



How to Create Predictable Tests in Jakarta EE Applications: A Guide to Reliable and Repeatable Testing



The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

User Guide

Contents

Guide Updated: **December 2023**

Isolating the Test Environment	1
The Power of Mocking	1
Leveraging Jakarta EE Testing Features	3
Managing Time and Date Challenges	3
Integration Testing with TestContainers	4
Standardizing the Build Process	5
Integrating Tests in Continuous Integration (CI) Frameworks	6
Keeping Test Documentation Updated and Regularly Reviewed	7
Conclusion	8

In the realm of enterprise software development, Jakarta EE stands out as a powerful framework, offering robustness and scalability for modern applications. Evolved from Java EE, Jakarta EE has a comprehensive set of specifications designed for building high-grade enterprise systems. Central to the development lifecycle in this environment is the practice of testing. Testing, in its various forms - unit, integration, system, and acceptance testing - plays a pivotal role in ensuring the reliability and performance of applications built on Jakarta EE.

However, a critical aspect of effective testing is predictability and repeatability. Predictable tests are those that consistently produce the same results under the same conditions, regardless of external variables. This level of predictability is essential for several reasons. First, it ensures that the tests are reliable indicators of software quality and functionality. Consistently predictable tests provide confidence that the application will perform as expected in the production environment. Second, it helps in early detection and resolution of bugs, leading to a more stable and robust application. Third, it enhances the efficiency of the development process, as developers spend less time troubleshooting erratic test failures and more time on actual development and enhancement of the application.

This guide aims to delve into the key strategies and best practices that can help you achieve predictability and repeatability in your Jakarta EE application testing. Whether it's unit testing individual components or conducting comprehensive integration tests, the goal is to establish a testing process that is not only thorough but also consistently reliable. Achieving this level of predictability in tests is a vital step toward developing stable, high-performing, and reliable enterprise applications.

Isolating the Test Environment

Begin by isolating your test environment from unpredictable external dependencies. Avoid reliance on external databases or network resources that may be prone to state changes or unavailability. Opt for in-memory databases like H2 or use mocking frameworks like Mockito to replicate these dependencies reliably within your tests.

The Power of Mocking

Mocking is a crucial technique for creating controlled, predictable test contexts. It allows you to replace real instances with mock objects, which simulate behaviours of external dependencies such as databases or services. This is especially useful in unit testing, where the goal is to test components in isolation. Libraries like Mockito facilitate the creation of these mocks, enabling the testing of various scenarios and edge cases. With mocking, you gain the ability to specify expected behaviours, such as specific return values or exceptions, under controlled conditions.

For instance, consider a scenario where you have a service class in your Jakarta EE application that depends on an external API for data retrieval. In a unit test for this service class, instead of making actual API calls, you could use Mockito to create a mock object of the API client. This mock can then be programmed to return a predictable set of data relevant to your test case. For example:

```
class MyServiceTest {
    @Mock
    private ExternalAPIClient mockApiClient;
    @InjectMocks
    private MyService serviceUnderTest;
    @BeforeEach
    void setUp() {
        MockitoAnnotations.initMocks(this);
    }
    @Test
    public void testServiceMethod() {
        // Define the behavior of the mock
        when(mockApiClient.fetchData()).thenReturn(new Data("Mocked Data"));
        Data result = serviceUnderTest.performAction();
        assertEquals("Mocked Data", result.getContent());
    }
}
```

In this example, `ExternalAPIClient` is a dependency of `MyService`. By using Mockito, we can dictate what `fetchData()` method of the `ExternalAPIClient` should return without making an actual call to the external API. This ensures that the test remains fast, reliable, and independent of external factors.

While mocking is a key testing tool, its limitations must be recognized. Overuse of mocks can create a false sense of security since they don't fully replicate real-world dependency behaviours, potentially leading to tests that succeed in isolation but fail during integration. Excessive reliance on mocking can also disconnect tests from real scenarios, reducing their effectiveness in identifying integration and system-level issues. Thus, while mocking is beneficial for specific testing scenarios, it should be complemented with other testing strategies for thorough and realistic coverage.

Leveraging Jakarta EE Testing Features

Jakarta EE offers several testing features. Notably, the Jakarta CDI specification provides annotations like `@Alternative` and `@Specializes` that can be used to swap out specific components during testing phases. The `@Alternative` annotation allows you to define alternate bean implementations for testing, enabling the simulation of various scenarios or the replacement of unsuitable production dependencies, such as external services. This flexibility is crucial for effective unit and integration testing, allowing a focus on specific application aspects in a controlled environment.

Furthermore, the `@Specializes` annotation extends the benefits of `@Alternative` by allowing a subclass to replace and enhance a bean's functionality. This is particularly useful for modifying component behaviour in specific test contexts, providing granular control over the application's components during testing. These features help you to create adaptable and robust testing contexts, ensuring the resilience and maintainability of your Jakarta EE applications.

Managing Time and Date Challenges

Dealing with time and date in tests presents unique challenges, especially in the context of enterprise applications that often operate across different time zones. A common pitfall is the inadvertent introduction of time zone dependencies in tests, which can lead to inconsistent test results depending on the environment in which they are run. For example, a test that passes in a developer's local time zone might fail in a continuous integration environment located in a different time zone.

Java SE's Date and Time API, introduced in Java 8 and enhanced in subsequent iterations of the language, offers robust solutions to these challenges. This API provides classes like `ZonedDateTime`, `OffsetDateTime`, and `Instant`, which are designed to handle time zones explicitly. For instance, in a global application where you need to schedule tasks based on users' local times, you can use `ZonedDateTime` to ensure that the time calculations are accurate regardless of the server's time zone.

Another example is daylight saving time transitions, which can cause issues in scheduling and time calculations. The Java Time API handles these transitions smoothly, avoiding common pitfalls like missing or duplicated times due to daylight saving shifts.

Moreover, for tests involving fixed dates and times, you can use `Clock` and `Instant` classes to create a fixed clock that ensures your tests run with the same time reference, regardless of the real-world time and date. This approach is particularly useful for tests involving expiration dates, scheduling, or any functionality where time is a critical factor.

With these capabilities of the Java SE Date and Time API, you can write tests that are resilient to time zone differences and daylight-saving transitions, thus enhancing the predictability and reliability of your Jakarta EE application tests.

Integration Testing with TestContainers

In the realm of Jakarta EE application testing, TestContainers play a pivotal role, especially in scenarios involving Jakarta Persistence (formerly JPA) entity testing and broader integration testing. This Java library allows the creation of lightweight, disposable instances of databases or other services within Docker containers, making it a valuable asset for both JPA testing and more extensive integration testing.

For Jakarta Persistence entity testing, TestContainers provide an authentic testing environment by simulating real database interactions. They enable each test to interact with a fresh, isolated instance of the actual database, closely resembling production scenarios. This ensures that JPA tests are not only consistent but also robust and representative of real-world database operations, which is vital in enterprise applications with complex database interactions.

Expanding its utility beyond Jakarta Persistence testing, TestContainers also excel in broader integration testing scenarios. It allows you to programmatically create and manage Docker containers, which can include real instances of databases, message brokers, and other services typically used in production environments. This feature is particularly advantageous for testing microservices or other distributed components that interact with various external systems. For example, in a microservices architecture, you can use TestContainers to launch individual services and their dependencies, thereby enabling end-to-end testing in an environment that mimics the actual deployment setup.

Moreover, TestContainers' ability to start and stop these services within test lifecycles provides a high degree of control over the testing environment. Each test can run in a clean, isolated setup, significantly reducing the risk of interference between tests and promoting more accurate and reliable results. This approach is particularly beneficial when testing complex workflows or scenarios that span multiple services or components.

By utilizing TestContainers for both JPA entity testing and broader integration testing, you can achieve a high level of confidence in the behaviour and performance of your Jakarta EE applications. TestContainers' versatility in handling a variety of testing scenarios makes it an essential tool in the modern Java developer's toolkit, ensuring that your applications are not only thoroughly tested but also ready for the challenges of real-world deployment.

Standardizing the Build Process

Establishing a standardized and repeatable build process is fundamental for creating predictable tests in Jakarta EE applications. Consistency in the build environment ensures that tests are executed under the same conditions each time, leading to more reliable and repeatable test results.

To achieve this, build tools like Maven or Gradle should be meticulously configured. These tools play a crucial role in automating the build process and ensuring that all dependencies are consistently managed. For instance, using Maven, you can specify exact versions of all dependencies in your `pom.xml` file. This practice prevents discrepancies that may arise from using different versions of libraries or tools, which can lead to unpredictable test outcomes.

Moreover, integrating a tool like Maven's Enforcer Plugin can further enhance consistency. This plugin can be used to enforce specific build rules, such as requiring certain Maven versions or dependency convergence. Such enforcement ensures that all team members and CI environments use the same build configurations, further reducing the risk of inconsistencies.

Additionally, consider defining specific profiles for different stages of development. For instance, a test profile in Maven can include specific configurations or dependencies that are only relevant for testing. This separation of concerns leads to cleaner, more focused builds, and ensures that test-specific configurations do not interfere with the production build.

By rigorously standardizing the build process with tools like Maven or Gradle, you can minimize the variability your testing environments. This approach is crucial for achieving predictability and reliability in tests, which in turn leads to more stable and trustworthy Jakarta EE applications.

Integrating Tests in Continuous Integration (CI) Frameworks

Continuous Integration (CI) frameworks are crucial in the realm of Jakarta EE application development, especially when it comes to maintaining the reliability and effectiveness of testing strategies. The absence of CI frameworks can have a profound impact on the software development lifecycle, often leading to delayed detection of issues and increased integration failures.

Without integrating tests into a CI framework, tests are typically run in a less controlled, more ad hoc manner. This approach can result in several challenges:

- **Inconsistency:** Tests run on your developers' local machines may pass due to environment-specific configurations that don't reflect the production environment. This inconsistency can lead to bugs that only surface in production or during later stages of development, which are more costly and time-consuming to fix.
- **Delayed Issue Detection:** When tests are not automatically and routinely executed as part of a CI pipeline, issues can go unnoticed for longer periods. This delay can lead to the accumulation of more complex, interconnected problems, making them harder to diagnose and resolve.
- **Integration Challenges:** In the absence of CI, merging code changes from different team members can become a risky process. Without regularly running tests against new code integrations, there's a higher chance of introducing integration bugs, which can disrupt the development workflow and delay releases.
- **Reduced Developer Confidence:** Developers may lack confidence in the stability and quality of the application if tests are not consistently run and validated. This lack of confidence can slow down development processes, as more time is spent on manual testing and validation.

On the other hand, incorporating tests within a CI framework brings numerous benefits:

- **Early Detection of Issues:** Regularly running tests in a controlled CI environment allows for early detection of bugs and integration issues, enabling quicker and easier fixes.
- **Standardized Testing Environment:** CI frameworks provide a consistent, standardized testing environment that closely mimics the production setting, leading to more reliable test results.
- **Streamlined Development Process:** With CI, developers can confidently merge code, knowing that tests will be automatically run to validate the integration, thus speeding up the development process.
- **Increased Software Quality:** Regular and automated testing within CI ensures continuous monitoring of the application's health, leading to higher overall software quality.

The integration of tests into Continuous Integration frameworks is not just a best practice but a critical component of a successful Jakarta EE development strategy. It ensures consistent, reliable, and efficient testing, which is fundamental in building robust and high-quality enterprise applications.

Keeping Test Documentation Updated and Regularly Reviewed

Documenting and reviewing test cases is a vital aspect of maintaining the integrity and relevance of your Jakarta EE applications. Effective documentation and review processes ensure that your testing strategies evolve alongside your application, maintaining their accuracy and effectiveness. Here are some strategies and tips for managing your test documentation and review processes:

- **What to Document:** For each test case, document its purpose, the specific conditions it tests, expected outcomes, and any special setup or teardown processes. Include information about the test environment and any dependencies or mock objects used. If a test case is complex or non-intuitive, add comments to clarify the logic.
- **Regular Reviews:** Establish a regular schedule for reviewing test cases. A good practice is to review them:
 - When significant changes are made to the application's codebase.
 - After the completion of a major development milestone.
 - Periodically, such as every quarter, to ensure they still align with the current application state and requirements.
- **Review Process:** During reviews, validate that each test case:
 - Is still relevant and necessary.
 - Accurately reflects the current functionality and requirements of the application.
 - Has up-to-date documentation.
- **Peer Reviews:** Encourage peer reviews of test cases. Different perspectives can help identify potential flaws or areas for improvement that the original author may have overlooked.

- **Automate Where Possible:** Utilize tools that can automate aspects of test documentation, such as generating reports of test coverage or automatically documenting the results of each test run.
- **Maintain a Change Log:** Keep a log of changes made to test cases, including the reason for the change and the date it was made. This historical record can be invaluable for understanding the evolution of your test suite.
- **Align Test Reviews with Code Reviews:** Integrate test case reviews with code reviews. When a new feature is added or a bug is fixed, the corresponding test cases should be reviewed and updated as part of the same process.
- **Leverage Version Control:** Store your test cases in a version control system alongside your application code. This practice not only helps in tracking changes over time but also ensures that tests are always in sync with the code they are testing.
- **Training and Knowledge Sharing:** Regularly train team members on the importance of test documentation and the specific processes your team uses. Sharing knowledge about effective testing strategies and documentation practices can help maintain high standards across the team.

These strategies can help ensure that your test documentation remains clear, relevant, and useful, ultimately contributing to the overall quality and reliability of your Jakarta EE applications. Remember, well-documented and regularly reviewed test cases are key to understanding and verifying the behaviour of your application, making them an indispensable part of the development lifecycle.

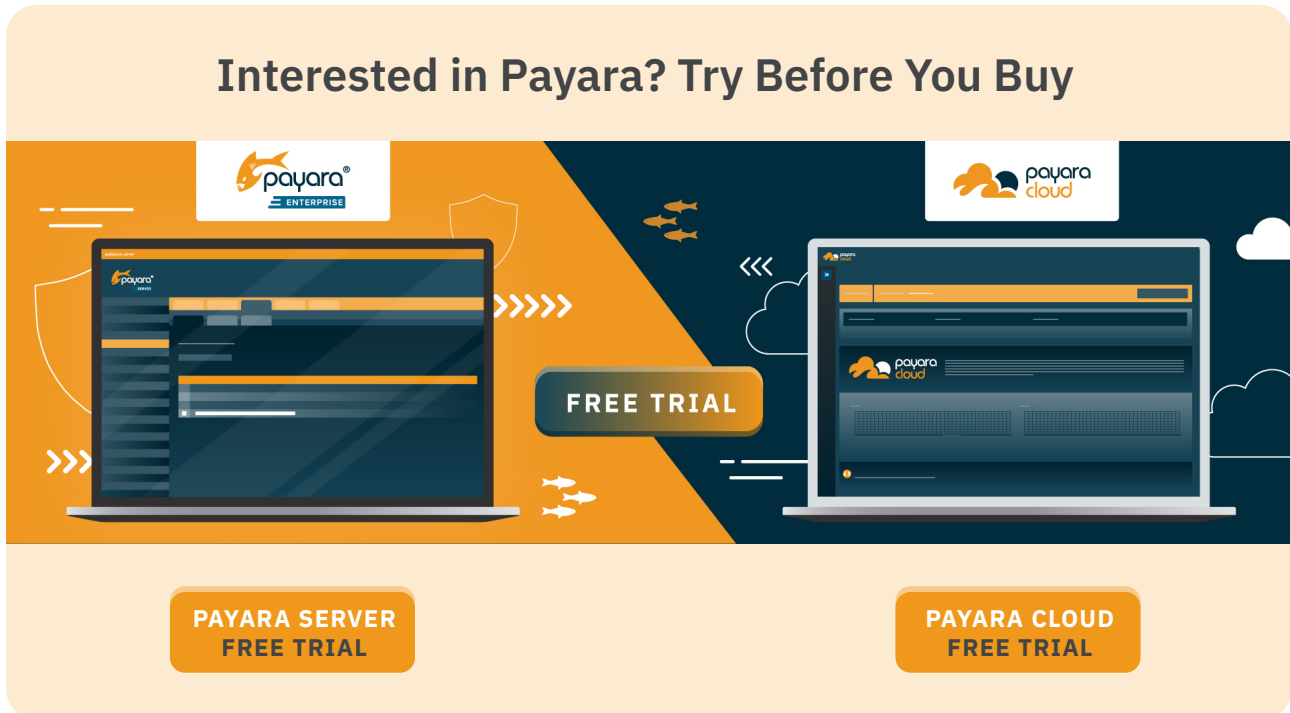
Conclusion

Creating predictable and reliable tests in Jakarta EE applications is an attainable goal that significantly enhances the quality and stability of your software. By isolating the test environment, leveraging the power of mocking judiciously, utilizing Jakarta EE's testing features, ensuring consistency in Jakarta Persistence testing with TestContainers, managing time and date challenges effectively, standardizing the build process, integrating tests into Continuous Integration (CI) frameworks, and maintaining up-to-date documentation and regular reviews of test cases, you can build a robust testing framework that stands the test of time and change.

The key to success in testing lies in the balance – between using mocks and real instances, between isolation and integration testing, and between automated and manual processes. It's crucial to continually evaluate and adapt your testing strategies to align with the evolving needs of your applications. The predictability and reliability of tests are not just about getting consistent results; they are about ensuring that these results meaningfully reflect the real-world performance and stability of your applications.

We encourage developers and teams to implement these practices and explore further resources to enhance their understanding and skills in Jakarta EE testing. Remember, every effort made towards improving the testing process is a step towards building more stable, reliable, and trustworthy software solutions. Embrace these practices and watch as they transform your development workflow and the quality of your Jakarta EE applications.

Interested in Payara? Try Before You Buy



The graphic illustrates the transition from Payara Server to Payara Cloud. On the left, a laptop displays the Payara Server interface with the 'payara ENTERPRISE' logo. On the right, a laptop displays the Payara Cloud interface with the 'payara cloud' logo. A central orange button labeled 'FREE TRIAL' is positioned between the two laptops. Arrows indicate a flow from the server to the cloud. The background is split into orange and dark blue sections with fish icons.

**PAYARA SERVER
FREE TRIAL**

**PAYARA CLOUD
FREE TRIAL**



sales@payara.fish



**UK: +44 800 538 5490
Intl: +1 888 239 8941**



www.payara.fish

Payara Services Ltd 2023 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ