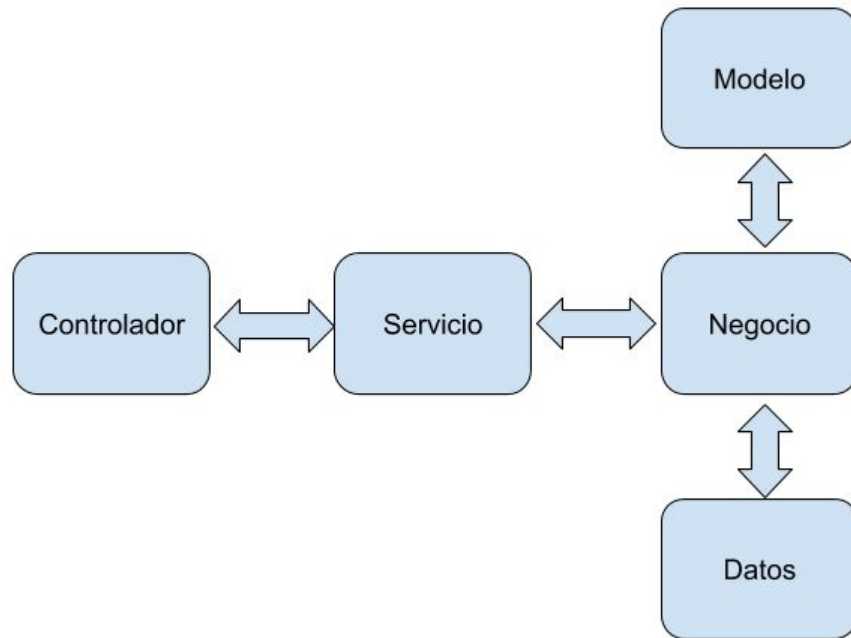


150 Minutos

/*La capa de servicios*/

La capa de servicio o Service

- Patrón de diseño que se aplica para separar las implementaciones de las capas del software que se está desarrollando.
- Se implementa para generar una interfaz entre el controlador y el negocio.



- Spring, implementa la anotación `@Service` para aplicar a las implementaciones en la capa de servicio de la aplicación.



Capa servicio.

Implementar la capa de servicio

1. Crear los siguientes package:

- a. `cl.desafiolatam.holamundospringmvc.service`
- b. `cl.desafiolatam.holamundospringmvc.service.impl`
- c. `cl.desafiolatam.holamundospringmvc.model`

2. En cl.desafiolatam.holamundospringmvc.model, crearemos la 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”).

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

3. En cl.desafiolatam.holamundospringmvc.service, crear la interfaz.

```
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 `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
    }
}
```

Se ha marcado con la anotación `@Service("mensajeService")`

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

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;
}
```

6. Inyectamos el servicio en el controlador HelloController.

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

7. Implementar la lógica de los métodos de la clase MensajeServiceImpl.java.

- Inyectamos a la clase MensajeServiceImpl.java, el modelo Mensaje.java

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

- Creamos el constructor de la clase MensajeServiceImpl.java

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

- Creamos la siguiente lógica en el método getDataMessageList().

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

8. Llamar al método `getDataMessageList` del servicio `MensajeService` y agregar el resultado al modelo de la vista.

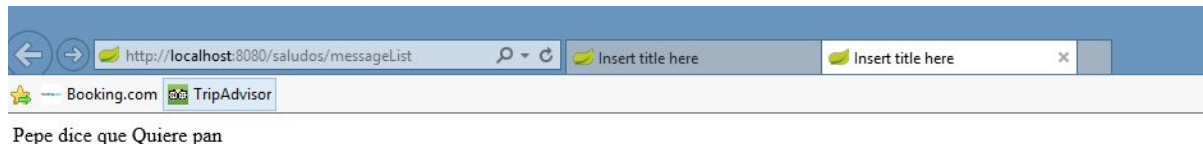
```
@RequestMapping(value="/messageList",
method=RequestMethod.GET)
public String getDataMessageList(ModelMap model) {
    // TODO Auto-generated method stub
    model.addAttribute("dataMessageList",
mensajeService.getDataMessageList());
    return "mensajes";
}
```


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

10. Iniciar el servidor e ingresar a la URL.

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



11. Crear el formulario que agrega mensajes a la lista.

```
<%@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.
Acción que queremos ejecutar.

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

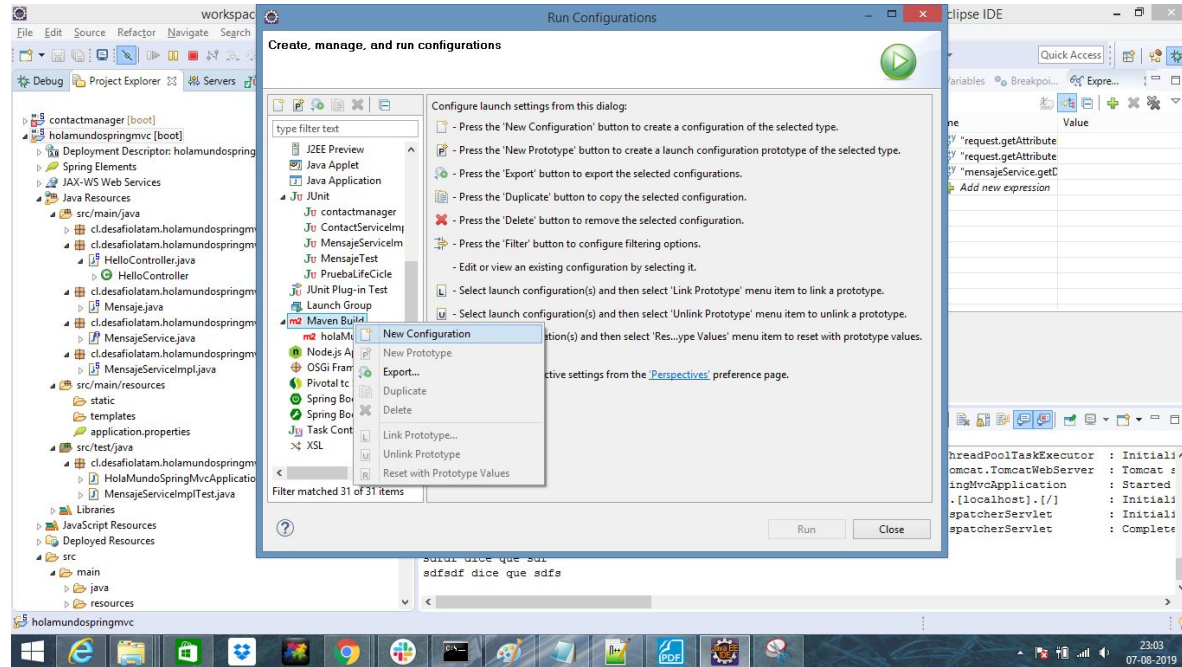
13. Dentro del formulario creado anteriormente, crearemos:
2 cuadros de textos.
Un botón que envíe el mensaje mediante el action del formulario.

```
<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>
```

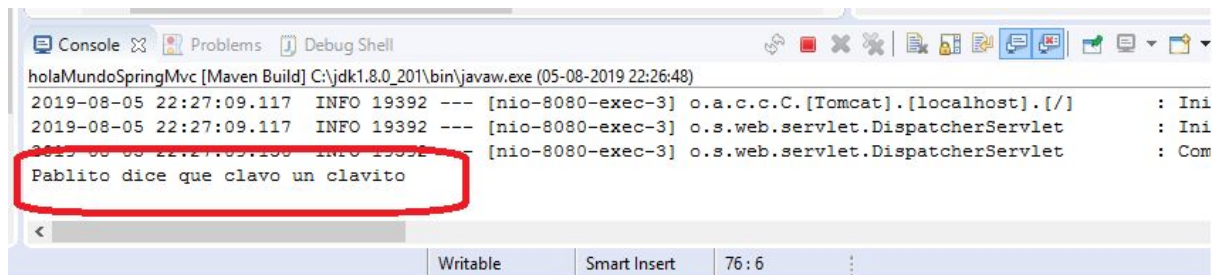
14. Comprobar, que todo está funcionando como lo esperamos.

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

15. Verificamos nuestra aplicación en la siguiente url.
<http://localhost:8080/saludos/messageList>.



Verificación en consola.



```
holaMundoSpringMvc [Maven Build] C:\jdk1.8.0_201\bin\javaw.exe (05-08-2019 22:26:48)
2019-08-05 22:27:09.117 INFO 19392 --- [nio-8080-exec-3] o.a.c.c.C.[Tomcat].[localhost].[/] : Ini
2019-08-05 22:27:09.117 INFO 19392 --- [nio-8080-exec-3] o.s.web.servlet.DispatcherServlet : Ini
2019-08-05 22:27:09.118 INFO 19392 --- [nio-8080-exec-3] o.s.web.servlet.DispatcherServlet : Com
Pablito dice que clavo un clavito
```

16. Nos queda solo implementar, el agregar mensaje en nuestra clase Service.

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

Método completo.

```
@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. Agregar la siguiente línea para inicializar la lista.

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

Código completo del constructor.

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

18. 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());
```

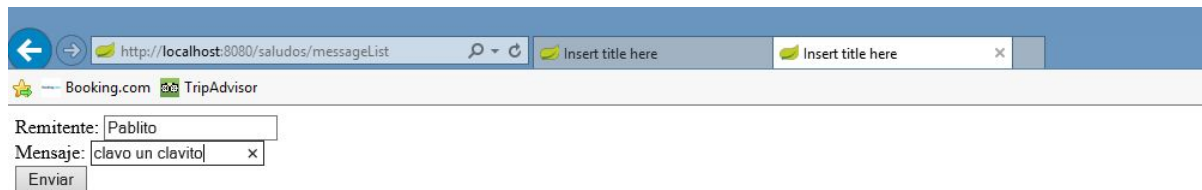
Código del método completo.

```
@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>

Enviando datos

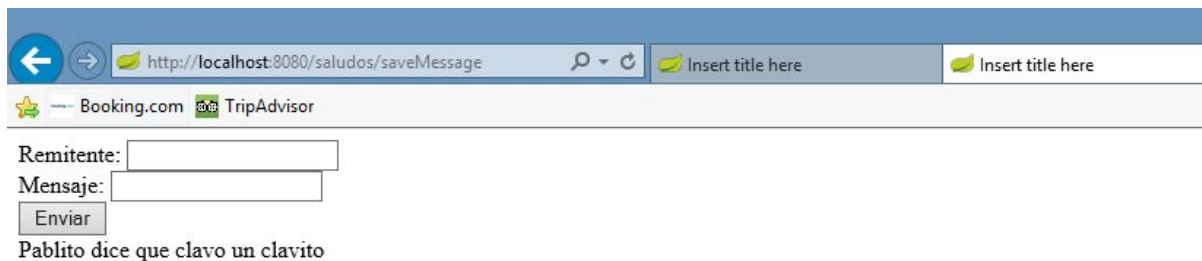


Remitente: Pablito

Mensaje: clavo un clavito

Enviar

Mensaje enviado.



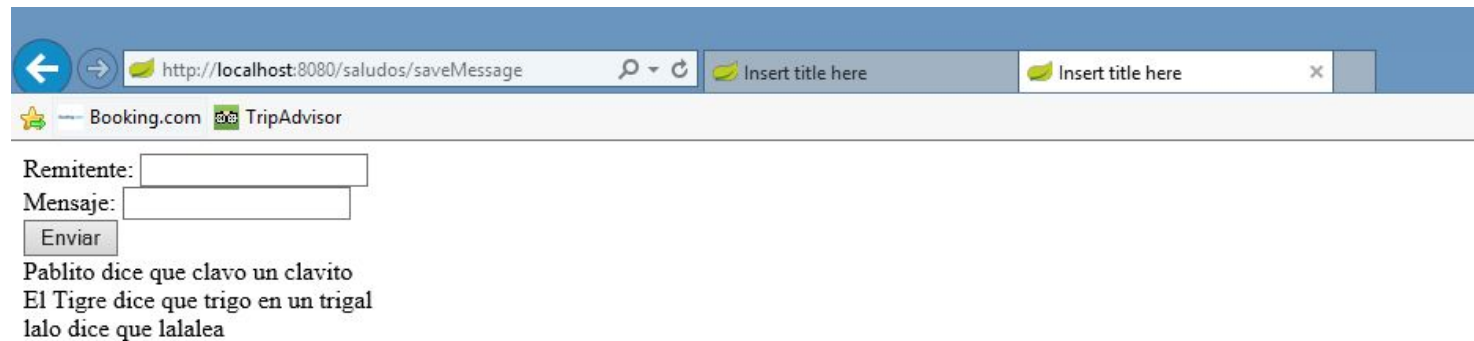
Remitente:

Mensaje:

Enviar

Pablito dice que clavo un clavito

Varios mensajes enviados.



← → http://localhost:8080/saludos/saveMessage 🔍 ↻

★ Booking.com 🌐 TripAdvisor

Remitente:

Mensaje:

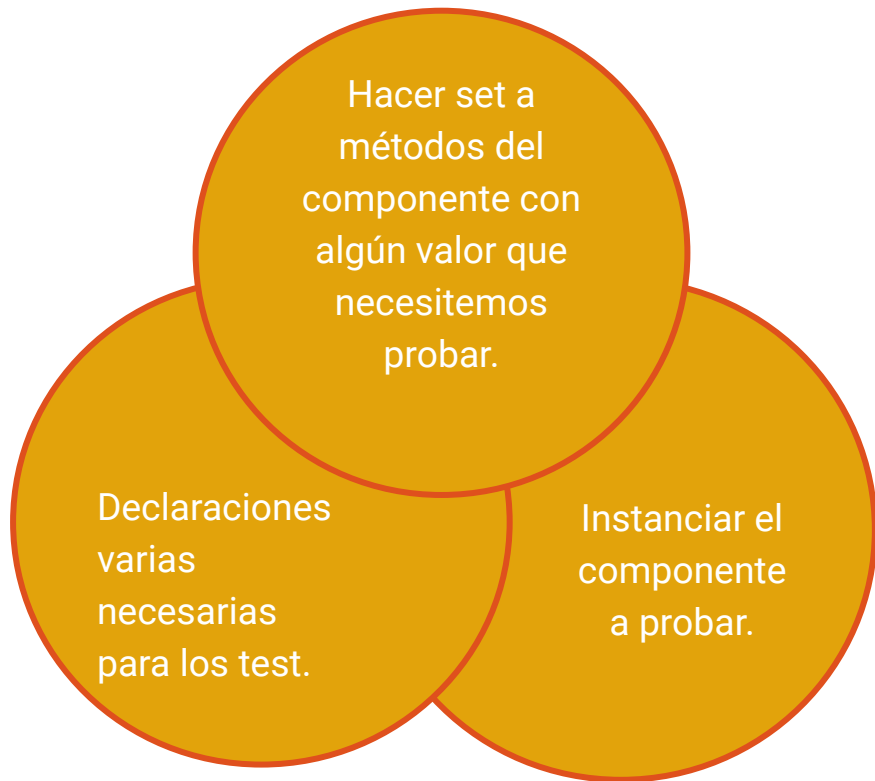
Pablito dice que clavo un clavito
El Tigre dice que trigo en un trigal
lalo dice que lalalea

/*Unidades de pruebas*/

Pruebas con la librería JUnit

- ¿Que se debe hacer antes de que se ejecute el test?

Implementar la anotación de JUnit @Before, nos sirve para:



¿Cuales son los test o pruebas relevantes que debemos hacer a nuestro componente?

- Usar la anotación @Test al inicio de cada prueba.
- Saber cuáles son las pruebas unitarias que deseemos y necesitemos certificar, escribir las pruebas unitarias antes de codificarlas, por ejemplo de la siguiente manera:

Caso 1:

El componente, al ser declarado, todos sus atributos deben tener el valor null.

Caso 2:

Al ejecutar el método setValor del Componente con un parámetro null, debería arrojar un error.

Estructura básica de un test unitario con JUnit:

```
public class NombreClaseTest {
    @Before
    public void setUp() throws Exception {
        /*Código que se ejecutará antes de los test*/
    }
    @Test
    public void test1() {
        /* Código de prueba para el caso 1 */
    }
    @Test
    public void test2() {
        /* Código de prueba para el caso 2 */
    }
    @Test
    public void testn() {
        /* Código de prueba para el caso n */
    }
}
```

Assert en JUnit

Realizan comparaciones en base a lo que se espera obtener de una prueba en particular.

- Tipos de assert:
 - **AssertEquals:** compara un valor esperado en base a un resultado.

```
@Test
public void resultadoEsCero() {
    assertEquals(0, 1-1);
}
```

- **AssertNotEquals:** evalúa si un valor esperado no es igual al resultado.

```
@Test
public void resultadoNoEsCero() {
    assertEquals(0, 1-2);
}
```

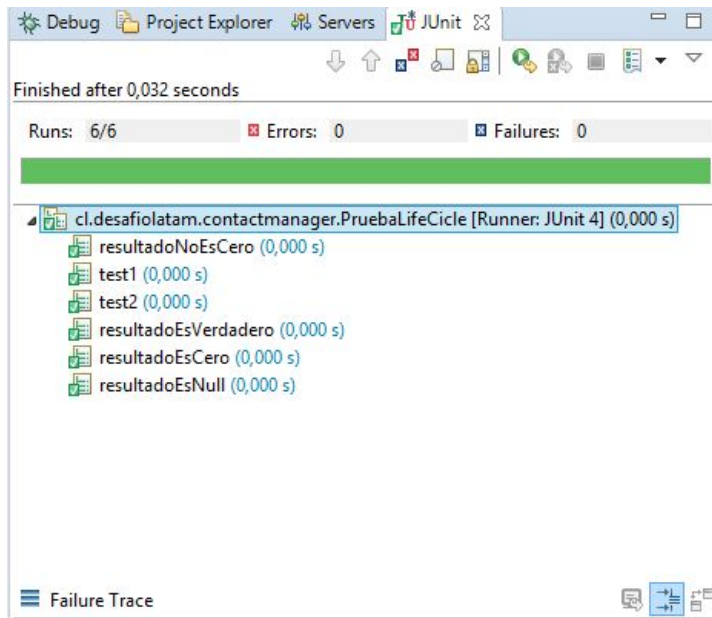
- **AssertTrue:** evalúa si una expresión es igual a verdadero.

```
@Test
public void resultadoEsVerdadero() {
    assertTrue(1<2);
}
```

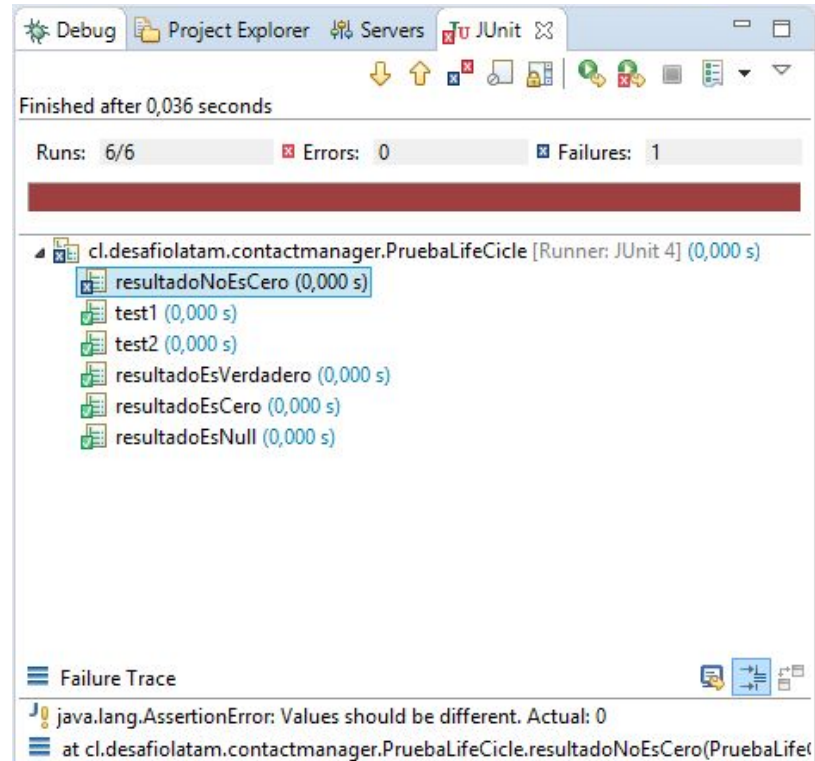

- **AssertNull:** evalúa si un objeto o algo en el código es null.

```
@Test
    public void resultadoEsNull() {
        assertNull(nombre);
    }
```

Casos de prueba JUnit.



Casos ejecutados, aprobados y fallados.

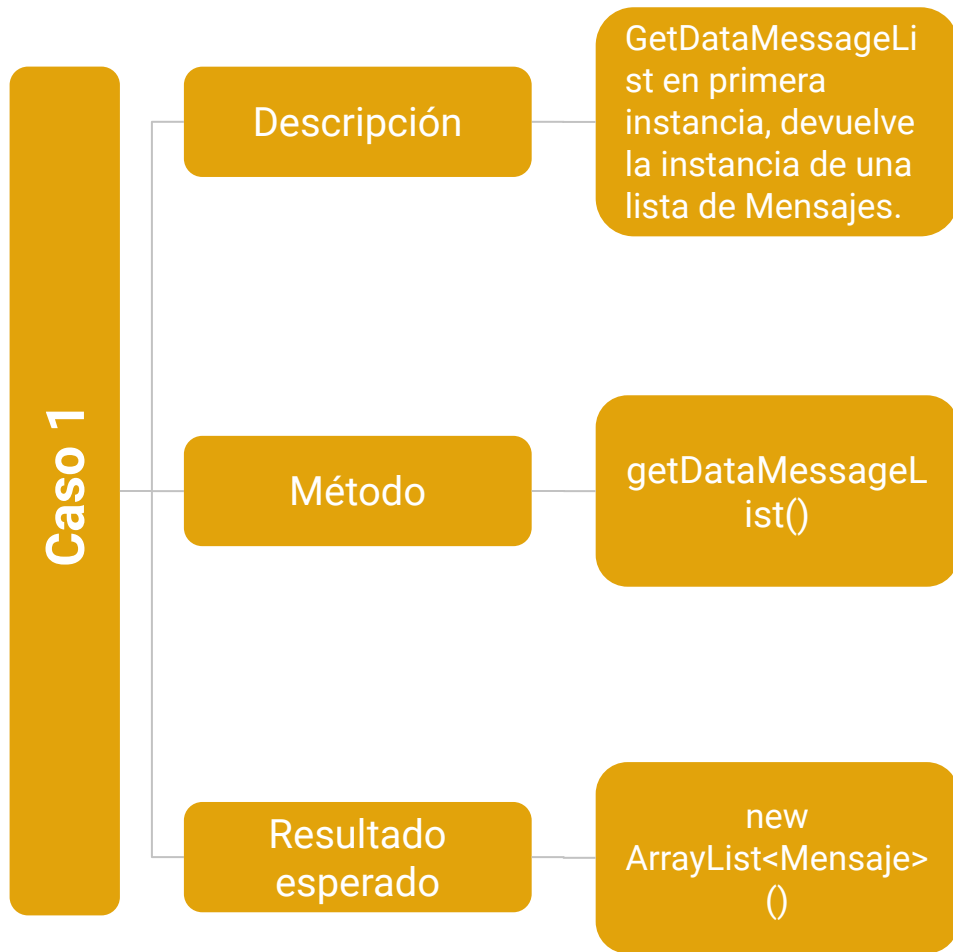


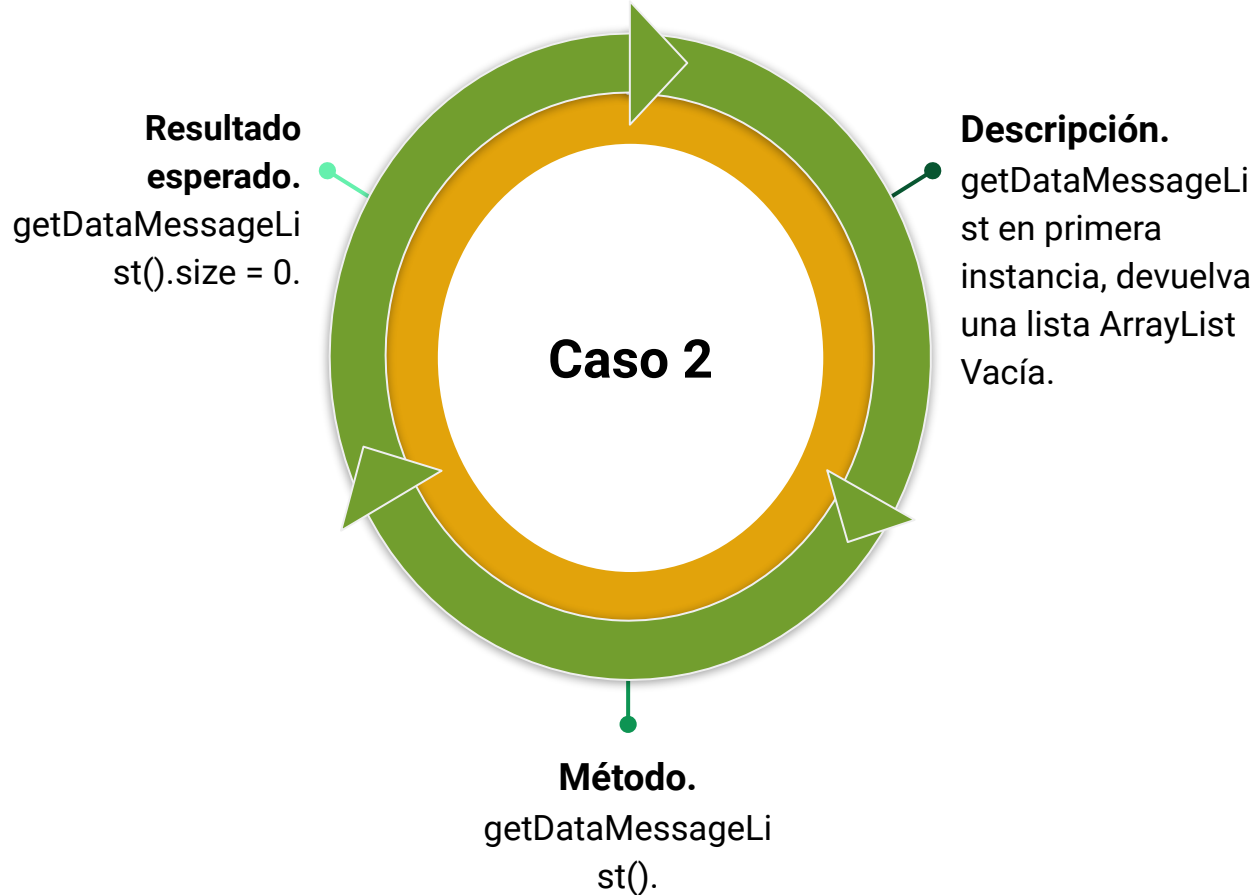
El test la imagen anterior:

```
@Test
    public void resultadoNoEsCero() {
        assertEquals(0, 1-1);
    }
```

Implementación JUnit a la clase MensajeServiceImpl

1. Definición de nuestros casos a probar:





Método.

saveDataMessage() -
getDataMessageList()

Descripción.

Agregar un nuevo
mensaje.

Resultado esperado.

Ejecutar
saveDataMessage
y luego
getDataMessageList.

2. Crear en cl.desafiolatam.holamundospringmvc la clase MensajeServiceImplTest.java.
3. Implementar el siguiente código en la clase test creada:

```
package cl.desafiolatam.holamundospringmvc;
import static org.junit.Assert.assertEquals;
import java.util.ArrayList;
import org.junit.Before;
import org.junit.Test;
import cl.desafiolatam.holamundospringmvc.model.Mensaje;
import cl.desafiolatam.holamundospringmvc.service.MensajeService;
import cl.desafiolatam.holamundospringmvc.service.impl.MensajeServiceImpl;
public class MensajeServiceImplTest {
    private Mensaje mensaje;
    private MensajeService mensajeService;
    @Before
    public void setUp() throws Exception {
        System.out.println("setUp");
        mensaje = new Mensaje();
        mensajeService = new MensajeServiceImpl(mensaje);
    }
}
```

Caso 1:

```
@Test
public void caso1_obtener_lista_mensajes()
{
    assertEquals(new ArrayList<Mensaje>(),
mensajeService.getDataMessageList());
}
```

Caso 1

Descripción

GetDataMessageList en primera instancia, devuelve la instancia de una lista de Mensajes.

Método

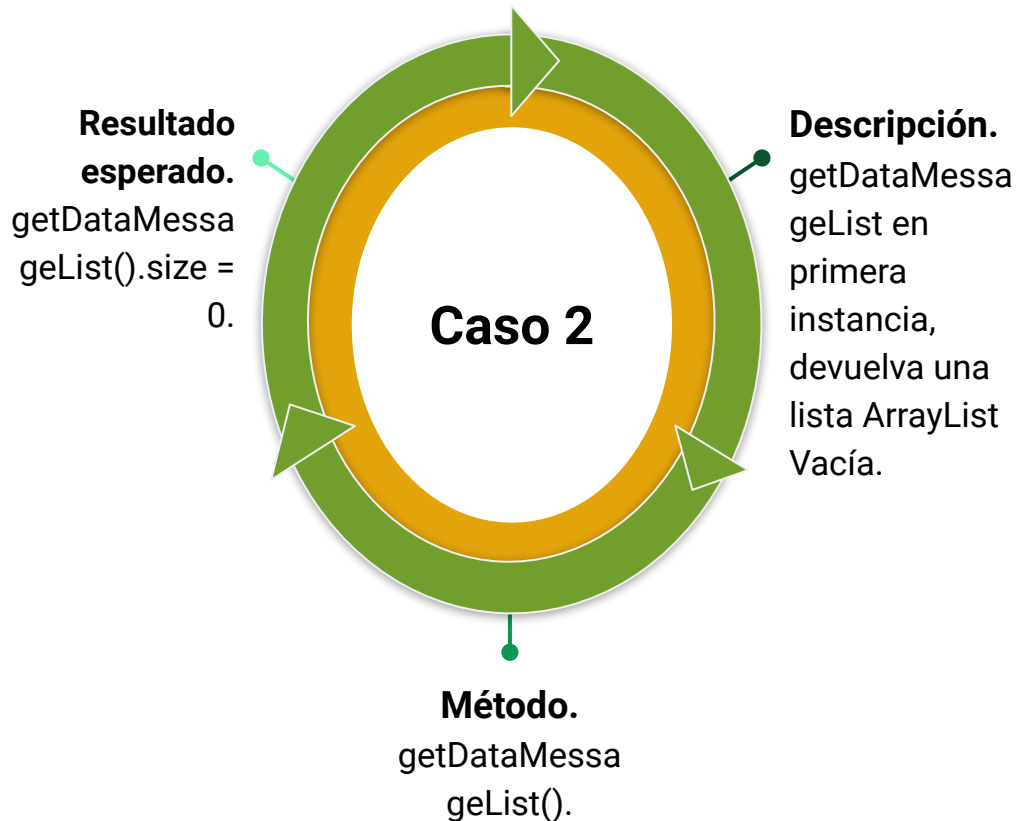
getDataMessageList()

Resultado esperado

new ArrayList<Mensaje>()

Caso 2:

```
@Test
public void
caso2_obtener_lista_contacto() {
    assertEquals(0,
mensajeService.getDataMessageList().size()
);
}
```



Caso 3:

```
@Test
public void
caso3_agregar_mensaje() {

    mensaje.setRemitente("Pablito");
    mensaje.setMensaje("clavo un
    clavito");

    mensajeService.saveDataMessage(mens
    aje);
    assertEquals(mensaje,
    mensajeService.getDataMessageList()
    .get(0));
}
```

{desario}
latam_

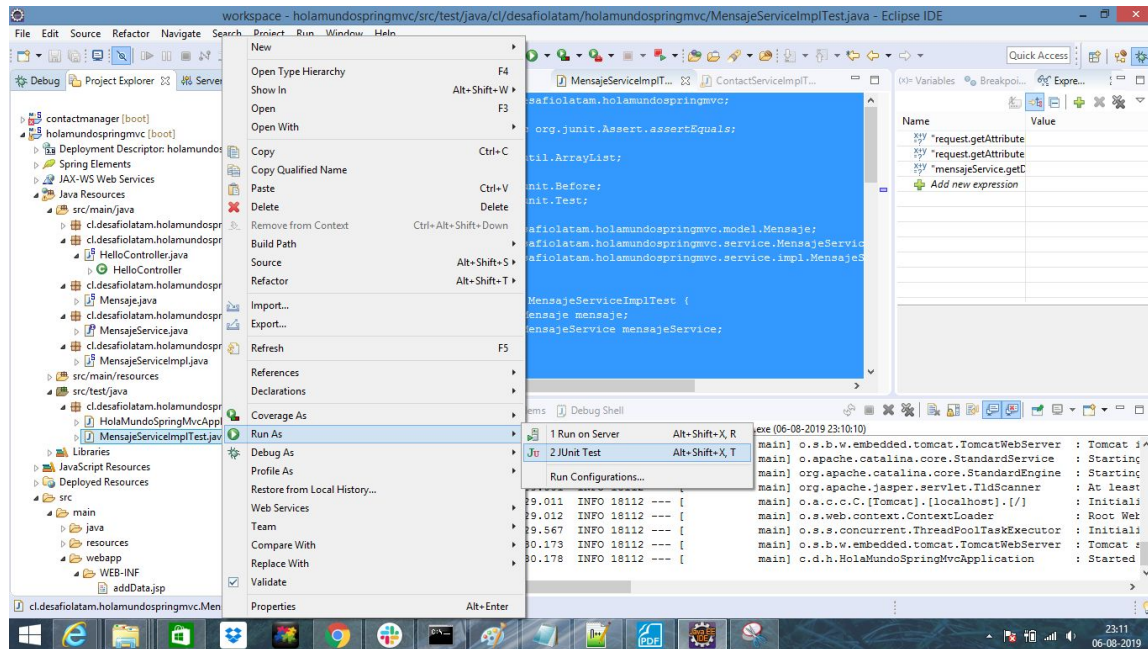
Método.
saveDataMes
sage() -
getDataMess
ageList()



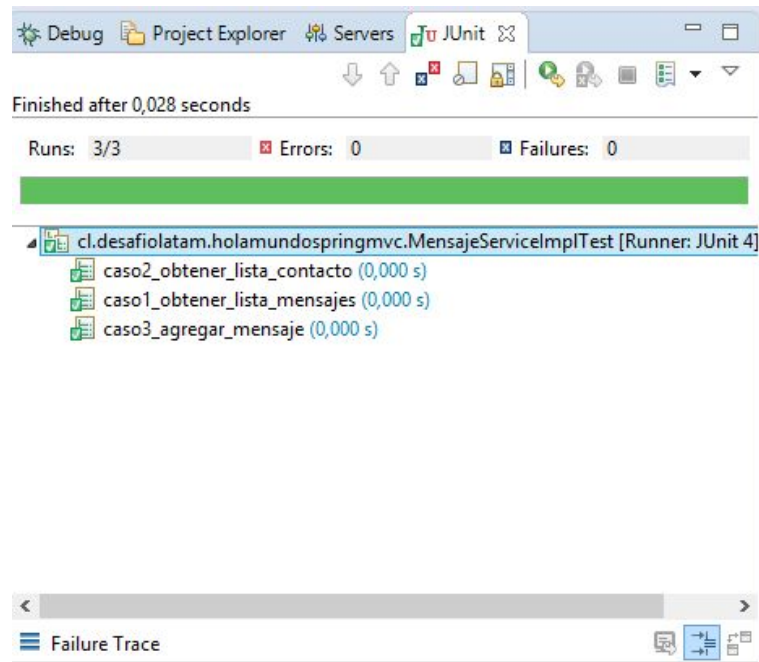
Descripción.
Agregar un
nuevo
mensaje.

**Resultado
esperado.
ejecutar.**
saveDataMes
sage y luego
getDataMess
ageList

4. Comprobar el test.
- A. Ejecutar casos de prueba.



B. Revisar los resultados.

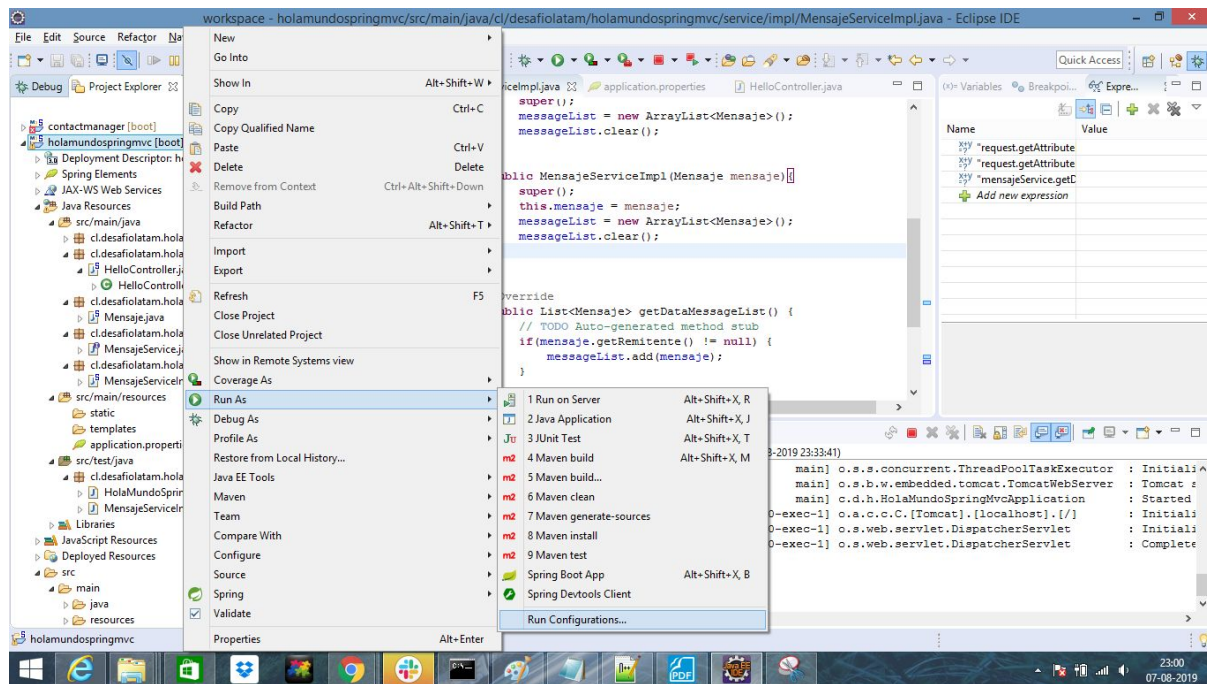


/*Empaquetando la aplicación*/

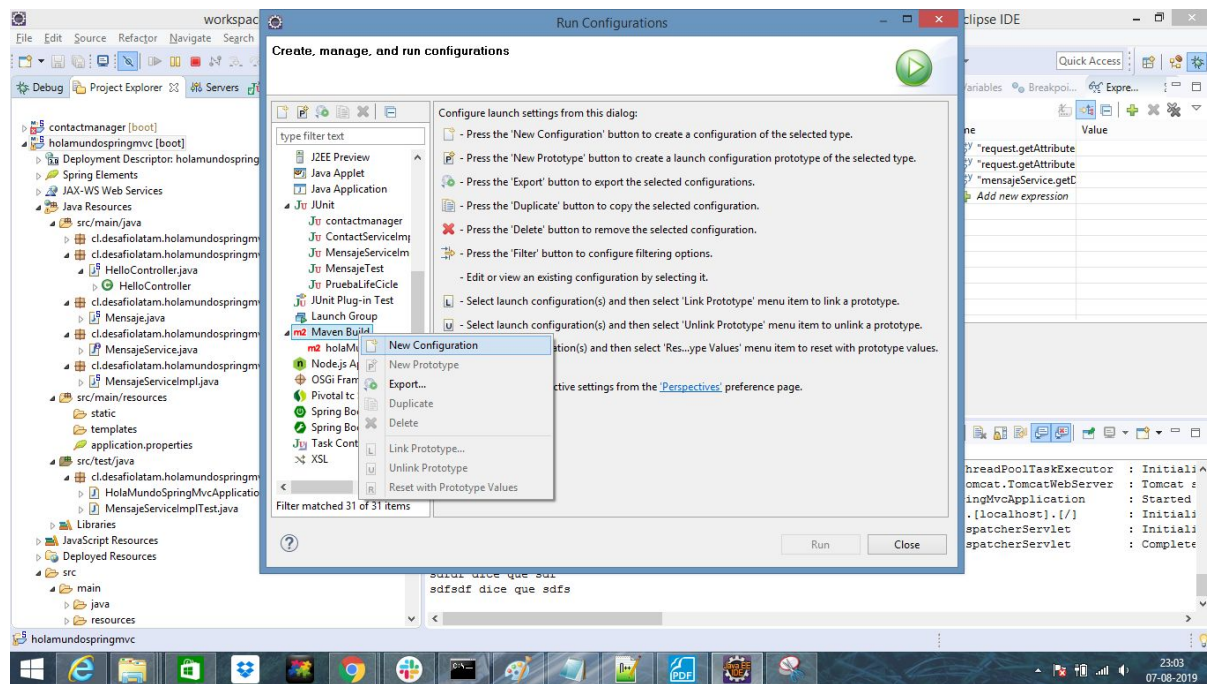
Compilar y Empaquetar la aplicación holamundospringmvc

- **Servidores de aplicaciones:** encargados de contener las aplicaciones desarrolladas para que los usuarios puedan hacer uso de ellas.
- Para compilar y empaquetar nuestra aplicación holamundospringmvc, con maven, debemos seguir los siguientes pasos:

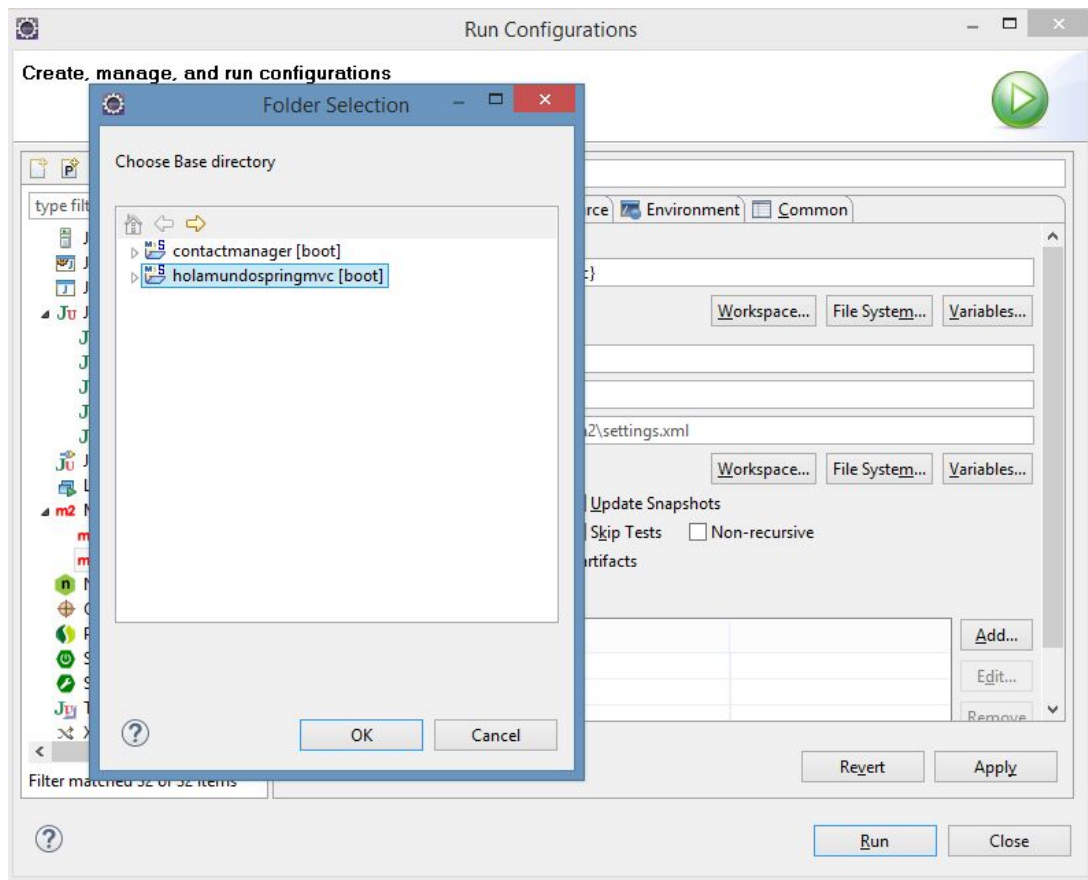
1. Configurar Eclipse IDE para que ejecute maven para empaquetar la aplicación.



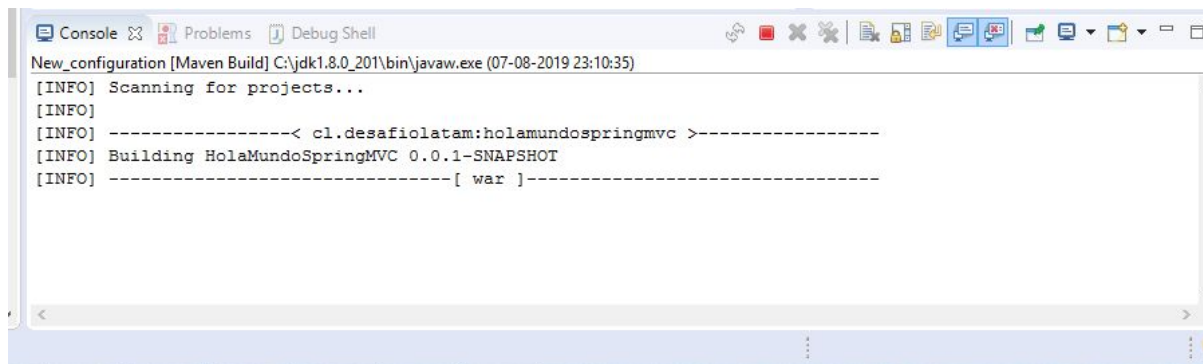
2. Buscar Maven Build > New Configuration.



3. Seleccionar el proyecto.



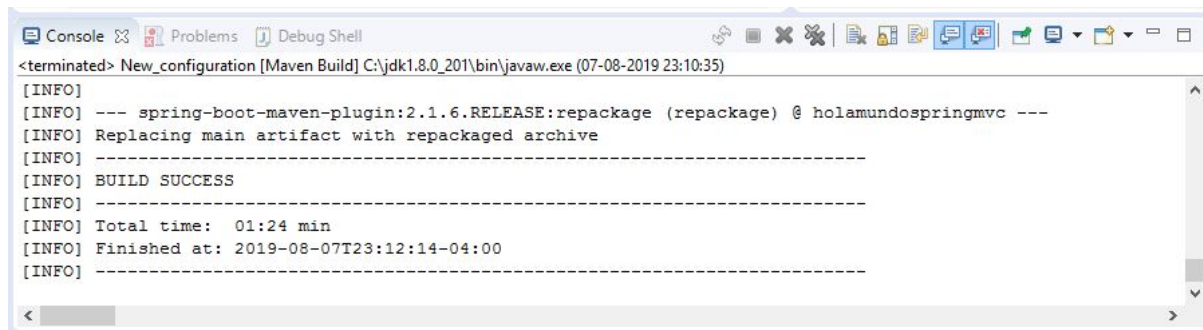
4. Resultado al aplicar los cambios.



The screenshot shows an IDE console window with the following text:

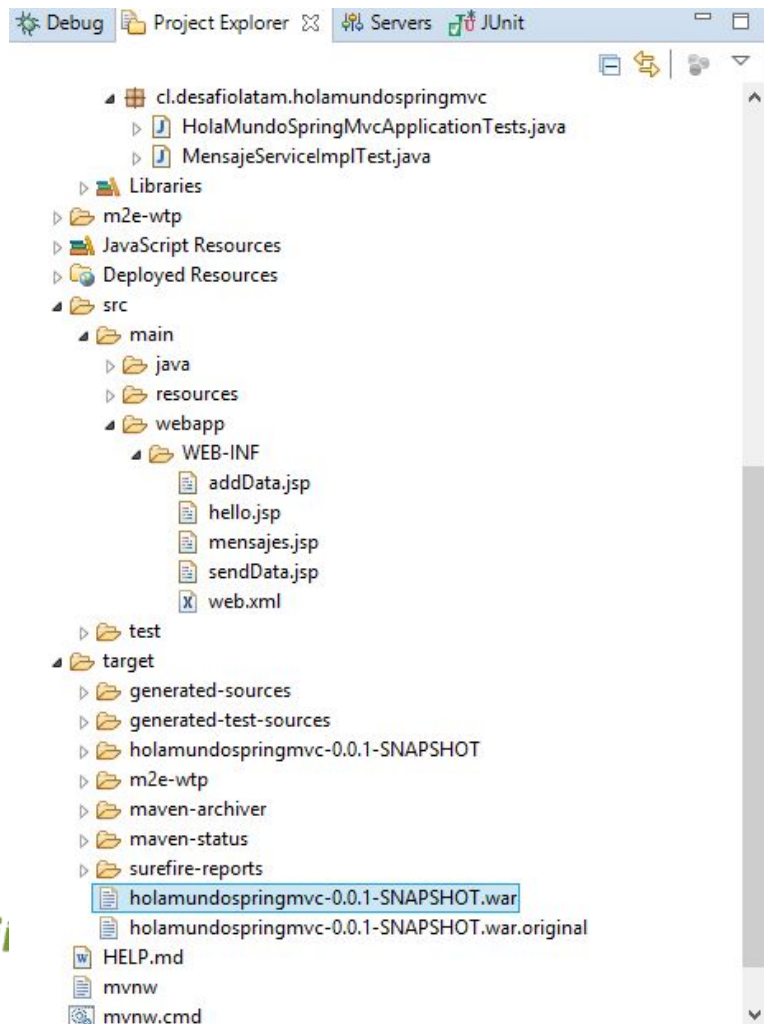
```
New_configuration [Maven Build] C:\jdk1.8.0_201\bin\javaw.exe (07-08-2019 23:10:35)
[INFO] Scanning for projects...
[INFO]
[INFO] -----< cl.desafiolatam:holamundospringmvc >-----
[INFO] Building HolaMundoSpringMVC 0.0.1-SNAPSHOT
[INFO] -----[ war ]-----
```

5. Mensaje de éxito.



The screenshot shows an IDE console window with the following text:

```
<terminated> New_configuration [Maven Build] C:\jdk1.8.0_201\bin\javaw.exe (07-08-2019 23:10:35)
[INFO]
[INFO] --- spring-boot-maven-plugin:2.1.6.RELEASE:repackage (repackage) @ holamundospringmvc ---
[INFO] Replacing main artifact with repackaged archive
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:24 min
[INFO] Finished at: 2019-08-07T23:12:14-04:00
[INFO] -----
```



{desafi
latam_

Archivo .war
generado



Quiz

{desafío}
latam_





Cierre

{desafío}
latam_



15 Minutos

**¿Existe algún concepto que
no hayas comprendido?**

**Volvamos a revisar los conceptos que más te
hayan costado antes de seguir adelante**

Reflexionemos





*Academia de
talentos digitales*

www.desafiolatam.com



/DesafioLatam



/DesafioLatam



/DesafioLatam



/DesafioLatam