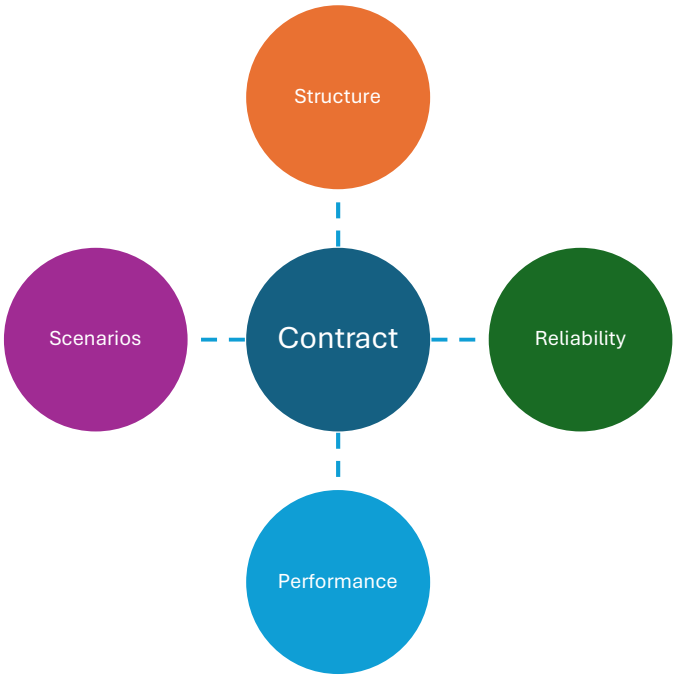
Heuristics for Testing Software Dependencies

Software systems are built of multiple components that depend on each other. Some of those dependency relationships may be complex, integrating libraries, modules, extensions, applications, or services outside control of the team building the system. This document offers a set of heuristics, guiding ideas to inspire test design, creation and product assessment.



This document breaks the heuristics into the following categories:

Contract: The rules of usage between dependencies with regard to interfaces, behavior, data and how to check for violations of contract, regression, and explore affects of those interactions.

Structure: Architecture and design of the dependency as it relates to the overall system and impact that has on testing approach.

Reliability: Ways in which dependencies affect overall system reliability and ways to examine and explore that affect.

Performance: Impact of the dependency on system performance in terms of speed, throughput, capacity and ways to explore the impact.

Scenarios: How usage of the system is affected by and affect multiple dependency relations and ways to understand the impacts on the system and end user.

Contract

Perform Evaluation: Create testing suite to evaluate dependencies as suitable for use.

Create Regression Coverage: Establish testing is in place to verify that dependent components respect their contracts (e.g., APIs), cover any areas that are at risk if something changes.

Establish Communication: Establish communication mechanisms with the team building the dependency to discuss breaking changes in dependencies.

Assess Documentation: Assess the API and data contract documentation for thoroughness and sufficiency. Pay close attention to things like missing data, tacit assumptions, errors in data state, expected order of operations, and other behaviors or attributes that go beyond simple functional state.

Analyze Data Definition: Define and analyze all the data structure, schema, and values which might come from or go to the dependency, what they mean, and impact on the system. Build testing data sets and to cover.

Explore Data Contract Rules: Define or map out the rules for data format, syntax, structure, and content coming from and going to the dependency and look for ways those rules may be violated and effect that has.

Error Handling in Contract: Enumerate all expected errors the dependency might emit and define impact of those errors and ways to cover them.

Behavioral Contract: Define the rules for actions and responses coming from and going to the dependency, look for ways those rules may be violated and effect that has and establish mechanisms to test those conditions.

Assess Data Control and Testability: Establish the degree you own and control the data in the dependency and establish whether that level of control or lack of control adds risk to the project. Address issues to bring data and contract behaviors under efficient and effective test control.

Mock Actions and Sequences: Create mocks that permit testing of each action or event the dependency can accept, respond to, or emit in whatever order the test wants to establish. Test for sequence orders that go beyond ones your system assumes will happen.

Mock Data Values: Mock the ability to return data that conforms to the data definition and type described in the contract. Cover as many permutations as possible as defined by the schema rules, going beyond just examples and into implied possibilities. Include missing

items, empty items, multiples of items, different order, references to other items which may or may not match. Include schematically and syntactically valid but perhaps semantically invalid data.

Mock Errors: Mock every know error that the dependency describes in its contract. Explore the dependency behaviors with further testing to find other errors that occur, adding those to your mock. Include implied errors (e.g. HTTP 5xx range) which might not be described in the contract.

Structure

Analyze Overall Dependency Structure: Describe the dependency architecture and where it resides, whether in an external service to your company, external service within your company, same machines and resources as yours, or even built into your code.

Test the Critical Dependency Points: Define the critical dependencies between components, how do they affect system functionality and define tests which target those dependencies.

Test for Tight Coupling: Look for instances where one side of a dependency relies on inner constructs, schema, and behaviors on the other side of the dependency and tries to control them.

Test for Dependency Chains: Assess the dependency for introduction of its own dependencies that might impact your system.

Testing for Cyclic Dependencies: Look for cyclical dependencies that could lead to deadlocks or infinite loops and try to exercise those relationships.

Look for Failures in Setup and Configuration: Determine how to set up and configure the dependency and ways in which that setup and configuration could go wrong.

Look for User Role and Capability Inconsistencies: Explore and cover differences between roles and capabilities of users and other entities in the dependent system and your own.

Look for Connection Management Problems: Determine how connections to the dependency are managed within the environment and your system and which scenarios that connection management impacts and explore ways connection management could go wrong.

Alter and Change Settings and Observe Impact: Make changes in the dependency for things like accounts, settings, behavior, access, storage, capacity and observe system behavior.

Attempt Observation of Dependency Behavior Via Interface, UI, Logs and Diagnostics: Assess how to see what the dependency is doing and whether that is sufficient to understand overall system behavior or diagnose problems.

Test Impact of Customization: Evaluate level of customization of dependency is allowed and which sorts of customizations might impact your system. If dependency does not allow customizations, assess where that may create problems for your system.

Assess Whether Specialized to Purpose Well or if Generalization Introduces Risks: Is the dependency a specialized application that handles exactly our case, or is for a broader use and purpose beyond what it is need it for? Is the dependency a platform meant to extend or host other applications, or is it an application with its own use cases?

Attempt to Replace Dependency or Install Alternate Dependencies: Assess what is required to replace the dependency with an alternative, or to extend the system with alternate dependencies at the same time as configurable options. Replace dependency or add alternates to test impact of doing so.

Design and Build a Mock: Determine how to mock the dependency for testing purposes. Define different kinds of testing problems which might require different mocking capabilities (see Contract section for examples of mocking ideas).

Test Connection and Communication Security: Determine how connections and communication are secured relative to the dependency. Establish what authorization levels and scopes exist. Explore ways security of the dependency could go wrong.

Reliability

Force and Test Fallback States: Analyze the system for redundant or fallback mechanisms for critical dependencies, test the fallback states.

Test System Behavior to Dependency Going Up and Down: Test the case where the dependency goes down or is unavailable.

Test With Dependency Disabled: Test the system in a degraded mode without certain dependencies available.

Test Monitors of Dependencies: Assess and test monitoring of external dependencies. Look for critical information to trace activities across the system, identify failures, or diagnose problems.

Test Error Prone Real-World Conditions: Establish testing coverage of real-world scenarios where dependencies may behave unexpectedly.

Introduce and Test Race Conditions and Concurrency: Assess potential for race conditions and concurrency issues the dependency may introduce and establish a way to test system behavior in that case.

Test Access and Authorization Failure: Create an authorization or access to the dependency failure or denial.

Execute Tests of Error Returns at Each Point of Dependency Interaction: Test the case the dependency returns an error. Try to cover each of the points where interaction happens with the dependency.

Test for Data and State Synchronization Issues: Look for ways data or state is different than the dependency's and try to explore behavior when that happens.

Test for Missing Operation and Messages: Examine behavior if messages or events sent from the dependency are not received, do not arrive, or are missed.

Test for Order of Operation Variance: Explore what happens if order of events, messages, or operations coming from or in dependency are different than we expect.

Introduce Interference to Simulate Reliability Problems: Test system behavior relative to reliability of the connection to the dependency via different mechanisms such as network noise, interrupted connections, or extensive load.

Performance

Test Dependency Impact on Performance: Analyze latency introduced by dependent components and whether it negative impacts system performance requirements.

Simulate Slow Performance State: Simulate or create the condition where the dependency is operating or responding slowly.

Test Extremely Fast/Heavy Load From System Against Dependency: Explore how fast or heavily we can interact with the dependency and what happens at different rates of usage.

Test for Data and Other Artifact Size: Determine big an object, session, state of data or other artifacts can the dependency handle and what happens at different sizes.

Test for Load Handling Behaviors: Determine how load is handled and shared within the dependency and effect that it has on the system.
Test Adding Capacity to System: Determine how to scale usage with the dependency and test behavior of the system when needing to scale load. Assess whether scalability needs meet system requirements, for example does the system need to scale in real time, or is it okay if adding resources takes up front planning?

Test and Measure Monetary Cost of Workload: Put the system under load and measure the volume of activity and data created. Extrapolate out the monetary cost if using a hosted service to accommodate that load and compare it to anticipated real-time usage.

Breakdown Real-Time Usage Costs: Track real time usage of resources for connection, processing, data storage and data transfer. Correlate each of those to monetary costs. Find instances where resources paid for may not be necessary.

Scenarios

Create Smoke Tests that Check Startup and Connection State: Create basic system startup and health scenarios that exercise paths through each of the dependency relationships.
Create tests which require investigating flow of action and data through each dependency in the system. Look for places where information is lost, not understood, control lost.

Test Cross-Dependency Scenarios and That Exacerbate the Crossing Point Impact: Describe a suite of user scenarios and actions and where the dependencies are utilized in each scenario. At points where steps cross from one side of a dependency to another, introduce ways to change the path of the scenario, add steps, alter course, change data.

Test Permutations of Users Crossing Systems to Complete Scenarios: Define and cover actions users need to take within the dependent system to complete scenarios. Explore permutations of those actions to extend the scenario coverage.

Test Handling of Sensitive Data: Evaluate what the dependency does with data it is given and whether those behaviors are suitable for your system requirements. Pay special attention to privacy, security, and other regulatory requirements.

Test for Regulatory Compliance: Assess whether the external dependency will comply with any audit regulatory needs for audit trails or other compliance you may be obligated to.

Test if Level of Control Is Sufficient: Establish how much control you have over what happens in the dependency, and whether or not that level of control is sufficient for system needs, and what impact that has on critical scenarios.

Test System Update and Data Migration Scenarios: Perform update and migration tests on parts of the system that interact with or rely on dependencies. Pay special attention to data or system state persisted before and after the update. Look for anything which may be inaccessible, lost, or unable to proceed through workflows. Look for dependency interactions, especially messaging and data writes, that might happen at the same time as a system update.