Streamlining Integration Testing with Test Containers: Addressing Limitations and Best Practices for Implementation

Sameer Shukla
Lead Software Engineer
Dallas, TX USA

Abstract: Integration Testing [1] [2] is an essential part of software development that ensures that different components of an application work correctly when integrated. However, setting up and managing external dependencies such as databases, message brokers, and web servers for integration testing can be challenging, time-consuming, and error prone.TestContainers is a Java library that simplifies the process of creating and managing Docker containers for testing purposes. In this paper, we discuss how to use TestContainers efficiently for integration testing.We begin by providing an overview of TestContainers, including its features, benefits, and supported technologies. We then discuss best practices for using TestContainers, such as choosing the right container for your testing needs, managing container lifecycles, and configuring containers for optimal performance.Next, we present several examples of how to use TestContainers in different scenarios, such as testing amicroservices-based application, testing a multi-database environment, and testing a message-driven application.Finally, we conclude the paper with a discussion of the benefits of using TestContainers for integration testing, including improved test reliability, faster test execution, and easier test setup and maintenance. Every aspect.

Keywords: Integration Testing, Test Containers, Spring Boot, Java, Junit, Docker, Kafka, Redis

Introduction:

The Agile development approach emphasizes on fast and frequent delivery of working software, and integration testing plays a crucial role in ensuring the quality and functionality of the software product. However, testing with external dependencies can often pose a challenge for developers, as they need to set up and manage these dependencies in their testing environments.

To simplify this process, TestContainers has emerged as a popular tool in the DevOps community. TestContainers provides a lightweight, disposable container environment for testing, allowing developers to easily spin up and manage external dependencies such as databases, message brokers, and web services.

In this research paper, we explore the use of TestContainers in optimizing integration testing and simplifying the management of external dependencies in the Agile development process. We will investigate the benefits of using TestContainers, including reducing testing time, increasing test coverage, and improving test reliability. We will also discuss practical examples of implementing TestContainers in real-world projects and provide guidelines for integrating TestContainers into your own Agile development workflow.

What are Test Containers?

Test Containers is a testing library that allows developers to easily create and manage Docker containers for integration testing. In short, it's a Wrapper API over Docker [4], Here's how it works behind the scenes:

1. Test Containers is initialized with a Docker client, which can be either a local Docker instance or a remote Docker host.
2. When a test is executed, TestContainers starts a new Docker [5] container [6] based on the specified image and sets up the container environment as specified in the configuration.
3. Once the container is up and running, TestContainers exposes its network ports and IP address to the host system so that the test can connect to it.
4. The test code can then use the exposed network ports and IP address to interact with the containerized service, just as if it were running locally.
5. After the test is completed, TestContainers stops and removes the container, ensuring that the environment is clean for subsequent tests.
6. Test Containers provides various hooks and configuration options to customize the container lifecycle, such as initializing the database schema or seeding data before starting the container.

Limitations Solve by Test Containers

Test Containers solve several limitations that developers face when performing integration testing with external dependencies, including:

1. **Complexity of setup:** Setting up external dependencies, such as databases and message brokers, for integration testing can be a time-consuming and error-prone task. TestContainers simplifies this process by providing a lightweight, disposable container environment that can be easily spun up and managed by developers.
2. **Inconsistency between testing environments:** Inconsistencies between testing environments can lead to unpredictable test results and make it difficult to reproduce bugs. TestContainers provides a consistent testing environment across different platforms, ensuring that tests can be reliably run in any environment.
3. **Difficulty in reproducing bugs:** Integration testing can make it difficult to reproduce bugs, as it can be hard to recreate the exact environment in which the issue occurred. TestContainers makes it easier to reproduce issues by providing a self-contained environment that can be easily replicated.
4. **Slow test execution:** Traditional integration testing can be slow due to the time it takes to set up and tear down external resources. TestContainers accelerates testing by providing a lightweight and fast solution for managing containerized resources.
5. **Addressing constraints of third-party software:** Typically, in the Spring Boot ecosystem, an approach for conducting end-to-end database operations testing involves utilizing an in-memory H2 database, yet this method presents various drawbacks. Some of them are.

- Limited SQL support: H2 Database [9] has limited SQL support compared to other databases, which can make it challenging to test complex SQL queries.
- Lack of compatibility with other databases: H2 Database [9]is not always compatible with other databases, which can be an issue if your application relies on multiple databases.
- Different behavior from production databases: H2 Database may not behave exactly the same way as production databases, which can lead to unexpected behavior during testing.
- Inability to test database-specific features: H2 Database does not support all features that are available in other databases, which can be a limitation when testing database-specific functionality.
- Limited scalability: H2 Database may not be as scalable as other databases, which can be an issue if your application requires high levels of scalability during testing.

Also, in general the architecture with H2DB looks like,

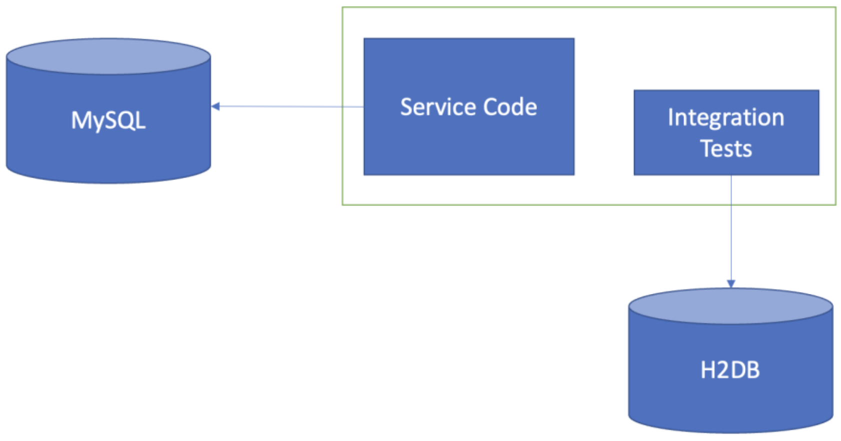


Fig1: Same application pointing to 2 different databases.

5. Streamlining Testing: Consider a situation where data is retrieved from the Redis Cache [6] [7] [8] if it exists, otherwise, the database is queried to populate the cache and then the data is returned. To test this flow without TestContainers we need to mock multiple external entities as Integration with actual setup is error prone. TestContainers simplifies such scenarios to great extent.

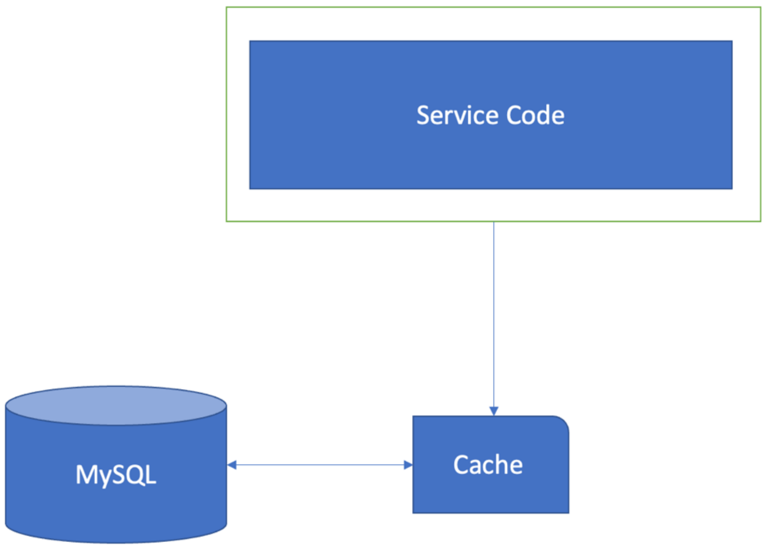


Fig2: Data is returned from Cache if present else go to DB and fetch.

Annotations in Test Containers

TestContainers provides several annotations [10] that can be used to configure and manage containerized resources for testing. Here are some of the most commonly used annotations:

@ Test containers: This annotation is used to initialize the TestContainers infrastructure before the tests are run. It is typically placed at the class level.

@ Container: This annotation is used to define a container that will be managed by TestContainers during testing. It is typically placed at the field level, and the type of the field should correspond to the type of container being defined.

@ Dynamic Property Source: This annotation is used to dynamically set properties for the container at runtime. This can be useful for setting environment variables or other configuration options that are required by the container.

The Test Containers library provides the aforementioned annotations, which can be found in the "org.testcontainers" package. Additionally, there are certain "junit" specific annotations that are crucial for the optimal execution of tests.

@Before All: This annotation is used to define setup code that will be run once before all the tests in the class. This can be useful for setting up resources that will be shared across multiple tests.

@After All: This annotation is used to define cleanup code that will be run once after all the tests in the class. This can be useful for tearing down resources that were set up during testing.

@Before Each: This annotation is used to define setup code that will be run before each test in the class. This can be useful for resetting the state of the container or other resources between tests.

@After Each: This annotation is used to define cleanup code that will be run after each test in the class. This can be useful for cleaning up resources that were created during testing.

Test Containers Library Methods

Test Containers is a Java testing library that provides several methods to manage containerized resources for integration testing. Here are some of the most commonly used TestContainers library methods:

“start”: This method starts the container and initializes its environment.

“stop”: This method stops the container and cleans up its resources.

“getContainerIpAddress”: This method returns the IP address of the container.

“getMappedPort”: This method returns the port number of the container that has been mapped to a port on the host system.

“withEnv”: This method is used to set environment variables for the container.

“withClasspathResourceMapping”: This method is used to map a resource in the test classpath to a file in the container.

“withFileSystemBind”: **This method is used to mount a directory on the host system as a volume in the** container.

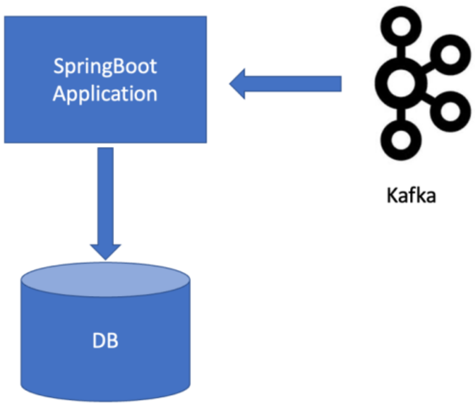
“withExposedPorts”: This method is used to specify which ports should be exposed by the container.

“waitingFor”: This method is used to specify a condition that must be met before the container is considered ready for testing.

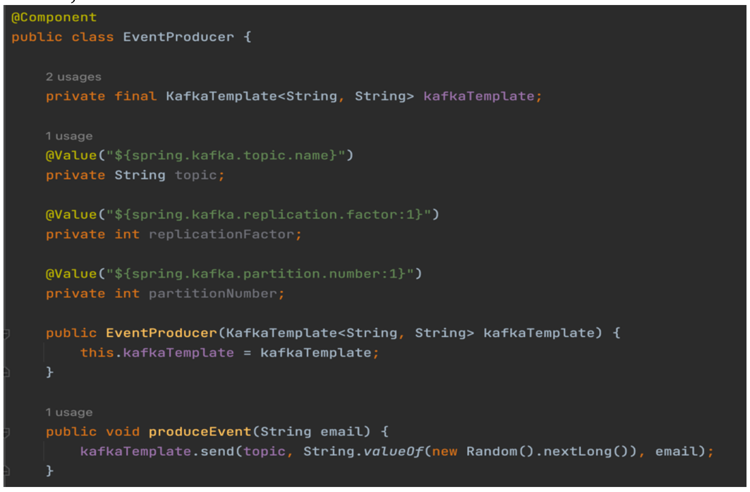
‘withCommand’: This method used to set the command that will be executed when the container [5] is started.

Use Case

Now we have sufficient information with us to dive into writing Integration Testcase using TestContainers.The use case is simple a SpringBoot service consumes event from Kafka [11] [12] topic and queries the data from the MySQL DB.

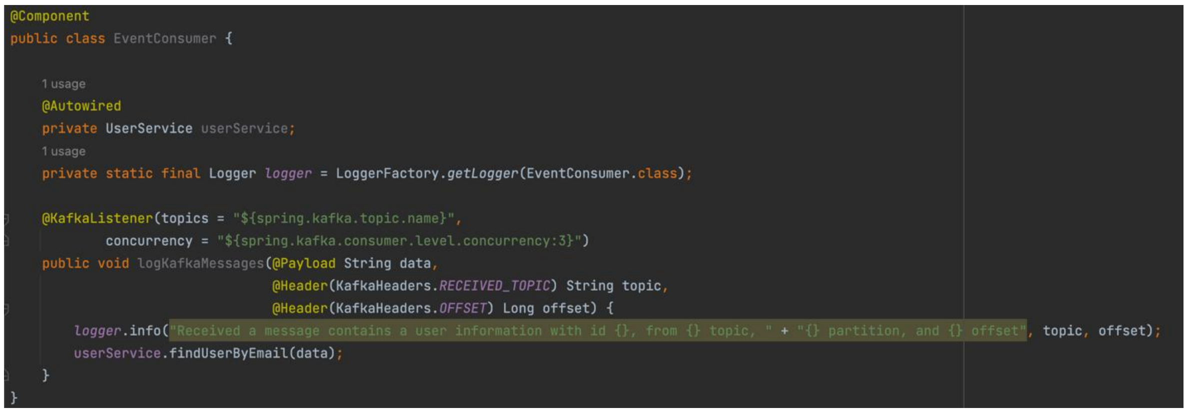


Code for producing event is,



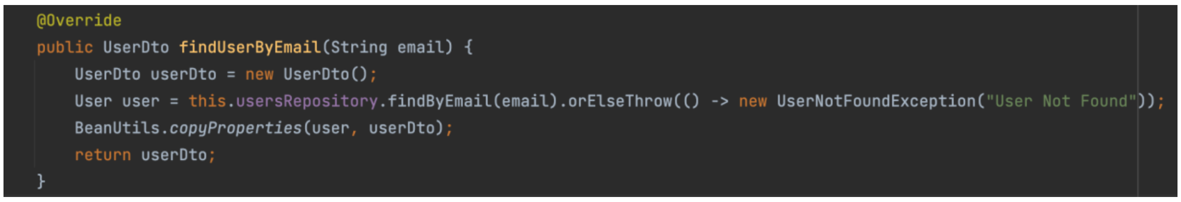
```
@Component
public class EventProducer {

    2 usages
    private final KafkaTemplate<String, String> kafkaTemplate;

    1 usage
    @Value("${spring.kafka.topic.name}")
    private String topic;

    @Value("${spring.kafka.replication.factor:1}")
    private int replicationFactor;

    @Value("${spring.kafka.partition.number:1}")
    private int partitionNumber;

    public EventProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    1 usage
    public void produceEvent(String email) {
        kafkaTemplate.send(topic, String.valueOf(new Random().nextLong()), email);
    }
```

The code for the “Event Consumer” class is, the Consumer class simply consumes the Event and query the Service class.



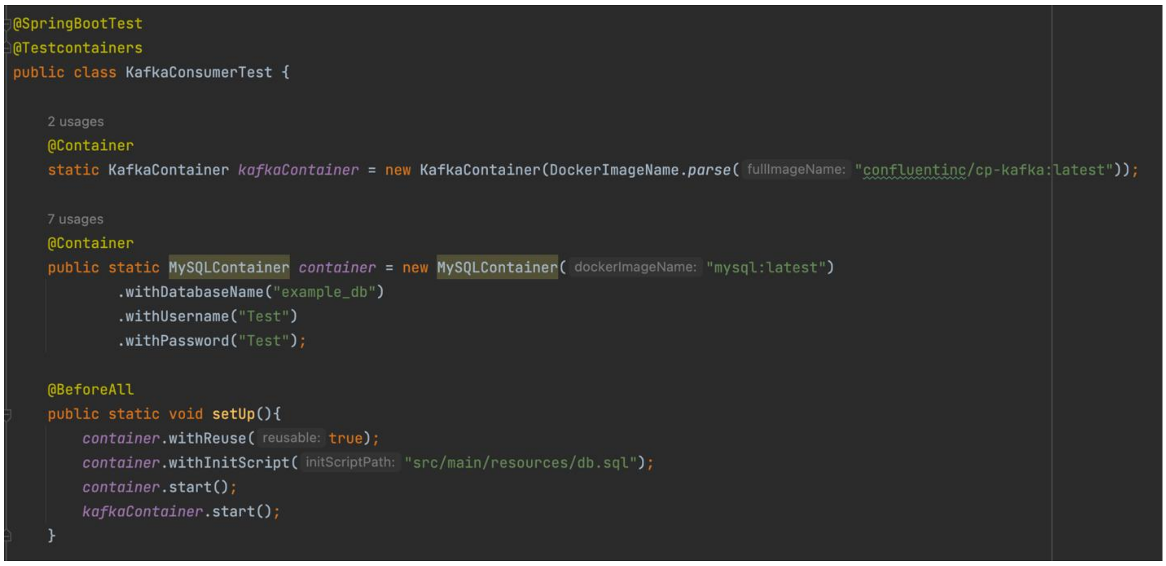
```java
@Component
public class EventConsumer {

    1 usage
    @Autowired
    private UserService userService;
    1 usage
    private static final Logger logger = LoggerFactory.getLogger(EventConsumer.class);

    @KafkaListener(topics = "${spring.kafka.topic.name}",
            concurrency = "${spring.kafka.consumer.level.concurrency:3}")
    public void logKafkaMessages(@Payload String data,
                                 @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
                                 @Header(KafkaHeaders.OFFSET) Long offset) {
        logger.info("Received a message contains a user information with id {}, from {} topic, " + "{} partition, and {} offset", topic, offset);
        userService.findUserByEmail(data);
    }
}
```

The "find User By Email" method in the service class is a simple implementation that retrieves a user from the database based on their email address.



```java
@Override
public UserDto findUserByEmail(String email) {
    UserDto userDto = new UserDto();
    User user = this.usersRepository.findByEmail(email).orElseThrow(() -> new UserNotFoundException("User Not Found"));
    BeanUtils.copyProperties(user, userDto);
    return userDto;
}
```

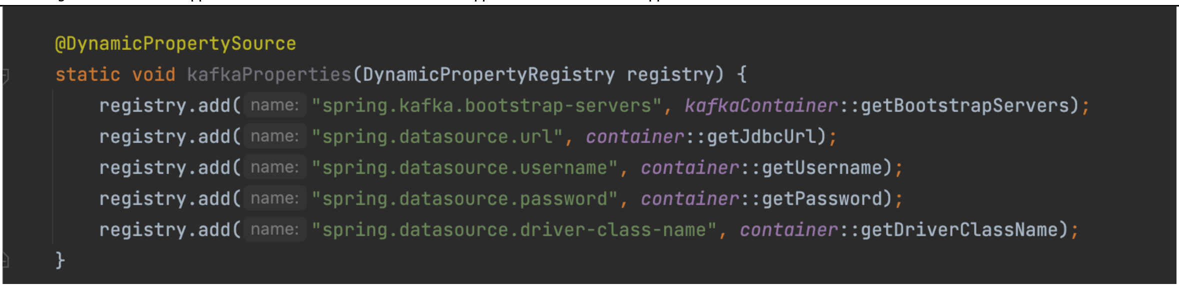
The Integration Test case is as follows,



```java
@SpringBootTest
@Testcontainers
public class KafkaConsumerTest {

    2 usages
    @Container
    static KafkaContainer kafkaContainer = new KafkaContainer(DockerImageName.parse( fullImageName: "confluentinc/cp-kafka:latest"));

    7 usages
    @Container
    public static MySQLContainer container = new MySQLContainer( dockerImageName: "mysql:latest")
            .withDatabaseName("example_db")
            .withUsername("Test")
            .withPassword("Test");

    @BeforeAll
    public static void setUp(){
        container.withReuse( reusable: true);
        container.withInitScript( initScriptPath: "src/main/resources/db.sql");
        container.start();
        kafkaContainer.start();
    }
```

The above class is marked with @TestContainers annotation [10] the @TestContainer annotation is a JUnit-Jupiter extension that facilitates the automatic starting and stopping of containers used in tests. This annotation identifies fields that are tagged with @Container and invokes the specific container life-cycle methods. In this case, the life-cycle methods of the My SQL Container and Kafka Container [12] will be invoked. Both KafkaContainer and MySQLContainer marked static because a single container is started and shared across all test methods.

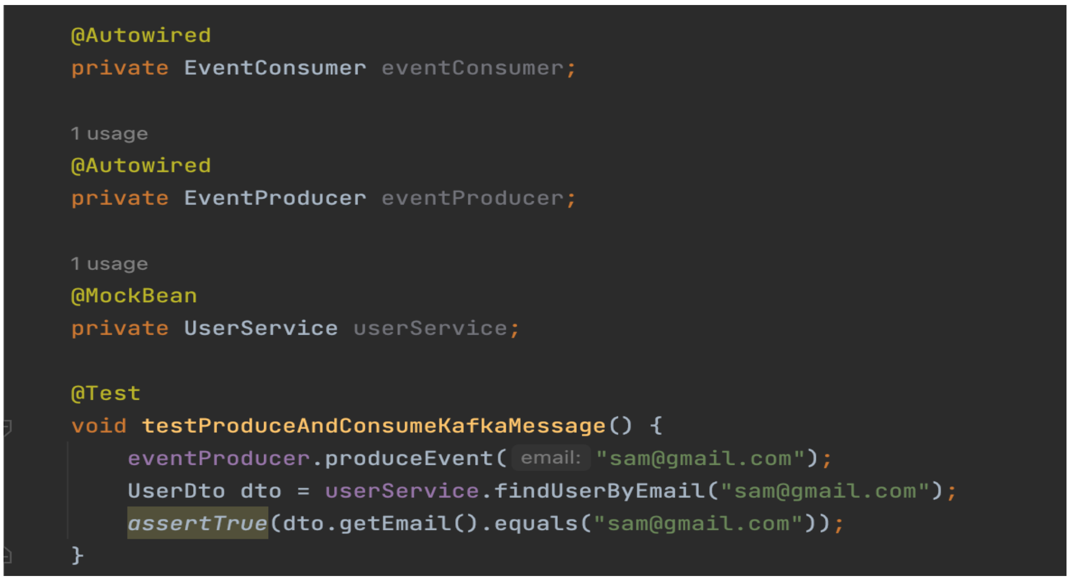
Next, we will create a setup method annotated with @BeforeAll, in which we enable the "with Reuse" method to allow reuse of existing containers. We will use the "withInitScript" method to execute the ".sql" file and then start the container.



```
@DynamicPropertySource
static void kafkaProperties(DynamicPropertyRegistry registry) {
    registry.add( name: "spring.kafka.bootstrap-servers", kafkaContainer::getBootstrapServers);
    registry.add( name: "spring.datasource.url", container::getJdbcUrl);
    registry.add( name: "spring.datasource.username", container::getUsername);
    registry.add( name: "spring.datasource.password", container::getPassword);
    registry.add( name: "spring.datasource.driver-class-name", container::getDriverClassName);
}
```

Next up is the @DynamicPropertySource annotation allows us to override the properties declared in a properties file. We can use this annotation to enable TestContainers to generate the URL, username, and password automatically to avoid potential errors that may occur. This method is written specifically for this purpose.

That's it, now we can run our usual Integration Test which will be talking to KafkaContainer and MySQL Container.



```
@Autowired
private EventConsumer eventConsumer;

1 usage
@Autowired
private EventProducer eventProducer;

1 usage
@MockBean
private UserService userService;

@Test
void testProduceAndConsumeKafkaMessage() {
    eventProducer.produceEvent( email: "sam@gmail.com");
    UserDto dto = userService.findUserByEmail("sam@gmail.com");
    assertTrue(dto.getEmail().equals("sam@gmail.com"));
}
```

Conclusion:

In conclusion, TestContainers is a popular testing library that provides a powerful and flexible way to create and manage lightweight, disposable test containers for integration testing. TestContainers is designed to solve the limitations of traditional integration testing by providing a consistent, reliable, and scalable environment that allows for easy and effective testing of complex applications.

One of the major advantages of TestContainers is its extensive use of annotations, such as @Container, @DynamicPropertySource, and @Testcontainers, which enable easy and intuitive setup and teardown of test containers. These annotations also allow developers to customize and configure their test environment to their specific needs.

TestContainers also provides a wide range of container classes and methods to support a variety of databases, messaging systems, and other components commonly used in modern application development. For example, TestContainers includes built-in support for Kafka and MySQL, allowing developers to easily test and validate these components in their applications.

TestContainers is a highly versatile and flexible testing library that provides support for a wide range of software components commonly used in modern application development. TestContainers supports popular databases such as MySQL, PostgreSQL, Oracle, SQL Server, and MongoDB, as well as message brokers such as Kafka, RabbitMQ, and ActiveMQ.

TestContainers also supports other components such as Selenium for web UI testing, Elasticsearch for full-text search testing, and even Docker itself for testing Dockerized applications.

This broad range of software support makes TestContainers a highly valuable tool for developers who need to test their applications in a wide variety of environments and configurations. With TestContainers, developers can easily create and manage disposable test containers for all of their software components, ensuring a consistent and reliable testing environment for their applications.

TestContainers is a valuable testing tool that can be used effectively in a Google Cloud Platform (GCP) environment. TestContainers provides a range of container classes and methods that support GCP services such as Cloud SQL, Cloud Spanner, and Cloud Pub/Sub.

Using TestContainers in a GCP environment can simplify the process of setting up and managing testing environments for these services, allowing developers to focus on writing and executing their tests without worrying about the underlying infrastructure.

Furthermore, TestContainers provides extensive support for Docker, which can be used to deploy applications in a containerized form to GCP's Kubernetes Engine or Google Compute Engine. This makes it easy to test containerized applications in a GCP environment, enabling developers to identify and resolve any issues before deployment to production.

Overall, TestContainers is an excellent tool for simplifying integration testing in Java applications. By providing a consistent and reliable testing environment, along with powerful annotations and container classes, TestContainers enables developers to write more effective and efficient integration tests, resulting in higher quality software and a more seamless development process.

References:

[1]. A Survey of Integration Testing in Software Engineering" by Lian Yu, Qing Gu, and Shanping Li. IEEE Access, vol. 6, 2018, pp. 41714-41733, doi: 10.1109/ACCESS.2018.2859836.

[2]. Model-Based Integration Testing of Real-Time Systems" by Andreas Ulbrich, Holger Giese, and Daniel Ziegenberg. Proceedings of the 20th IEEE International Conference on Automated Software Engineering, 2005, pp. 239-248, doi: 10.1109/ASE.2005.25.[3]. Agrawal, A., Gans, J., Goldfarb, A., 2018b. Prediction Machines: The Simple Economics ofArtificial Intelligence. Harvard Business Review Press, Cambridge, MA.

[3]. A Performance Analysis of Docker Containers for Scientific Applications" by M. Yousaf Javed, NouredineMelab, and Daniel Tuyttens. Journal of Grid Computing, vol. 15, no. 1, 2017, pp. 19-39, doi: 10.1007/s10723-016-9378-3.

[4]. Docker Containers as a Service: A Platform for Building and Sharing Scientific Tools" by Keith James, Robert L. Grossman, and Ravi Madduri. Proceedings of the 2015 IEEE International Conference on Big Data, 2015, pp. 2335-2340, doi: 10.1109/BigData.2015.7364022.[5]. Argote, L., Greve, H.R., 2007. A Behavioral Theory of the Firm —40 years and counting:introduction and impact. Organ. Sci. 18, 337–349.https://doi.org/10.1287/orsc.1070.0280.

[5]. Towards an Efficient Implementation of Docker Containers for HPC Applications" by David E. Singh, Karl Fuerlinger, and Edgar Solomonik. Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium, 2016, pp. 907-916, doi: 10.1109/IPDPS.2016.83.[7]. Autodesk, 2016. Airbus: reimagining the future of air travel [WWW Document]. AutodeskURL. http://www.autodesk.com/customer-stories/airbus (accessed 1.29.19).

[6]. Redis: a database for the cloud era, Salvatore Sanfilippo, 2015. Citation: Sanfilippo, S. (2015). Redis: a database for the cloud era. Communications of the ACM, 58(4), 44-51.

[7]. Redis: Lightweight Key-value Store with Persistence, Salvatore Sanfilippo and Pieter Noordhuis, 2010. Citation: Sanfilippo, S., &Noordhuis, P. (2010). Redis: lightweight key-value store with persistence. In Proceedings of the Conference on Innovative Data Systems Research (pp. 1-6).

[8]. Performance Analysis of Redis for Large Scale Web Applications, F. Chong, P. Krishnamurthy, and C. E. Wills, 2013. Citation: Chong, F., Krishnamurthy, P., & Wills, C. E. (2013). Performance analysis of Redis for large scale web applications. In Proceedings of the 12th International Conference on Networks (pp. 7-12).

[9]. H2 Database Engine: An Overview and Performance Evaluation, Luka Bekić, Matija Novak and Maja Matijašević, 2015. Citation: Bekić, L., Novak, M., &Matijašević, M. (2015). H2 database engine: an overview and performance evaluation. Journal of Information and Organizational Sciences, 39(1), 61-74.

[10]. An Overview of Java Annotations: A New Feature Introduced in Java 5.0, P. Krishna Kumar and J. Soundarya, 2015. Citation: Kumar, P. K., & Soundarya, J. (2015). An Overview of Java Annotations: A New Feature Introduced in Java 5.0. International Journal of Computer Applications, 119(17), 8-12.

[11]. Apache Kafka: A Distributed Streaming Platform" by Jay Kreps, Neha Narkhede, and Jun Rao (2011): This paper introduces Apache Kafka, a distributed streaming platform designed to handle large-scale data processing in real-time. The paper describes Kafka's architecture, design principles, and use cases. [Source: https://kafka.apache.org/documentation/

[12]. Towards Elastic and Fine-Grained Scaling of Stream Processing Systems" by Marta Patiño-Martínez, Kaiwen Zhang, and Bettina Kemme (2020): This paper proposes a technique for elastic and fine-grained scaling of stream processing systems, using Apache Kafka and Apache Flink as case studies. The paper describes the technique's architecture, algorithms, and evaluation results. [Source: https://ieeexplore.ieee.org/abstract/document/9208982/]