Testing Microservices with Testcontainers

Spring Boot integration tests that rock!

Jonathan Manera · Follow

7 min read · Mar 23

Listen Share

Photo by frank mckenna on Unsplash

When writing integration tests for Spring Boot applications, we usually need to access external resources such as databases, message brokers, webservices, etc. **One of the**

goals of integration testing should be to verify precisely how the different parts of an application behave in combination with these external resources.

We will explore in this article how a library like Testcontainers can help us to achieve a better integration testing design.

How Testcontainers Works?

This library provides lightweight containerized instances of resources, like databases, messages brokers, and web servers. **In order to use it, it requires a Docker environment up and running on your machine.**

Testcontainers is supported by *Junit4*, *Junit5* and *Spock*. However, **in the following examples we will use *Junit5* as the testing framework and Java 17.**

The SUT

To show how to use Testcontainers, let me propose **the following System Under Test (SUT), which is intended to be a simplistic approximation of what a real-life microservice would be.**

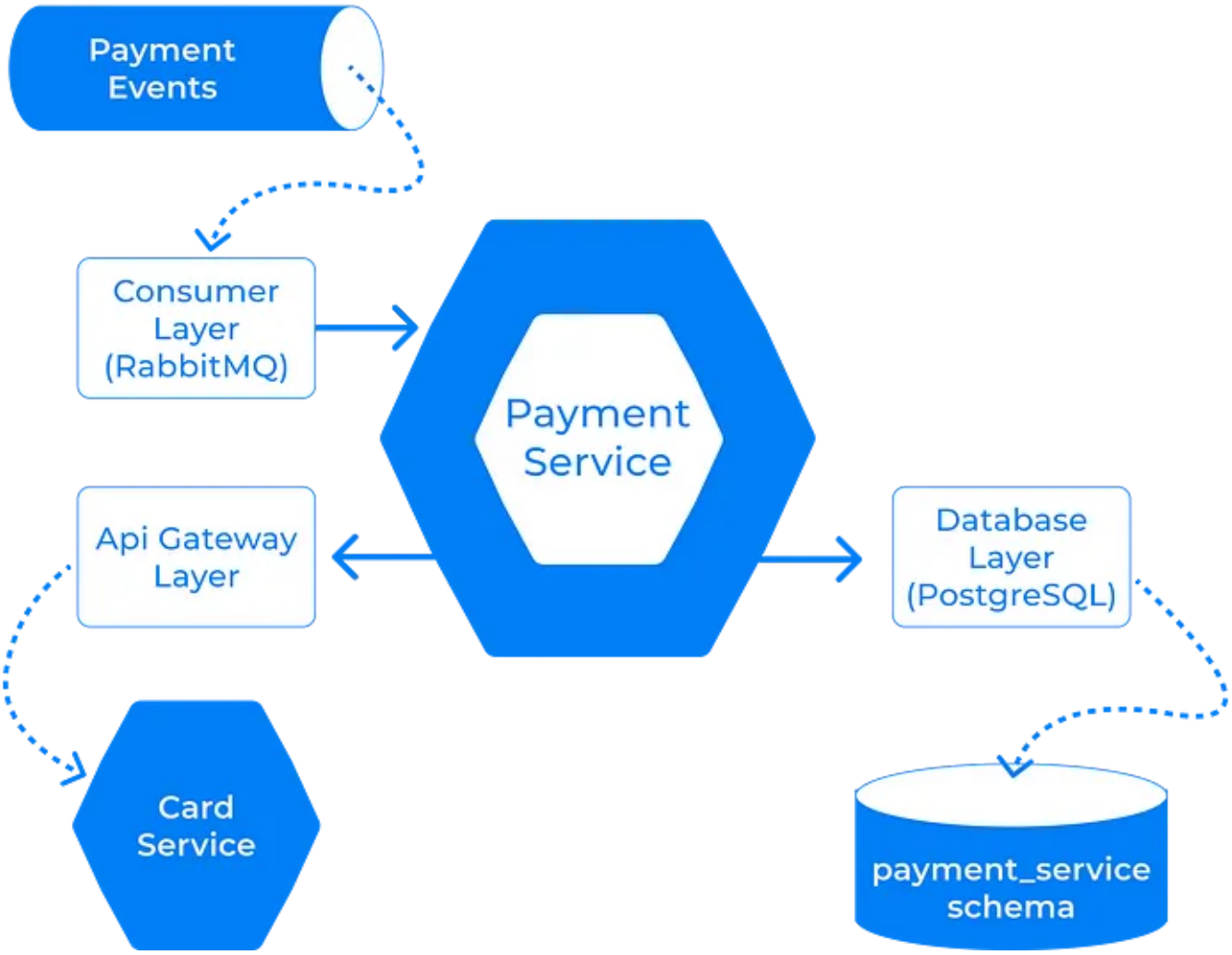


Image by Author

Here, the *Payment Service* consumes *Payment Events* from a RabbitMQ queue, stores the transactional data on a PostgreSQL database, and calls the *Card Service* to verify if the *Card Details* provided are valid.

The Core Business

The SUT described above includes the following **Core Business** classes:

- *Payment Method*:

```
public enum PaymentMethod {
    CARD, CASH
```

```
}
```

- *Payment Status*:

```
public enum PaymentStatus {
    PENDING_VALIDATION, OK, ERROR
}
```

- *Payment*:

```
public record Payment(BigDecimal amount,
                      PaymentMethod paymentMethod,
                      @Nullable CardDetails card) {

    public Payment(BigDecimal amount, PaymentMethod paymentMethod) {
        this(amount, paymentMethod, null);
    }
}
```

- *Card Details*:

```
public record CardDetails(String number, int expDate, int cvc) {
}
```

- *Payment Service*

```
@Service
public class PaymentService {

    private final PaymentDao paymentDao;
    private final CardServiceProxy cardService;
```

```
    public PaymentServiceImpl(PaymentDao paymentDao, CardServiceProxy cardService)
        this.paymentDao = paymentDao;
        this.cardService = cardService;
    }

    @Transactional
    public void registerPayment(Payment payment) {
        if (Objects.equals(payment.paymentMethod(), PaymentMethod.CARD)) {
            UUID id = paymentDao.create(payment, PaymentStatus.PENDING_VALIDATION)
            boolean isValidCard = cardService.validateCard(payment.card());
            paymentDao.updateStatus(id,
                 isValidCard ? PaymentStatus.OK : PaymentStatus.ERROR);
        } else {
            paymentDao.create(payment, PaymentStatus.OK);
        }
    }
}
```

Here, **Business Logic** is encapsulated in the `registerPayment()` method:

- **If the *Payment Method* is *CASH*,** it creates a *Payment* with a Status set to `OK`
- **If the *Payment Method* is *CARD*,** it creates a *Payment* with a status set to `PENDING_VALIDATION`, validates the *Card Details* through the `CardServiceProxy` class, and updates the *Payment* to status `OK` or `ERROR`, depending on whether the card is valid or not.

Setting Up the Project

Testcontainers is distributed along separate dependencies.

In this tutorial, we will use Gradle as the build tool. So, let's add the following test dependencies to the `build.gradle` file:

```
testImplementation "org.testcontainers:testcontainers:${testcontainersVersion}"
testImplementation "org.testcontainers:junit-jupiter:${testcontainersVersion}"
```

```
testImplementation "org.testcontainers:postgresql:${testcontainersVersion}"
testImplementation "org.testcontainers:rabbitmq:${testcontainersVersion}"
```

Let's also configure the `application.yml` file with the following properties:

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
  datasource:
    url: jdbc:postgresql://localhost:5432/mydb?currentSchema=payment_service
    username: username
    password: password
    driver-class-name: org.postgresql.Driver
ws:
  card:
    base-url: http://localhost:8181
    validate-uri: v1/cards/validate
```

Integration Testing with PostgreSQL Containers

The Database Layer

- *Payment JPA Entity*:

```
@Entity
@Table(schema = "payment_service", name = "payment")
@Access(AccessType.FIELD)
public class PaymentJpaEntity {

    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(name = "UUID", strategy = "org.hibernate.id.UUIDGenerator")
    private UUID id;
```

```
    private BigDecimal amount;

    @Enumerated(EnumType.STRING)
    private PaymentMethod paymentMethod;

    @Enumerated(EnumType.STRING)
    private PaymentStatus paymentStatus;

    @CreationTimestamp
    private Instant paymentDate;

    // methods removed for simplicity
}
```


- *Payment JPA Repository*:

```
public interface PaymentJpaRepository
        extends JpaRepository<PaymentJpaEntity, UUID> {
}
```

- *Payment DAO* (Data Access Object):

```
@Component
public class PaymentDao {

    private static final Logger log = LoggerFactory.getLogger(PaymentDao.class);

    private final PaymentJpaRepository jpaRepository;

    public PaymentDao(PaymentJpaRepository jpaRepository) {
        this.jpaRepository = jpaRepository;
    }

    public UUID create(Payment payment, PaymentStatus status) {
        PaymentJpaEntity paymentCreated = jpaRepository.save(
                new PaymentJpaEntity(
                        payment.amount(),
                        payment.paymentMethod(),
                        status
```

```
                )
        );
        log.debug("Payment created: {}", paymentCreated);
        return paymentCreated.getId();
    }

    public boolean updateStatus(UUID paymentId, PaymentStatus status) {
        AtomicBoolean updated = new AtomicBoolean(false);
        jpaRepository.findById(paymentId)
                .ifPresent(payment -> {
                    payment.setPaymentStatus(status);
                    jpaRepository.save(payment);
                    log.debug("Payment updated: {}", payment);
                    updated.set(true);
                });
        return updated.get();
    }
}
```

Creating the Integration Test

First, let's create a Java Interface with a *container* to reuse with the different test classes.

```
@Testcontainers
public interface PostgresTestContainer {

    String DOCKER_IMAGE_NAME = "postgres:15";

    @Container
    PostgreSQLContainer<?> container =
            new PostgreSQLContainer<>(DOCKER_IMAGE_NAME);

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

Here:

- The `@Testcontainers` annotation is required to use the Testcontainers extension for JUnit.
- The `@Container` annotation is required for the instance of the container.
- The `PostgreSQLContainer` is the specific container class that we are using for PostgreSQL. This class receives the docker image name as an argument in the constructor, and creates an instance with a dynamic url, username and password.
- The `@DynamicPropertySource` annotation allows to add properties with the dynamic values provided by Testcontainer. By declaring an instance of `DynamicPropertyRegistry` as a method argument, you can use the `add()` method to replace the `spring.datasource.*` properties with the following values:
  - `container.getJdbcUrl()` (gets the JDBC URL to connect to)
  - `container.getUsername()` (gets the database username)
  - `container.getPassword()` (gets the database password)

Next, to test the `PaymentDao` class, let's implement the `PostgresTestContainer` interface in the test class.

```
@DataJpaTest
@ComponentScan(basePackages = {"ports.output.jpa"})
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class PaymentDaoIntegrationTest implements PostgresTestContainer {

    static UUID paymentId;

    @Autowired
    PaymentDao dao;

    @Test
    @Order(1)
    @Commit
    void create_test() {
        paymentId = dao.create(
                new Payment(BigDecimal.TEN, PaymentMethod.CARD),
                PaymentStatus.PENDING_VALIDATION
        );
        assertThat(paymentId).isNotNull();
```

```
    }

    @Test
    @Order(2)
    void updateStatus_test_withExistingPaymentId() {
        assertThat(dao.updateStatus(paymentId, PaymentStatus.OK))
                .isTrue();
    }

    @Test
    @Order(3)
    void updateStatus_test_withNonExistingPaymentId() {
        UUID randomPaymentId = UUID.randomUUID();
        assertThat(dao.updateStatus(randomPaymentId, PaymentStatus.OK))
                .isFalse();
    }
}
```

Here:

- The `@DataJpaTest` annotation applies only configurations relevant to JPA tests.
- The `@ComponentScan` annotation allows us to define the package(s) that Spring should scan and that are relevant to JPA tests.
- The `@AutoConfigureTestDatabase` annotation with the `replace` field set to `NONE` will prevent the auto-configuration of any Datasource other than the application's default.
- The `@TestMethodOrder` annotation facilitates running the tests in a specific order, using the `@Order(1..N)` annotation on each test method.

Note that, by default, each test will *roll-back* the transaction once it has finished, to persist the data use the `@Commit` annotation on the test.

Integration Testing with Generic Containers

The API Gateway Layer

- The *Card Proxy Properties* class:

```
@ConfigurationProperties(prefix = "ws.card")
public record CardProxyProperties(String baseUrl, String validateUri) {

    public String validateUri() {
        return String.format("%s/%s", this.baseUrl, this.validateUri);
    }
}
```

- The *Card Validation Request:*

```
public record CardValidationRequest(String number, int expDate, int cvc) {
}
```

- The *Card Validation Response*:

```
public record CardValidationResponse(boolean isValid) {
}
```

- The *Card Service Proxy* class:

```
@Component
@EnableConfigurationProperties(CardProxyProperties.class)
public class CardServiceProxy implements CardRepository {

    private static final Logger log = LoggerFactory.getLogger(CardServiceProxy.cl

    private final WebClient client;
    private final CardProxyProperties properties;

    public CardServiceProxy(WebClient client, CardProxyProperties properties) {
        this.client = client;
        this.properties = properties;
    }
```

```
    @Override
    public boolean validateCard(CardDetails card) {
        try {
            CardValidationRequest request = new CardValidationRequest(
                    card.number(),
                    card.expDate(),
                    card.cvc()
            );

            return client.post()
                    .uri(properties.validateUri())
                    .body(BodyInserters.fromValue(request))
                    .retrieve()
                    .toEntity(CardValidationResponse.class)
                    .map(response -> response.getBody().isValid())
                    .block();

        } catch (Exception e) {
            log.warn("There was an error calling the card service");
            log.error(e.getMessage(), e);
        }
        return false;
    }
}
```

Creating the Integration Test

One of the many advantages of using Testcontainers is that you can stub APIs and create Docker images for testing purposes. For further information about API Stubbing, see the tutorial I wrote about it.

In the example below, in order to create a `GenericContainer` class, we use an image uploaded on my DockerHub with the Card Service Mocks.

```
@Testcontainers
public interface CardServiceTestContainer {

    String DOCKER_IMAGE_NAME = "manerajona/card-service";
    int PORT = 8080;

    @Container
    GenericContainer<?> container =
            new GenericContainer<>(DOCKER_IMAGE_NAME)
```

```
                    .withExposedPorts(PORT);

    @DynamicPropertySource
    static void registerProperties(DynamicPropertyRegistry registry) {
        registry.add("ws.card.base-url",
                () -> "http://localhost:" + container.getMappedPort(PORT));
    }
}
```

Here, the `GenericContainer` class receives the image name as an argument in the constructor.

You can expose ports within your containers using the `withExposedPorts()` method, and obtain where the exposed port is mapped with the `getMappedPort()` method.

Now, to test the `CardServiceProxy` class, let's implement the `CardServiceTestContainer` interface in the test class.

```
@SpringBootTest(classes = {CardServiceProxy.class, WebClientConfig.class})
class CardServiceProxyIntegrationTest implements CardServiceTestContainer {

    @Autowired
    CardServiceProxy proxy;

    @Test
    void validateCard_test() {
        boolean isValid = proxy.validateCard(
                new CardDetails("1234567890123456", 1129, 123)
        );
        assertThat(isValid).isTrue();
    }
}
```

Here the `@SpringBootTest(classes = {...})` annotation prevents the full auto-configuration, and applies only the classes relevant to this particular test: `CardServiceProxy.class` and `WebClientConfig.class`.

Integration Testing with RabbitMQ Containers

The Message Consumer Layer

- The *Payment Event*:

```
public record PaymentEvent(Payment payment) {
}
```

- The *Payment Event Listener*:

```
@Component
public class PaymentEventListener {

    private static final Logger log = LoggerFactory.getLogger(PaymentEventListene

    private final PaymentService paymentService;

    public PaymentEventListener(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    @RabbitListener(queues = {"paymentEvents"})
    public void onPaymentEvent(PaymentEvent event) {
        try {
            paymentService.registerPayment(event.payment());
        } catch (Exception e) {
            log.warn("There was an error on event {}", event);
            log.error(e.getMessage(), e);
        }
    }
}
```

Creating the Integration Test

Let's first create a Java Interface with the RabbitMQ container.

```
@Testcontainers
public interface RabbitTestContainer {

    String DOCKER_IMAGE_NAME = "rabbitmq:3";

    @Container
    RabbitMQContainer container = new RabbitMQContainer(DOCKER_IMAGE_NAME);

    @DynamicPropertySource
    static void registerProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.rabbitmq.host", container::getHost);
        registry.add("spring.rabbitmq.port", container::getAmqpPort);
        registry.add("spring.rabbitmq.username", container::getAdminUsername);
        registry.add("spring.rabbitmq.password", container::getAdminPassword);
    }
}
```

Here, the `RabbitMQContainer` is the specific container class for RabbitMQ containerization. This instance is created with a dynamic host, port, username and password.

Next, we will use all three containers for the integration test.

```
@SpringBootTest
@ExtendWith(OutputCaptureExtension.class)
class PaymentEventListenerIntegrationTest implements CardServiceTestContainer,
        PostgresTestContainer, RabbitTestContainer {

    @Autowired
    RabbitTemplate rabbitTemplate;

    @Test
    void onPaymentEvent_test(CapturedOutput output) {

        Payment payment = new Payment(
                BigDecimal.TEN,
                PaymentMethod.CARD,
                new CardDetails("1234567890123456", 1129, 123)
        );

        rabbitTemplate.convertAndSend(
```

```
                "paymentEvents",
                new PaymentEvent(payment)
        );

        await().atMost(Duration.ofSeconds(5))
                .until(paymentUpdated(output), is(true));

        assertThat(output.getErr()).isEmpty();
        assertThat(output.getOut()).contains(
                "payment_status=PENDING_VALIDATION",
                "payment_status=OK"
        );
    }

    private Callable<Boolean> paymentUpdated(CapturedOutput output) {
        return () -> output.getOut().contains("Payment updated");
    }
}
```

Here, the `@SpringBootTest` will auto-configure the tests with all we need.

The `OutputCaptureExtension` class provides parameter resolution for the `CapturedOutput` instance. In the code example, the `output` parameter allows us to assert that the correct output has been written.

The first assertion on `output.getErr()` will check there are no errors on the application logs, while the second assertion on `output.getOut()` checks that the payment status has changed from `PENDING_VALIDATION` to `OK`.

Summary

In this article, we have explored how to use containerized instances of a PostgreSQL database, a RabbitMQ queue, and a custom image with Generic Containers for integration testing.

However, **Testcontainers is a rich and powerful library that offers a lot of functionalities for testing**. Check all the Modules available in the Testcontainers

official page: https://www.testcontainers.org/

Thanks for reading. I hope this was helpful!

The example code is available on GitHub.

Testcontainer Java Spring Boot Microservices

Follow

Written by Jonathan Manera

112 Followers

If you wish to make a Java app from scratch, you must first invent the universe.

More from Jonathan Manera

Jonathan Manera

Mapping Bidirectional Object Associations using MapStruct

Two strategies to map bi-directional relationships between entities

4 min read · Jan 7

2

Jonathan Manera

The Holy Trinity: JUnit5, Mockito, and Instancio

Unit tests that are heavenly easy to write and maintain

8 min read · Apr 20

21

Jonathan Manera

CI/CD Databases with Liquibase

A solution for continuous database integration

4 min read · Oct 29, 2022

10

Jonathan Manera in Better Programming

Hexagonal Architecture on Spring

Hexagonal architecture is replacing the layered style

4 min read · Jul 26, 2022

165 1

See all from Jonathan Manera

Recommended from Medium

```
DockerClientFactory - Checks are enabled
DockerClientFactory - Checking the system...
DockerClientFactory - ✔ Docker server version should be at leas
utility.PrefixingImageNameSubstitutor - No prefix is configure
utility.ImageNameSubstitutor - Did not find a substitute image
rationFileImageNameSubstitutor' and 'PrefixingImageNameSubstitu
images.AbstractImagePullPolicy - Using locally available and n
Starting container: mongo:4.4.2
Trying to start container: mongo:4.4.2 (attempt 1/1)
Starting container: mongo:4.4.2
reating container for image: mongo:4.4.2
```

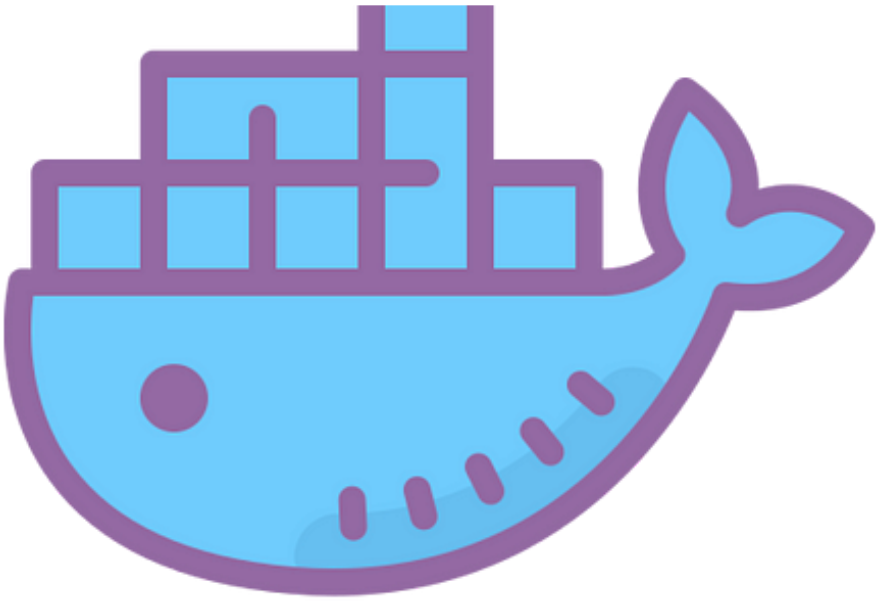
Soham Sen

Database Integration Tests with Mongo and Testcontainers

What are Testcontainers?

4 min read · Mar 3

 3  1 



Jonathan Manera

Building a Stub Server with WireMock and Docker

A practical approach to API stubbing and virtualization

4 min read · Feb 15

1

Lists

It's never too late or early to start something

13 stories · 66 saves

General Coding Knowledge

20 stories · 192 saves

New_Reading_List

174 stories · 61 saves

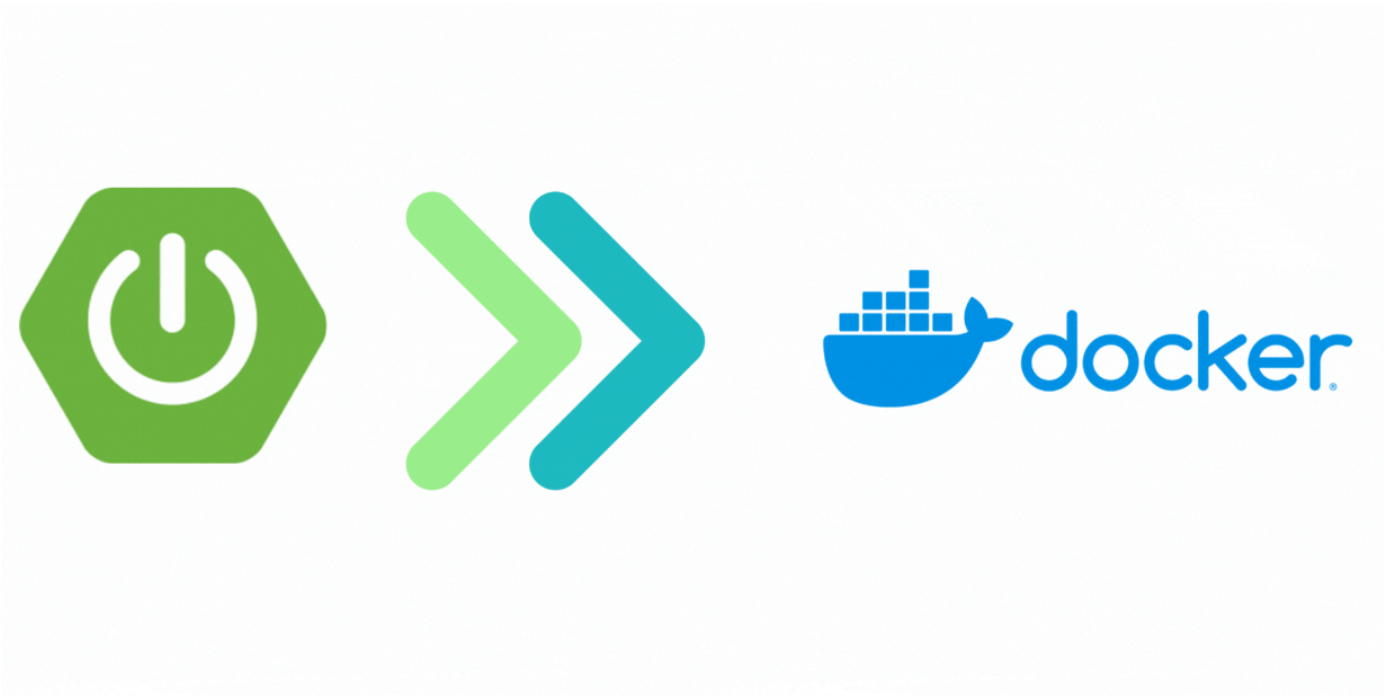
Rupert Waldron

Create a non-blocking REST Api using Spring @Async and Polling

Get the great asynchronous, non-blocking experience you deserve with Spring's basic Rest Api, polling and @Aysnc annotation.

12 min read · Feb 25

16



Mónika Lombos in Level Up Coding

Spring Boot 3.1 comes with docker-compose support

dev-ready feature

· 3 min read · May 23

88 1

William Rees

Maximizing Test Coverage: Using TestContainers to Simplify Integration Testing in .NET

Testing is a critical aspect of software development as it helps to ensure that we're delivering high quality and maintainable software...

5 min read · Feb 16

7

Tejreddymamidi

Integration Tests using TestContainers with Redis

What are Integration Tests?

3 min read · Mar 20

3

See more recommendations