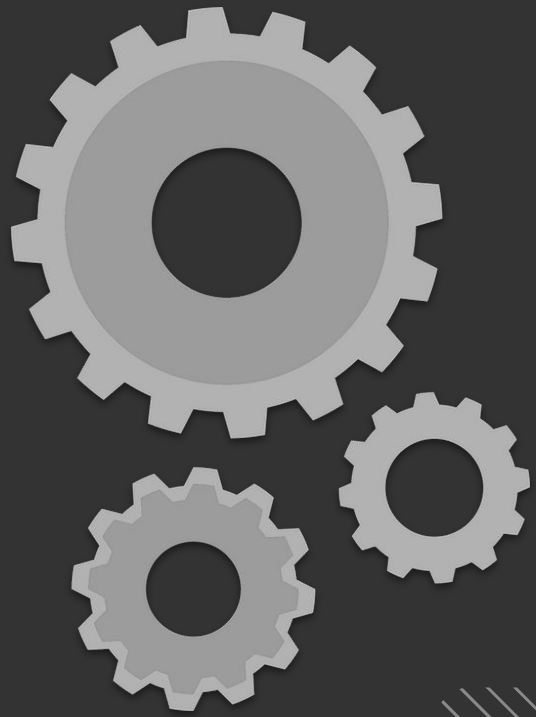
Testing:

// Integration Test con Mocks



IT BOARDING

BOOTCAMP

Índice

IT BOARDING
BOOTCAMP

TESTING

// Integration Test (Pruebas de Integración)

IT BOARDING

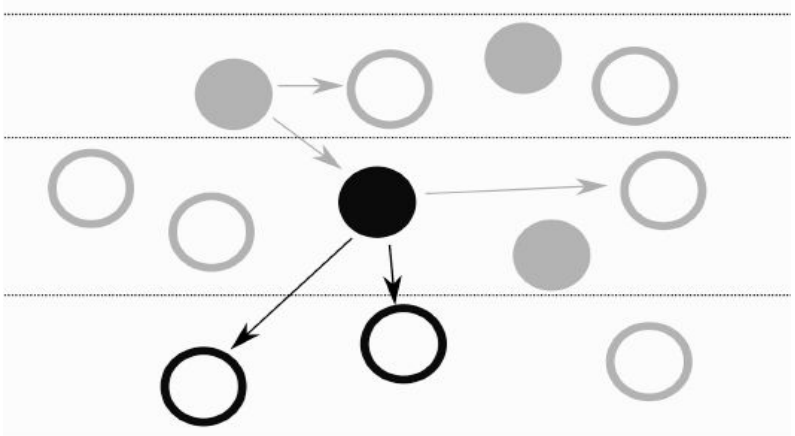
BOOTCAMP

Integration Test (Prueba de Integración)

Validar la **interacción** de módulos de software dependientes entre sí probandolos en **conjunto**.

Cubren un área mayor de código, del que a veces no tenemos control (como librerías de terceras partes), o una conexión a una base de datos, o a otro web service.

Corren más lento y suelen ser el paso siguiente a los tests unitarios.



MockMvc

MockMvc es un framework parte de Spring MVC que provee de una forma elegante y fácil de implementar mecanismos que permiten efectuar tests de integración para una API con llamados (requests) a endpoints.

Como alternativa a **MockMvc** se pueden utilizar frameworks como **RestTemplate** o **Rest-assured**.

Escribiendo Integration Tests con MockMvc 1/6

Se comienza por establecer el contexto inicial de la clase de testeo, levantando la aplicación tal cual se ejecuta en el contexto de desarrollo, e inyectando todas las dependencias que se requieran.

```
@SpringBootTest
@AutoConfigureMockMvc
public class HelloWorldIntegrationTest {

   @Autowired
   private MockMvc mockMvc;
```

@SpringBootTest: Levanta el contexto completo de la aplicación Spring.

@AutoConfigureMockMvc: Permite la inyección de un objeto MockMvc completamente configurado.

@Autowired: Inyecta la dependencia requerida.

Escribiendo Integration Tests con MockMvc 2/6

Testeando un método GET y verificando el contenido de la respuesta.

- Se hará un pedido (request) a la URL: http://localhost:8080/sayHello
- La salida esperada es:

```
{
   "id": 1,
   "message": "Hello World!"
}
```

```
@Test
public void testHelloWorldOutput () throws Exception {
   MvcResult mvcResult =
            this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
           .andDo(print()).andExpect(status().isOk())
           .andExpect(jsonPath("$.message").value("Hello World!"))
           .andReturn() ;

   Assertions.assertEquals("application/json",
mvcResult.getResponse().getContentType()) ;
}
```

- **perform()** va a efectuar el método GET request, que devuelve ResultActions. A este objeto se le podrán efectuar las assertions sobre la response, content, HTTP status o header.
- **andDo(MockMvcResultHandlers.print())** imprime request y response por consola. Útil para obtener detalles en caso de error.
- **andExpect(MockMvcResultMatchers.status().isOk())** verifica que la respuesta (response) sea HTTP status OK (200).
- **andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!!!"))** verifica que el contenido de la respuesta coincida con la salida esperada. jsonPath extrae parte de esa respuesta para proveer del valor a chequear.
- **andReturn()** devuelve el objeto MvcResult completo por si hiciera falta chequear algo por fuera de los métodos anteriores.

Escribiendo Integration Tests con MockMvc 3/6

Testeando un método GET con una PathVariable.

- Se hará un pedido (request) a la URL: http://localhost:8080/sayHello/George
- La salida esperada es:

```
{
   "id": 1,
   "message": "Hello George!"
}
```

```
@Test
public void testHelloGeorgeOutput () throws Exception {
   this.mockMvc.perform(MockMvcRequestBuilders. get("/sayHello/{name}" , "George"))
           .andDo( print()).andExpect( status().isOk())
           .andExpect( content().contentType( "application/json" ))
           .andExpect(MockMvcResultMatchers. jsonPath("$.message").value("Hello
George!"));
}
```

- **MockMvcRequestBuilders.get(“/greetWithPathVariable/{name}”, “John”)** va a efectuar el método GET request con su PathVariable en el path de la URL..

Escribiendo Integration Tests con MockMvc 4/6

Testeando un método GET con parámetros Query.

- Se hará un pedido (request) a la URL: **http://localhost:8080/sayHelloWithParam?name=George**
- La salida esperada es:

```
{
   "id": 1,
   "message": "Hello George!"
}
```

```
@Test
public void testHelloWithParamGeorgeOutput () throws Exception {
   this.mockMvc.perform(MockMvcRequestBuilders. get("/sayHelloWithParam" )
                .param( "name", "George"))
           .andDo( print()).andExpect( status().isOk())
           .andExpect( content().contentType( "application/json" ))
           .andExpect(MockMvcResultMatchers. jsonPath("$.message").value("Hello
George!"));
}
```

- **param(“name”, “John Doe”)** va a agregar el parámetro Query en el request GET.

Escribiendo Integration Tests con MockMvc 5/6

Testeando un método POST y verificando el contenido de la respuesta.

- Se hará un pedido (request) a la URL: **http://localhost:8080/sayHelloPost**
- El body de **entrada** es:

```
{
   "name": "George"
}
```

- La **salida** esperada es:

```
{
   "id": 1,
   "message": "Hello George!"
}
```

```
@Test
public void testHelloPostGeorgeOutput() throws Exception {
   NameDTO payloadDTO =new NameDTO("George");

   ObjectWriter writer =new ObjectMapper().
           configure(SerializationFeature.WRAP_ROOT_VALUE, false).
           writer().withDefaultPrettyPrinter();
   String payloadJson = writer.writeValueAsString(payloadDTO);

   this.mockMvc.perform(MockMvcRequestBuilders.post("/sayHelloPost")
               .contentType(MediaType.APPLICATION_JSON)
               .content(payloadJson))
           .andDo(print()).andExpect(status().isOk())
           .andExpect(content().contentType("application/json"))
           .andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello George!"));
}
```

- Se incopora el ObjectMapper, que se utiliza para convertir un objeto de tipo DTO en un String con su representación en JSON.
- **contentType(MediaType.APPLICATION.JSON)** especifica el formato del payload de entrada.
- **content(payloadJson)** agrega el payload en formato Json al POST request.

Escribiendo Integration Tests con MockMvc 6/6

Testeando un método POST y verificando el contenido completo de la respuesta.

- Se hará un pedido (request) a la URL: **http://localhost:8080/sayHelloPost**

- El body de **entrada** es:

```
{
   "name": "George"
}
```

- La **salida** esperada es:

```
{
   "id": 1,
   "message": "Hello George!"
}
```

```
@Test
public void testHelloPostGeorgeOutput() throws Exception {
   NameDTO payloadDTO =new NameDTO("George");
   HelloDTO responseDTO =new HelloDTO(1, "Hello George!");

   ObjectWriter writer =new ObjectMapper()
           .configure(SerializationFeature.WRAP_ROOT_VALUE, false)
           .writer();

   String payloadJson = writer.writeValueAsString(payloadDTO);
   String responseJson = writer.writeValueAsString(responseDTO);


   MvcResult response =this.mockMvc.perform(MockMvcRequestBuilders.post("/sayHelloPost")
               .contentType(MediaType.APPLICATION_JSON)
               .content(payloadJson))
           .andDo(print()).andExpect(status().isOk())
           .andExpect(content().contentType("application/json"))
           .andReturn();

   Assertions.assertEquals(responseJson, response.getResponse().getContentAsString());
}
```

Otras anotaciones útiles para Test de Integración

@WebMvcTest: Se utiliza para pruebas MockMVC. Deshabilita la auto-configuración y permite una configuración determinada de por ejemplo Spring Security.

@MockBean: Permite la simulación de Beans.

@InjectMocks: Permite la inyección de Beans.

@ExtendWith: Usualmente se le propociona SpringExtension.class, inicializa el contexto de testeo Spring.

@ContextConfiguration: Permite cargar una clase de configuración custom.

@WebAppConfiguration: Permite cargar el contexto web de la aplicación.

TESTING

// Code Coverage (Cobertura de Código)

IT BOARDING

BOOTCAMP

Code Coverage (Cobertura de Código)

La cobertura de código describe el porcentaje de código cubierto por los tests automatizados.

Es decir, chequea que partes del código son ejecutadas durante los tests y cuales NO.

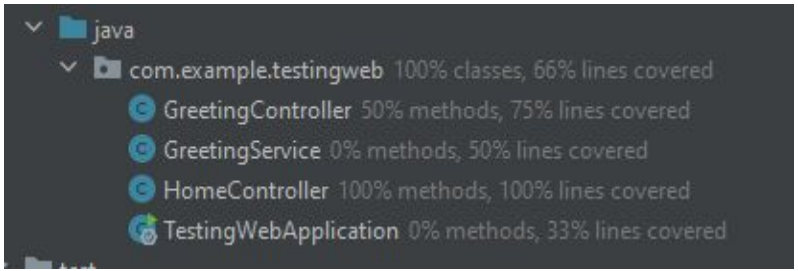
La importancia de una buena cobertura

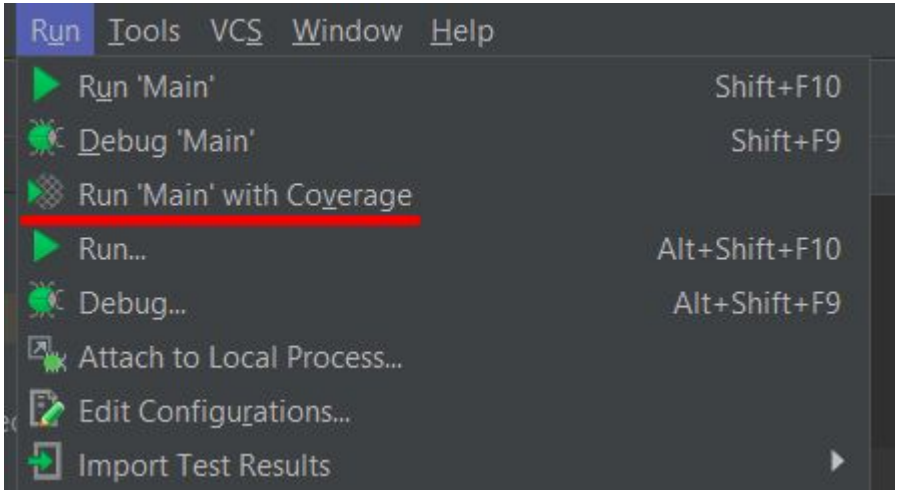
Tomar un enfoque de testeo basado en la cobertura desde el principio del proyecto elimina posibles BUGS en un estadio incial del ciclo de desarrollo.

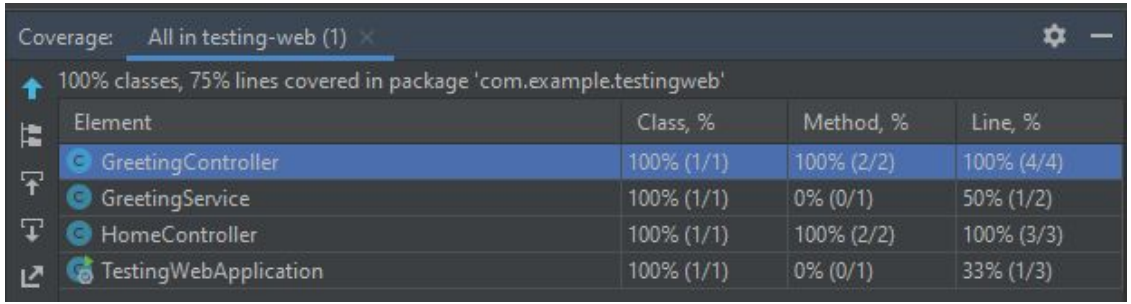
Una buena cobertura de test debería superar el 80%

Code Coverage (Cobertura de Código)

Nos permite saber el porcentaje de cobertura que tiene nuestro proyecto con los test desarrollados.







Gracias.

IT BOARDING

BOOTCAMP