JUNIT and MOKITO

developers do the unit testing after writing the code not testers

1997 -junit 1

junit platform ,junit jupiter ,junit vintage these are the modules in junit 5

unit small peace of code it can be module ,function or a procedure

legacy projects use junit 4

In this package junit 5 is added auto matically :

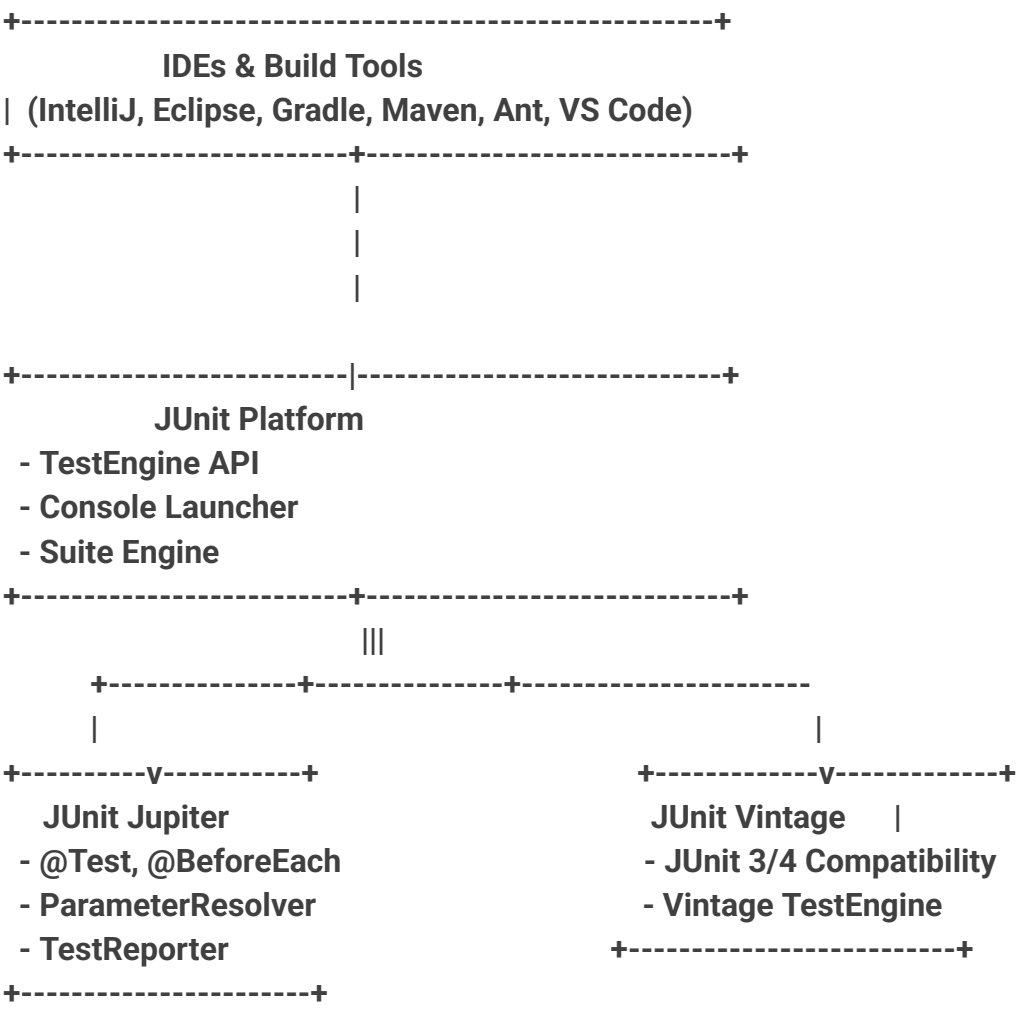
```
<dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
</dependency>
```

it is not needed in new projects unless a legacy project you want to work with

```
<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
</dependency>
```

→ in the image you can see the request hit the ui then it went to controller then service then repository then data base and come back in the same way as it goes

-> after every method test it first test repo
-> next service then
-> controller



✅ JUnit 5 Architecture: Platform, Jupiter, Vintage

JUnit 5 is built on a modular architecture, composed of three main parts:

---

1. JUnit Platform

🔧 The foundation layer.

- **What it does:**
  - **Launches test frameworks.**
  - **Provides APIs for test discovery, execution, and reporting.**

    - Used by IDEs (like IntelliJ, Eclipse), build tools (Maven, Gradle), and CI/CD pipelines.

- Example component:
  `org.junit.platform.launcher`

- Use case: You can even run non-JUnit tests (like Cucumber, TestNG) through the platform.

---

2. JUnit Jupiter

🚀 The test engine for writing and running JUnit 5 tests.

- What it provides:
    - New annotations like `@Test`, `@BeforeEach`, `@AfterEach`, `@BeforeAll`, etc.
    - The `Assertions` and `Assumptions` classes.
    - Extensions model for customizing behavior (`@ExtendWith`, `TestInstance`, etc.)

- Module:
  `org.junit.jupiter:junit-jupiter-engine`

- Use case: This is where your JUnit 5 tests live.

---

3. JUnit Vintage

🕰️ The bridge for running JUnit 3 and 4 tests in JUnit 5.

- What it does:
    - Allows backward compatibility for projects with older JUnit tests.
    - Supports annotations like `@Test`, `@Before`, `@After` from JUnit 4.

- Module:
  `org.junit.vintage:junit-vintage-engine`

- Use case: Lets you gradually migrate a legacy codebase to JUnit 5.

🛠️ Maven/Gradle Dependency Setup

Maven

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.10.0</version>
  <scope>test</scope>
</dependency>
```

To add support for old tests (Vintage):

```
<dependency>
  <groupId>org.junit.vintage</groupId>
  <artifactId>junit-vintage-engine</artifactId>
  <version>5.10.0</version>
  <scope>test</scope>
</dependency>
```

🚀 Deployment and Execution in Different Environments

1. IDE (IntelliJ, Eclipse, VS Code)

- The IDE uses the JUnit Platform to discover and run tests.

- It automatically picks the Jupiter engine for JUnit 5 tests.

- Backward compatibility with Vintage is also supported if configured.

2. Command Line / Build Tools

```
mvn test
```

- Uses the `maven-surefire-plugin` to run tests via the JUnit Platform.
  it

```
./gradlew test
```

- Uses `testImplementation 'org.junit.jupiter:junit-jupiter'`
- Supports JUnit Platform out of the box in recent versions.

3. CI/CD Pipelines (GitHub Actions, Jenkins, GitLab CI, etc.)

- Run tests as part of your build job.
- Example (GitHub Actions):

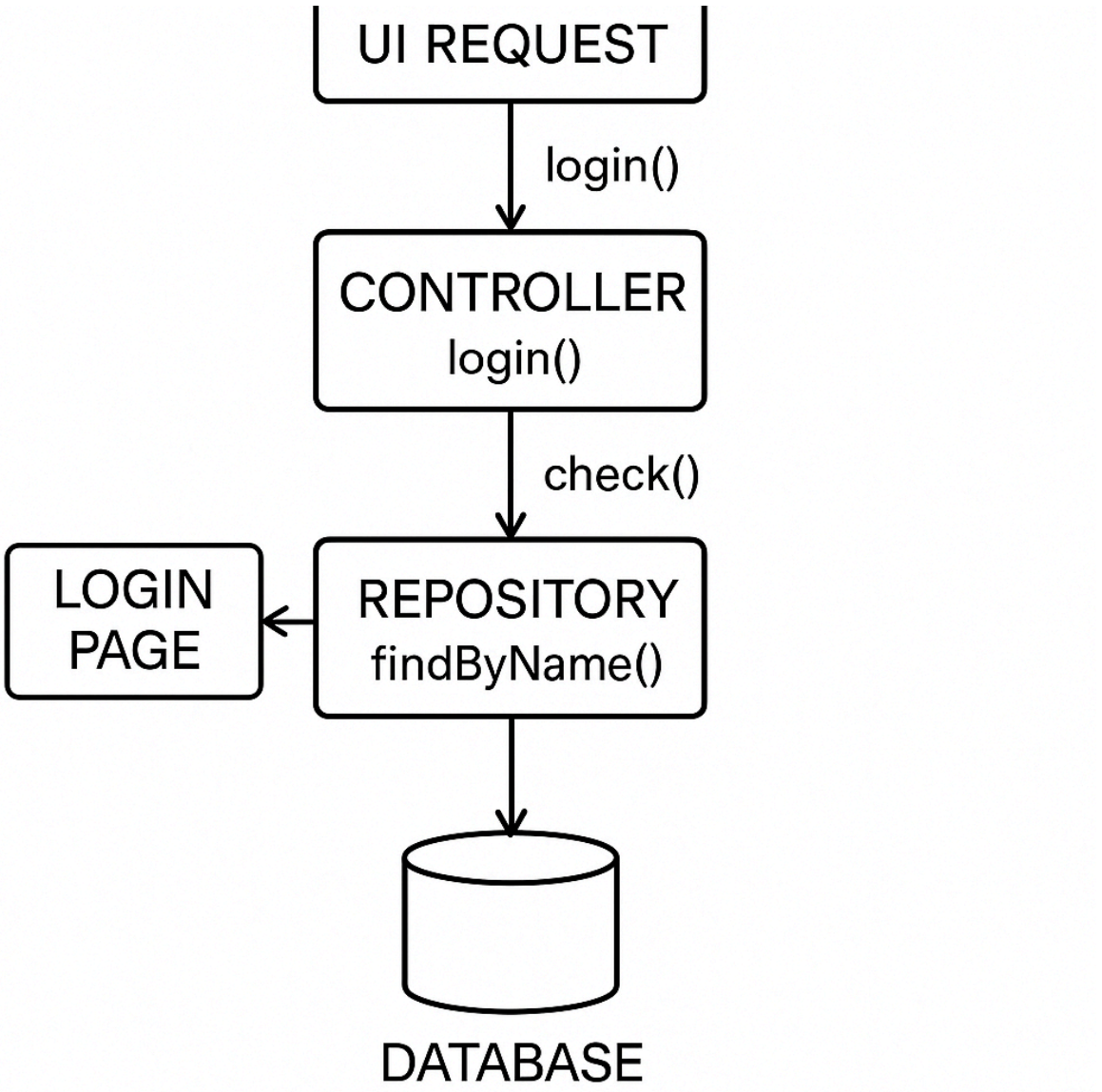
```
- name: Run Tests
  run: mvn test
```

- Results are reported using JUnit XML format, often processed by plugins like `junit.xml` readers.

💡 Summary Table

Component	Role	Typical Use
Platform	Test launcher & discovery	IDE, CLI, CI
Jupiter	JUnit 5 test engine	New tests
Vintage	Runs JUnit 3 & 4 tests	Legacy support

Would you like a working Maven or Gradle project structure to try out all three (Jupiter, Vintage, Platform) together?



📁 Project Structure (Maven or simple folder-based)

```
calculator_project/
│
├── src/
│   └── main/
│       └── java/
│           └── calculator/
│               └── Calculator.java
```

```
|
├── src/
|   └── test/
|       └── java/
|           └── calculator/
|               └── CalculatorTest.java
```

---

1. ✅ Calculator Code (`Calculator.java`)

```
// src/main/java/calculator/Calculator.java
package calculator;

public class Calculator {

    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    public double divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by
zero.");
        }
        return (double) a / b;
    }
}
```

---

2. 🧪 Unit Tests with JUnit (`CalculatorTest.java`)

```
package calculator;
```

```
import org.junit.jupiter.api.*;

import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {

    static Calculator calc;

// here it is static because you must use static for before all and
after all

    @BeforeAll
    public static void setUp() {
        System.out.println("Setting up Calculator instance...");
        calc = new Calculator();
    }

    @AfterAll
    public static void tearDown() {
        System.out.println("Tearing down Calculator instance...");
        calc = null; // Make eligible for garbage collection
    }

    @Test
    public void testAdd() {
        assertEquals(5, calc.add(2, 3));
    }

    @Test
    public void testSubtract() {
        assertEquals(6, calc.subtract(10, 4));
    }

    @Test
    public void testMultiply() {
        assertEquals(21, calc.multiply(3, 7));
    }

    @Test
    public void testDivide() {
```

```
        assertEquals(5.0, calc.divide(10, 2));
    }

    @Test
    public void testDivideByZero() {
        assertThrows(IllegalArgumentException.class, () ->
calc.divide(10, 0));
    }
}
```

✅ What does `assertEquals()` do?

In JUnit, `assertEquals(expected, actual)` checks that the `actual` result is equal to the `expected` result.

If they aren't equal, the test fails and shows what was expected vs what was returned.

```
assertEquals(4, 2 + 2);  // Passes
assertEquals(5, 2 + 2);  // Fails
```

🏃 How to Run the Tests

If you're using Maven:

```
mvn test
```

If you're using an IDE like IntelliJ or Eclipse:

- Right-click the test file or class and click Run.