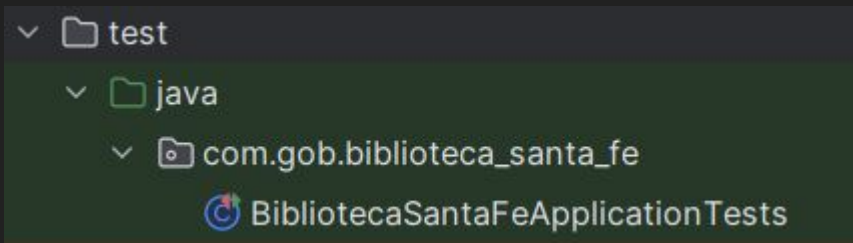


# Testing Automatizado

# Testing automatizado

- Una de las principales ventajas de la inyección de dependencia es que nos permite instanciar objetos simulados, facilitando las pruebas unitarias.
- Cuando creamos un proyecto de Spring con el Initializer, tenemos creado por defecto el directorio donde se ubicará todo el código que tiene como objetivo automatizar el test.



# JUnit

- JUnit es un framework para escribir y ejecutar Tests
- Cada test está vinculado con un método y cada método debe representar un escenario de test específico para el que se compara la salida esperada con la realmente recibida.
- Un método es identificado como método de testing si tiene una anotación “@Test”, y obviamente, contiene código que realiza la invocación a lógica de negocio a testear

# JUnit

- Los resultados esperados se evalúan a través de “afirmaciones”
- Una “afirmación” es un predicado que verifica una situación que asumimos como cierta es igual al resultado obtenido de la ejecución de una porción del programa a probar
- Ejemplos:

```
– assertEquals
– assertEquals
– assertSame    - assertNotSame;
– assertTrue   - assertFalse;
– assertNull    - assertNotNull;
```

# Ejemplos de Asserts

```
public class AssertTests {  
    @Test  
    public void testAssertEquals() {  
        assertEquals("Fallo no son iguales", "text", "text");  
    }  
  
    @Test  
    public void testAssertArrayEquals() {  
        byte[] expected = "trial".getBytes();  
        byte[] actual = "trial".getBytes();  
        assertEquals("Falla no son iguales", expected, actual);  
    }  
  
    @Test  
    public void testAssertFalse() {  
        assertFalse("failure - should be false", false);  
    }  
}
```

# Asserts agregados en JUnit5

- En JUnit5 se agregó un `assertAll` que recibe una lista de funciones lambdas que implementan la interface funcional “Executable” que tiene el método “execute”.
- Otras incorporadas son:
  - `assertTimeOut`
  - `assertThrows`

```
@Test
void verificarAll() {
    Producto p= new Producto();
    p.setDescripcion("PProducto1");
    p.setPrecio(100.0);
    assertAll("producto",
        () -> assertEquals("PProducto1", p.getDescripcion()),
        () -> assertEquals(100.0, p.getPrecio())
    );
}
```

# Ignorar un test

- Si una falla hay en un bloque de código y no queremos testearlo hasta que no esté corregido, pero si queremos ejecutar el resto de los test podemos:
  - Comentar el test
  - Borrarlo
  - JUnit propone agregar la anotación `@Disabled` pasando como parámetro la razón

– `@Test`

– **`@Disabled ("John's holiday stuff failing")`**

– `public void when_today_is_holiday_then_stop_alarm() {}`

# Anotaciones

@Test	Declara que un metodo puede ser testeado
@ParameterizedTest	Permite configurar un método para ser testeado en multiples ejecuciones con distintos parametros
@RepeatedTest	Permite repetir N veces un test
@TestMethodOrder	Orden de ejecucion
@DisplayName	Declares a custom <a href="#">display name</a> for the test class or test method. Such annotations are not <i>inherited</i> .
@BeforeEach	Ejecuta ante/despues de cada test
@AfterEach	
@BeforeAll	Ejecuta antes/despues de todos los test
@AfterAll	
@Disabled	Especifica que un test no se ejecute
@Timeout	Falla el test si demora más de N milisegundos



# Crear un TestUnitario

- Creamos una clase que contenga nuestros tests y lo anotamos con `@SpringBootTest` para que pueda inyectar las dependencias necesarias
- Creamos los métodos que representan cada caso de prueba y los anotamos con `@Test` para que Spring entienda que representan un test ejecutable.
- Los métodos siempre son public, retornan void y no tienen parámetros.

```
@SpringBootTest new *
public class LibroServiceTests {

    @Test new *
    public void findLibroByNameNoEncontrado(){
        |
    }
}
```

# @SpringBootTest

- Esta anotación nos permite inyectar las clases que requerimos.
- Se encarga de generar un ApplicationContext de prueba.

```
@SpringBootTest new *
public class LibroServiceTests {

    @Autowired 1 usage
    LibroService libroService;

    @Test new *
    public void findLibroByNameExito(){

        Libro libro = new Libro(
            id: 1L,
            isbn: "978-1-56619-909-5",
            nombre: "1984",
            autor: "George Orwell",
            cantidad: 12,
            fechaCreacion: null,
            fechaModificacion: null);

        Libro libroBuscado= libroService.findByNombre("1984");

        assertEquals(libro, libroBuscado);
    }
}
```

# Test Unitario

- Necesitamos simular componentes que aún no tenemos desarrollados, o que no queremos usar en su implementación real porque es muy costoso o no queremos alterar los datos.
- Por ejemplo, la llamada a un api o una llamada a la base de datos
- El objetivo es romper la dependencia con componentes externos y de esta manera probar requerimientos de manera independiente.
- El costo de escribir estos objetos falsos puede ser muy alto por lo que existen diversos frameworks que facilitan esto siendo Mockito el más popular

# Test Unitario

- Spring boot incluye dos anotaciones propias, por defecto, para objetos falsos
- Pertencen al paquete `org.springframework.boot.test.mock.mockito`:
  - `@MockBean`
  - `@SpyBean`
- Si usamos la anotación `@SpringBootTest` se habilita automáticamente el uso de Mockito
- Sino debemos incluirlo con la siguiente anotación:
  - `@TestExecutionListeners ({MockitoTestExecutionListener.class, ResetMocksTestExecutionListener.class})`

# Spy

- Estos objetos guardan las acciones que se hacen sobre ellos
- Hace una especie de seguimiento sobre qué métodos se han llamado y con qué parámetros
- Cuando para que un test sea un éxito no es suficiente ver el estado de los objetos disponibles, podemos usar un spy y comprobar cosas como cuántas veces se ha llamado a un método, qué valores han devuelto, etc.

# Mock

- Muy similar a un spy, pero no solo guardan las acciones que se hacen sobre ellos, también es necesario configurar qué comportamiento se espera cuando se invoque alguno de sus métodos
- Se usan para probar que se realizan correctamente llamadas a otros métodos, por ejemplo, a una web API, por lo que se utilizan para verificar el comportamiento de los objetos
- Los mocks también pueden guardar un registro de las invocaciones realizadas
- Si usamos la anotación `@SpringBootTest` se habilita automáticamente el uso de Mockito

# Determinemos qué dependencias son reales y cuáles falsas

```
@SpringBootTest
public class UsuarioServiceTests {

    UsuarioDTO usuarioDT0Busqueda;
    String id;
    Usuario usuario;
    UsuarioDTO usuarioComplDto;

    @Autowired
    UsuarioService usuarioService;

    @MockBean
    UsuarioRepository usuarioRepository;
```






# Mejoremos nuestros test

- Es momento de determinar qué deben hacer nuestros objetos falsos
- Por defecto los objetos Mock, cuando reciben una invocación a un método “ejecuta el método real de la clase que ocultan”
- Este comportamiento no es el deseado dado que queremos probar la lógica de nuestro método con situaciones puntuales, por lo que se configurarán las acciones a realizar ante cada método
- Para esto usaremos el método `when()`



# Mock when

- El método when nos permite especificar que se retorna
- Nos provee una serie de métodos

```
 thenReturn(Optional<Usuario> t)  
 thenReturn(Optional<Usuario> t, 0...  
 thenAnswer(Answer<?> answer)  
 thenThrow(Throwable... throwables)  
 thenCallRealMethod()
```

## Ejemplo del uso de when

```
when( usuarioRepository.findByEmail("example@test.com.ar")).thenReturn(Optional.of(usuario));
```

```
when( usuarioRepository.findById("123456789")).thenReturn( t: Optional.empty());
```

```
when( usuarioRepository.findById("123456789")).thenThrow(new Exception("Error"));
```

# Mejorando el uso de los when

- Podemos mejorar el uso de los when usando matchers genericos como:
  - anyString()
  - anyInt()
  - anyDouble()
  - any(Class<T> type)
- Por ejemplo

```
when( usuarioRepository.findById(anyString()) ).thenReturn( t: Optional.empty());
```

```
when(usuarioRepository.findByIdByEmail(anyString())).thenReturn(Optional.of(usuario));
```

# Verificar invocaciones

- Para verificar la cantidad de invocaciones de un método podemos usar el método `verify()`
- Se lo puede invocar sin parámetros y verificará que el método se invocó al menos una vez
- Con el parámetro `never()` verifica que nunca se haya invocado
- Con el parámetro `times(int n)` verifica que se haya invocado `n` veces
- `atLeast(int n)` verifica que se haya invocado al menos `n` veces
- `atMost(int n)` verifica que se haya invocado como máximo `n` veces
- `after(long millis)` sirve para diferir la verificación una cantidad de milisegundo, esto se utiliza para verificar métodos asíncronicos.

```
verify(mock, after(100).times(1)).someMethod();
```

# Un ejemplo completo

@SpringBootTest

```
public class UsuarioServiceTests {
```

```
    UsuarioDTO usuarioDTOBusqueda; 4 usages
```

```
    String id; 8 usages
```

```
    Usuario usuario; 7 usages
```

```
    UsuarioDTO usuarioCompleto; 7 usages
```

```
@Autowired 13 usages
```

```
    UsuarioService usuarioService;
```

```
@MockBean 38 usages
```

```
    UsuarioRepository usuarioRepository;
```

```
@BeforeEach
```

```
public void setUp(){
```

```
    id = UUID.randomUUID().toString();
```

```
    usuarioDTOBusqueda = new UsuarioDTO(
```

```
        id,
```

```
        email: "example.email@test.com",
```

```
        nombre: "example",
```

```
        apellido: "email",
```

```
        password: null,
```

```
        LocalDate.now().toString());
```

```
    usuarioCompleto = new UsuarioDTO(
```

```
        id,
```

```
        email: "example.email@test.com",
```

```
        nombre: "example",
```

```
        apellido: "email",
```

```
        password: "123456789",
```

```
        LocalDate.now().toString());
```

```
    usuario = new Usuario(
```

```
        id,
```

```
        email: "example.email@test.com",
```

```
        password: "123456789",
```

```
        nombre: "example",
```

```
        apellido: "email",
```

```
        LocalDate.now(),
```

```
        RolUsuario.USER,
```

```
        resetToken: null);
```

```
}
```

```
@Test
```

```
public void findByIdNotFoundErrorTest(){
```

```
    String idIncorrecto = "1";
```

```
    EntidadNoEncontradaException entidadNoEncontradaException = null;
```

```
    Exception exception = null;
```

```
    when( usuarioRepository.findById(anyString()) ).thenReturn( Optional.empty());
```

```
    try{
```

```
        usuarioService.findById(idIncorrecto);
```

```
    }
```

```
    catch(EntidadNoEncontradaException notFoundException){
```

```
        entidadNoEncontradaException = notFoundException;
```

```
    }
```

```
    catch(Exception e){
```

```
        exception = e;
```

```
    }
```

```
    assertNull(exception);
```

```
    assertNotNull(entidadNoEncontradaException);
```

```
    assertEquals( expected: "No existe usuario con id " + idIncorrecto,
```

```
        entidadNoEncontradaException.getLocalizedMessage());
```

```
    verify(usuarioRepository, times( wantedNumberOfInvocations: 1)).findById(anyString());
```

```
}
```

# Un ejemplo completo

```
@Test
public void findByIdOkTest() {

    UsuarioDTO retornoServicio = null;
    EntidadNoEncontradaException entidadNoEncontradaException = null;
    Exception exception = null;

    when(usuarioRepository.findById(anyString())).thenReturn(Optional.of(usuario));

    try {
        retornoServicio = usuarioService.findById(id);
    } catch (EntidadNoEncontradaException notFoundException) {
        entidadNoEncontradaException = notFoundException;
    } catch (Exception e) {
        exception = e;
    }

    assertNull(exception);
    assertNull(entidadNoEncontradaException);
    assertEquals(usuarioDTOBusqueda, retornoServicio);
    verify(usuarioRepository, times(wantedNumberOfInvocations: 1)).findById(anyString());
}
```

# Un ejemplo completo

```
@Test
public void crearUsuarioEmailRepetidoTest(){

    when(usuarioRepository.findByEmail(anyString())).thenReturn(Optional.of(usuario));

    InvalidPasswordException invalidPasswordException= null;
    Exception exception= null;
    EntidadRepetidaException entidadRepetidaException= null;

    try{
        usuarioService.crearUsuario(usuarioComplDto);
    }
    catch(EntidadRepetidaException repetidaException){
        entidadRepetidaException= repetidaException;
    }
    catch(InvalidPasswordException iPasswordException){
        invalidPasswordException= iPasswordException;
    }
    catch(Exception e){
        exception= e;
    }

    assertNotNull(entidadRepetidaException);
    assertNull(invalidPasswordException);
    assertNull(exception);
    assertEquals( expected: "Ya existe un usuario con email "+usuarioComplDto.getEmail(), entidadRepetidaException.getLocalizedMessage());

    verify(usuarioRepository, times( wantedNumberOfInvocations: 1)).findByEmail(anyString());
    verify(usuarioRepository, times( wantedNumberOfInvocations: 0)).save(any(Usuario.class));
}
```

# Plugins de Maven

- Importante: Maven encuentra para la fase de test las clases cuyo nombre termina en `*Test.java`
- Tenemos varios plugins que podemos utilizar para test unitarios
- Los dos más utilizados son
  - Surefire
  - Jacoco



# Surefire

- El plugin Surefire se utiliza durante la fase de test del ciclo de vida de build para ejecutar las pruebas unitarias de una aplicación
- Genera reportes en formato xml y txt
- Por defecto la ruta en donde se deja los informes es `${basedir} / target / surefire-reports / TEST - *. Xml`.

# Configurar Surefire para reportes

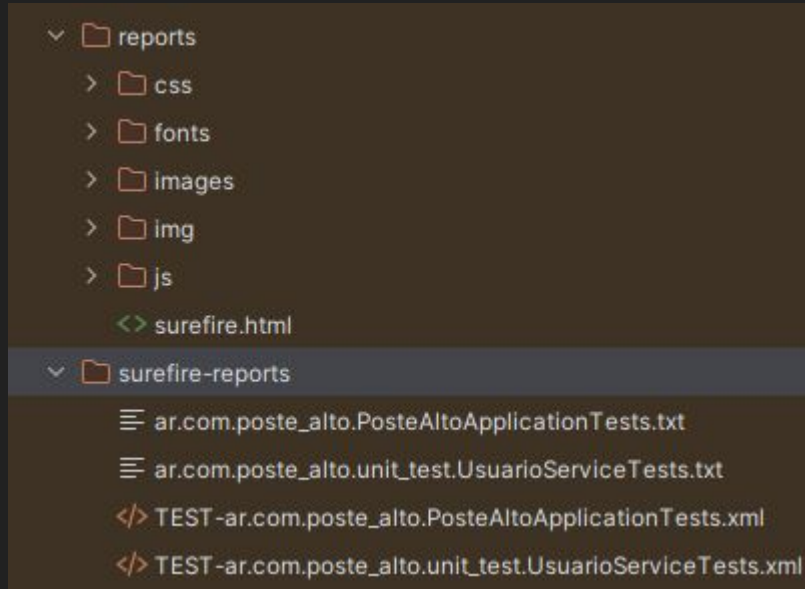
- Para agregar los reportes de Surefire al reporte general de maven debemos agregar el siguiente plugin

```
<reporting>
  <plugins>
    . . .
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-report-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
```

# Configurar Surefire para reportes

- El plugin se ejecuta dentro del ciclo de vida del componente
- Podemos crear un html con el reporte a través del comando “mvn surefire-report:report”
- No debemos indicar versión dado que el pom.xml padre de spring boot ya posee configurada la misma

# Surfire reports



```
-----  
Test set: ar.com.poste_alto.unit_test.UsuarioServiceTests  
-----
```

```
Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.723 s -- in  
ar.com.poste_alto.unit_test.UsuarioServiceTests
```

# Surfire reports en formato html

## Surefire Report

### Summary

[\[Summary\]](#) [\[Package List\]](#) [\[Test Cases\]](#)

Tests	Errors	Failures	Skipped	Success Rate	Time
14	0	0	0	100%	16.04 s

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

### Package List

[\[Summary\]](#) [\[Package List\]](#) [\[Test Cases\]](#)

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
<a href="#">ar.com.poste_alto</a>	1	0	0	0	100%	11.45 s
<a href="#">ar.com.poste_alto.unit_test</a>	13	0	0	0	100%	4.586 s

# Surfire reports en formato html

## UsuarioServiceTests

☀	crearUsuarioEmailRepetidoTest	0.113 s
☀	crearUsuarioPasswrodVacioTest	0.009 s
☀	borrarUsuarioTest	0.009 s
☀	crearUsuarioTest	0.143 s
☀	findByEmailTest	0.007 s
☀	crearUsuarioPasswordNullTest	0.005 s
☀	modificarUsuarioTest	0.009 s
☀	modificarUsuarioEmailRepetidoTest	0.008 s
☀	findByIdOkTest	0.008 s
☀	modificarUsuarioNoEncontradoTest	0.007 s
☀	findByEmailNotFoundErrorTest	0.008 s
☀	findByIdNotFoundErrorTest	0.008 s
☀	borrarUsuarioNoEncontradoTest	0.008 s

# Jacoco

- Nos permite realizar reportes de cobertura de código de nuestros tests
- Los reportes de cobertura se hacen basados en dos conceptos
  - Instrucciones
  - Caminos lógicos
- La cobertura de instrucciones proporciona información sobre la cantidad de código que se ha ejecutado o se ha perdido.
- Para la cobertura de caminos lógicos, el plugin calcula la complejidad ciclomática para cada método no abstracto.

# Complejidad ciclomática

- Según McCabe 1996, la complejidad ciclomática es el número mínimo de caminos que pueden, en combinación, generar todos los caminos posibles a través de un método
- El valor de esta complejidad sirve como una indicación del número de casos de prueba unitarios para cubrir completamente una determinada pieza de software
- La complejidad ciclomática se calcula como número de ramas (Branches) menos la cantidad de decisiones (D) más uno.
- $v(G) = B - D + 1$



# Ejemplo de cálculo de complejidad ciclomática

La cantidad de branches en el código es 6

La cantidad de decisiones son 3

Por tanto la complejidad ciclomática de este código es  $v(G) = 6 - 3 + 1 = 4$

De esto deducimos que con 4 test unitarios deberíamos tener un coverage del 100% para este código

```
if(a==1){ //decision point 1
    //branch 1
    if(b==2) { //decision point 2
        // branch 2
        if (c==3) { //decision point 3
            // branch 3
        } else {
            // branch 4
        }
    } else {
        // branch 5
    }
} else {
    // branch 6
}
```

# Configurar Jacoco

- Debemos agregar el siguiente plugin en el elemento “build”

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.12</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Indica que hay que ejecutar el goal “report” en la fase “test” del ciclo de vida “build”

# Reporte resultados

- El resultado se puede disponer en `target/site/jacoco/index.html`

## poste-alto

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
<a href="#">ar.com.poste_alto.exceptions</a>		16 %		n/a	6	10	30	38	6	10	1	5
<a href="#">ar.com.poste_alto.controller</a>		7 %		n/a	5	6	10	11	5	6	0	1
<a href="#">ar.com.poste_alto</a>		15 %		n/a	3	4	4	5	3	4	1	2
<a href="#">ar.com.poste_alto.config.security</a>		65 %		n/a	2	7	2	10	2	7	0	1
<a href="#">ar.com.poste_alto.model</a>		0 %		n/a	2	2	2	2	2	2	1	1
<a href="#">ar.com.poste_alto.services</a>		100 %		100 %	0	14	0	48	0	11	0	1
<a href="#">ar.com.poste_alto.enums</a>		100 %		n/a	0	1	0	3	0	1	0	1
Total	178 of 449	60 %	0 of 6	100 %	18	44	48	117	18	41	3	12

# Plugins para análisis estático

- Además de testear el funcionamiento de nuestro código, también es importante verificar estáticamente, si adhiere a las mejores prácticas de desarrollo con java
- Para realizar esto basta con inspeccionar aleatoriamente algunas muestras de código en forma manual
- O usar herramientas automatizadas que realizan estas tareas
- Entre las más populares se encuentran:
  - PMD: <https://maven.apache.org/plugins/maven-pmd-plugin/>
  - Checkstyle: <http://maven.apache.org/plugins/maven-checkstyle-plugin/checkstyle.html>
  - SpotBug: <https://spotbugs.github.io/>

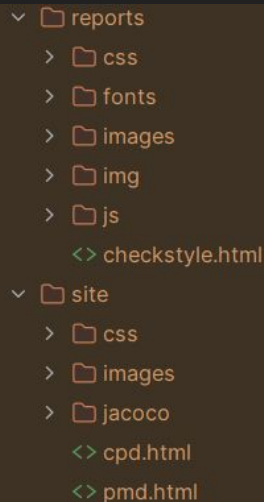
# Configuremos algunos

- dentro de build agregamos los plugins

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <version>3.24.0</version>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>3.5.0</version>
</plugin>
```

# Corremos pmd y checkstyle

- Para correr checkstyle usamos `mvn checkstyle:checkstyle`
- Para generar el reporte de pmd usamos `mvn pmd:pmd`
- Para buscar código duplicado podemos usar `mvn pmd:cpd`
















```

▼ reports
  > css
  > fonts
  > images
  > img
  > js
  <> checkstyle.html
▼ site
  > css
  > images
  > jacoco
  <> cpd.html
  <> pmd.html
```

# Checkstyle report

## ar/com/poste\_alto/controller/UsuarioController.java






Severity	Category	Rule	Message	Line
 Error	javadoc	JavadocPackage	Falta el archivo package-info.java.	1
 Error	javadoc	JavadocVariable	Falta el comentario Javadoc.	24
 Error	design	VisibilityModifier	La variable 'usuarioService' debe ser privada y tener métodos de acceso.	25
 Error	regexp	RegexpSingleline	Line has trailing spaces.	26
 Error	design	DesignForExtension	Clase 'UsuarioController' se parece a diseñada para la extensión (puede ser una subclase), pero el método 'findUsuarioByld' no tiene javadoc que explica cómo hacerlo de forma segura. Si la clase no está diseñado para la extensión de considerar la posibilidad de la clase 'UsuarioController' final o haciendo el método 'findUsuarioByld' anotación static/final/abstract/empty, o la adición permitido para el método.	27
 Error	javadoc	MissingJavadocMethod	Falta el comentario Javadoc.	27
 Error	misc	FinalParameters	El parámetro id debería ser final.	28
 Error	whitespace	WhitespaceAround	'{' no está precedido de espacio en blanco.	28
 Error	design	DesignForExtension	Clase 'UsuarioController' se parece a diseñada para la extensión (puede ser una subclase), pero el método 'findUsuarioByEmail' no tiene javadoc que explica cómo hacerlo de forma segura. Si la clase no está diseñado para la extensión de considerar la posibilidad de la clase 'UsuarioController' final o haciendo el método 'findUsuarioByEmail' anotación static/final/abstract/empty, o la adición permitido para el método.	36
 Error	javadoc	MissingJavadocMethod	Falta el comentario Javadoc.	36
 Error	sizes	LineLength	La línea es mayor de 80 caracteres (encontrado 85).	37
 Error	misc	FinalParameters	El parámetro email debería ser final.	37
 Error	whitespace	WhitespaceAround	'{' no está precedido de espacio en blanco.	37

# Pmd report

## Violations By Priority


### Priority 3

#### ar/com/poste\_alto/interfaces/UsuarioService.java



Rule	Violation	Line
UnnecessaryModifier 	Unnecessary modifier 'public' on method 'findById': the method is declared in an interface type	7
UnnecessaryModifier 	Unnecessary modifier 'public' on method 'findByEmail': the method is declared in an interface type	8
UnnecessaryModifier 	Unnecessary modifier 'public' on method 'crearUsuario': the method is declared in an interface type	9
UnnecessaryModifier 	Unnecessary modifier 'public' on method 'modificarUsuario': the method is declared in an interface type	10
UnnecessaryModifier 	Unnecessary modifier 'public' on method 'borrarUsuario': the method is declared in an interface type	11

### Priority 4

#### ar/com/poste\_alto/config/security/SecurityBeansInjector.java

Rule	Violation	Line
UnnecessaryImport 	Unused import 'org.hibernate.ObjectNotFoundException'	3

#### ar/com/poste\_alto/model/Usuario.java

Rule	Violation	Line
UnnecessaryImport 	Unused import 'jakarta.persistence.GeneratedValue'	7
UnnecessaryImport 	Unused import 'jakarta.persistence.GenerationType'	8