Getting started with Testcontainers for Java

- Prerequisites
- Create the Java project
  - Set up the Maven project
  - Implement the business logic
- Write tests with Testcontainers
  - Add Testcontainers dependencies
  - Write the test
- Run tests and next steps
  - Run the tests
  - Summary
  - Further reading

In this guide, you will learn how to:

- Create a Java project with Maven
- Implement a `CustomerService` that manages customer records in PostgreSQL
- Write integration tests using Testcontainers with a real Postgres database
- Run the tests and verify everything works

Prerequisites

- Java 17+
- Maven or Gradle
- A Docker environment supported by Testcontainers

> [!NOTE] If you're new to Testcontainers, visit the Testcontainers overview to learn more about Testcontainers and the benefits of using it.

Create the Java project

Set up the Maven project

Create a Java project with Maven from your preferred IDE. This guide uses Maven, but you can use Gradle if you prefer. Add the following dependencies to `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.3</version>
    </dependency>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.5.6</version>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.2.5</version>
        </plugin>
```

```
    </plugins>
</build>
```

This adds the Postgres JDBC driver, logback for logging, JUnit 5 for testing, and the latest `maven-surefire-plugin` for JUnit 5 support.

Implement the business logic

Create a `Customer` record:

```
package com.testcontainers.demo;


public record Customer(Long id, String name) {}
```

Create a `DBConnectionProvider` class to hold JDBC connection parameters and provide a database `Connection`:

```
package com.testcontainers.demo;

import java.sql.Connection;
import java.sql.DriverManager;

class DBConnectionProvider {

  private final String url;
  private final String username;
  private final String password;

  public DBConnectionProvider(String url, String username, String password) {
    this.url = url;
    this.username = username;
    this.password = password;
  }

  Connection getConnection() {
    try {
      return DriverManager.getConnection(url, username, password);
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
```

```
  }
}
```

Create the `CustomerService` class:

```
package com.testcontainers.demo;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class CustomerService {

  private final DBConnectionProvider connectionProvider;

  public CustomerService(DBConnectionProvider connectionProvider) {
    this.connectionProvider = connectionProvider;
    createCustomersTableIfNotExists();
  }

  public void createCustomer(Customer customer) {
    try (Connection conn = this.connectionProvider.getConnection()) {
      PreparedStatement pstmt = conn.prepareStatement(
        "insert into customers(id,name) values(?,?)"
      );
      pstmt.setLong(1, customer.id());
      pstmt.setString(2, customer.name());
      pstmt.execute();
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
  }

  public List<Customer> getAllCustomers() {
    List<Customer> customers = new ArrayList<>();

    try (Connection conn = this.connectionProvider.getConnection()) {
```

```
      PreparedStatement pstmt = conn.prepareStatement(
        "select id,name from customers"
      );
      ResultSet rs = pstmt.executeQuery();
      while (rs.next()) {
        long id = rs.getLong("id");
        String name = rs.getString("name");
        customers.add(new Customer(id, name));
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return customers;
  }

  private void createCustomersTableIfNotExists() {
    try (Connection conn = this.connectionProvider.getConnection()) {
      PreparedStatement pstmt = conn.prepareStatement(
        """
        create table if not exists customers (
            id bigint not null,
            name varchar not null,
            primary key (id)
        )
        """
      );
      pstmt.execute();
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Here's what `CustomerService` does:

- The constructor calls `createCustomersTableIfNotExists()` to ensure the table exists.
- `createCustomer()` inserts a customer record into the database.
- `getAllCustomers()` fetches all rows from the `customers` table and returns a list of `Customer` objects.

Write tests with Testcontainers

You have the `CustomerService` implementation ready, but for testing you need a PostgreSQL database. You can use Testcontainers to spin up a Postgres database in a Docker container and run your tests against it.

Add Testcontainers dependencies

Add the Testcontainers PostgreSQL module as a test dependency in `pom.xml`:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers-postgresql</artifactId>
    <version>2.0.4</version>
    <scope>test</scope>
</dependency>
```

Since the application uses a Postgres database, the Testcontainers Postgres module provides a `PostgreSQLContainer` class for managing the container.

Write the test

Create `CustomerServiceTest.java` under `src/test/java`:

```
package com.testcontainers.demo;


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


import java.util.List;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.testcontainers.postgresql.PostgreSQLContainer;


class CustomerServiceTest {


  static PostgreSQLContainer postgres = new PostgreSQLContainer(
```

```
    "postgres:16-alpine"
  );

  CustomerService customerService;

  @BeforeAll
  static void beforeAll() {
    postgres.start();
  }

  @AfterAll
  static void afterAll() {
    postgres.stop();
  }

  @BeforeEach
  void setUp() {
    DBConnectionProvider connectionProvider = new DBConnectionProvider(
      postgres.getJdbcUrl(),
      postgres.getUsername(),
      postgres.getPassword()
    );
    customerService = new CustomerService(connectionProvider);
  }

  @Test
  void shouldGetCustomers() {
    customerService.createCustomer(new Customer(1L, "George"));
    customerService.createCustomer(new Customer(2L, "John"));

    List<Customer> customers = customerService.getAllCustomers();
    assertEquals(2, customers.size());
  }
}
```

Here's what the test does:

- Declares a `PostgreSQLContainer` with the `postgres:16-alpine` Docker image.
- The `@BeforeAll` callback starts the Postgres container before any test methods run.

- The `@BeforeEach` callback creates a `DBConnectionProvider` using the JDBC connection parameters from the container, then creates a `CustomerService`. The `CustomerService` constructor creates the `customers` table if it doesn't exist.
- `shouldGetCustomers()` inserts 2 customer records, fetches all customers, and asserts the count.
- The `@AfterAll` callback stops the container after all test methods finish.

Run tests and next steps

Run the tests

Run the tests using Maven:

```
$ mvn test
```

You can see in the logs that Testcontainers pulls the Postgres Docker image from Docker Hub (if not already available locally), starts the container, and runs the test.

Writing an integration test using Testcontainers works like writing a unit test that you can run from your IDE. Your teammates can clone the project and run tests without installing Postgres on their machines.

Summary

The Testcontainers for Java library helps you write integration tests using the same type of database (Postgres) that you use in production, instead of mocks. Because you aren't using mocks and instead talk to real services, you're free to refactor code and still verify that the application works as expected.

In addition to Postgres, Testcontainers provides dedicated modules for many SQL databases, NoSQL databases, messaging queues, and more. You can use Testcontainers to run any containerized dependency for your tests.

To learn more about Testcontainers, visit the Testcontainers overview.

Further reading

- Testcontainers container lifecycle management using JUnit 5
- Replace H2 with a real database for testing

- Getting started with Testcontainers in a Java Spring Boot project

---

□□□ #2
□ □□□ □□□ 2026-10-31 18:48:10 UTC
□ □□□ □□□ 2026-10-31 18:48:12 UTC