INTERSECT / DApp Testing Strategy - Reference Doc

Draft, v0.1

In attn GovTool / March 19th, 2024

A. Best Practices

1. JavaScript Testing

1. Follow the AAA Pattern: Structure your tests into three clear steps - Arrange (setup your testing objects and preconditions), Act (execute the function to be tested), and Assert (verify that the outcome matches the expectation). This pattern helps maintain a clear structure and makes tests easier to understand.

2. Use Declarative Style: Favor a declarative approach to writing tests, focusing on describing what should happen, rather than how (imperative) with detailed steps and conditions. Declarative tests are easier to read and maintain.

3. Emphasize Black-Box Testing: Concentrate on testing the public interfaces of modules or components without considering the internal workings. This approach ensures that tests validate the expected behavior from an external perspective, maintaining a focus on the user experience.

4. Utilize Test Doubles: Incorporate test doubles like dummies, fakes, mocks, and stubs to simulate and test interactions with other components. Prefer using stubs and spies over mocks to simplify tests and focus on behavior rather than implementation.

5. Use Realistic Input Data: Instead of using arbitrary values like “foo” or “bar,” use realistic data for more accurate and meaningful tests. This approach helps in uncovering issues that might only surface with valid input scenarios.

6. Implement Property-Based Testing: Employ property-based testing to evaluate your code against a broad set of inputs, enhancing the test coverage by checking for invariants and behaviors across many input combinations.

7. Adopt Snapshot Testing Judiciously: Snapshot tests are useful for components that rarely change, capturing and comparing the rendered output over time. Use inline snapshots sparingly to keep tests manageable and focused.

8. Expect Errors Explicitly: Design tests to expect errors as outcomes, rather than catching them. This method ensures that error handling is actively tested and not merely circumvented.

9. Tag Tests for Organization: Apply tags (e.g., #cold-test, #sanity) to categorize tests, facilitating selective test execution and clearer organization.

10. Categorize Tests in Layers: Organize tests into at least two levels (e.g., unit, integration) to enhance the clarity and comprehensiveness of test reports.

11. Prefer Teardown in After-All Hook: Conduct resource teardowns (like database connections) after all tests have run in a suite for more efficient test execution, especially in multi-process environments.

12. Isolate Test Data: Avoid using global fixtures and seeds; instead, create or add data on a per-test basis to prevent tests from affecting each other, leading to more reliable and independent tests.

13. Utilize Selectors for DOM Interactions: Use specific selectors (e.g., data-testId) rather than generic text, class, or tag selectors, for more stable and reliable DOM element targeting.

14. Test with Realistically Rendered Components: Ensure components are tested in a fully rendered state to mimic real-user scenarios closely, improving the accuracy of the tests.

15. Avoid Using Sleep for Asynchrony: Instead of using sleep functions to wait for asynchronous operations, leverage the testing framework's built-in support for handling async events to avoid flakiness and improve test reliability.

16. Stub Out Flaky and Slow Resources: When possible, stub out resources that are slow or prone to flakiness to ensure tests run reliably and quickly.

17. Optimize End-to-End Tests: Speed up end-to-end tests by reusing login credentials and other session data where possible, reducing test execution time.

18. Implement a Smoke Test for Site Navigation: Have at least one end-to-end smoke test that navigates through the site map to ensure basic functionality across the application.

Measuring Test Effectiveness:

- Coverage Testing: Assess the extent to which your tests cover the codebase, identifying untested paths.

- Mutation Testing: Introduce changes (mutations) to the code and check if the tests catch these modifications, validating the effectiveness of your tests.
- Utilize Test Linters: Apply linters specifically designed for tests to enforce best practices and identify potential issues early.

B. Haskell Testing

Adopt a dual approach for testing:

- Utilize both Unit Testing and Property-Based Testing.
- Store tests in the root directory under /test or /tests to maintain a clear distinction from your primary code.

Organize test files meticulously, ensuring they're separate from your main code. Align the structure of your test modules with the structure of your application modules. For example, tests for the Plutus.V1.Ledger module should ideally reside in Plutus.V1.Ledger.Tests. For broader tests that span multiple modules, consider housing them in a unified module, such as Plutus.V1 for overarching Plutus V1 tests.

Employ specific Test Frameworks tailored to your testing needs: Use HUnit for unit tests, QuickCheck for property-based tests, Tasty to orchestrate and execute tests, Hspec for writing and conducting tests efficiently, and HPC to track code coverage.

When leveraging Testy, ensure that each Tests module exposes a tests:: TestTree value.

For functionalities entailing IO/Network actions, aim to segregate the IO element from the pure logic to the fullest extent. Apply strategies like monadic composition to delineate responsibilities, thereby simplifying the testing process.

In Property-Based Testing for Invariants, ensure the consistent validity of invariants across varied inputs using property-based testing. For instance, an essential invariant for an insertion sort function is maintaining the sorted segment of the list in order after every iteration.

For property-based testing:

- Define clear and concise properties that capture specific behaviors or invariants your program should adhere to, with each property focusing on a singular aspect of your code's behavior.
- Employ a naming convention like prop_<...> for properties.
- Leverage Haskell's robust typing system to represent properties as type constraints, minimizing errors.
- Utilize generators judiciously to ensure adequate randomness and comprehensive coverage of potential edge cases, avoiding hardcoded values.
- Implement a shrinking mechanism to simplify failing test cases.

Adhere to standard test naming practices similar to those in other programming languages:

- Choose descriptive names for tests.
- Maintain a consistent naming convention.
- Use descriptive function names and include context.

Regularly refactor tests as necessary.

B. DApp Types

In the context of this document, DApps are defined as any application that interacts with the blockchain. The interaction with the blockchain can be reading from the chain or writing to the chain by composing transactions. To compose transactions, the application likely requires access or interface to the wallet or private key material.

Based on the access to the wallet, we classify DApps into the following four categories:

A. DApps reading data from the chain
B. DApps interacting with the wallet via CIP defined interfaces - Read Only
C. DApps interacting with the wallet via CIP defined interfaces - Read/Write
D. Custodian DApps holding private key material

A. DApps reading data from the chain

These applications read data from the chain either in raw form or pre-processed form using external applications like DbSync, Chain Indexers, APIs etc and represent the data in the user interface.

B. DApps interacting with the wallet via CIP defined interfaces - Read Only

These applications interact with the wallet, usually browser extensions, and fetch data associated with the wallet. The data could be different accounts, stake keys, transactions etc. The applications that use wallet to login to the DApp also fall under this category.

C. DApps interacting with the wallet via CIP defined interfaces - Read/Write

These applications interact with the wallet and actively compose transactions of some sort and involve signing transactions or data using the wallet private key.

D. Custodian DApps holding private key material

These applications hold access to the private key material, thus being able to compose/sign transactions on behalf of the associated wallet. These applications usually could be backend or back office applications and are almost always a major security risk.

Different tests will be applicable for different categories of applications. The following section includes the different testing strategies for applications.

C. Different Tests

1. Unit Testing

Purpose:
To test individual components or functions in isolation to ensure they work as expected.

Common Tools:
Jest/Mocha etc. for Javascript-based projects, JUnit for Java-based projects, PyTest for Python projects, and other language-specific frameworks.

Responsible:
Dev Team

DApp Category:
A, B, C, D

Approach to Testing:
The developer writing the code for feature or bug fix is responsible for writing the unit test. The logic of the test would depend on the functionality being developed. However, the structure usually follows: Arrange, Act and Assert pattern.

For the dependencies, mocks are used.

Acceptance Criteria:

- The code coverage shows that the code is covered by the test either fully (100%) or most achievable.

2. Integration Testing

Purpose:
To verify that different modules or services work together as intended.

Approach:
This involves testing APIs, database interactions, and other integration points.

Responsible: Dev Team and QA Team

DApp Category:
A, B, C, D

Approach to Testing:
The developer working on the feature is responsible to make sure that developed feature works well with the rest of the system. If the system is complex with multiple modules and is not possible to cover, then the separate QA team may be responsible for testing the system.

Acceptance Criteria:

- Module/Component interfaces and data transfer objects are tested to make sure data exchange format is not broken
- Testing of the system with minimal dependencies is working. This could include partial mocks of dependencies as well.

3. Functional Testing

Purpose:

- To check that the software meets the specified requirements and behaves as expected in scenarios close to real-world use.
- *DApp Front-End Testing*: Ensuring compatibility across different browsers and devices, and testing the integration with various blockchain wallets and extensions.

Common Tools:
Playwright, Cypress and Selenium like tools are useful for these tests.

Responsible:
Dev Team and QA Team

DApp Category:
A, B, C, D

Approach:
Automated scripts are used to perform these tests, simulating user interactions with the DApp. The testing of the user stories are covered here.

For some complex scenarios, it may be required to perform manual testing and/or verification.

Acceptance Criteria:

- User stories are well written and cover the use cases well.

4. Performance and Scalability Testing

Purpose:
To ensure that the software performs well under expected and peak load conditions.

Common Tools:
Tools like Gatling and Locust are often for performance testing.

Responsible: Dev Team and QA Team

DApp Category: All

Approach:
Performance and scalability tests are done in specific environments. These tests are run against the application running on such test environments. The tools like Gattling and Locust are used to simulate the user behavior and the measurement of the system is recorded.

Performance tests measure the performance of the system by recording different metrics like API response time, time to load etc under different load conditions.

Scalability tests measure how well the system can scale to accommodate the different load (eg. concurrent users or connections) and still provide expected quality of service.

5. Security Testing

Purpose:
To identify vulnerabilities and ensure that the software is secure against attacks.

Approach:
This includes regular security audits, using tools for static and dynamic analysis, and adhering to security best practices in development.

Responsible:
Dev Team, Code Quality Analysis Tool and QA/Security Team

DApp Category:
A, B, C, D

Approach:

- Setup the Static Code Analysis Tools like SonarCloud
- Setup Quality Gate (Security Hotspots and vulnerability) and run the analysis to know whether or not the quality gate passes. Fix any reported issues.

- Setup dependency scanning on the codebase. Eg. Enable Dependabot on the repository.
- Add additional security tests below that are application specific.

Here is a list of additional security tests to do for DApp

5.1. Test for Overflows, Valid/Invalid Data, and Headers

- **Objective:** Ensure system stability and data integrity.
- **Method:** Inject extreme, unexpected, or boundary data.
- **Expected Outcome:** System should handle the data without crashing, corrupting data, or behaving unexpectedly.

5.2. DoS and Rate Limits

- **Objective:** Validate the application's resilience to denial-of-service attacks.
- **Method:** Simulate a flood of requests to the system.
- **Expected Outcome:** The system should not crash and should block or delay suspicious traffic.

5.3. Cross-Site Scripting (XSS)

- **Objective:** Ensure the application is resistant to script injection attacks.
- **Method:** Attempt to run scripts in fields and parameters.
- **Expected Outcome:** Scripts should not be executed; instead, they should be sanitized or blocked.

5.4. SQL Injection

- **Objective:** Ensure that the backend database is immune to injection attacks.
- **Method:** Introduce malicious SQL code into input fields.
- **Expected Outcome:** The system should sanitize inputs and prevent unauthorized database access.

5.5. Cross-Site Request Forgery & Server-Side Request Forgery

- **Objective:** Ensure the system is safeguarded against unauthorized actions on behalf of an authenticated (wallet) user (Ada holder or DRep) and prevent from making requests to the backend.
- **Method:** Simulate fake requests to the system.
- **Expected Outcome:** The system should block or deny such requests.

6. Regression Testing

Purpose:
To make sure that new changes don't adversely affect existing functionality.

Approach:
Automated test suites are run to detect regressions as new code is integrated.

Responsible: First, CI/CD workflow should detect this then QA Team

DApp Category:
B, C, D

7. Cross-DApp and Interoperability Testing

Purpose:

- Ensuring seamless operation across related DApps and services. Making sure the implementation is based on the standards rather than DApp specific implementation, thus avoiding interoperability issues.
- Testing the DApp's ability to interact with other DApps, smart contracts, and services across blockchain ecosystems.

Approach:
Testing with multiple implementations if available. If the DApp is following certain standards (CIP), then it is important to test the DApp with related applications. An approach to ensure is to follow producer/consumer templates.

Your DApp could produce the data or transaction that could be consumed by related other DApp, and vice versa.

Another important component to test is integration with the wallet. Always being aware of specific implementations in wallet providers. DApp working on one wallet may not work on

other wallets. Therefore, testing on different wallets ensures greater compatibility and awareness of known issues.

Acceptance Criteria:

- Producer/Consumer tests work on related DApps
- DApp works on multiple wallet providers

DApp Category: B, C, D

8. Cross-chain Integration Testing

Purpose:

- Verifying the DApp's interaction with blockchain data and transactions, including testing across different networks (mainnet, testnets), gas usage optimization, transaction processing, and error handling.
- Especially important when there are major language upgrades on smart contracts. Compiler version difference, node compatibility can make DApp to fail working. Imagine users not being able to redeem transactions.
- Identifying logic issues because of different protocol parameters on different chains early on.

Approach:

- Plan, prepare and select tests to run on different network chains. It might not be possible to run all tests on all networks.
- Understand the difference between different networks, eg. protocol versions, parameters, compiler versions, supported primitives
- Check if defensive programing and fallback strategies are implemented and do not lead to crash of the DApp or unexpected scenarios.

Acceptance Criteria:

- DApp deployment is successful in multiple relevant network chains
- Functional tests pass on those deployments.

Responsible: QA Team + DevOps Team

DApp Category:

C, D

D. Test Strategy Implementation

Test Selection and Execution Strategy

1. Select Which Tests Make Sense for Your Project

The selection of tests is foundational to your testing strategy. It involves identifying the types of tests that are most beneficial for your project based on its size, complexity, and the criticality of various features.

2. Define When to Run Which Tests

Determining the appropriate timing for each type of test is crucial for optimizing the development workflow and resource utilization. Consider automating the execution of tests based on specific triggers:

On Pull Request (PR) Open: Run unit and integration tests to catch issues early before merging code changes.

On Merge: After code is merged into the main branch, run a broader test suite, including system and E2E tests, to ensure the integrity of the application.

Scheduled Nightly Builds: Utilize off-peak hours to run extensive testing, including performance and security tests, to minimize disruption and allocate time for analysis.

3. Define Which Environment to Run Those Tests On

Selecting the right environment for your tests is important for ensuring the validity and relevance of the test outcomes. Different environments can simulate various stages of the

deployment pipeline, offering insights into how the application behaves under different conditions.

Developers Local Machine: Ideal for running unit tests during the development phase to catch issues early without requiring network resources.

Deployed Dev Environment: Use this environment for initial integration tests. It's a shared environment that closely mirrors production settings but is used for development.

Deployed Staging Environment: This environment is a near replica of the production environment and is used for running E2E, performance, and security tests. It's crucial for final validation before production deployment.

Deployed Production Environment: Running tests in production, such as canary testing or A/B testing, can be beneficial for validating real-world user scenarios and performance under live conditions. However, this requires careful planning to avoid disrupting the user experience and is not considered under the scope of this document.

Test Development

This step focuses on creating a structured approach to developing and refining your test suite. By defining the scope, writing tests, measuring metrics, and iterating, you can build a comprehensive test suite that effectively assesses the quality and performance of your DApp.

1. Define Scope of Tests and Metrics

Before diving into test writing, it's crucial to outline the scope of your testing efforts and the metrics you will use to measure success. This step involves:

Identifying Test Objectives: Determine what you aim to achieve with your testing. Are you focusing on unit testing, integration, functionality, performance, security, or a combination of these?
Selecting Key Metrics: Choose metrics that best represent the outcome of the Test. This could include code coverage, bug density, load times, security vulnerability counts, and more.
Prioritizing Features and Components: Not all parts of your application will require the same level of testing. Identify critical features that need more rigorous testing based on their complexity or importance to the overall system.

2. Write Test or Scaffolds to Expand Upon Later

With a clear scope in mind, start writing your tests. At this stage, it's beneficial to create test scaffolds for your entire scope, which you can expand upon iteratively.

This includes:
Creating Test Cases: Write test cases that cover identified objectives and features. Ensure each test is focused on a single aspect or behavior to simplify debugging.
Implementing Test Scaffolds: For more complex features, implement test scaffolds – basic structures of tests that outline how tests will be conducted without fully implementing the test logic. This allows for early planning of how tests will interact with the codebase.

3. Measure the Intended Metrics

As you develop your tests, continuously measure the metrics you've identified as indicators of success.

This iterative measurement helps in:

Evaluating Test Effectiveness: Use the metrics to assess how well your tests are covering the intended objectives. This can inform adjustments in your testing strategy.
Identifying Gaps in Coverage: Metrics like code coverage can highlight areas of your application that are not adequately tested, directing your efforts where they are most needed.

4. Repeat Until All the Tests Are Developed

Developing a comprehensive test suite is an iterative process. Continue through these steps, expanding upon your test scaffolds, refining existing tests, and measuring outcomes until you've developed a full suite of tests that meets your project's needs.

This process involves:
Refinement and Expansion: Based on the metrics and outcomes of initial tests, refine and expand your tests to increase coverage, enhance precision, and address any identified weaknesses.
Continuous Integration: Integrate testing into your development process, running tests automatically on code commits to detect issues early and often.
Feedback and Adjustment: Use feedback from test results to continually adjust your testing scope, priorities, and strategies to align with evolving project goals and challenges.

Test execution and Automation

The Test Execution and Automation phase ensures that tests are run efficiently and consistently to maintain high-quality standards.

1. Automate the Test Execution

Automating test execution involves setting up systems that automatically run your tests at specified triggers or intervals, reducing the need for manual intervention and ensuring that tests are consistently executed.

Using CI/CD - GitHub Actions: Integrate your testing suite with GitHub Actions within your CI/CD pipeline. Configure actions to run tests on every pull request, commit, or merge to main branches. This ensures that changes are automatically tested, helping catch and resolve issues early in the development cycle.

Scheduled Cron Jobs: For tests that do not need to be run on every change (like some integration or performance tests), set up scheduled cron jobs. These can run tests at off-peak hours or at specific intervals, ensuring regular validation without disrupting ongoing development work.

Other Scripts: Utilize custom scripts for more complex testing scenarios that might not fit directly into CI/CD pipelines or cron jobs. Scripts can handle tasks like setting up test environments, running tests in a specific sequence, or cleaning up after tests are completed.

2. Measure the Intended Metrics

Once test automation is in place, the next step is to measure key performance indicators (KPIs) or metrics to evaluate the effectiveness of your testing efforts. This could include metrics such as test coverage, pass/fail rates, time taken to run tests, and the number of bugs uncovered.

3. Report the Measured Metrics to a Reporting Server

Automated reporting of these metrics to a central reporting server or dashboard facilitates real-time monitoring and analysis. It enables teams to quickly identify trends, pinpoint areas of concern, and make informed decisions on where to focus their testing and development efforts for improvement.

4. Repeat Until All the Tests Are Automated

Test automation is an iterative process. Start with automating the most critical tests to get quick wins and gradually expand to cover more areas. Continuously evaluate the effectiveness of your automation, adding new tests as features evolve and refining existing tests to enhance coverage and efficiency.

Reporting Dashboard

Reporting Dashboard helps for maintaining transparency and facilitating quick decision-making based on test outcomes.

Visibility of Test Results and Metrics

Ensure that your reporting dashboard is set up to display comprehensive test results and key performance metrics prominently. This should include pass/fail statuses, coverage reports, performance benchmarks, and any custom metrics relevant to your project. The goal is to make these insights accessible and actionable for the development team, QA team, and stakeholders.

Integration with Issue Reporting System

Implement a mechanism to automatically report failing tests to your issue tracking system, such as GitHub Issues. This ensures that every test failure is promptly documented as an issue, assigned a priority, and routed to the appropriate team

member for investigation and resolution. Automating this process helps in reducing the manual overhead and accelerates the feedback loop between detecting and addressing defects. Automating might not always be desirable though, make decisions wisely based on your DApp expectations on quality.

Community Testing Initiatives

Leveraging community testing initiatives can significantly enhance your project's quality by tapping into diverse perspectives and expertise.

Contribution Guidelines and Disclosure Forms
Clearly define how external contributors can participate in testing your project. This includes detailed contribution guidelines that outline the process for submitting test cases, reporting bugs, and the criteria for valid submissions.

Additionally, incorporate disclosure forms to manage expectations regarding privacy, intellectual property, and the handling of sensitive information.

Reward and Recognition Mechanisms
Establish a system of rewards and recognition to motivate community involvement in testing activities. This could range from monetary rewards, swag, public acknowledgment, or privileged access to future project features for notable contributions.

E. Bug Tracking and Reporting

A bug is a deviation from the intended behavior of the system. Sometimes, it is referred to as an issue or defect. A mechanism should be in place to detect, report and fix the bugs.

Steps in Bug Tracking and Reporting

1. Identification

Bugs can occur in any stage of the software development. Everybody involved in the projects including the users should be encouraged to report them if encountered.

For effectiveness and to avoid duplicated work or wasted effort, conducting tests to identify bugs should be done only when it is marked as ready for test.

2. Documentation

Once a bug is identified, it is important to document it thoroughly.

This documentation should include a detailed description of the bug, steps to reproduce it, the severity level, the environment in which it was found, and any other relevant information that can help in its resolution.

3. Prioritization

Bugs are prioritized based on their severity, impact on the system, and the complexity of the fix. This helps in managing the resolution process efficiently, ensuring that critical issues are addressed first.

4. Assignment

After prioritization, bugs are assigned to the appropriate development team or team member for resolution based on their area of expertise.

5. Resolution and Verification

The assigned developer works on fixing the bug. Once resolved, the fix is verified by the QA team to ensure that the defect has been adequately addressed and that there are no side effects.

Using GitHub for Bug Tracking and Reporting

GitHub has built-in features like Issues, Projects, Labels, and Milestones to establish an effective workflow for identifying, documenting, prioritizing, and tracking bugs. GitHub offers a collaborative space that supports the entire bug management lifecycle from report to resolution.

Here's a proposed approach using GitHub:

Creation: Initiate by creating a new Issue for each bug found during testing. Provide a descriptive title and a comprehensive comment detailing steps to reproduce the bug, expected versus actual outcomes, screenshots if relevant, and any additional pertinent details.

Maintain clarity and efficiency by ensuring all team members adhere to a standardized format when documenting issues. A template is attached in the Appendix for GitHub bug report.

Labels: Employ Labels to categorize issues based on severity (e.g., Critical, High, Medium, Low), type (e.g., bug, enhancement, documentation), or any other project-specific classifications you might need.

Assignees: Assign the issue to a team member tasked with resolving the bug. GitHub allows for multiple assignees, facilitating collaborative efforts in fixing issues.

Projects: Use Projects within GitHub to arrange issues and pull requests. The Kanban board feature enables visual tracking of progress. Shift issues across columns like To do, In progress, and Done to reflect their current state.

Milestones: Compile related issues and pull requests under Milestones to monitor progress towards specific objectives or project phases. Milestones help in setting deadlines and prioritizing tasks for upcoming releases.

Communication: Utilize the comment section of each Issue for ongoing discussion about the bug. Team members can post updates, request further details, and discuss resolution strategies. Mention (@username) specific team members to alert them to an issue or to seek their feedback. Within issue comments or descriptions, use GitHub's task lists to outline steps for bug resolution.

Review and Merge with Pull Requests: After a bug fix has been implemented, the changes should be committed to a branch and a Pull Request (PR) created. Link the PR to the corresponding issue by referencing the issue number in the PR description (e.g., "Fixes #123").

Perform code reviews within the PR to ensure the bug is adequately addressed and that the fix does not introduce new issues.

Once the PR is approved through review, merge it into the main/master branch. If the "Fixes #issue" syntax was used, the linked issue will automatically close.

Track and Report with Insights and Analytics

Access GitHub's Insights tab to monitor activities on issues and pull requests, assess contribution patterns.

Generate reports based on Labels, Milestones, or Assignees to track bug resolution progress and identify areas needing focus.

-
-

Appendix Bug report template:

---

name: Bug Report

about: Create a report to help us improve

title: '[BUG] Short, Descriptive Title'

labels: bug

assignees: ''

Issue Description

<!-- A clear and concise description of what the bug is. -->

Steps to Reproduce

<!-- Detail the steps required to reproduce the issue (if applicable). -->

1.

2.

3.

Expected Behavior

<!-- Describe what you expected to happen. -->

Actual Behavior

<!-- Describe what actually happened. Include screenshots or error messages if possible. -->

Environment

<!-- Please complete the following information: -->

- OS: [e.g., Windows, Linux, macOS]

- Browser (if applicable): [e.g., Chrome, Firefox]

- Version of the project, package, or library:

Possible Solution

<!-- Suggest a fix/reason for the bug, if you have one in mind. -->

Additional Context

<!-- Add any other context about the problem here. This might include logs, specific situations where the issue occurs more frequently, etc. -->