Testing Strategy for Multi-Tenant Web Applications Using TestContainers with Use Case: MealMatrix

Davor Dimoski[1,*,†], Bojana Koteska[1,*,†] and Anastas Mishev[1,†]

[1]Faculty of Computer Science and Engineering, "Rugjer Boshkovikj" 16, P.O. Box 393 1000 Skopje, North Macedonia

Abstract

This paper addresses the problem of missing a standardized approach for verifying the architectural setup of multi-tenant applications by offering a testing strategy that covers scenarios of common problems that multi-tenant applications face (divided in 3 areas: data isolation, data integrity and constraints, tenant context). Using TestContainers to create a replica of a production environment, with a big bang integration testing approach, we showcase the practical usage of the proposed testing strategy with a Spring Boot and Kotlin web application for managing meal orders – MealMatrix. The results show that the testing approach is effective in identifying faulty setup for multi-tenant environments, with a limitation that TestContainers does not cover an easy-setup for the multiple databases, multiple schemas model with a single instance of the application serving multiple tenants. This work contributes to the field of software testing by offering an easily applicable, high-level testing approach for multi-tenant web applications.

Keywords

multi-tenant, integration testing, TestContainers, software testing, testing strategy

1. Introduction

In today's software landscape, multi-tenant applications have become a go-to choice for Software-as-a-Service (SaaS) platforms. Their popularity comes from their ability to serve multiple clients from a single running application instance, which helps in reducing costs and offers a simplified maintenance. Testing this type of applications is often a challenge, due to the fact that aside from needing to verify the correct behavior of the features, we also need to verify that the tenants are correctly isolated and that the integrity of the data is managed properly without leaks. Integration testing is a crucial part of this process. Setting up a proper production-like environment is necessary to ensure quality of tests, and that is where the open source library TestContainers comes into play.

Despite the popularity of multi-tenancy, there is a lack of standardized testing strategies that cover the common scenarios that arise from the infrastructure of these systems. Testing efforts are usually concentrated on verifying the business logic and as a result of that, the architectural setup is often overlooked and not validated enough during the testing stage. Although previous work has examined similar issues [1], their focus has been on parallel and automated user acceptance testing (UAT), whereas here, we will focus on providing generic test scenarios and show a use case with integration testing to address this issue. This paper aims to fill the gap of missing standardization and provide a high-level testing strategy for multi-tenant applications by covering the most common problems that come from applications that use multi-tenant architecture: data isolation on CRUD operations, data integrity and constraints, and tenant context. We will try to demonstrate its effectiveness through a practical use case, implementing integration tests using TestContainers with the proposed strategy in a multi-tenant web application for managing food orders – MealMatrix. This approach lays the groundwork for building more robust and repeatable integration testing practices in multi-tenant environments.

SQAMIA 2025: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 20–12, 2025, Maribor, Slovenia

*Corresponding author.

†These authors contributed equally.

davor.dimoski@students.finki.ukim.mk (D. Dimoski); bojana.koteska@finki.ukim.mk (B. Koteska); anastas.mishev@finki.ukim.mk (A. Mishev)

0009-0004-0363-4057 (D. Dimoski); 0000-0001-6118-9044 (B. Koteska); 0000-0001-7271-6655 (A. Mishev)

2. Multi-tenancy

2.1. Definition

Multi-tenancy is an architectural pattern in which a single instance of the application serves multiple clients (aka tenants) [2]. This means that the tenants are sharing resources such as servers (and sometimes databases) while keeping data secure and separated. The basic functionality that a multi-tenant application needs to cover is to support the usage of multiple tenants and ensure that there is a separation of concerns for each client [3]. In addition to this, in order to sustain long-term usage, it needs to be easily scalable, robust, and secure. Multi-tenancy is an attractive solution for models like SaaS (Software as a Service) and PaaS (Platform as a Service) due to the fact that it provides a low-cost solution that optimizes resource utilization.

2.2. Types of multi-tenancy

As previously stated, in a multi-tenant environment, numerous clients utilize the same application instance, hardware, and storage mechanism [2]. Database architecture varies, offering three distinct approaches. This setup optimizes resource utilization and security by ensuring isolation between the tenants. Understanding these distinctions is very important for both businesses and software developers when selecting the model that best aligns with their specific requirements. The three types of database structure in multi-tenancy are the following:

1. **Shared database, shared schema** – the main characteristics of this type are the cost-effectiveness, resource optimization, and uniformity. The data isolation between the tenants in this style is done with tenant-specific identifiers in each of the tables in the database. The limitations here are mainly about customization and scalability when introducing new tenants.
2. **Shared database, multiple schemas** – here we have a single database and a separate schema for each tenant. This offers better data isolation than the shared schema model because the data for each tenant is kept in a separate space. This also allows a higher level of customization while still keeping efficiency in resource use.
3. **Multiple databases, multiple schemas** – in this model we have separate database and schema instances for each tenant. This approach can be executed in 2 different variations: using a single instance for multiple tenants that solves the database access in its implementation, or by using separate instances with custom configurations for each tenant. While this model provides the highest possible data isolation, customization and flexibility, it comes with higher costs. This method is usually the most useful where data security and isolation are the highest priority and each tenant requires wide-ranging customization.

2.3. Benefits and challenges

When it comes to multi-tenancy, its most significant advantages are sharing physical resources and high configurability [4]. In the single-tenant architecture, each tenant operates on a separate virtual machine that's tailored to its specific needs. But this is often very restricting because every virtual machine requires significant resources from the single physical machine on which it's operating. The use of one instance for multiple tenants is the core of the multi-tenancy paradigm and this offers cheaper maintenance and overall costs. In addition to this, the multi-tenancy setup requires the applications to be highly configurable and customizable to cater to the specific needs of different tenants and these configuration capabilities are built into the application's design.

On the opposite side, some of the most notable disadvantages associated with implementing a multi-tenancy architecture are performance, scalability, security, and maintenance. The tenants in a multi-tenant application often have varied demands – one tenant can excessively consume a lot of the resources, which can have an impact on the performance of other tenants. Aside from the performance issue, multi-tenant applications get progressively harder to scale as the number of tenants grows overtime, and with that the maintenance becomes harder to manage. Multi-tenant applications

can also face challenges with security, in the sense that a security breach of one tenant could lead to the unintended exposure of data to other tenants, potentially even competitors.

3. Integration testing

In today's age, it's rare for a software to work in isolation, without having to communicate with other modules or components [5]. These units can range from databases, 3rd party applications, and so on, and it's crucial for them to work well together in order to be able to deliver software. One of the issues that occurs often is that components work individually, but not when combined. The conflicts may arise due to incompatibility between versions, different processing logic, etc. All of these issues lead to the need for integration testing.

Integration testing is a technique where combined modules or components are tested together in one big system to see if they interact with each other as expected [6]. The main objective in integration testing is to make sure that the integrated units function correctly as a group and that the flow of the application is not disrupted.

The reasons why integration testing is sometimes considered difficult in comparison to unit testing are that you have to make sure that the infrastructure is fully functional and that the data is pre-configured to meet specific requirements [5]. Additionally, if there are several builds running in parallel, it's important for the execution of each pipeline not to interfere with the test data of the other pipeline. Because of challenges like this, in-memory databases are often used for the process of integration testing. In-memory databases help in speeding up the process of testing by not having to set up specific environments for the database in the testing process. However, a lot of issues arise from using in-memory databases, the most notable being the fact that we're straying away from the real-life environment where our application will be run in production. Considering that in-memory databases are almost never used in production environments, this causes our testing process to be less reliable, and often sets the feedback cycle back by providing successful tests but improper function in the production environment. The need for setting up an environment for integration testing that is as close as possible to the production environment leads to the development of the library TestContainers, which will be covered in the next section.

Integration testing has proven to be beneficial in the areas of verifying reliability between components, their compatibility, and early detection of bugs, among other things [7]. On the flip side, some of the common downsides with integration testing are the intensive use of resources due to the involvement of several components - this can involve setting up complex test environments and hardware, and additionally, it also is a slower feedback loop due to the slower execution times [8].

There are several strategies for integration testing, and they are mostly dependent on the integration of the modules in a system [9, 10, 11]. The four main approaches are the following:

1. **The big bang approach** involves all of the modules being tested at once as a single unit. This type of testing simulates a complete system as it covers all modules that are integrated into it. It is often used in smaller systems or, conversely, in very complex and large systems where it might not be practical to test all modules together incrementally.

2. **The bottom-up approach** involves testing each module from the lower levels until all modules are covered. This way, the stability of the foundation of the system is verified by gradually reaching the higher levels. The testing process can be slow depending on the layers in the architecture.

3. **The top-down approach** is used when we want to test modules from the highest to the lowest levels of the application. This technique is used to simulate the behavior of lower-level layers that have not been integrated yet. In this methodology, stubs are needed to showcase the behavior of low-level modules, which can make testing more time-consuming.

4. **Sandwich/Hybrid approach** is a combination of both top-down and bottom-up strategies, by testing high-level and low-level modules at the same time in order to provide. The goal of this approach is to provide a comprehensive view of the application.

4. TestContainers

TestContainers is a set of open-source libraries that offer simple and lightweight APIs that help developers use real services (like databases or message brokers) for the execution of tests, all wrapped in a disposable Docker container [5]. The usage of Docker containers allows for reproducing an environment as close to production as possible, avoiding mocks or in-memory services, while being in a disposable and isolated state. Even though TestContainers is a Java-based library, it offers support for a high variety of popular technologies, including .NET, Go, NodeJS, Python, Rust, Haskell and more.

The lifecycle of TestContainers can be boiled down to three stages. Before the tests run, the configuration needs to be set up for the container and the necessary services (like the database or messaging systems). Once the setup is finished, the containers are started so that they can support the execution of the tests. During the test stage, the Docker container that was spun up by TestContainers is being used to execute the tests. After the test run is completed, the containers are automatically destroyed, regardless of whether the tests executed successfully or not. The test results are shown before the container is destroyed.

The biggest advantage of TestContainers is their ability to easily create test environments that are isolated and disposable, constructed in a way to replicate the production environment of the application as close as possible. A very common issue for testing environments is that they can be unreliable and inconsistent when executed on different locations. TestContainers offers a stable and consistent testing environment that will act the same way locally, as well as CI/CD pipelines. To further prove their reliability, Testcontainers implements several out-of-the-box wait strategies to make sure that the application is fully started before executing any tests inside the container.

On the other hand, a common downside of this library is that it has slower test execution and higher resource usage compared to the testing done with mocks or a H2 database [12] and it is also heavily dependent on a correct installation of Docker in order to be used. The dependency for a correct Docker installation can be avoided with an alternative approach - the use of TestContainers Cloud which is a paid service.

4.1. Pre-requisites

In order to set up TestContainers, you need to meet the following pre-requisites [12]:

- **Docker-Compatible Container runtime** – the main requirement in order for TestContainers to execute is having Docker installed beforehand. You can either install Docker locally on the machine where tests are going to be executed, or use TestContainers Cloud – an alternative paid cloud service provided by TestContainers. Supported container runtimes include Docker Desktop on macOS and Windows, and Docker on Linux. Alternative runtimes like Colima and Podman can also be configured with some additional required setup.
- **TestContainers Library** – the TestContainers library is a necessary dependency that's needed in the project in order to be able to utilize the service. The TestContainers library is available in a variety of programming languages: Java, .NET, Python, Go, Node.js etc.

5. Problem and proposed solution

As mentioned before, multi-tenant applications have a lot of benefits like cost efficiency and scalability, but they come with a unique set of challenges in integration testing. It is crucial to ensure proper isolation between the data of each tenant, to make sure there are no leaks between them and that the system stores and manages the data of each client correctly. Writing integration tests for applications with mocked services and in-memory databases often don't reflect the same environment as the one the application will be when using when in production state, therefore the tests can be inaccurate and not provide a real picture of the validity of the system. In this paper, we strive to provide a high-level

testing strategy for multi-tenant applications that can be applied to systems that use a multi-tenant architecture.

In this section, we will propose a testing strategy that will focus on the most common problems in integration testing for multi-tenant applications. The problems are divided into 3 separate areas: data isolation on CRUD operations, data integrity and constraints, and tenant context.

The testing strategy can be used for each type of multi-tenancy setup. All of the test scenarios are recommended to be executed using TestContainers, as a library that offers an easy replication of a production environment in an isolated container. Furthermore, this testing strategy is best applied for a big bang integration testing approach due to the fact that it covers all areas of the system that are being tested.

It is important to note that in this paper, the focus of the proposed testing strategy is on the architecture of a multi-tenant application, not on the differences between the business logic of each tenant. Writing tests that cover the differences in the business logic is dependent on the requirements that the clients have, and that will not be covered here. This paper offers a testing strategy that provides a high-level overview of the most common problems that arise from the setup of multi-tenant applications in order to mitigate these issues as soon as possible in the development lifecycle.

The two types of multi-tenancy: *shared database, shared schema* and *shared database, separate schema* are implementations that use the same database for storing data for each tenant. Considering that these implementations have the lowest separation of data between the tenants and they are heavily reliant on how it's handled in the code base, naturally, the main focus of the integration tests will be the **data isolation** between each tenant. Additionally, even though the databases are separated into *multiple databases, multiple-schemas* approach, the data isolation concerns remain if there is a single instance that is serving multiple tenants.

The test scenarios for this area that should be covered are the following:

1. The tenant can access its own data
2. The tenant can access ONLY its own data (no data is shown from other tenants)
3. Creating new data is saved under the correct tenant
4. Updating existing data modifies the correct record
5. Deleting data deletes the correct record

Here we should note that scenario number 1 and scenario number 2 are very similar, but there is a distinct difference between them and they should be covered separately. The first scenario covers a case where you're retrieving a specific record and getting the correct data back, whilst the second scenario would cover a case where you're fetching a collection of data and all the retrieved data belongs to the correct tenant.

The data isolation that we want to cover with our testing is tightly coupled with CRUD operations. The reason for this is that CRUD operations are the primary points where tenant data could leak or be accessed improperly, therefore, we want to make sure that our application is covering the basic scenarios of accessing and manipulating data.

The next area of testing that we want to cover with our testing scenarios is relating to data again – this time in the context of **data integrity and constraints**. These scenarios are applicable mainly to multi-tenancy of type *shared database, shared schema*, due to the fact that this implementation uses a discriminatory tenantId column, which is present in all of the tables in the database. If the application is not set up correctly, the integrity of the data could easily be compromised and the database-level constraints could cause issues. This section covers mainly constraints that are related to the way we store data, and not constraints that are dependent on business logic. This is an important distinction because, as mentioned earlier, the proposed testing strategy in this paper isn't reliant on the business logic of the clients, but is a bare-bones structure that can work in any multi-tenant environment.

The test scenarios in this section are the following:

1. Referential integrity check – the relationship between different entities should be allowed only if they both share the same tenant

2. Unique constraint check – the constraint should be restricted specifically to the tenant, not across all tenants

The first scenario covers the risk of data leaks between tenants. Referential integrity in the scope of a multi-tenant application is not just ensuring the referenced data exists in another table, it's important that the data is scoped to the appropriate tenant as well. This scenario is applicable only if there is a link between entities in the system.

The second scenario covers unique constraints that also have to be scoped on a tenant level and not on a global level. This scenario is applicable if there is a unique constraint on a field in the system.

The third and last area of testing scenarios covers the system's behavior regarding the multi-tenancy implementation – **tenant context**. This is applicable to all types of multi-tenancy that use a single instance of the application. Mainly, they are scenarios for error-handling that cover the system's response when we don't use the tenant implementation as expected.

Here we have 2 scenarios:

1. Missing tenant information – check if the system responds accordingly if the tenant information is not sent on request
2. Wrong tenant information – check the system's behavior when the passed tenant information is wrong, i.e., doesn't correspond to an existing tenant

These scenarios make sure that the core setup of a multi-tenant application is working as expected. Sending tenant information is the entry point of how the system deals with the user's request, so it is important that any improper data does not cause faulty behavior in the system.

In *Table 1*, you can find a summary of the testing strategy. The table covers the 3 abovementioned areas of testing, their test scenarios and the types of multi-tenancy that they can be applied to.

6. Use case for the proposed test strategy through a multi-tenant application – MealMatrix

We will demonstrate the usage of the proposed testing strategy through a practical example – a multi-tenant web application called MealMatrix. The code snippets in this paper are released under the GNU license [1]. MealMatrix is a backend web application built with Kotlin and Spring Boot whose main purpose is the management of food orders and employees. The application is built with the 3 types of multi-tenancy, each type implemented on a separate branch.

For the showcase in this section, we will provide the setup and some test examples from the testing of a multi-tenant implementation with the *shared database, shared schema* approach. The implementation here is reliant on the tenant's name being sent in the header of each request for the proper actions to be executed under the correct tenant. We will consider that the application is running for 2 tenants: restaurant and take-away. We will take the big bang approach for the integration testing, as it has been proven to be a reliable verification process for detecting system defects [11]. We will also use TestContainers in our process, a widely used tool by companies for reliable and containerized testing that gives consistent results[13]. Additionally, we will utilize jUnit for the writing of the test cases.

6.1. Setting up the environment

The setup for TestContainers consists of installing Docker on a local machine where the tests will be executed, and adding dependencies for the library in the build.gradle file, as well as a dependency for the database driver we will use in the container (PostgreSQL in our case). Since we are using Spring Boot, we will also need the Spring Boot Test library and jUnit for writing of the tests. The core libraries that are needed for writing and executing the integration tests are shown in Listing 1.

[1] https://www.gnu.org/licenses/gpl-3.0.html

Table 1
Summary of the testing strategy for multi-tenant web applications

Area	Test scenarios	Type of multi-tenancy
Data isolation on CRUD operations	1. The tenant can access its own data 2. The tenant can access ONLY its own data (no data is shown from other tenants) 3. Creating new data is saved under correct tenant 4. Updating existing data modifies the correct record 5. Deleting data deletes the correct record	• Shared database, shared schema • Shared database, separate schema • Multiple databases, multiple schemas
Data integrity and constraints	1. Referential integrity check – relationship between different entities should be allowed only if they both share the same tenant 2. Unique constraint check – the constraint should be restricted specifically to the tenant, not across all tenants	• Shared database, shared schema
Tenant context	1. Missing tenant information – check if the system responds accordingly if tenant information is not sent on request 2. Wrong tenant information – check the system's behavior when the passed tenant information is wrong i.e., doesn't correspond to an existing tenant	• Shared database, shared schema • Shared database, separate schema • Multiple databases, multiple schemas

```
testImplementation("org.springframework.boot:spring-boot-starter-test")
testImplementation("org.springframework.boot:spring-boot-test:3.4.2")
testImplementation("org.springframework.boot:spring-boot-testcontainers")
testImplementation("org.jetbrains.kotlin:kotlin-test-junit5")
testImplementation("org.testcontainers:junit-jupiter")
testImplementation("org.testcontainers:postgresql")
```

Listing 1: Core dependencies used for writing and executing tests with TestContainers

Next, we will set up the configuration for TestContainers, which will spin up the container with our needed database driver to replicate the environment as close to the production one as possible. The configuration class is shown in Listing 2. This configuration will later be imported into the specific test classes and it's used on a global level in the test environment.

The annotations used in this class and their purpose are the following:

- **@ExtendWith(SpringExtension::class)** – This annotation wires up the Spring context so that it can be used in tests. It enables features like dependency injection, and commonly used annotations like @MockBean and @Autorwired.
- **@Testcontainers** – this is the annotation that marks the usage of TestContainers. It manages the startup and teardown of containers automatically and it makes sure that containers are shared correctly between tests.
- **@TestConfiguration** – Spring annotation that marks the class as a test configuration (e.g.,

```
@ExtendWith(SpringExtension::class)
@Testcontainers
@TestConfiguration
class TestContainersConfiguration {

    davor *
    companion object {
        davor
        @Bean
        fun dataSource(): DataSource? {
            return DataSourceBuilder.create() DataSourceBuilder<*>!
                .username(postgresContainer.username) DataSourceBuilder<out DataSource!>!
                .password(postgresContainer.password)
                .driverClassName(postgresContainer.driverClassName)
                .url(postgresContainer.jdbcUrl)
                .build()
        }

        val postgresContainer = PostgreSQLContainer( dockerImageName: "postgres:15.3").apply { this: PostgreSQLContainer<out PostgreSQLContainer<*>!>
            start()
        }

        davor
        @JvmStatic
        @DynamicPropertySource
        fun registerProperties(registry: DynamicPropertyRegistry) {
            registry.add( name: "spring.datasource.url", postgresContainer::getJdbcUrl)
            registry.add( name: "spring.datasource.username", postgresContainer::getUsername)
            registry.add( name: "spring.datasource.password", postgresContainer::getPassword)
        }
    }
}
```

Listing 2: Configuration class for the tests

overriding beans like DataSource only for tests). It helps isolate test dependencies from the configuration of the main application.

Inside the companion object of the class, we have defined the following:

- **PostgreSQL container** – we create a PostgreSQL container using TestContainers that can be used by all tests
- **Registering properties** – we dynamically register property values at runtime before the Spring context loads for the database-related variables in the active profile
- **Custom bean for Data source** – this is used to connect the PostgreSQL container as the main data source used in the test context

6.2. Test case examples

Before writing the tests, we fill up the database with initial data for the tenants with an SQL script that's executed before the start of the tests. In Figures 3, 4, and 5 below, we have example tests from each area in the proposed testing strategy.

The test in Listing 3 covers the scenario "Creating new data is saved under the correct tenant" from Data Isolation. In this test, we create a new employee and check if the call for fetching an employee by ID under the correct tenant returns the correct data.

The test in Listing 4 covers the scenario "Unique constraint check – the constraint should be restricted specifically to the tenant, not across all tenants" from Data Integrity and Constraints. This test checks the ability to create discount codes that have the same name under 2 separate tenants.

The test in Listing 5 covers the scenario "Missing tenant information" from Tenant Context. This test checks if the system is behaving correctly when tenant information is missing from the request.

```
davor
@Test
fun `createEmployee - newly created employee is later returned when fetched under the correct tenant`() {
    val employeeRequest = createEmployeeRequest()

    val gson = Gson()

    val mvcResult = mockMvc.perform(
        post( urlTemplate: "/api/v1/employee/create")
            .content(gson.toJson(employeeRequest))
            .contentType(MediaType.APPLICATION_JSON)
            .header( name: "X-Tenant-Name",  ...values: "Restaurant")
    ).andReturn()

    val employeeId = gson.fromJson(mvcResult.response.contentAsString, EmployeeResponse::class.java).id

    mockMvc.perform(
        get( urlTemplate: "/api/v1/employee/$employeeId")
            .header( name: "X-Tenant-Name",  ...values: "Restaurant")
    )
        .andExpect(status().isOk)
        .andExpect(jsonPath( expression: "firstName").value( expectedValue: "Katniss"))
        .andExpect(jsonPath( expression: "lastName").value( expectedValue: "Everdeen"))
        .andExpect(jsonPath( expression: "isActive").value( expectedValue: true))
        .andExpect(jsonPath( expression: "type").value( expectedValue: "WAITER"))
}
```

Listing 3: Example test for a scenario from Data Isolation

```
davor *
@Test
fun `createDiscountCode - you should be able to create 2 discount codes with same code name under separate tenants (unique constraint)`() {
    val employeeRequest = DiscountCodeRequest(
        code = "DISCOUNT10",
        percent = 10.0,
        startActiveDate = LocalDate.now(),
        endActiveDate = LocalDate.now().plusDays( daysToAdd: 30)
    )

    mockMvc.perform(
        MockMvcRequestBuilders.post( urlTemplate: "/api/v1/discount-code/create")
            .content(gson.toJson(employeeRequest))
            .contentType(MediaType.APPLICATION_JSON)
            .header( name: "X-Tenant-Name",  ...values: "Restaurant")
    )

    mockMvc.perform(
        MockMvcRequestBuilders.post( urlTemplate: "/api/v1/discount-code/create")
            .content(gson.toJson(employeeRequest))
            .contentType(MediaType.APPLICATION_JSON)
            .header( name: "X-Tenant-Name",  ...values: "Take-away") // different tenant
    )
        .andExpect(status().is2xxSuccessful)
}
```

Listing 4: Example test for a scenario from Data Integrity and Constraints

6.3. Findings

The practical implementation of the testing strategy has proven to be efficient for validating the integrity of the architectural setup of multi-tenant web applications. The testing strategy is useful in identifying issues related to the areas of data isolation, data integrity and constraints, as well as the context of the tenant itself.

However, there is a restriction that we came across that limits this approach of using the testing strategy together with TestContainers. For the multi-tenancy model of *multiple databases, multiple schemas*, TestContainers is able to provide the correct setup for the testing strategy only if the model is

```
davor
@Test
fun `getEmployeeById - Exception is thrown if no tenant name header is present`() {

    mockMvc.perform(
        get( urlTemplate: "/api/v1/employee/1")
    )
        .andExpect(status().is4xxClientError)
        .andExpect(jsonPath( expression: "$.message").value( expectedValue: "Missing X-Tenant-Name header in request"))
}
```

Listing 5: Example test for a scenario from Tenant Context

set up by using separate application instances for each tenant. Docker (and therefore TestContainers as well) follows the principle of "one service per container" as a best practice [14]. It does not offer an out-of-the-box solution for this problem. While it is possible to manually configure a container that will support multiple database services, that approach shifts the responsibility of managing the container's lifecycle entirely onto the developer. That type of manual handling defies the original purpose of a library like TestContainers, which is intended to simplify and automate the setup and management of test environments.

7. Conclusion

This paper introduced a testing strategy that covers the common problems any multi-tenant application can face. By doing so, we standardized the approach for testing the architectural setup of multi-tenant web applications. We showed its practical usage by implementing integration tests using TestContainers on a web application MealMatrix. The findings from the use case showed that although the strategy does hold up in verifying the proper implementation of a multi-tenant structure, the usage of TestContainers falls short on providing a proper out-of-the-box support for the model *multiple databases, multiple schemas* for an application that serves multiple tenants on a single instance. Regardless of this, the usage of TestContainers in integration testing is still very valuable by offering a fast solution for creating disposable and isolated containers for testing.

There are some limitations to this research that can be further improved. The proposed testing strategy does not cover scenarios for testing other parts of multi-tenant application, such as the security layer of the application or the lifecycle of its tenants (e.g. creating new tenants, migrating tenant data, or deactivating existing tenants). These things can also serve as a possibility to extend this research in the future. In addition to expanding the testing strategy, there is also a potential to define a standardized approach for integrating these methods into CI/CD workflows and enabling automated test generation, as well as explore how testing would work for multi-tenant applications that are not web-based.

The testing strategy presented in this paper serves as a foundational step toward what we hope will become a broader effort to advance research in the testing of multi-tenant applications.

8. Acknowledgments

This research is partially supported by the Faculty of Computer Science and Engineering in Skopje.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] V. H. S. C. Pinto, R. R. Oliveira, R. F. Vilela, S. R. Souza, Evaluating the user acceptance testing for multi-tenant cloud applications., in: CLOSER, 2018, pp. 47–56.

[2] V. Cheng, Multi-tenancy architecture, 2004. URL: https://www.saasceo.com/multi-tenancy-architecture/.

[3] N. H. Bien, T. Dan Thu, Multi-tenant web application framework architecture pattern, in: 2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS), 2015, pp. 40–48. doi:10.1109/NICS.2015.7302219.

[4] C.-P. Bezemer, A. Zaidman, Multi-tenant saas applications: maintenance dream or nightmare?, IWPSE-EVOL '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 88–92. URL: https://doi.org/10.1145/1862372.1862393. doi:10.1145/1862372.1862393.

[5] Testcontainers, 2025. URL: https://testcontainers.com/.

[6] H. Leung, L. White, A study of integration testing and software regression at the integration level, in: Proceedings. Conference on Software Maintenance 1990, 1990, pp. 290–301. doi:10.1109/ICSM.1990.131377.

[7] J. E. T. Akinsola, M. Adeagbo, Qualitative comparative analysis of software integration testing techniques 7 (2022) 67–82.

[8] I. Alazzam, A. M. R. AlSobeh, B. B. Melhem, Enhancing integration testing efficiency through ai-driven combined structural and textual class coupling metric, Online Journal of Communication and Media Technologies 14 (2024) e202460.

[9] BrowserStack, Integration testing: A detailed guide, 2025. URL: https://www.browserstack.com/guide/integration-testing.

[10] GeeksforGeeks, Software engineering | integration testing, 2025. URL: https://www.geeksforgeeks.org/software-engineering-integration-testing/.

[11] J. Solheim, J. Rowland, An empirical study of testing and integration strategies using artificial software systems, IEEE Transactions on Software Engineering 19 (1993) 941–949. doi:10.1109/32.245736.

[12] R. North, other authors, Testcontainers for java, 2025. URL: https://java.testcontainers.org/.

[13] S. Timonen, M. Sroor, R. Mohanani, T. Mikkonen, Anomaly detection through container testing: A survey of company practices, in: International Conference on Product-Focused Software Process Improvement, Springer, 2023, pp. 363–378.

[14] Run multiple processes in a container, 2024. URL: https://docs.docker.com/engine/containers/multi-service_container/.