Replace H2 with a real database for testing

- Prerequisites
- The problem with H2 for testing
  - Example: PostgreSQL-specific syntax
  - The Spring Boot test using H2
- Replace H2 with the Testcontainers JDBC URL
  - How the special JDBC URL works
  - Initialize the database with a script
  - Testing JdbcTemplate-based repositories
- Use the JUnit 5 extension for more control
  - Summary
  - Further reading

In this guide, you will learn how to:

- Understand the drawbacks of using H2 in-memory databases for testing
- Replace H2 with a real PostgreSQL database using the Testcontainers special JDBC URL
- Use the Testcontainers JUnit 5 extension for more control over the container
- Test both Spring Data JPA and JdbcTemplate-based repositories

Prerequisites

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

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

The problem with H2 for testing

A common practice is to use lightweight databases like H2 or HSQL as in-memory databases for testing while using PostgreSQL, MySQL, or Oracle in production. This approach has significant drawbacks:

- The test database might not support all features of your production database.
- SQL syntax might not be compatible between H2 and your production database.
- Tests passing with H2 don't guarantee they'll work in production.

Example: PostgreSQL-specific syntax

Consider implementing an "upsert" — insert a product only if it doesn't already exist. In PostgreSQL, you can use:

```
INSERT INTO products(id, code, name) VALUES(?,?,?) ON CONFLICT DO NOTHING;
```

This query doesn't work with H2 by default:

```
Caused by: org.h2.jdbc.JdbcSQLException: Syntax error in SQL statement
"INSERT INTO products (id, code, name) VALUES (?, ?, ?) ON[*] CONFLICT DO NOTHING";
```

You can run H2 in PostgreSQL compatibility mode, but not all features are supported. The inverse is also true — H2 supports `ROWNUM()` which PostgreSQL doesn't.

Testing with a different database than production means you can't trust your test results and must verify after deployment, defeating the purpose of automated tests.

The Spring Boot test using H2

A typical H2-based test looks like this:

```
@DataJpaTest
class ProductRepositoryTest {
```

```
    @Autowired
    ProductRepository productRepository;

    @Test
    @Sql("classpath:/sql/seed-data.sql")
    void shouldGetAllProducts() {
        List<Product> products = productRepository.findAll();
        assertEquals(2, products.size());
    }
}
```

Spring Boot uses H2 automatically when it's on the classpath. The test passes, but it doesn't catch PostgreSQL-specific issues.

Replace H2 with the Testcontainers JDBC URL

Replacing H2 with a real PostgreSQL database requires two test properties:

```
@DataJpaTest
@TestPropertySource(properties = {
  "spring.test.database.replace=none",
  "spring.datasource.url=jdbc:tc:postgresql:16-alpine:///db"
})
class ProductRepositoryWithJdbcUrlTest {

  @Autowired
  ProductRepository productRepository;

  @Test
  @Sql("classpath:/sql/seed-data.sql")
  void shouldGetAllProducts() {
    List<Product> products = productRepository.findAll();
    assertEquals(2, products.size());
  }
}
```

That's it — two properties and your tests run against a real PostgreSQL database.

How the special JDBC URL works

A standard PostgreSQL JDBC URL looks like:

```
jdbc:postgresql://localhost:5432/postgres
```

The Testcontainers special JDBC URL inserts `tc:` after `jdbc:`:

```
jdbc:tc:postgresql:///db
```

The hostname, port, and database name are ignored — Testcontainers manages them automatically. You can specify the Docker image tag after the database name:

```
jdbc:tc:postgresql:16-alpine:///db
```

This creates a container from the `postgres:16-alpine` image.

Initialize the database with a script

Pass `TC_INITSCRIPT` to run an SQL script when the container starts:

```
jdbc:tc:postgresql:16-alpine:///db?TC_INITSCRIPT=sql/init-db.sql
```

Testcontainers runs the script automatically. For production applications, use a database migration tool like Flyway or Liquibase instead.

The special JDBC URL also works for MySQL, MariaDB, PostGIS, YugabyteDB, CockroachDB, and other databases with Testcontainers JDBC support.

Testing JdbcTemplate-based repositories

The same approach works for `JdbcTemplate`-based repositories. Use `@JdbcTest` instead of `@DataJpaTest`:

```
@JdbcTest
@TestPropertySource(properties = {
  "spring.test.database.replace=none",
  "spring.datasource.url=jdbc:tc:postgresql:16-alpine:///db?TC_INITSCRIPT=sql/init-db.sql"
```

```
})
class JdbcProductRepositoryTest {

  @Autowired
  private JdbcTemplate jdbcTemplate;

  private JdbcProductRepository productRepository;

  @BeforeEach
  void setUp() {
    productRepository = new JdbcProductRepository(jdbcTemplate);
  }

  @Test
  @Sql("/sql/seed-data.sql")
  void shouldGetAllProducts() {
    List<Product> products = productRepository.getAllProducts();
    assertEquals(2, products.size());
  }
}
```

Use the JUnit 5 extension for more control

If the special JDBC URL doesn't meet your needs, or you need more control over container creation (for example, to copy initialization scripts), use the Testcontainers JUnit 5 extension:

```
@DataJpaTest
@TestPropertySource(properties = {
    "spring.test.database.replace=none"
})
@Testcontainers
class ProductRepositoryTest {

  @Container
  static PostgreSQLContainer postgres =
    new PostgreSQLContainer("postgres:16-alpine")
```

```
      .withCopyFileToContainer(
        MountableFile.forClasspathResource("sql/init-db.sql"),
        "/docker-entrypoint-initdb.d/init-db.sql");

  @DynamicPropertySource
  static void configureProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url", postgres::getJdbcUrl);
    registry.add("spring.datasource.username", postgres::getUsername);
    registry.add("spring.datasource.password", postgres::getPassword);
  }

  @Autowired
  ProductRepository productRepository;

  @Test
  @Sql("/sql/seed-data.sql")
  void shouldGetAllProducts() {
    List<Product> products = productRepository.findAll();
    assertEquals(2, products.size());
  }

  @Test
  @Sql("/sql/seed-data.sql")
  void shouldNotCreateAProductWithDuplicateCode() {
    Product product = new Product(3L, "p101", "Test Product");
    productRepository.createProductIfNotExists(product);
    Optional<Product> optionalProduct = productRepository.findById(
      product.getId()
    );
    assertThat(optionalProduct).isEmpty();
  }
}
```

This approach:

- Uses `@Testcontainers` and `@Container` to manage the container lifecycle.
- Copies `init-db.sql` into the container's init directory so PostgreSQL runs it at startup.
- Uses `@DynamicPropertySource` to register the container's connection details with Spring Boot.
- Tests PostgreSQL-specific features like `ON CONFLICT DO NOTHING` that wouldn't work with H2.

Summary

- Use the **special JDBC URL** (`jdbc:tc:postgresql:...`) for the quickest way to switch from H2 to a real database — it's a one-property change.
- Use the **JUnit 5 extension** when you need more control over the container (custom init scripts, environment variables, etc.).
- Both approaches work with Spring Data JPA (`@DataJpaTest`) and JdbcTemplate (`@JdbcTest`) tests.

Further reading

- Testcontainers Postgres module
- Testcontainers JDBC support
- Testing a Spring Boot REST API with Testcontainers

□□□ #2
□ □□□ □□□ 2026-10-31 18:48:17 UTC
□ □□□ □□□ 2026-10-31 18:48:19 UTC