How `PostgreSQLContainer` Works in TestContainers

`PostgreSQLContainer` from **TestContainers** allows us to spin up a **real PostgreSQL database in a Docker container** during tests. This helps simulate real-world database interactions without needing a separate local or shared test database.

How It Works (Step-by-Step)

1. Define and Start the Container

java
CopyEdit
```
static PostgreSQLContainer<?> postgres = new
PostgreSQLContainer<>("postgres:13")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

@BeforeAll
static void startContainer() {
    postgres.start(); // Start PostgreSQL container
    System.setProperty("spring.datasource.url",
postgres.getJdbcUrl());
    System.setProperty("spring.datasource.username",
postgres.getUsername());
    System.setProperty("spring.datasource.password",
postgres.getPassword());
}
```

- **Creates a new PostgreSQL container** (using `postgres:13` image).
- **Configures database credentials** (`testdb`, `test`, `test`).
- **Starts the container** before running tests.
- **Overrides Spring Boot's `application.properties` dynamically** using `System.setProperty(...)`.
    - This ensures that the test environment uses this container instead of a real DB.

---

2. Run Tests with an Isolated Database

Since TestContainers runs **each test in a fresh container**, all database operations are **isolated**.
Example:

java
CopyEdit
```
@Test
void testSaveAndFindTransaction() {
    PaymentTransaction transaction = new
PaymentTransaction("txn123", "sender123", "receiver123", 50.0,
"PENDING");
    transactionRepository.save(transaction);

    Optional<PaymentTransaction> found =
transactionRepository.findById("txn123");
    assertTrue(found.isPresent());
    assertEquals("sender123", found.get().getSender());
}
```

- Saves a transaction **inside the container database**.
- Queries the same database to **verify persistence**.
- The container **automatically shuts down after tests**, ensuring a **clean database state** for the next run.

Advantages of Using `PostgreSQLContainer`

✅ **No need for a separate test DB** (eliminates DB cleanup issues).
✅ **Simulates real PostgreSQL behavior** (indexes, constraints, transactions).
✅ **Fully isolated per test run** (avoids test contamination).
✅ **Works seamlessly with CI/CD pipelines** (GitHub Actions, Jenkins, etc.).

Would you like a **Kafka TestContainer setup** as well? 🚀

You said:
any other way to test the jparepository

ChatGPT said:

Yes! Apart from using **TestContainers**, there are other ways to test `JpaRepository` effectively in a Spring Boot application. Here are some of the best approaches:

1️⃣ Using H2 In-Memory Database (Fast & Lightweight)

H2 is an in-memory database that mimics PostgreSQL but runs entirely in memory.

How to Use It?

Add H2 dependency to `pom.xml`:

xml
CopyEdit
```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

Then, configure `application-test.properties`:

ini
CopyEdit
```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
```

Example Test
java
CopyEdit
```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@TestPropertySource(locations =
"classpath:application-test.properties")
class TransactionRepositoryTest {
    @Autowired
    private TransactionRepository transactionRepository;

    @Test
    void testSaveAndFindTransaction() {
        PaymentTransaction transaction = new
PaymentTransaction("txn123", "sender123", "receiver123", 50.0,
"PENDING");
        transactionRepository.save(transaction);
```

```
        Optional<PaymentTransaction> found =
transactionRepository.findById("txn123");
        assertTrue(found.isPresent());
        assertEquals("sender123", found.get().getSender());
    }
}
```

✅ **Pros:** Fast, lightweight, easy to set up
❌ **Cons:** H2 doesn't fully replicate PostgreSQL behavior

2️⃣ Using `@DataJpaTest` (Efficient Unit Testing)

`@DataJpaTest` loads only JPA-related components, making it **faster** than a full Spring Boot test.

Example Test

java
CopyEdit

```
@DataJpaTest
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE) // Use real DB if needed
class TransactionRepositoryTest {
    @Autowired
    private TransactionRepository transactionRepository;

    @Test
    void testSaveAndFindTransaction() {
        PaymentTransaction transaction = new
PaymentTransaction("txn123", "sender123", "receiver123", 50.0,
"PENDING");
        transactionRepository.save(transaction);

        Optional<PaymentTransaction> found =
transactionRepository.findById("txn123");
        assertTrue(found.isPresent());
        assertEquals("sender123", found.get().getSender());
    }
}
```

✅ **Pros:** Minimal setup, faster than full Spring Boot tests
❌ **Cons:** No external database (unless configured with TestContainers)

3️⃣ Using `@MockBean` with Mockito (Isolated Testing)

If you want **pure unit tests without a database**, use **Mockito** to mock `JpaRepository`.

Example Test
java
CopyEdit
```
@ExtendWith(MockitoExtension.class)
class TransactionRepositoryMockTest {
    @Mock
    private TransactionRepository transactionRepository;

    @InjectMocks
    private PaymentService paymentService;

    @Test
    void testSaveTransactionWithMock() {
        PaymentTransaction transaction = new
PaymentTransaction("txn123", "sender123", "receiver123", 50.0,
"PENDING");


when(transactionRepository.save(any(PaymentTransaction.class))).then
Return(transaction);

        String transactionId = paymentService.processPayment(new
PaymentRequest("receiver123", 50.0), "sender123");

        assertNotNull(transactionId);
        verify(transactionRepository,
times(1)).save(any(PaymentTransaction.class));
    }
}
```

✅ **Pros:** No database needed, fast execution
❌ **Cons:** Doesn't test actual database behavior

Which One Should You Use?

Approach	Pros	Cons	Best For
TestContainers (PostgreSQLContainer)	Real DB behavior	Slower startup	Integration tests
H2 In-Memory Database	Fast, lightweight	Doesn't fully match PostgreSQL	Quick repository testing
@DataJpaTest	Minimal setup, isolated	No external DB support (unless configured)	Focused JPA unit tests
Mockito (@MockBean)	No DB needed, fast	Doesn't test real DB queries	Isolated service layer tests

Implementation Breakdown: Real Time Payment system

1. `PaymentController` → Handles API requests (`/payments`)
2. `PaymentService` → Processes transactions, saves to DB, and sends to Kafka
3. `PaymentTransaction` → Entity for transaction storage (PostgreSQL)
4. `KafkaTemplate` → Sends transactions to Kafka for async processing
5. `TransactionRepository` → JPA repository for database operations
6. **OAuth2/JWT Security** → Authenticates users using JWT tokens.
7. **Kafka Consumer (`PaymentConsumer`)** → Listens to Kafka topic and updates transaction status.
8. **Resilience4J Retry Mechanism** → Retries payments on failures with a fallback method.
9. **API Rate Limiting (Bucket4j)** → Limits requests to **10 per minute** per user.
10. **Graceful Response on Rate Limit** → Returns **HTTP 429 (Too Many Requests)** if exceeded.
11. integration tests (Spring Boot Test + TestContainers)
12. **JUnit & Mockito Tests**:

    PaymentServiceTest: Verifies that transactions are saved and published to Kafka.

    TransactionRepositoryTest: Tests database save and retrieval operations.

```
package com.bank.payment;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```java
import org.springframework.web.bind.annotation.*;
import org.springframework.http.ResponseEntity;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import
org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationToken;
import io.github.resilience4j.retry.annotation.Retry;
import io.github.bucket4j.Bucket;
import io.github.bucket4j.Bandwidth;
import io.github.bucket4j.Refill;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.testcontainers.junit.jupiter.Testcontainers;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.utility.DockerImageName;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mockito;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;
import javax.persistence.*;
import java.time.Duration;
import java.util.Optional;
import java.util.UUID;

@SpringBootApplication
public class PaymentApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentApplication.class, args);
    }
}

@RestController
@RequestMapping("/payments")
class PaymentController {
    @Autowired
    private PaymentService paymentService;
    private final Bucket bucket;

    public PaymentController() {
        Bandwidth limit = Bandwidth.classic(10, Refill.greedy(10, Duration.ofMinutes(1)));
        this.bucket = Bucket.builder().addLimit(limit).build();
```

```
    }

    @PostMapping
    public ResponseEntity<String> processPayment(@RequestBody PaymentRequest
request, JwtAuthenticationToken auth) {
        if (bucket.tryConsume(1)) {
            String userId = auth.getName();
            String transactionId = paymentService.processPayment(request, userId);
            return ResponseEntity.ok("Transaction initiated with ID: " + transactionId);
        } else {
            return ResponseEntity.status(429).body("Too many requests. Please try again
later.");
        }
    }
}

class PaymentRequest {
    public String receiver;
    public double amount;
}

@Service
class PaymentService {
    @Autowired
    private TransactionRepository transactionRepository;
    @Autowired
    private KafkaTemplate<String, PaymentTransaction> kafkaTemplate;

    @Transactional
    @Retry(name = "paymentService", fallbackMethod = "fallbackProcessPayment")
    public String processPayment(PaymentRequest request, String sender) {
        String transactionId = UUID.randomUUID().toString();
        PaymentTransaction transaction = new PaymentTransaction(transactionId, sender,
request.receiver, request.amount, "PENDING");
        transactionRepository.save(transaction);
        kafkaTemplate.send("payments", transaction);
        return transactionId;
    }

    public String fallbackProcessPayment(PaymentRequest request, String sender,
Throwable t) {
        return "Transaction failed. Please try again later.";
    }
}

@Entity
@Table(name = "transactions")
class PaymentTransaction {
```

```java
    @Id
    private String transactionId;
    private String sender;
    private String receiver;
    private double amount;
    private String status;

    public PaymentTransaction() {}

    public PaymentTransaction(String transactionId, String sender, String receiver, double
amount, String status) {
        this.transactionId = transactionId;
        this.sender = sender;
        this.receiver = receiver;
        this.amount = amount;
        this.status = status;
    }
}

interface TransactionRepository extends JpaRepository<PaymentTransaction, String> {}

@Service
class PaymentConsumer {
    @Autowired
    private TransactionRepository transactionRepository;

    @KafkaListener(topics = "payments", groupId = "payment_group")
    public void processTransaction(PaymentTransaction transaction) {
        transaction.setStatus("COMPLETED");
        transactionRepository.save(transaction);
        System.out.println("Transaction processed: " + transaction.getTransactionId());
    }
}

@ExtendWith(SpringExtension.class)
@SpringBootTest
class PaymentServiceTest {
    @MockBean
    private TransactionRepository transactionRepository;
    @MockBean
    private KafkaTemplate<String, PaymentTransaction> kafkaTemplate;

    @Autowired
    private PaymentService paymentService;

    @Test
    void testProcessPayment() {
        PaymentRequest request = new PaymentRequest();
```

```
        request.receiver = "receiver123";
        request.amount = 100.0;

        String sender = "sender123";
        String transactionId = paymentService.processPayment(request, sender);

        assertNotNull(transactionId);
        verify(transactionRepository, times(1)).save(any(PaymentTransaction.class));
        verify(kafkaTemplate, times(1)).send(eq("payments"), any(PaymentTransaction.class));
    }
}

@ExtendWith(SpringExtension.class)
@SpringBootTest
@Testcontainers
class TransactionRepositoryIntegrationTest {
    static PostgreSQLContainer<?> postgres = new
PostgreSQLContainer<>(DockerImageName.parse("postgres:13"))
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");

    @BeforeAll
    static void startContainer() {
        postgres.start();
        System.setProperty("spring.datasource.url", postgres.getJdbcUrl());
        System.setProperty("spring.datasource.username", postgres.getUsername());
        System.setProperty("spring.datasource.password", postgres.getPassword());
    }

    @Autowired
    private TransactionRepository transactionRepository;

    @Test
    void testSaveAndFindTransaction() {
        PaymentTransaction transaction = new PaymentTransaction("txn123", "sender123",
"receiver123", 50.0, "PENDING");
        transactionRepository.save(transaction);

        Optional<PaymentTransaction> found = transactionRepository.findById("txn123");
        assertTrue(found.isPresent());
        assertEquals("sender12", found.get().getSender());
    }
}
p
```

. Design a Real-Time Payment Processing System

💡 *Scenario:* Build a high-throughput, low-latency system to process transactions in real time.

Key Discussion Points:

✅ **System Components:**

- API Gateway (Spring Cloud Gateway)
- Transaction Processing Service (Spring Boot, gRPC for low latency)
- Event-Driven Architecture (Kafka for async processing)
- ACID-compliant Database (PostgreSQL/MySQL with read replicas)
- Fraud Detection Module (Kafka Streams + ML models)
- Notifications (WebSockets, Webhooks, SMS/Email)

✅ **Concurrency & Consistency**

- Use **Optimistic Locking** or **Pessimistic Locking** for database updates
- Implement **idempotency keys** for retrying failed transactions

✅ **Scalability & Fault Tolerance**

- Use **Kafka with partitions** for event-driven transaction processing
- Implement **circuit breakers** (Resilience4J) to handle failures
- Deploy with **Kubernetes (K8s) + Auto-scaling**

✅ **Security & Compliance**

- **OAuth2 + JWT** for authentication
- **TLS & AES Encryption** for securing transactions
- **PCI-DSS Compliance** for storing sensitive data

---

2. Design a High-Frequency Trading (HFT) Platform

💡 *Scenario:* Design a system that can process millions of stock trades per second.

Key Discussion Points:

✅ **Low Latency Considerations:**

- Use **Netty or gRPC** for ultra-fast communication
- Leverage **Zero-Copy Mechanisms (e.g., Java NIO, memory-mapped files)**
- Use **LMAX Disruptor** for in-memory event processing

✅ **Market Data Processing**

- Subscribe to **Kafka streams** for real-time stock market updates
- Maintain **Order Books** in **Redis or Aerospike** for fast access

✅ **Risk Management & Compliance**

- Implement **circuit breakers** to throttle high-volume trades
- Use **pre-trade risk checks** to avoid violations

---

3. Design a Fraud Detection System for Banking Transactions

💡 *Scenario:* Design a system to detect fraudulent transactions in real time.

Key Discussion Points:

✅ **Real-Time Fraud Detection Pipeline:**

- **API Gateway** → **Kafka** → **Stream Processing (Flink/Spark)** → **ML Model** → **Alerts**

✅ **ML-Based Anomaly Detection:**

- Train fraud detection models (XGBoost, Random Forest)
- Deploy **model inference using TensorFlow Serving**

✅ **Data Storage & Retrieval:**

- Store historical transactions in **Cassandra or BigQuery**
- Maintain real-time session data in **Redis**

✅ **High Availability & Scalability:**

- Use **Kafka partitions** for scaling fraud detection
- Implement **fallback mechanisms** for delayed ML model responses

---

4. Design a Scalable Banking Ledger System

💡 *Scenario:* Build a ledger system that handles millions of transactions.

Key Discussion Points:

✅ **Data Model**

- **Event-Sourcing Architecture** (store immutable transactions)

- Use **CQRS (Command Query Responsibility Segregation)**

✅ **Storage & Consistency**

- ACID-compliant DB for critical data (PostgreSQL)
- Append-only **Kafka topics** for audit logs

✅ **Concurrency Handling**

- **Optimistic concurrency control**
- **Idempotency tokens** for retry handling

I'll create **Java classes** for a **Real-Time Payment Processing System** using **Spring Boot, Kafka, and PostgreSQL**. The system will include:

1. **Transaction API** (handles payment requests)
2. **Transaction Service** (business logic, validation)
3. **Kafka Producer & Consumer** (for async processing)
4. **Database Entity** (JPA for PostgreSQL)
5. **Security & Idempotency Handling**

Here's the complete **Java backend implementation**:

Real Time Payment
Edited