IT Bootcamp

Testing,

Intregration Testing

DigitalHouse>

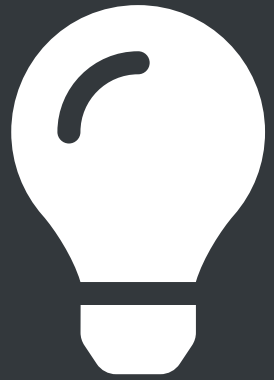
Índice

1. Integration Test
   (Pruebas de integración)
2. Tests Unitarios vs. Tests de Integración

1 Integration Testing
Pruebas de Integración

Los Test de componentes integrados, prueban la interacción entre las distintas partes del sistema.

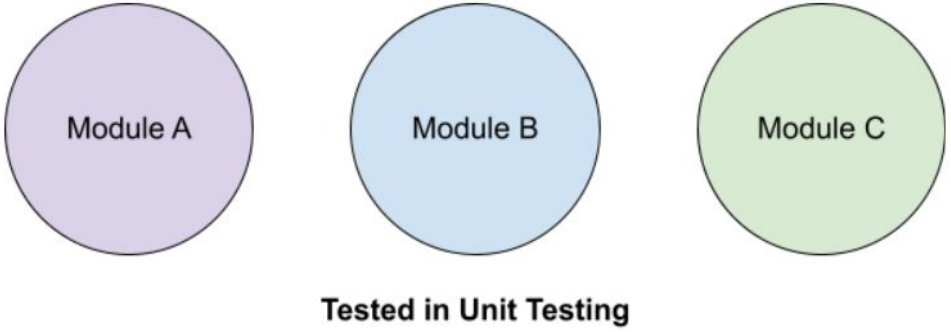
Este tipo de pruebas, verifican que los componentes de la aplicación funcionan correctamente actuando en conjunto



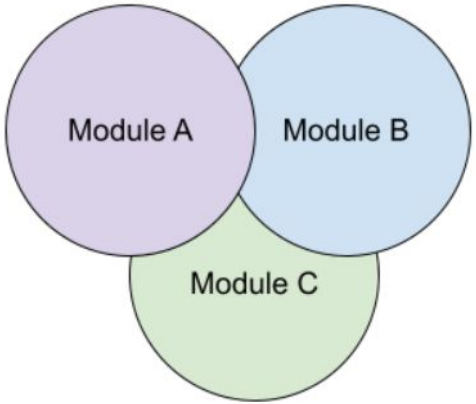
Los test de integración cubren un área mayor de código, del que a veces no tenemos control (como librerías de terceros) y comprueban, por ejemplo, que se ha enviado un email, la conexión real con la base de datos, la conexión con otro web service, etc.

- En una arquitectura de microservicios se utilizan típicamente para verificar las interacciones entre las distintas capas de código y componentes externos con los que se está integrando (otros microservicios, bases de datos, caches, etc.)
- El objetivo es verificar la correcta comunicación entre los diferentes módulos, en lugar de realizar una prueba de aceptación del componente externo.
- Debe apuntar a cubrir los caminos básicos de éxito y error a través del módulo de integración.

Una vez que todos los componentes o módulos funcionan correctamente de manera independiente, debemos verificar el correcto flujo de datos entre los módulos dependientes.

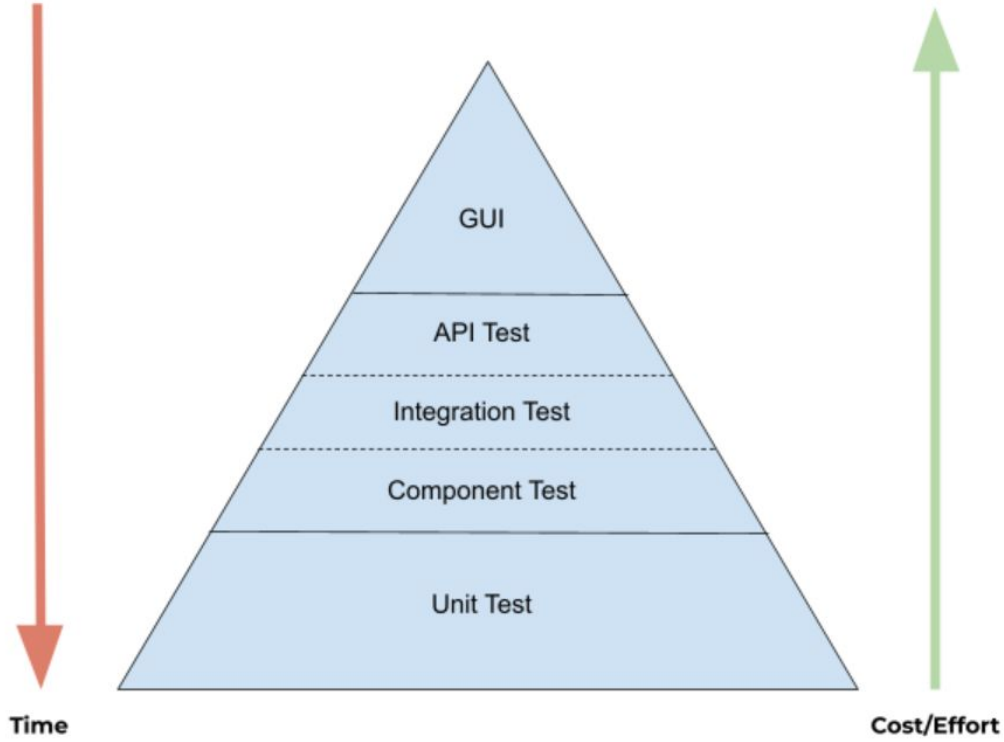


Tested in Unit Testing



Integration Testing

Pirámide ideal del Testing



Guía para Integration Testing

- Optamos por las pruebas de integración luego de que se completen las pruebas funcionales en cada módulo de la aplicación.
- Realizar pruebas de integración seleccionando módulo por módulo para que se siga una secuencia adecuada, y así, no perder ningún escenario de integración.
- Se debe determinar la estrategia de casos de prueba a través de la cual se pueden preparar casos de prueba ejecutables de acuerdo a la test data.
- Examinar la estructura y arquitectura de la aplicación para identificar los módulos cruciales para probarlos primero, y también así identificar posibles escenarios.
- Diseñar los casos de prueba para verificar cada interfaz en detalle.
- Seleccionar los datos de entrada para la ejecución del caso de prueba.
- Si se encuentra algún bug, comunicar los errores y corregir los defectos, para luego volver a realizar la prueba.
- Realizar integration testing positivo y negativo.

¿Por qué hacer tests de integración?

Aunque todos los módulos de la aplicación se hayan probado en las pruebas unitarias, aún pueden existir errores por diversas razones.

- Cada módulo suele estar diseñado por un desarrollador de software cuya lógica de programación puede diferir de los desarrolladores de otros módulos.
  Las pruebas de integración se vuelven esenciales para determinar el funcionamiento de los módulos en conjunto.
- Comprobar si la interacción de los módulos con la base de datos es correcta o no.
- Verificar incompatibilidad entre los módulos que pueda generar errores.
- Testear la compatibilidad del hardware con el software.
- Evitar errores si el manejo de excepciones es inadecuado entre los módulos.

Beneficios de las Pruebas de Integración

- Se asegura de que todos los módulos de aplicación estén bien integrados y funcionen juntos según lo esperado.
- Detecta problemas y conflictos interconectados para resolverlos antes de generar un problema mayor.
- Valida la funcionalidad, fiabilidad y estabilidad entre diferentes módulos.
- Detecta excepciones ignoradas para mejorar la calidad del código.
- Admite la canalización de CI/CD.

¡A tener en cuenta!
Como se puede apreciar en la pirámide ideal de testing presentada anteriormente, las pruebas de integración son más costosas de realizar que las pruebas unitarias.

Mock MVC

Spring provee una gran herramienta para testear aplicaciones Spring Boot: **MockMVC**. Este framework provee una forma fácil de implementar tests de integración para aplicaciones web.

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

Para comenzar a testear endpoints declarados en un controller utilizando MockMVC, necesitamos agregar las dependencias de **spring-boot-starter-web** y **spring-boot-starter-test** al archivo pom.xml

Escribir tests de integración con MockMVC

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.

Testear un método GET y verificar 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(MockMvcResultMatchers.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(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.

Testear un método GET con un 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.

Testear un método GET con QueryParam

- 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.

Testear un método POST y verificar 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 incorpora 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.

Testear un método POST y verificar 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());
}
```

Anotaciones para Test de Integración

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

@MockBean: Permite la simulación de Beans.

@InjectMocks: Permite la inyección de Beans.

@ExtendWith: Usualmente se proporciona la extensión 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.

2

Tests Unitarios vs. de Integración

Test Unitarios vs. Test de Integración

Entonces…¿Es correcto decir que las pruebas unitarias y las pruebas de integración son lo mismo?

La respuesta es **NO**, a continuación se presentarán las principales diferencias entre estos tipos de pruebas…

Test Unitarios vs. Test de Integración

Pruebas Unitarias	Pruebas de Integración
Prueba una clase/unidad en solitario, es decir, prueba una unidad de forma aislada.	Prueba los componentes del sistema trabajando juntos, es decir, prueba la colaboración de múltiples unidades.
Fáciles de escribir y verificar.	El setup de las pruebas de integración puede ser complicado.
Toda dependencia externa es mockeada o eliminada de ser necesario en el escenario de prueba.	Requiere interacción con dependencias externas (por ejemplo, base de datos, hardware, etc.)
Se llevan a cabo desde el inicio del proyecto y luego se puede realizar en cualquier momento.	Debe realizarse después de la prueba unitaria y antes de la prueba del sistema.

Test Unitarios vs. Test de Integración

Pruebas Unitarias	Pruebas de Integración
Los resultados dependen del código Java, solo verifican si cada pequeño fragmento de código está haciendo lo que se pretende que haga.	Los resultados también dependen de sistemas externos. Una prueba de integración fallida, puede indicar también que el código continúa siendo correcto, pero el entorno ha cambiado.
Permite identificar problemas en la funcionalidad de módulos individuales. No expone errores de integración ni problemas en todo el sistema.	Permite identificar los errores que surgen cuando diferentes módulos interactúan entre sí. Tienen un alcance más amplio.
Se ejecutan más rápido.	Su ejecución puede demorar mucho más tiempo.

DigitalHouse>