Simulation Testing
Solution Blueprint

Michael T. Nygard, Cognitect

QA IS A **NO-WIN SITUATION**

01/ THE CHALLENGE

More organizations are moving to continuous delivery. As they do, QA must somehow shorten testing cycles and reduce the fraction of bugs that slip through. Manual testing cycles can't get done fast enough, no matter how much you offshore. And yet, scripted testing can't catch the "weird" bugs because scripts always act more reasonably than real-world users.

The situation gets even harder when we talk about microservices. More systems to test, more deployments, more interfaces where bugs can creep in. Unit tests help at the micro level, but we need something better at the macro level.

Above all else, we know that QA is not the objective. Creating a robust system is. That means we need more realistic testing methods. We can get there by simulating the world around the system.

02/ SIMULATION TESTING ESSENTIALS

Simulation testing creates the illusion of activity from real world usage.
Realism is the key.

The activity can mimic human users or it can take the place of system-to-system traffic. In either case, it is based on a model of the real world. We use the *model* to generate a recorded stream of repeatable activity. We then *execute* the activity stream against the system under test and *capture* the system's output and relevant internal state. That captured output is recorded so we can *validate* it later.

Each of these steps is decoupled through a write-once database. It turns out that separating the stages this way brings benefits at each individual step as well as overall benefits.

This blueprint will explain the benefits of each step and show you how to put the pieces together.

Every simulation test follows the same basic process. Each of these steps is decoupled through a write-once database. Because each step is isolated from the others, it immediately gains a degree of simplicity.

03/ SUCCEEDING WITH SIMULATION TESTING

Unit tests and other forms of example-based tests may increase our confidence in the local correctness of fragments of our application, but they offer no way to understand whether the system as a whole is correct.

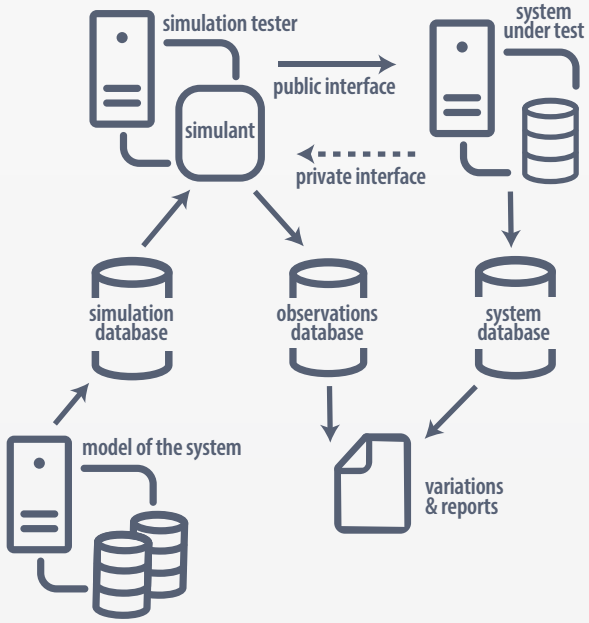
Simulation Testing gives you this ability through randomness designed to mimic reality: agents run through serial, timestamped action streams, and any number of agents can run in parallel to simulate real, multi-user activity.

Your particular needs will lead you to more or less customization in each step:

1. **Model -** Identify facts about your system
2. **Generate** - Create serialized, time-stamped activity streams
3. **Execute** - Run driver programs, scaled horizontally and coordinated through a database, to call your system.
4. **Capture** - Record your system's outputs into a database.
5. **Validate** - Query the results database to find undesirable conditions.

04/ COMPONENTS OF A SIMULATION TEST SYSTEM

Simulation testing separates responsibilities that are combined in example-based testing. Each step of the process reads from or writes to a database. For instance, the generator writes scripts into a database, which the runner later reads from. That way we get randomness in the generator, and repeatability by using the same script more than once.



05/ SYSTEM UNDER TEST

The "system under test" is your application, service, or set of services.

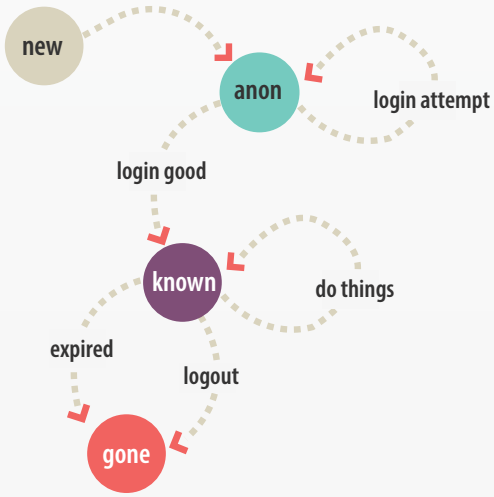
If you just want to test for correct behavior, you can run the system under test together with the simulation runner itself.

More often, the system under test should be deployed in a realistic way, including both the configuration and topology that it will have in production. Where the real environment will have a load balancer, the simulation should go through a load balancer. The same goes for firewalls.

The system under test does not need to be written in Clojure or Java. It just needs to be callable from the simulation.

06/ MODELING THE SYSTEM

Example-based tests rely on scripted sequences of actions, either captured via recording or hand written by testers. These scripts may include some variable data with randomized pools or some random "think time" between actions, but they are essentially linear.



In simulation testing, we use a model of the system to decide what sequences of actions to take. We can then sample that model over and over again to create activity.

It's best to start simple. Often, a simple state machine will do for a model. Put realistic probabilities on the transitions and generate some activity. Causatum is a useful library you can use to create models that don't need a "memory" of what states they have already visited.

Some cases require more sophisticated models. You may even end up training your model with logs recorded from an existing system.

07/ MAKING THE SYSTEM TESTABLE

Actions in the simulation call the system under test via the public interface. To make the system testable, however, it is often necessary to provide a private interface for:

1. Setup prior to running the simulation
2. Data collection during or at the conclusion of a simulation
3. Cleanup after the simulation.

Data setup and cleanup is an important part of making simulations repeatable. For instance, if the simulation includes steps of "account creation," then those accounts must not exist when the simulation starts. Conversely, if the simulation relies on certain entities to exist, then the setup must create them. Sometimes, the need for this setup actually indicates a missing administrative interface for the system under test. In other cases, it just means there is a private API available only to the simulation runner.

08/ RUNNING THE SIMULATION

The next major block is the simulation tester. It reads and executes the script of actions, calling into the code you provide to perform each step. Almost all of these call the system under test, often by making an HTTP request.

As the simulation tester proceeds, it records the results, but does not validate them. That happens later.

The simulation runner executes on one or more hosts. You can scale the number of runners out as needed for the scale of traffic you're simulating. One runner can handle a lot of virtual users. Still, if it's Black Friday you want to simulate, you'd better plan on using some extra machines. The runner works very well on cloud-based virtual machines.

09/ STORING OBSERVATIONS

When you run an example-based test suite, it reports failures immediately via console or GUI. The results of that test run are ephemeral. In contrast, each run of a simulation test gets stored in a database for inspection, validation, and reporting.

Each run of the simulation gets recorded in the results database. This holds all the raw observations in the form of an activity log per agent. Each record links back to the action that produced it.

"Observations" is deliberately broad. It includes responses to the actions such as HTTP replies or API return values. It may also include any other instrumentation you need for validation.

The main thing here is to store the raw observations. This step does not judge whether the observation indicates a desirable or undesirable response.

10/ VALIDATING AND REPORTING

Where example-based tests immediately report test failures, a simulation stores all the results in a database. This includes responses to the public interface as well as any private information that the system under test can provide. A separate program validates the results by searching that database for bad conditions.

This can be as simple as querying for any HTTP request that returned an error code, or it can be much more sophisticated. The history of all actions is available, so it's possible to ask about cause-effect relations. For example, did each virtual user receive the promotional price that was offered? This is an example of a query that follows sessions across interactions.

It's also possible to query across user sessions to check global properties. For example, you can validate the percentage of sessions placed into a test group for A/B testing and personalization.

It's even possible to query across multiple test runs.

ABOUT SIMULANT

Simulant is a library and schema for constructing simulation-based tests, developed by Cognitect. It takes advantage of Clojure and Datomic, giving you the power of a versioned, time-aware database to capture all Simulation Test artifacts, from the model to the execution results.

Simulant provides support for the Generate, Execute, and Capture steps in the process. When you build an application with Simulant, you add your own model and validations.

NEXT STEPS

- Check out Simulant and an example test system
- Listen to episode 59 of the Cognicast
- Watch this video recorded at Strange Loop 2014
- Get started with Datomic