Testing

Why Testing

- Tests allow us to find flaws in our software
- Good tests document the code by describing the intent
- Automated tests saves time, compared to manual tests
- Automated tests allow us to safely change and refactor our code without introducing regressions

The Fundamentals

- We consider code to be incomplete if it is not accompanied by tests
- We write unit tests (tests without external dependencies) that can run before every PR merge to validate that we don't have regressions
- We write Integration tests/E2E tests that test the whole system end to end, and run them regularly
- We write our tests early and block any further code merging if tests fail.
- We run load tests/performance tests where appropriate to validate that the system performs under stress

Build for Testing

Testing is a critical part of the development process. It is important to build your application with testing in mind. Here are some tips to help you build for testing:

- **Parameterize everything.** Rather than hard-code any variables, consider making everything a configurable parameter with a reasonable default. This will allow you to easily change the behavior of your application during testing. Particularly during performance testing, it is common to test different values to see what impact that has on performance. If a range of defaults need to change together, consider one or more parameters which set "modes", changing the defaults of a group of parameters together.

- **Document at startup.** When your application starts up, it should log all parameters. This ensures the person reviewing the logs and application behavior know exactly how the application is configured.
- **Log to console.** Logging to external systems like Azure Monitor is desirable for traceability across services. This requires logs to be dispatched from the local system to the external system and that is a dependency that can fail. It is important that someone be able to console logs directly on the local system.
- **Log to external system.** In addition to console logs, logging to an external system like Azure Monitor is desirable for traceability across services and durability of logs.
- **Log all activity.** If the system is performing some activity (reading data from a database, calling an external service, etc.), it should log that activity. Ideally, there should be a log message saying the activity is starting and another log message saying the activity is complete. This allows someone reviewing the logs to understand what the application is doing and how long it is taking. Depending on how noisy this is, different messages can be associated with different log levels, but it is important to have the information available when it comes to debugging a deployed system.
- **Correlate distributed activities.** If the system is performing some activity that is distributed across multiple systems, it is important to correlate the activity across those systems. This can be done using a Correlation ID that is passed from system to system. This allows someone reviewing the logs to understand the entire flow of activity. For more information, please see Observability in Microservices.
- **Log metadata.** When logging, it is important to include metadata that is relevant to the activity. For example, a Tenant ID, Customer ID, or Order ID. This allows someone reviewing the logs to understand the context of the activity and filter to a manageable set of logs.
- **Log performance metrics.** Even if you are using App Insights to capture how long dependency calls are taking, it is often useful to know long certain functions of your application took. It then becomes possible to evaluate the performance characteristics of your application as it is deployed on different compute platforms with different limitations on CPU, memory, and network bandwidth. For more information, please see Metrics.

Map of Outcomes to Testing Techniques

The table below maps outcomes (the results that you may want to achieve in your validation efforts) to one or more techniques that can be used to accomplish that outcome.

When I am working on...	I want to get this outcome...	...so I should consider
Development	Prove backward compatibility with existing callers and clients	Shadow testing
Development	Ensure telemetry is sufficiently detailed and complete to trace and diagnose malfunction in End-to-End testing flows	Distributed Debug challenges; Orphaned call chain analysis
Development	Ensure program logic is correct for a variety of expected, mainline, edge and unexpected inputs	Unit testing; Functional tests; Consumer-driven Contract Testing; Integration testing
Development	Prevent regressions in logical correctness; earlier is better	Unit testing; Functional tests; Consumer-driven Contract Testing; Integration testing; Rings (each of these are expanding scopes of coverage)
Development	Quickly validate mainline correctness of a point of functionality (e.g. single API), manually	Manual smoke testing Tools: postman, powershell, curl
Development	Validate interactions between components in isolation, ensuring that consumer and provider components are compatible and conform to a shared understanding documented in a contract	Consumer-driven Contract Testing
Development	Validate that multiple components function together across multiple interfaces in a call chain, incl network hops	Integration testing; End-to-end (End-to-End testing) tests; Segmented end-to-end (End-to-End testing)

When I am working on...	I want to get this outcome...	...so I should consider
Development	Prove disaster recoverability – recover from corruption of data	DR drills
Development	Find vulnerabilities in service Authentication or Authorization	Scenario (security)
Development	Prove correct RBAC and claims interpretation of Authorization code	Scenario (security)
Development	Document and/or enforce valid API usage	Unit testing; Functional tests; Consumer-driven Contract Testing
Development	Prove implementation correctness in advance of a dependency or absent a dependency	Unit testing (with mocks); Unit testing (with emulators); Consumer-driven Contract Testing
Development	Ensure that the user interface is accessible	Accessibility
Development	Ensure that users can operate the interface	UI testing (automated) (human usability observation)
Development	Prevent regression in user experience	UI automation; End-to-End testing
Development	Detect and prevent 'noisy neighbor' phenomena	Load testing
Development	Detect availability drops	Synthetic Transaction testing; Outside-in probes

When I am working on...	I want to get this outcome...	...so I should consider
Development	Prevent regression in 'composite' scenario use cases / workflows (e.g. an e-commerce system might have many APIs that used together in a sequence perform a "shop-and-buy" scenario)	End-to-End testing; Scenario
Development; Operations	Prevent regressions in runtime performance metrics e.g. latency / cost / resource consumption; earlier is better	Rings; Synthetic Transaction testing / Transaction; Rollback Watchdogs
Development; Optimization	Compare any given metric between 2 candidate implementations or variations in functionality	Flighting; A/B testing
Development; Staging	Prove production system of provisioned capacity meets goals for reliability, availability, resource consumption, performance	Load testing (stress); Spike; Soak; Performance testing
Development; Staging	Understand key user experience performance characteristics – latency, chattiness, resiliency to network errors	Load; Performance testing; Scenario (network partitioning)
Development; Staging; Operation	Discover melt points (the loads at which failure or maximum tolerable resource consumption occurs) for each individual component in the stack	Squeeze; Load testing (stress)
Development; Staging; Operation	Discover overall system melt point (the loads at which the end-to-end system fails) and which component is the weakest link in the whole stack	Squeeze; Load testing (stress)
Development; Staging; Operation	Measure capacity limits for given provisioning to predict or satisfy future provisioning needs	Squeeze; Load testing (stress)

When I am working on...	I want to get this outcome...	...so I should consider
Development; Staging; Operation	Create / exercise failover runbook	Failover drills
Development; Staging; Operation	Prove disaster recoverability – loss of data center (the meteor scenario); measure MTTR	DR drills
Development; Staging; Operation	Understand whether observability dashboards are correct, and telemetry is complete; flowing	Trace Validation; Load testing (stress); Scenario; End-to-End testing
Development; Staging; Operation	Measure impact of seasonality of traffic	Load testing
Development; Staging; Operation	Prove Transaction and alerts correctly notify / take action	Synthetic Transaction testing (negative cases); Load testing
Development; Staging; Operation; Optimizing	Understand scalability curve, i.e. how the system consumes resources with load	Load testing (stress); Performance testing
Operation; Optimizing	Discover system behavior over long-haul time	Soak
Optimizing	Find cost savings opportunities	Squeeze
Staging; Operation	Measure impact of failover / scale-out (repartitioning, increasing provisioning) / scale-down	Failover drills; Scale drills

When I am working on...	I want to get this outcome...	...so I should consider
Staging; Operation	Create/Exercise runbook for increasing/reducing provisioning	Scale drills
Staging; Operation	Measure behavior under rapid changes in traffic	Spike
Staging; Optimizing	Discover cost metrics per unit load volume (what factors influence cost at what load points, e.g. cost per million concurrent users)	Load (stress)
Development; Operation	Discover points where a system is not resilient to unpredictable yet inevitable failures (network outage, hardware failure, VM host servicing, rack/switch failures, random acts of the Malevolent Divine, solar flares, sharks that eat undersea cable relays, cosmic radiation, power outages, renegade backhoe operators, wolves chewing on junction boxes, ...)	Chaos
Development	Perform unit testing on Power platform custom connectors	Custom Connector Testing

Technology Specific Testing

- Using DevTest Pattern for building containers with AzDO
- Using Azurite to run blob storage tests in pipeline

Last update: August 26, 2024