Universidad Nacional de Rio Negro

Automatización, Testing y Calidad de código en arquitectura limpias aplicado al proyecto “Plataforma intrainstitucional de servicios digitales para simplificación organizacional”

Alumno: *Cruz Jose Luis* - *joseluiscruz1914@gmail.com*
Director: *Ing. Enrique Molinari* - *emolinari@unrn.edu.ar*
Co-Director: *Lic. Horacio Muñoz Abbate* - *hmunuz@unrn.edu.ar*

Automatización, Testing y Calidad de código

Agradecimientos

En primer lugar, me gustaria agradecer a la Universidad Nacional de Río Negro por las oportunidades que me brindó para desarrollar mis estudios y mi carrera profesional.

Agradezco especialmente a mis directores, quienes han sido una fuente de inspiración y un modelo a seguir profesionalmente para mí.

Quiero expresar mi profundo agradecimiento a mi familia, amigos y compañeros por su apoyo incondicional durante todos estos años. Su apoyo me ha permitido recorrer este hermoso camino con alegría y motivación.

Automatización, Testing y Calidad de código

Automatización, Testing y Calidad de código

Resumen

En el ámbito del desarrollo de software, surge con frecuencia la pregunta: ¿Cómo garantizar la calidad del software? Este trabajo aborda este interrogante, proponiendo tres pilares esenciales para alcanzar dicho objetivo: la automatización, el testing y la calidad de código. Se dirige principalmente a desarrolladores, líderes técnicos y arquitectos, adoptando una perspectiva técnica sin adentrarse en estándares gerenciales como las normas ISO.

***Palabras clave**:* Automatización, Testing, Calidad de Código

1 Introducción

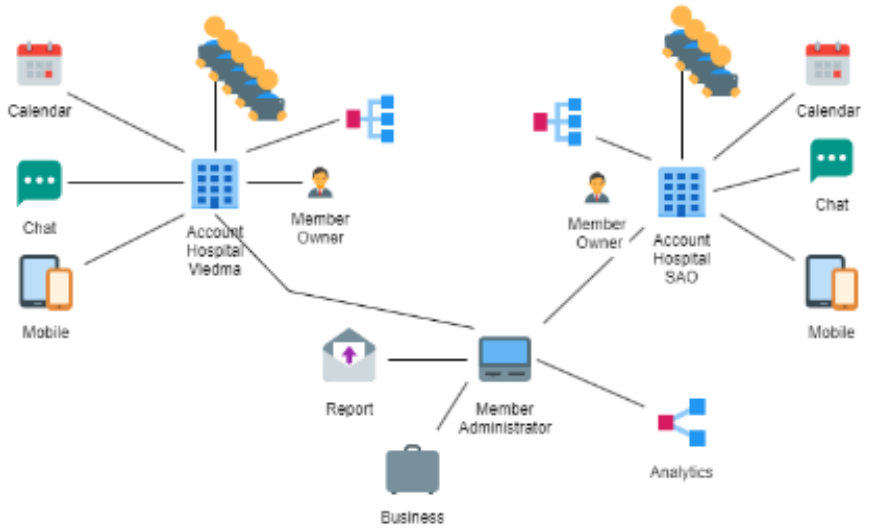
En el desarrollo de software actual, la calidad del código y la eficiencia en los procesos de prueba desempeñan un papel crucial para determinar el éxito de un proyecto. El presente trabajo final de carrera se centra en el análisis, mejora y automatización de pruebas para el sistema denominado *Plataforma intrainstitucional de servicios digitales para simplificación organizacional* enmarcado en la convocatoria *Ideas Proyecto de Desarrollo y Transferencia de Tecnología* (IPDTT).

El enfoque de este trabajo se basa en tres pilares fundamentales para el desarrollo moderno de software: testing, calidad del código y automatización. Estos elementos son esenciales para asegurar la robustez, mantenibilidad y escalabilidad de cualquier sistema a medida que evoluciona.

El sistema IPDTT consistió en desarrollar una plataforma digital con el objetivo de establecer una red de perfiles del personal de salud del Hospital de Viedma, facilitando la comunicación e interacción entre los miembros del personal de salud y la comunidad.

El principal objetivo es crear y gestionar los perfiles miembros de la institución y la comunidad, creando un canal de comunicación directo entre los mismos. Además, se busca:

- Representar la estructura orgánica funcional -EOF- de la institución.
- Identificar al personal de la institución que tiene un rol en la EOF.
- Contribuir a generar una mejora en la comunicación organizacional.
- Crear canales de servicios sobre: eventos, avisos, notificaciones, actividades, comunicación en general.
- Analizar los niveles de interacción entre las áreas de la institución.
- Generar una comunidad digital que permita relacionar miembros de la institución y la comunidad en general.



- Figura 1: Comunidad digital

1.1 Funcionamiento y estructura de módulos funcionales

El sistema IPDTT se encuentra dividido en dos áreas funcionales, el backoffice y el frontoffice. El backoffice se encarga de las tareas administrativas y operativas, mientras que el frontoffice está en contacto directo con el usuario y se encarga de gestionar sus solicitudes y necesidades.

En el contexto del sistema IPDTT, la distribución de funciones se llevó a cabo de la siguiente manera:

- En el backoffice, los usuarios que acceden a este componente son aquellos que ostentan el rol de "Miembro Owner" y utilizan navegadores web para ingresar.
- Por otro lado, en el frontoffice, los usuarios con el rol de "Miembro" que forman parte del personal de salud acceden al sistema a través de dispositivos móviles.

Esta separación de funciones permitió que ambos equipos se centrarán en sus tareas principales y que la colaboración entre ambos fuera más eficiente. Además el sistema IPDTT cuenta con los siguientes módulos funcionales:

- Gestionar Cuentas (Instituciones – Ciudadanos)
- Gestionar Perfiles (Red de contactos) para personal de las Instituciones
- Gestionar mensajería entre perfiles

- Gestionar Estructura Orgánica Funcional (EOF)
- Gestionar Notificaciones
- Gestionar Actividades/eventos

1.2 Arquitectura

Para describir la arquitectura del sistema IPDTT, utilizaremos el modelo C4, desarrollado por Simon Brown [1]. Este modelo es una técnica de documentación que se utiliza para describir la arquitectura de sistemas de software. Es muy interesante porque nos permite explicar una arquitectura desde un punto de vista muy abstracto hasta niveles de implementación muy detallados.

Como podemos ver en la Figura 2, el sistema cuenta con dos tipos de usuarios principales *Administration Owne*r y *Member Owner*, ambos interactúan con el sistema.

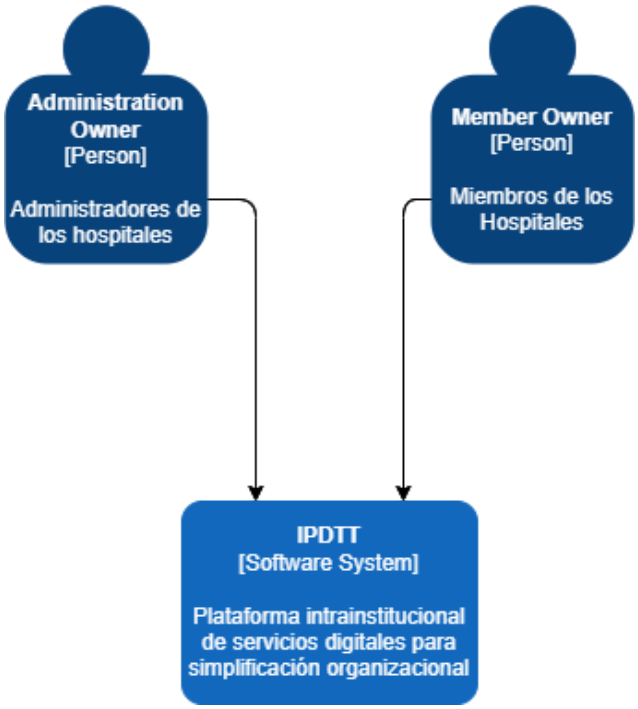


Figura 2 - Modelo C4 Contexto

Según el modelo C4, el sistema IPDTT cuenta con tres aplicaciones (containers): una Single-Page Application (SPA) que utiliza el framework Angular, una API (del inglés, application programming interface) desarrollada en el framework Spring Boot de Java y una base de datos PostgresSQL. Para más información, ver figura 3.

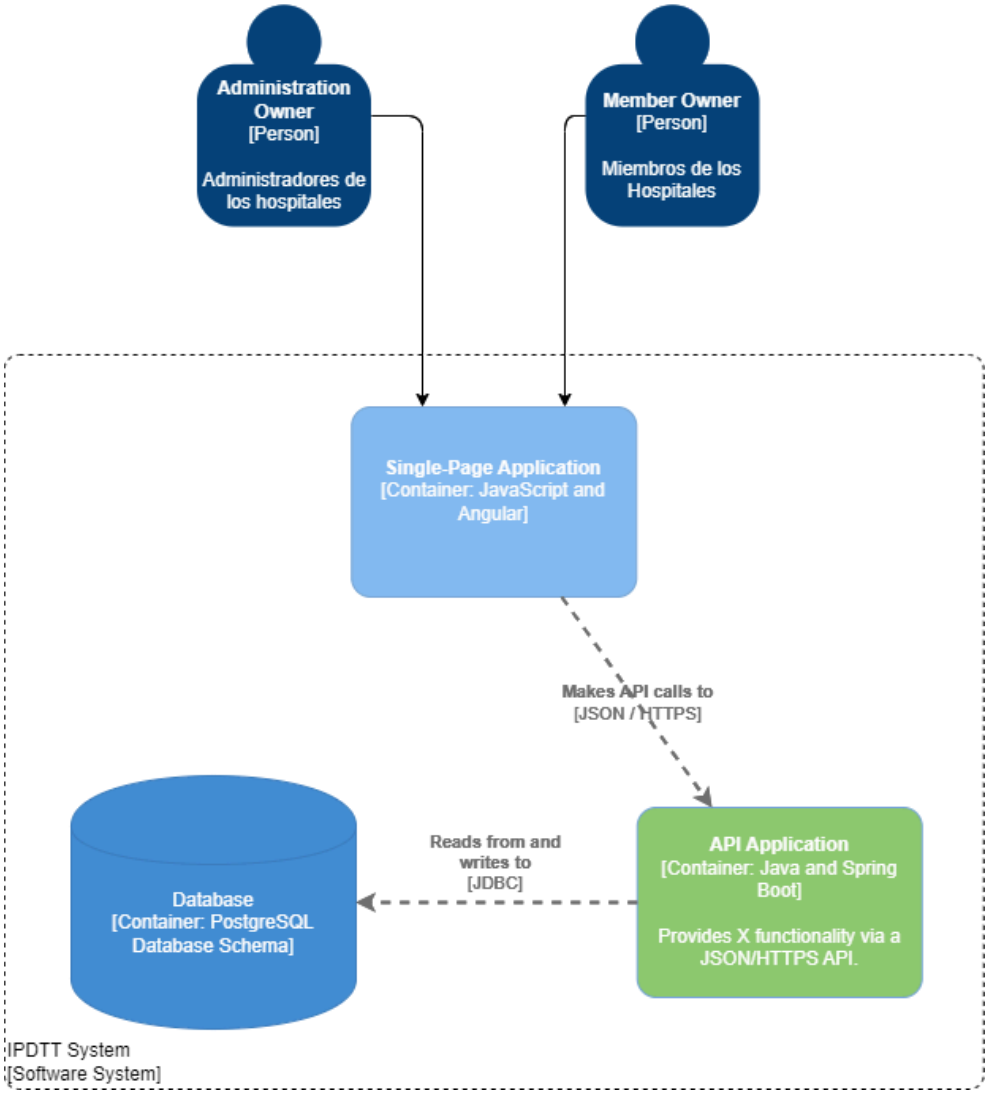


Figura 3 - Modelo C4 - Diagrama de contenedores

Este trabajo se centra en el testing, la verificación de calidad de código y la automatización del componente *bc-account* de la aplicación *API Application* desarrollada en Spring Boot. La aplicación cuenta con seis componentes (ver Figura 4) implementados mediante módulos maven [2], de los cuales el componente *bc-account* es el encargado de gestionar las cuentas de los usuarios.

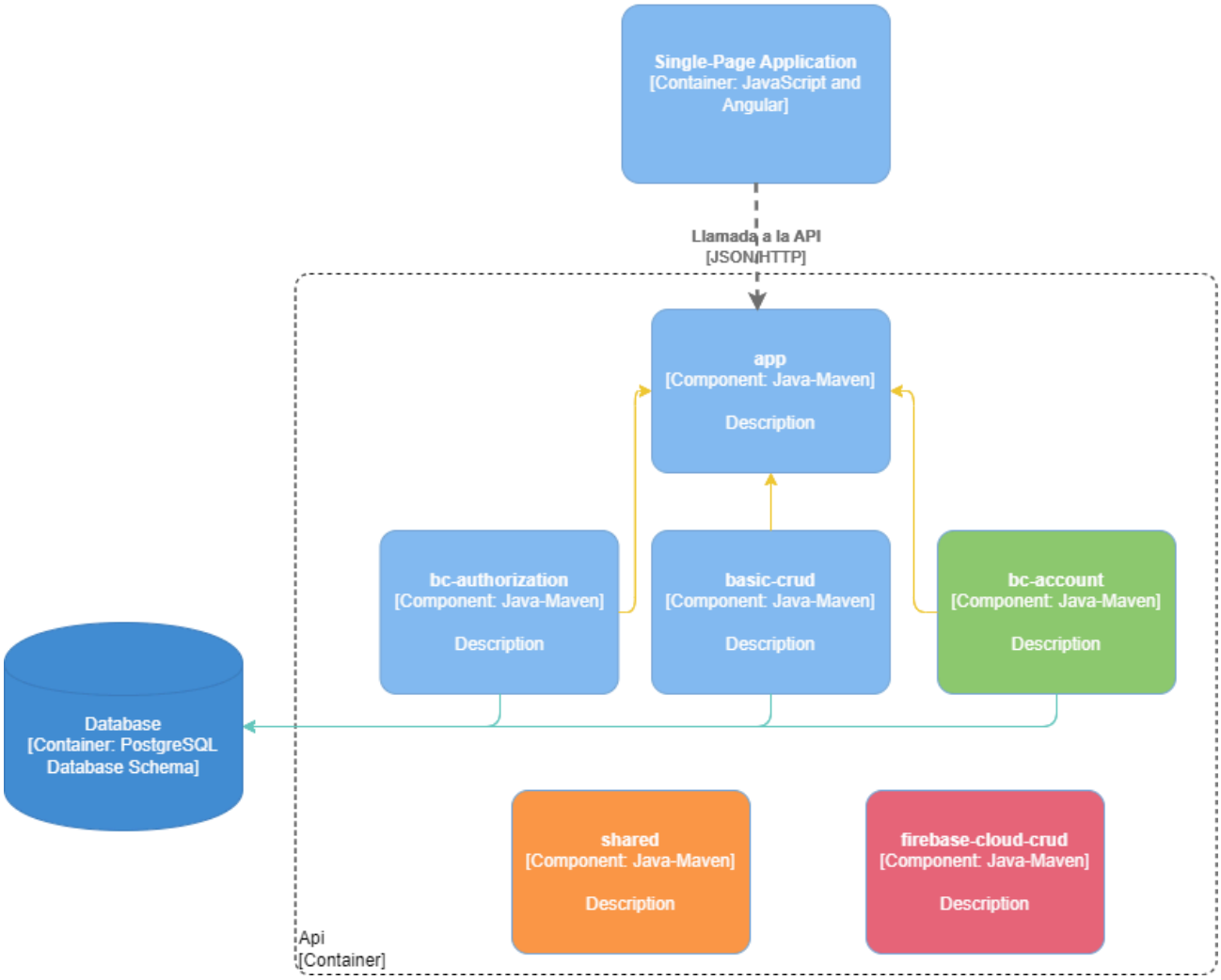


Figura 4 - Modelo C4 - Diagrama de componente del container API

El componente “bc-account” está implementado con arquitectura hexagonal y consta de tres subcomponentes (ver figura 5):

- bc-account-domain
- bc-account-web
- bc-account-persistence

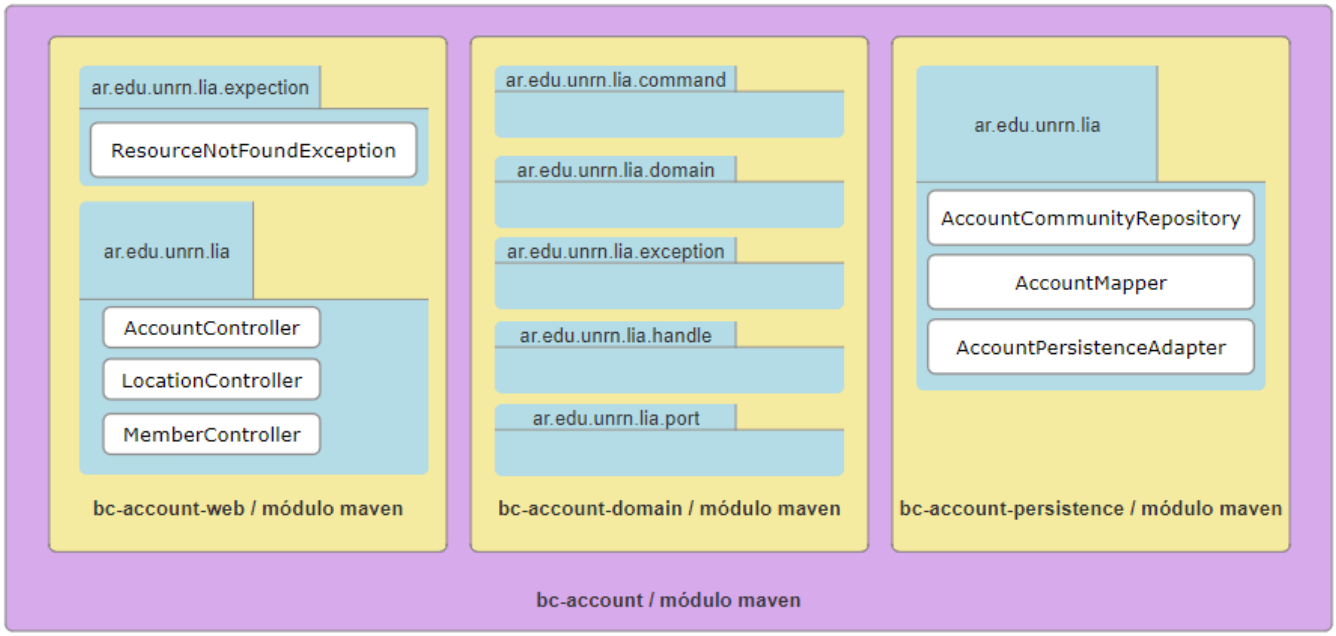


Figura 5 - sub módulos y paquetes de bc-account

1.2.b Arquitectura Hexagonal

La arquitectura hexagonal, también conocida como arquitectura de puertos y adaptadores, es un patrón de diseño de software que fue descrito por Alistair Cockburn [3]. La arquitectura hexagonal aborda un problema común en el diseño de software: la necesidad de separar la lógica de negocio de la interfaz de usuario y de las dependencias externas.

La arquitectura hexagonal resuelve este problema creando dos elementos: puertos y adaptadores. Los puertos definen la interfaz de la aplicación (ver figura 6), mientras que los adaptadores implementan la interfaz utilizando dependencias externas.

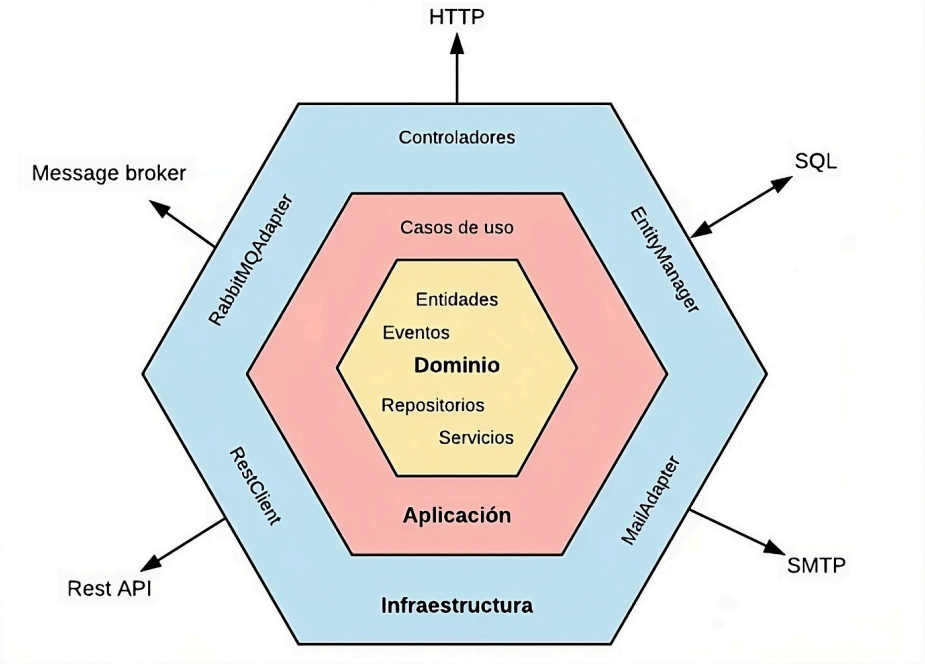


Figura 6 - Arquitectura hexagonal

La Arquitectura Hexagonal también persigue el desacoplamiento del framework, en este caso, Spring Boot. Esto permite que la aplicación pueda ser utilizada con otros frameworks o sin ninguno.

2 Testing

La aplicación de pruebas en un sistema, como en el caso de IPDTT, se vuelve esencial para asegurar la calidad y confiabilidad del software. El proceso de testing, que implica la evaluación y verificación minuciosa de cada componente, desempeña un papel crucial en la identificación y corrección de posibles errores. Además, valida la conformidad con los requisitos establecidos, asegurando así el correcto desempeño del sistema. Al facilitar la detección temprana de fallos, mejorar la mantenibilidad del código y proporcionar confianza en la estabilidad y rendimiento del sistema, el testing se posiciona como un paso indispensable previo a la implementación del software.

Una definición muy interesante de las pruebas de software o testing es la siguiente:

“El proceso que consiste en todas las actividades del ciclo de vida, tanto estáticas como dinámicas, relacionadas con la planificación, preparación y evaluación de un componente o sistema y productos de trabajo relacionados para determinar que satisfacen requisitos específicos, para demostrar que son aptos para el propósito y para detectar defectos.” [11]

Para evaluar el impacto de los distintos tipos de pruebas en un proyecto, Mike Cohn introdujo la *Pirámide de Pruebas de Software*[4]. Este modelo sugiere que la atención principal en las pruebas de software debe centrarse en las capas inferiores de la pirámide, donde los errores pueden tener un impacto más significativo.

Figura 7 - Piramide de Mike Cohn - Extraído de https://10pines.gitbook.io/

Las capas inferiores de la pirámide se centran en las unidades individuales de código, mientras que las capas superiores se centran en la interfaz de usuario y la integración del sistema.

La pirámide de pruebas de software se presenta como una herramienta valiosa tanto para desarrolladores de software como para profesionales en el campo de pruebas. Su utilidad radica en dirigir las pruebas hacia las áreas críticas del software, contribuyendo así a elevar la calidad del producto y disminuir el riesgo de posibles errores.

2.1 Principios y Prácticas

Un principio muy utilizado cuando implementamos testing sobre nuestros proyectos es el principio FIRST que determina principios y prácticas básicas para las pruebas.

- Aislamiento: Las pruebas unitarias deben aislar la unidad de código que se está probando del resto del sistema. Esto se hace para que las pruebas se puedan ejecutar de forma rápida y sencilla, y para que no se vean afectadas por los cambios en el código de otros componentes.
- Completitud: Las pruebas unitarias deben cubrir todos los casos de uso posibles de la unidad de código. Esto se hace para garantizar que la unidad de código funcione correctamente en todas las condiciones.
- Automatización: Las pruebas unitarias deben ser automatizadas para que se puedan ejecutar de forma rápida y sencilla. Esto permite a los desarrolladores ejecutar pruebas con frecuencia y detectar errores rápidamente.
- Independencia: Las pruebas unitarias deben ser independientes entre sí para que se puedan ejecutar en cualquier orden. Esto facilita la ejecución de pruebas de forma paralela y reduce el tiempo de ejecución total de las pruebas.
- Resistencia al refactoring: Las pruebas unitarias deben ser resistentes al refactoring para que no se rompan cuando se cambia el código. Esto se hace para garantizar que las pruebas sigan siendo válidas a medida que el código evoluciona.

La implementación del principio *FIRST* al realizar pruebas en nuestros proyectos no solo promueve la calidad del software, sino que también fortalece la eficiencia y la confiabilidad del proceso de desarrollo. Al centrarnos en la completitud, la automatización, la independencia y el aislamiento, maximizamos la efectividad de las pruebas, contribuyendo a la entrega de productos más robustos y satisfactorios.

2.2 Testing unitario

El testing unitario evalúa unidades individuales para confirmar su correcto funcionamiento según el diseño previsto. Su objetivo principal es verificar componentes antes de su integración en el programa completo, permitiendo una rápida identificación de posibles errores. En el desarrollo ágil de software, el testing unitario desempeña un papel crucial en el control de calidad, realizándose frecuentemente de manera automática para detectar y corregir errores en las primeras fases del desarrollo.

2.2.1 JUnit

Para llevar a cabo la implementación de los test unitarios utilizaremos JUnit [5], que es una herramienta que ayuda a los desarrolladores de Java a escribir pruebas unitarias para sus aplicaciones. JUnit proporciona un conjunto de clases y herramientas que facilitan la escritura de pruebas unitarias.

Para integrar Junit 5 en nuestro proyecto se debe agregar la siguiente dependencia en el **pom.xml**

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
```

Figura 8 - Dependencia JUnit 5 para Maven

2.2.2 Test de Controladores

Los tests de controladores son cruciales para verificar que los controladores RESTful operen adecuadamente y generen los resultados esperados. En términos simples, *RESTful* se refiere a un enfoque de diseño que utiliza servicios web basados en HTTP para facilitar la comunicación entre sistemas distribuidos, favoreciendo la eficiencia y la interoperabilidad.

2.2.2.1 MockMvc

Para implementar los test de controladores nos apoyaremos en MockMvc [6], que nos proporciona soporte para probar aplicaciones Spring MVC. Realiza el manejo completo de solicitudes, pero a través de objetos simulados de solicitud y respuesta en lugar de un servidor en ejecución.

Para escribir tests de controladores en Spring Boot, puedes usar la anotación *@WebMvcTest*. Esta anotación creará un contexto de Spring Boot con los beans necesarios para probar el controlador, incluyendo un MockMvc que puedes usar para realizar peticiones HTTP simuladas al controlador.

Para escribir una prueba de controlador, primero debes crear una clase de prueba que extienda de JUnit. Luego, puedes usar la anotación *@Test* para marcar los métodos de prueba. Dentro de cada método de prueba, puedes usar el objeto MockMvc para realizar peticiones HTTP al controlador y verificar los resultados.

A continuación se muestra un ejemplo de un test de controlador para un controlador RESTful simple:

```
// static import of MockMvcRequestBuilders.* and MockMvcResultMatchers.*

mockMvc.perform(get("/accounts/1")).andExpectAll(
    status().isOk(),
    content().contentType("application/json;charset=UTF-8"));
```

Figura 9 - Ejemplo de test de controlador - extraído de Spring boot MockMvc

Para poder utilizar MockMvc en nuestro aplicación basta con agregar la siguiente dependencia:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

Figura 10 - Dependencia Spring boot que incluye MockMvc - extraído de Spring boot MockMvc

2.3 Testing de integración

Según Martin Fowler, en su artículo *Integration Test*, señala que las aplicaciones suelen interactuar con diversos componentes, como bases de datos, sistemas de archivos o realizar llamadas a otras aplicaciones. Al escribir pruebas unitarias, estas interacciones a menudo se omiten para lograr un aislamiento más efectivo y pruebas más rápidas.

Las pruebas de integración, por otro lado, tienen como objetivo verificar si las unidades de software desarrolladas de forma independiente funcionan correctamente cuando se conectan entre sí.

En el contexto de la aplicación IPDTT, que se integra con una base de datos PostgreSQL, es crucial asegurarse de que la conexión con la base de datos sea correcta. Para llevar a cabo esta verificación, realizaremos una prueba de integración que involucra la ejecución tanto de nuestra aplicación como de la base de datos. Para

facilitar este proceso, emplearemos la librería TestContainers, que nos posibilitará crear una instancia de PostgreSQL de manera eficiente.

2.3.1 TestContainers

Testcontainers es una librería Java que permite ejecutar contenedores Docker para ejecutar tests de integración [13]. Esta librería proporciona una serie de ventajas para las pruebas de persistencia, entre las que se incluyen:

- Aislamiento: Nos permite aislar los tests de persistencia del entorno de desarrollo. Esto significa que los tests no dependen de la base de datos real, lo que puede evitar conflictos y problemas de concurrencia.
- Simulabilidad: Nos permite simular diferentes escenarios de persistencia. Esto permite a los desarrolladores probar su código en diferentes condiciones, lo que puede ayudar a identificar errores y problemas de rendimiento.
- Eficiencia: Es una librería eficiente que permite ejecutar tests de persistencia de forma rápida y sencilla.

Para poder utilizar TestContainers es necesario Docker y JUnit e incluir la dependencia mediante Maven de la siguiente forma:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.19.1</version>
    <scope>test</scope>
</dependency>
```

Figura 11 - Dependencia de TestContainers

2.4 Test de regresión

Las pruebas de regresión constituyen un subconjunto específico de las pruebas planificadas que se eligen para llevar a cabo de manera periódica, especialmente al aproximarse cada nueva liberación del producto. Estas pruebas están diseñadas con el propósito fundamental de confirmar que el software en desarrollo o modificación no ha experimentado ningún retroceso o regresión en su funcionalidad [16].

En esencia, cuando se realiza una actualización, mejora o cambio en el código fuente de una aplicación, existe la posibilidad de que estas modificaciones impacten en otras áreas del sistema, incluso generando errores inesperados en funcionalidades previamente probadas y consideradas estables. Las pruebas de regresión se

encargan de detectar cualquier deterioro en el rendimiento o mal funcionamiento que pueda surgir como consecuencia de las modificaciones implementadas.

Al ejecutar periódicamente estas pruebas, se asegura que el software mantenga su integridad y calidad general, garantizando que las nuevas implementaciones no hayan introducido problemas inesperados en áreas previamente verificadas. En este sentido, las pruebas de regresión actúan como un salvaguarda frente a posibles retrocesos, contribuyendo así a la estabilidad continua del producto a lo largo de su evolución.

2.5 Implementación testing

En esta sección, se detallará la aplicación de pruebas en el sistema IPDTT, abordando la estructura de paquetes seleccionada y describiendo la implementación tanto de las pruebas unitarias como de las pruebas de integración.

2.5.1 Estructura de paquetes

Con respecto a la estructura de paquetes destinada a las pruebas, se sigue consistentemente la estrategia de mantener la coherencia con la estructura general del proyecto, como se ilustra en las siguientes capturas de los módulos *bc-account-domain*, *bc-account-persistence* y *bc-account-web*.

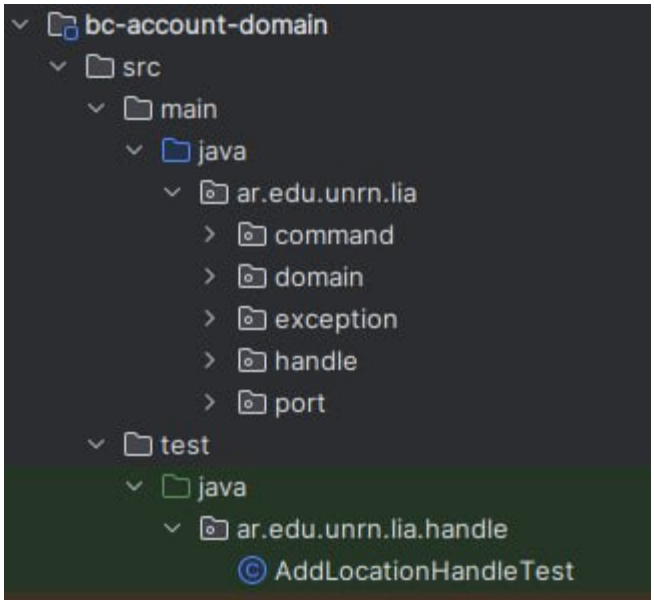


Figura 13 - Estructura de paquetes módulo bc-account-domain

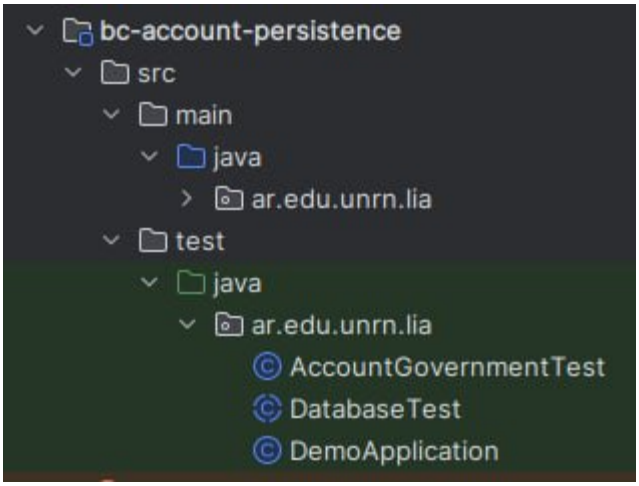


Figura 14 - Estructura de paquetes módulo bc-account-persistence

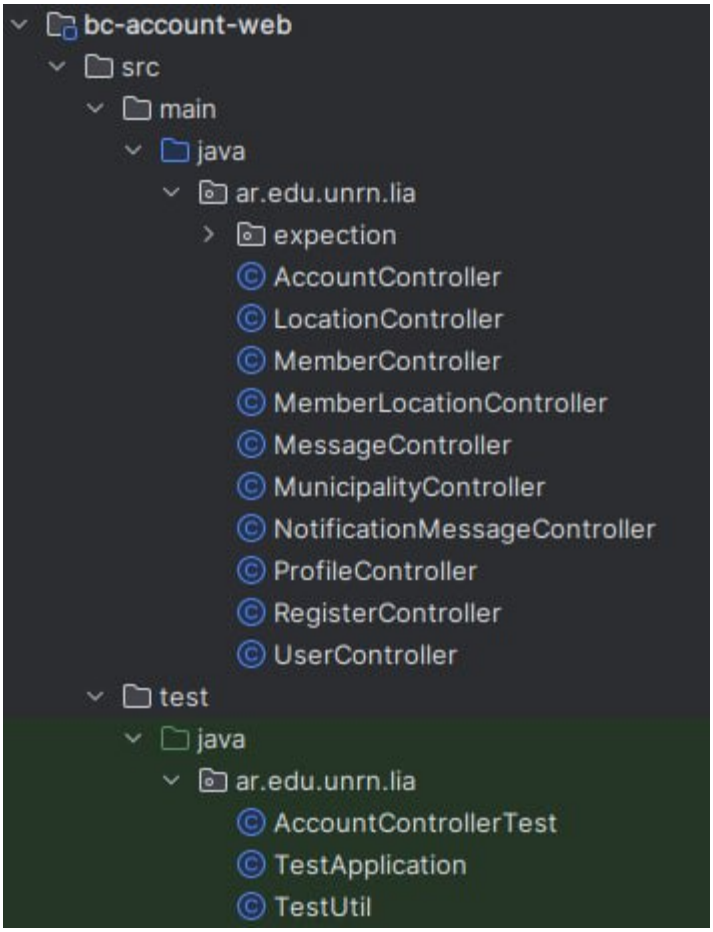


Figura 15 - Estructura de paquetes módulo bc-account-web

2.5.2 Test unitario

A continuación, se presenta una imagen destacando la implementación ejemplar de un caso de uso específico, detallando el código correspondiente enfocado en la creación de una ubicación. Este código se presenta como un ejemplo ilustrativo que representa la implementación real de toda la batería de tests unitarios implementados en el proyecto IPDTT.

```
@ServiceDomain
@RequiredArgsConstructor
@Feature(context = "bc-account")
class AddLocationHandle implements CommandHandler<AddLocationCommand> {

    private final MunicipalityPort municipalityPort;
    private final LocationPort locationPort;

    @Override
    @Transactional
    public void handle(AddLocationCommand command) {
        locationPort.validationByName(command.getName());
        Municipality municipality = municipalityPort.get(command.getMunicipalityId());
        LocationAccount locationPadre = command.getSuperior() != null ?
                locationPort.findById(command.getSuperior()) : null;

        LocationAccount locationAccount;
        if (command.getType().equals("UnitDependency"))
            locationAccount = UnitDependencyAccount.withoutId(command.getName(), command.getContact(),
                    municipality, locationPadre, command.getLinkInfo(), command.getLatitude(), command.getLongitude());

        else
            locationAccount = LocationAccount.withoutId(command.getName(),
                    municipality, locationPadre, command.getType());

        locationPort.create(locationAccount);
    }
}
```

Figura 16 - Implementación del caso de uso “Agregar Localidad”

Posteriormente, se exhibe el test unitario diseñado para verificar y validar el correcto funcionamiento de este caso de uso particular. El énfasis recae en utilizar este test como un ejemplo práctico para evaluar que el caso de uso implementado cumple con todas las especificaciones, es decir, los *requerimientos funcionales*, subrayando así la aplicación efectiva del testing unitario en la validación de la funcionalidad implementada.

```java
@ExtendWith(MockitoExtension.class)
public class AddLocationHandleTest {

    @InjectMocks
    private AddLocationHandle addLocationHandle;

    @Mock
    private MunicipalityPort municipalityPort;

    @Mock
    private LocationPort locationPort;

    @Test
    @DisplayName("El nombre de la localidad es invalido")
    public void invalidName() {
        AddLocationCommand command = new AddLocationCommand(
                1L,
                null, // Nombre invalido
                "type",
                1L,
                1L,
                "contact",
                12,
                12,
                "linkInfo"
        );

        BDDMockito
                .willThrow(new RuntimeException("localidad.exists"))
                .given(locationPort)
                .validationByName(ArgumentMatchers.anyString());

        RuntimeException thrown = Assertions.assertThrows(RuntimeException.class, () -> {
            addLocationHandle.handle(command);
        });
    }
```

Figura 17 - Test nombre de ubicación invalida

Conforme se señaló en la sección 2.2, la ejecución de pruebas unitarias se subdivide en dos categorías: pruebas de casos de uso y pruebas de controladores. A continuación, se describirá el fragmento de código(“endpoint”) utilizado para realizar una prueba unitaria utilizando MockMVC quedando el mismo como ejemplos de los todo los test de controladores restantes.

```java
@GetMapping(path = relativePath + "/{id}")
Account get(@PathVariable("id") Long id) throws QueryHandlerExecutionError {
    GetAccountCommand query = new GetAccountCommand(id);
    return queryBus.ask(query);
}
```

Figura 18 - Endpoint búsqueda de cuenta por ID

En la Figura 18 se exhibe el fragmento de código del endpoint "búsqueda de cuenta por ID", seguido por una explicación en la Figura 19 sobre la implementación de la prueba correspondiente.

```java
@WebMvcTest(controllers = AccountController.class)
class AccountControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private CommandBus commandBus;
    @MockBean
    private QueryBus queryBus;
    @MockBean
    private QueryFindAllBus queryFindAllBus;

    private final static String relativePath = "/api/account";

    @Test
    void testGetAccount() throws Exception {
        String url = relativePath + "/{id}";
        String idAccount = "1";

        when(queryBus.ask(any(GetAccountCommand.class)))
            .willReturn(new Account(1L, true, "Jose Luis", "http://localhost/image.png"))

        mockMvc.perform(get(url, idAccount)
                .header("Content-Type", "application/json"))
                .andExpect(status().isOk());
    }

}
```

Figura 19 - Test del endpoint con MockMvc

2.5.2 Test integración

En esta sección del proyecto, nos sumergimos en las pruebas de integración, utilizando Testcontainers como nuestra herramienta. Nos enfocaremos en evaluar la interacción y la interoperabilidad de los distintos módulos que componen el sistema IPDTT. La adopción de Testcontainers nos brinda la capacidad única de crear entornos de contenedores Docker temporales y reproducibles para simular situaciones del mundo real.

En la siguiente figura se específica cómo se realizó una configuración general que se utilizó como base para todo los test de integración.

```java
@DataJpaTest
@Testcontainers
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@ContextConfiguration(classes = {
        DemoApplication.class,
        AccountGovernmentTest.Initialiser.class
})
public abstract class DatabaseTest {

    @Container
    static PostgreSQLContainer POSTGRE_SQL = new PostgreSQLContainer("postgres:9.6.19-alpine")
            .withDatabaseName("springboot")
            .withPassword("springboot")
            .withUsername("springboot");

    static class Initialiser implements ApplicationContextInitializer<ConfigurableApplicationContext> {
        @Override
        public void initialize(ConfigurableApplicationContext applicationContext) {
            TestPropertyValues.of(
                    "spring.datasource.url=" + POSTGRE_SQL.getJdbcUrl(),
                    "spring.datasource.username=" + POSTGRE_SQL.getUsername(),
                    "spring.jpa.hibernate.ddl-auto=create-drop"
            ).applyTo(applicationContext.getEnvironment());
        }
    }
}
```

Figura 20 - Configuración Testcontainers

A continuación se indica como hacer uso de Testcontainers para implementar los test de integración.

```java
@Import({AccountPersistenceAdapter.class, AccountMapper.class, MunicipalityMapper.class})
public class AccountGovernmentTest extends DatabaseTest {

    @Autowired
    private AccountPersistenceAdapter adapter;

    @Autowired
    private AccountGovernmentRepository accountGovernmentRepository;
    @Autowired
    private AccountCommunityRepository accountCommunityRepository;
    @Autowired
    private AccountGovernmentRepository governmentRepository;
    @Autowired
    private MemberRepository memberRepository;
    @Autowired
    private MemberOwnerGovernmentRepository memberOwnerGovernmentRepository;

    @Test
    public void shouldStoreOneAccount() {

        Long idAccountGovernment = 1L;

        AccountGovernment account = AccountGovernment.withId(
                idAccountGovernment,
                true, //active
                AccountType.Name.GOVERNMENT, //type
                "http://localhost:123/photo", //url photo
                Municipality.withoutId("Municipalidad de Viedma", "", "", "", "")
        );

        adapter.create(account);

        Optional<AccountJpaEntity> accountGovernment = accountGovernmentRepository.findById(idAccountGovernment);

        Assertions.assertTrue(accountGovernment.isPresent());
    }
```

Figura 21 - Implementación test de integración

3 Calidad de código

La calidad del código en cualquier sistema es considerado una prioridad, ya que un código bien estructurado no solo mejora la legibilidad, sino que también facilita el mantenimiento y la detección temprana de posibles errores. La implementación de buenas prácticas de desarrollo se vuelve crucial para garantizar la integridad del sistema y su capacidad de adaptación.

Para lograr incrementar la calidad de código en el sistema IPDTT es necesario contar con anterioridad con la implementación testing que nos permite realizar inspecciones sobre el código, también conocidas como análisis estático del código fuente, son una de las primeras etapas de verificación de calidad en los entornos de desarrollo continuo.

Estas inspecciones se utilizan para medir atributos de calidad en el código fuente, de acuerdo con buenas prácticas de codificación. Por lo general, de estas mediciones se obtienen valores que podrán asociarse al grado de mantenibilidad del código fuente. Las inspecciones se ejecutan justo después de la integración del código fuente al repositorio.

La inspección de código fuente es un tipo de análisis de software que se lleva a cabo sin que el producto software esté en ejecución, o incluso, sin que este pueda ser compilado. Por esta razón se lo llama *estático*.

3.1 Conceptos

La calidad del código es un aspecto esencial en el desarrollo de software que va más allá de la simple ejecución de instrucciones por parte de un computador. En esta sección, se explora los conceptos fundamentales asociados con la calidad del código, destacando su importancia en la creación de sistemas robustos y sostenibles

Martin Fowler proporciona una perspectiva valiosa que contribuye a la comprensión de la calidad del código. Según él, "Any fool can write code that a computer can understand. Good programmers write code that humans can understand". Esta cita resalta la idea de que el código debe ser funcional no solo para las computadoras, sino también comprensible y colaborativo para los programadores.

3.2 Métricas

La forma de medir los atributos de calidad en el código fuente recibe el nombre de “métrica”. Las métricas pueden ser de alto nivel en todo el código o pueden centrarse en pequeños bloques de código en la clase, el método o incluso en un nivel inferior. Estas métricas generalmente se recopilan mediante herramientas durante las fases de análisis estático o dinámico.

La principal diferencia entre análisis estático o dinámico, es que el análisis dinámico ocurre mientras el sistema está en funcionamiento mientras que el análisis estático se ejecuta sin que el sistema esté activo y se centra en la estructura del código en sí, frente al funcionamiento del sistema.

Las herramientas de análisis estático examinan el código o sistema y miden preocupaciones específicas durante el análisis. Con el tiempo, la industria ha descubierto pautas generales en torno a varias métricas que indican si esa medida en particular se alinea o no con los rangos generalmente aceptados. Cada herramienta de análisis moderna proporcionará medidas y alguna referencia en cuanto a los valores estándar aceptados para esa métrica.

Algunas de las principales métricas y la evaluación de buenas prácticas obtenidas de diferentes herramientas que han demostrado estar relacionadas con la mantenibilidad son:

- Sentencias de código fuente no comentadas
- Complejidad del código fuente
- Métricas de diseño orientado a objetos
- Código duplicado
- Código no utilizado
- Validación de estilos
- Dependencias cíclicas

3.3 Cobertura

La cobertura de las pruebas es una idea muy importante porque proporciona una evaluación cuantitativa del alcance y la calidad de las pruebas [17]. En otras palabras, responde a la pregunta "¿cuántas pruebas has realizado?" de una manera que no está abierta a interpretación. Declaraciones como "Ya casi termino", o "He hecho pruebas de dos semanas" o "He hecho todo lo que estaba en el plan de pruebas" generan más preguntas de las que responden. Son declaraciones sobre cuántas pruebas se han realizado o cuánto esfuerzo se ha aplicado a las pruebas, en lugar de declaraciones sobre cuán efectivas han sido las pruebas o qué se ha logrado.

Necesitamos saber sobre la cobertura de las pruebas por dos razones muy importantes:

- Proporciona una medida cuantitativa de la calidad de las pruebas que se han realizado midiendo lo que se ha logrado.
- Proporciona una forma de estimar cuántas pruebas más deben realizarse. Utilizando medidas cuantitativas podemos establecer objetivos para la cobertura de las pruebas y medir el progreso en función de ellos.

3.3.1 JaCoCo

Para medir la cobertura de los tests se utilizó JaCoCo, que es una biblioteca de cobertura de código gratuita para Java, que ha sido creada por el equipo de EclEmma basándose en las lecciones aprendidas del uso y la integración de bibliotecas existentes durante muchos años [18].

Para poder configurar JaCoCo en nuestro proyecto es necesario aplicar los siguientes pasos:

- Agregar plugin de JaCoCo para maven

```xml
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.7.7.201606060606</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Figura 22 - Configuración JaCoCo plugin - Extraído de https://www.baeldung.com/jacoco

3.4 Sonarqube

La calidad del software es un elemento fundamental en la industria del desarrollo. Disponer de herramientas que nos ayuden a evaluar nuestro código se convierte en una tarea imprescindible para garantizar el desarrollo de una manera correcta y la aplicación de buenas prácticas. Para poder analizar las métricas antes mencionadas utilizaremos la herramienta Sonarqube.

SonarQube es una plataforma de código abierto para la inspección continua de la calidad del código a través de diferentes herramientas de análisis estático de código fuente. Proporciona métricas que ayudan a mejorar la calidad del código de un programa permitiendo a los equipos de desarrollo hacer seguimiento y detectar errores y vulnerabilidades de seguridad para mantener el código limpio.

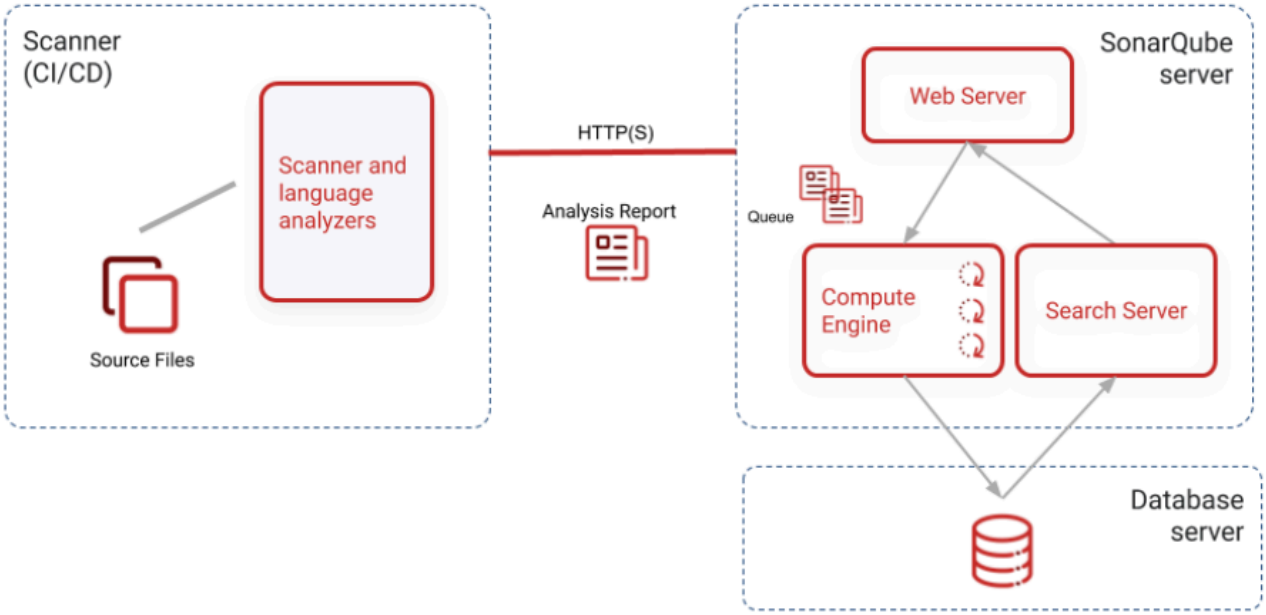


Figura 23 - Ecosistema de SonarQube - extraido de
https://docs.sonarsource.com/sonarqube/latest/setup-and-upgrade/install-the-server/introduction/

Para realizar un escaneo de nuestro código se basa en tres reglas principales:

1. Dominio de mantenimiento (maintainability)
2. Dominio de confianza (reliability)
3. Dominio de seguridad (security)

Un aspecto crucial consiste en establecer las condiciones indispensables para verificar que el proyecto está preparado para su implementación y que las reglas analizadas se han cumplido de manera precisa. En este sentido, haremos uso de las Quality Gates proporcionadas por SonarQube.

En términos más detallados, los Quality Gates de SonarQube son un conjunto de criterios y umbrales de calidad que se aplican durante el ciclo de vida del desarrollo de software. Estos criterios son esenciales para evaluar la salud general del código, garantizando que cumpla con los estándares predefinidos. Las Quality Gates de SonarQube actúan como un mecanismo de control de calidad, permitiendo que el despliegue del proyecto se realice únicamente cuando se cumplen todas las condiciones establecidas, asegurando así un software más robusto y fiable.

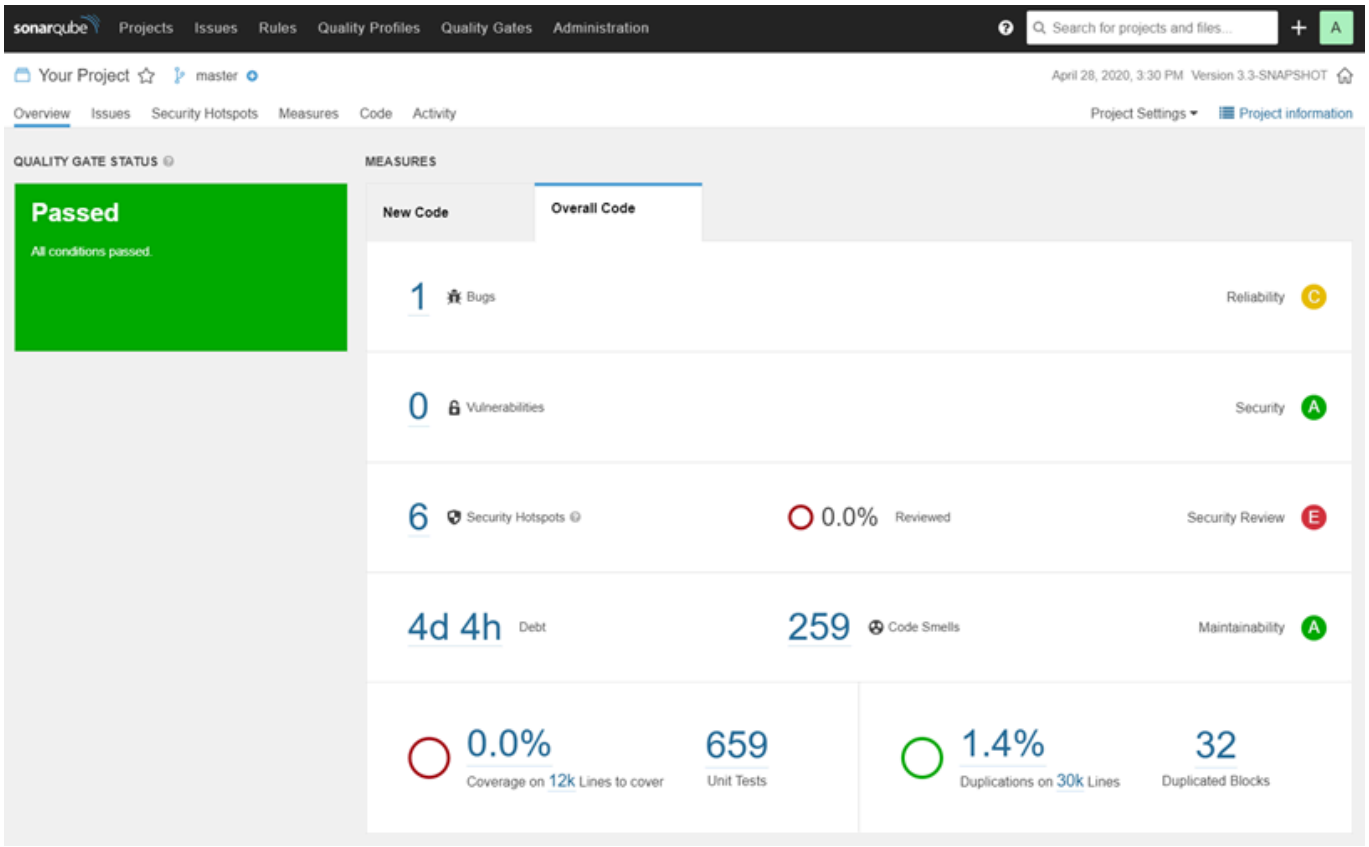


Figura 24 - Dashboard de SonarQube

3.4 Implementación

Para realizar un escaneo, es necesario ejecutar Sonar Scanner mediante el plugin de Maven de Sonar. El ejemplo siguiente se ejecuta manualmente en una máquina local, y este mismo procedimiento se detalla en la sección siguiente, donde se explica cómo se logró automatizar mediante herramientas de canalización.

```
mvn clean verify sonar:sonar
-Dsonar.projectKey=Prueba-Back
-Dsonar.host.url=http://localhost:9000
-Dsonar.login=sqp_463f42f0ffb2da667f3b56175e05de97206d5412
```

Figura 25 - Sonar Scanner mediante plugin de maven

4 Automatización

Históricamente, la automatización surgió para reducir el esfuerzo humano requerido en actividades que podrían ser replicadas por un sistema o máquina programable.

Cuando hablamos de automatización en la actualidad nos basamos en tres pilares fundamentales, la integración, entrega e implementación continuas que son prácticas de desarrollo relativamente nuevas que han ganado mucha popularidad en los últimos años. En el desarrollo del proyecto IPDTT backend, se implementó la automatización con el objetivo de agilizar la verificación de pruebas y la evaluación de la calidad del código.

4.1 Integración continua

El punto de partida esencial para proporcionar software coherente y de alta calidad radica en la implementación de la Integración Continua (CI). La CI se centra en asegurar que el software se encuentre en un estado de implementación constante.

El proceso de Integración Continua se inicia mediante el uso de un repositorio; en el caso del proyecto IPDTT, se emplea GitLab [1]. Los sistemas de control de código fuente garantizan la centralización de todo el código en un único lugar, facilitando a los desarrolladores la verificación, modificación y revisión del código fuente.

Según Sander Rossell, los modernos sistemas de control de fuente, como GitLab, admiten múltiples ramas (branch) del mismo software. Esto posibilita el trabajo en diversas etapas del software sin afectar, e incluso detener, otras fases del desarrollo.

Una vez que nuestro código ha sido integrado en el repositorio, el siguiente paso consiste en compilarlo. Este proceso genera un archivo *.jar* [2] que encapsula la aplicación. Para almacenar este paquete, aprovechamos los artefactos de GitLab, los cuales podremos utilizar posteriormente en los procesos de despliegue como se muestra en la siguiente figura.

---

[1] Gitlab - Es una plataforma para gestionar proyectos de desarrollo de software con funciones como control de versiones, seguimiento de problemas y CI/CD. Facilita la colaboración y automatiza procesos en el desarrollo de software. Puede utilizarse en la nube o en instalaciones locales.

https://about.gitlab.com/why-gitlab/

[2] Jar - Un archivo JAR (Java Archive) es un formato comprimido utilizado para empaquetar y distribuir aplicaciones Java. Contiene archivos de código fuente, archivos compilados, recursos y metadatos en un solo archivo. https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html

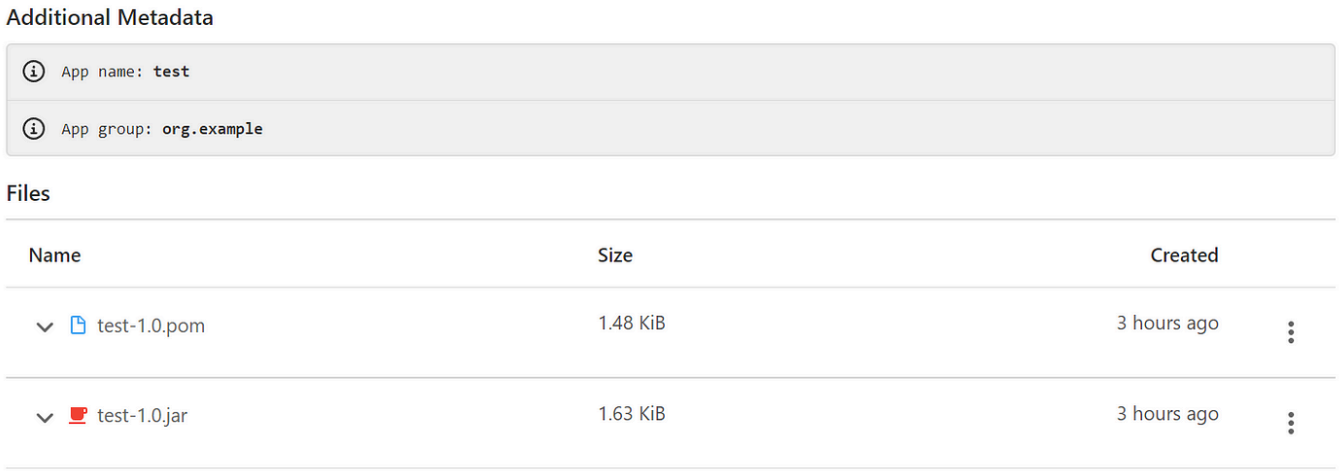


Figura 26 - Ejemplo de empaquetamiento

Si la compilación y empaquetamiento fue exitoso en el servidor de CI se procede con la verificación de calidad del software, que se divide en dos procesos, la ejecución de los test y el análisis estático de código.

4.1.1 Testing automatizado

El testing automatizado consiste en que una máquina logre ejecutar los casos de prueba en forma automática, en nuestro caso es el servidor de Gitlab CI es el encargado de ejecutarlas mediante un script. Al automatizar pruebas de software se busca simplificar el trabajo repetitivo o complejo, haciéndolo más efectivo y más productivo.

Humble y Farley (2010) afirman que se deben automatizar todos los niveles de pruebas posibles, además otros autores realizaron estudios demostrando que la automatización de pruebas es una pieza clave en la entrega incremental de software (Shahin, Babar y Zhu, 2017).

4.1.2 Análisis de calidad de código

Tras la ejecución de las pruebas unitarias y de integración, se lleva a cabo la validación de la cobertura del código, generando así un informe detallado con los resultados de la ejecución. Este informe se empleará en conjunto con el código fuente por Sonarqube para llevar a cabo el análisis estático del proyecto. Los resultados obtenidos de este análisis proporcionarán la información necesaria para tomar decisiones sobre la necesidad de

realizar refactoring en el proyecto.

Figura 27 - Beneficios de la integración continua - Extraído de Grupoica

En la integración continua, el último paso es crear una imagen Docker a partir del artefacto de compilación. Una vez creada la imagen, se sube a un repositorio de imágenes Docker integrado en GitLab.

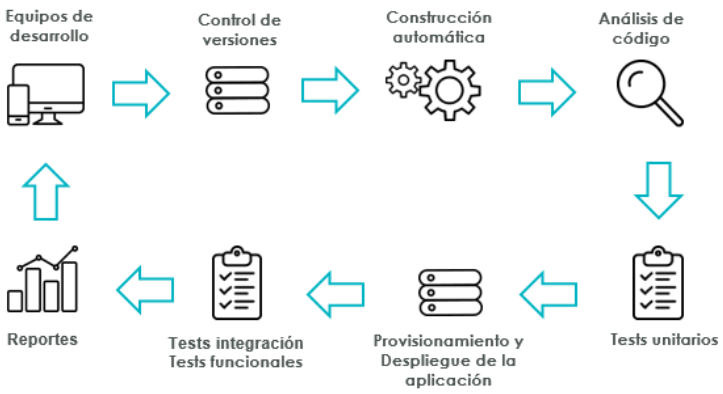


Figura 28 - Proceso de integraciòn continua - Extraído de Grupoica

4.2 Despliegue continuo

Cuando nos referimos a despliegue de aplicaciones utilizando mecanismos de automatización la variante que suele surgir es la entrega continua (CD, del inglés “Continuous Delivery”), que se basa en el hecho de que siempre hay una rama principal estable(Master o Main) del código y la implementación en producción puede realizarse en cualquier momento desde esa rama. La rama principal o branch, se mantiene lista para producción, lo que se ve facilitado por la automatización de implementaciones y pruebas.

Un artefacto se construye sólo una vez y se recupera de un repositorio central. Las implementaciones en entornos de prueba y producción se realizan de la misma manera y se utiliza el mismo artefacto para todos los entornos de destino.

4.3 Beneficios

Según Henry van Merode, los beneficios inherentes a la integración y entrega continuas radican en la capacidad de acelerar la entrega del código de la aplicación al entorno de producción mediante la automatización de la cadena de suministro de software. Este enfoque no solo agiliza el tiempo de comercialización del producto (Time to Market), sino que también conlleva beneficios adicionales.

La automatización no solo elimina el error humano en el proceso de implementación, sino que también fortalece la seguridad del código, mitigando posibles vulnerabilidades. Además, al seguir prácticas de integración y entrega continuas, se logra una reproducibilidad mejorada en todo el ciclo de vida del software. Estos elementos se

traducen en un código más seguro, de mejor calidad y en un ciclo de desarrollo que puede reproducirse de manera consistente, ofreciendo una respuesta más eficiente y rápida en la evolución del producto.

4.4 Implementación

Para llevar a cabo la automatización del backend IPDTT, es imprescindible generar un archivo llamado ".gitlab-ci.yml" y colocarlo en el directorio raíz. Esto permitirá que Gitlab CI lo identifique automáticamente y ejecute los scripts que se encuentran en dicho archivo.

El proyecto en cuestión ya incorporaba un sistema de automatización, aunque este se limitaba a la compilación y despliegue. El propósito final de este trabajo consistió en añadir fases adicionales que posibilitarán la ejecución y verificación de pruebas, así como la evaluación de la calidad del código a través de SonarQube.

Durante el intento de ejecutar las pruebas de integración en Gitlab, nos enfrentamos a un inconveniente: era necesario implementar un contenedor dentro de nuestro proceso de canalización. Para abordar esta necesidad, aprovechamos un servicio conocido como Docker Dind (Docker In Docker), el cual nos otorga la capacidad requerida.

```yaml
services:
  - name: docker:dind
    command: ["--tls=false"]

variables:
  DOCKER_HOST: "tcp://docker:2375"
  DOCKER_TLS_CERTDIR: ""
  DOCKER_DRIVER: overlay2
```

Figura 29 - Configuración DIND en .gitlab-ci.yml

Para completar la verificación mediante SonarQube, es esencial suministrar dos variables que son fundamentales para que pueda llevar a cabo su tarea:

- Token de Sonar: Se debe generar un token específico de SonarQube para GitLab y luego crear una variable de entorno personalizada en GitLab denominada SONAR_TOKEN. En esta variable, la clave sería el nombre, y el valor sería el token recién generado.

- URL del host de Sonar: Asimismo, es necesario crear otra variable de entorno personalizada, llamada SONAR_HOST_URL, donde la clave sería el nombre y el valor sería la dirección del servidor SonarQube.

La figura subsecuente detalla el script empleado para realizar la ejecución de pruebas unitarias e integración, además de la verificación de la calidad del código. Con esto, se concluye la configuración de las etapas adicionales en la automatización o canalización.

```yaml
stages:
  - sonarqube-check
  - build-dev
  - deploy-dev
  - build-testing
  - deploy-testing

sonarqube-check:
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"
    GIT_DEPTH: "0"
  cache:
    key: "${CI_JOB_NAME}"
    paths:
      - .sonar/cache
  script:
    - mvn verify sonar:sonar -Dsonar.qualitygate.wait=true
  allow_failure: true
  rules:
    - if: $CI_COMMIT_REF_NAME == $BRANCH || $CI_PIPELINE_SOURCE == 'merge_request_event'
```

Figura 30 - Etapas de la pipeline. Script de los test y SonarQube.

5 Conclusiones

En conclusión, el presente trabajo ha abordado de manera integral la mejora y automatización de pruebas en el sistema "Ideas Proyecto de Desarrollo y Transferencia de Tecnología" (IPDTT), enfocándose en tres pilares fundamentales del desarrollo de software: testing, calidad del código y automatización.

El proceso de testing se revela como un componente esencial para garantizar la calidad y confiabilidad del software, permitiendo la identificación temprana y corrección de posibles errores. La aplicación de pruebas no sólo valida la conformidad con los requisitos establecidos, sino que también facilita la detección temprana de fallos, mejorando así la mantenibilidad y proporcionando confianza en la estabilidad y rendimiento del sistema.

La calidad del código, considerada una prioridad, se ha abordado mediante la implementación de buenas prácticas de desarrollo. La realización de inspecciones estáticas del código fuente ha sido esencial para medir atributos de calidad y asociarlos al grado de mantenibilidad del código, contribuyendo así a la integridad y adaptabilidad del sistema.

La automatización, centrada en la integración, entrega e implementación continuas, ha sido implementada en el proyecto IPDTT backend para agilizar la verificación de pruebas y evaluar la calidad del código de manera eficiente.

En conjunto, estos esfuerzos han fortalecido la base del desarrollo de software, asegurando la robustez, mantenibilidad y escalabilidad del sistema IPDTT. Este enfoque integral no solo ha mejorado el proceso de desarrollo, sino que también sienta las bases para futuras implementaciones y actualizaciones, consolidando así el éxito y la eficiencia del proyecto en el ámbito de la tecnología y la salud.

Referencias

1. Enrique Pablo Molinari. (2020). Coding An Architecture Style - A practical guide to learn Software Architecture by coding in Java. pag 12-15.
2. Timothy M. O'Brien, John Casey. (2011). Maven: The Complete Reference. cap 1.1. https://books.sonatype.com/mvnref-book/reference/index.html
3. Alistair Cockburn. (2005). Hexagonal architecture. https://alistair.cockburn.us/hexagonal-architecture/
4. Mike Cohn. (2009). Succeeding with Agile: Software Development Using Scrum, pag 312
5. JUnit 5. Documentación oficial. https://junit.org/junit5/docs/current/user-guide/
6. https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html
7. Martin Fowler. (2018). Integration Test. https://martinfowler.com/bliki/IntegrationTest.html
8. Humble, Jez y Farley, David. (2010). Continuous delivery: reliable software releases through build, test, and deployment automation. EE.UU.: Pearson Education.
9. Shahin, Mojtaba, Babar, Muhammad Ali y Zhu, Liming. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access, 5, 3909-3943
10. Louridas, Panagiotis. (2006). Static code analysis. IEEE Software, 23 (4), 58-61
11. International Software Testing Qualifications Board. https://www.istqb.org/
12. Desarrollo de Software Ágil en 10Pines. https://10pines.gitbook.io/
13. Testcontainers. Documentación oficial. https://testcontainers.com/guides/introducing-testcontainers/
14. Sander Rossel. (2017). Continuous Integration, Delivery, and Deployment.
15. Giménez, M y Espínola, A. (2014). Automatización de Pruebas para Interfaces de Aplicaciones Web. http://www.cc.pol.una.py/wpfg2014/
16. Abstracta Team.(2024, 2 de enero). ¿Qué son las Pruebas de Regresión en Agile? https://cl.abstracta.us/blog/pruebas-regresion-entorno-agile/
17. Brian Hambling, Peter Morgan, Angelina Samaroo, Geoff Thompson y Peter Williams. (2010). Software testing. An ISTQB-ISEB Foundation Guide Revised Second Edition.
18. JaCoCo. (2024). Documentación oficial. https://www.jacoco.org/jacoco/trunk/doc/
19. Grupoica. Arquitectura e Integración Digital. https://www.grupoica.com/transformacion-digital/arquitectura-integracion-digital
20. Henry van Merode. (2023). Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines. Cap 2.