On the Maintenance of Large-Scale Elixir Tests

Livia Almeida Barbosa

Remote Technology, Inc
Wilmington, DE – United States

livia@remote.com

Abstract. *This paper discusses how a global company maintains its test suite, which currently has over 95,000 test cases. While growing its products using an Elixir monolithic codebase, the team developed and evolved guidelines and patterns for writing tests and managing unreliable (flaky) tests while keeping the developer experience and the CI/CD time up to standards.*

1. Introduction

Classified as the second most-admired language for the last three years in the StackOverflow Developer Survey [StackOverflow 2022] [StackOverflow 2023] [StackOverflow 2024], Elixir is a general-purpose functional language that runs on top of BEAM (Erlang's virtual machine) and is used to build fault-tolerant, scalable, and distributed applications [The Elixir Team 2012]. Created in 2012, it already has a vast ecosystem with frameworks and libraries for web development, data processing pipelines, machine learning, and embedded systems.

Remote, a company that enables employment through recruitment, pay, and management of international teams across more than 190 countries, debuted in 2019. Its founders chose Elixir from the beginning, and the monolithic application has evolved to support several products. Its codebase holds more than 19,000 source code files and 95,000 Elixir tests, including unit and integration (99.20%), doctests (0.79%), and property-based (<0,01%) tests. Besides leveraging Elixir's built-in test framework, EXUNIT,[1] Remote also relies on other important libraries, as EX_MACHINA[2] for factories, MOX[3] and MIMIC[4] for test doubles and dependency injection.

In this paper, we discuss how to maintain an ever-growing test suite and how the knowledge and challenges evolved to accommodate the product and the team's needs.

2. Guidelines and Testing Patterns

The backend team doesn't have a target code coverage because a well-established test culture already guarantees that all change requests will have tests covering the modifications. In addition to following software testing best practices and guidelines of the EXUNIT framework [Leopardi and Matthias 2021], the backend team agreed on patterns to maintain consistency, prevent slowness and unreliable tests across the test suite.

[1] https://hexdocs.pm/ex_unit/ExUnit.html
[2] https://hexdocs.pm/ex_machina/ExMachina.html
[3] https://hexdocs.pm/mox/Mox.html
[4] https://hexdocs.pm/mimic/Mimic.html

The most relevant instruction is to write tests following the application organization[5] and exercise the correct scope. Each of the monolith layers should have a narrow scope. Thus, writing key tests should not take long. Tests should at least assert on the API contract and exercise side effects introduced by logic in the function under test. This helps to balance the extent of tests, avoiding redundant coverage, and changing multiple test files when introducing a single change.

It is common for a developer to be part of a temporary cross-functional team or be reallocated to a different team. Since teams keep their domain tests up-to-date, most often with explicit setups, new developers can rely on them as another source for documentation. When creating or editing a module, developers add, at least, unit tests for the happy path, expected error paths, and some edge cases. If there is a bug in production, one ships the fix and a test case to prevent regressions.

3. Unreliable Tests

Unreliable, or flaky, tests are one of the pain points for developers in change requests. Remote has a distributed team across the globe, with more than 300 engineers. Faults are usually related to dates and time precision, unique key collisions in fixtures or seeds, unexpected unsorted lists, and timeouts in the sandbox test database. Developers in the Asia–Pacific region are the first affected by flaky tests, especially the datetime ones, due to timezone differences. Addressing these issues is key to ensuring productivity for the entire Engineering department.

To handle flaky tests caused by imprecise datetime, the team agreed to hardcode the datetime values in the fields under tests, or better yet, pass it as a parameter to the module or function under test when feasible. However, hardcoding datetime can also decrease the possibility of finding other edge cases, for example, scenarios that don't work well when changing the day or the year.

Listing database records without an `order by` clause can also lead to flaky tests. To address that, the team leverages and strongly recommends using the ASSERTIONS[6] library, which provides flexible and concise assertions for elements in lists. We can assert that all desirable elements are present, but not necessarily in the same order.

ECTO_SQL,[7] part of a component for database management and interaction, provides a test sandbox that runs database tests concurrently within transactions, giving developers a secure way to run tests. However, database timeouts still occur, leading to test failures. Another guideline related to running concurrent tests is not to assert on logs when it is not the main functionality under test. The default configuration runs tests concurrently; therefore, one execution might affect assertions against other test logs.

Despite efforts, the team still experiences flaky tests. The approach to handle them was to write a script to re-run all failed tests in the continuous integration pipeline. Those that fail again are finally reported as failed. A flaky test report is consolidated in the continuous integration tool, and results are sent over to the company's main communication tool, automatically creating a new issue in the issue tracking tool to ensure quick action to

[5]https://www.youtube.com/watch?v=xWqOR-cdIUQ
[6]https://github.com/devonestes/assertions
[7]https://hexdocs.pm/ecto_sql

address them. The team also set up static code analysis rules using CREDO[8] and enforces guidelines on code reviews to avoid introducing potential flaky tests.

4. Developer Experience

Having helper modules with commonly needed fixtures, test setups, custom assertions, and helper functions is essential to have a good and maintainable test suite. For example, the team developed specific helper functions to change the environment (test, development, or production) and to mock feature flags, which prevents changing the global state and interfering with other test runs.

Remote incentivizes automation and efficiency in all areas. Driven by these values and noticing the time spent between running one `mix test` command and another, a couple of engineers developed a helper module that made the iteration between development and test execution faster. The module implements helper functions that allow running tests directly in the Elixir interactive shell (IEX), bypassing the BEAM startup time and avoiding around 5 seconds for each `mix test` run. This brought a major quality-of-life improvement for the backend team during development.

Because of the size of the test suite, it is unfeasible to run all tests frequently when writing code. The team also leverages `mix test.watch`, [9] a task that runs the project's tests whenever a source file changes, and `mix test.diff`, another task that, given a change in a file, runs related tests based on the dependency graph of that change.

5. Continuous Integration and Delivery (CI/CD)

Running all tests in a single CI/CD pipeline job is also impractical. Although most Elixir test files run concurrently, each test case in a file is executed synchronously. The infrastructure team was key to keeping the deployment time within a reasonable time, which was defined to be less than an hour. They have split the test suite into multiple jobs to be run in our CI tool, leveraging both Elixir's concurrency and the CI's parallelism. The number of jobs and the amount of tests running on them are a tradeoff between the tolerance for slowness, the ideal scenario, and available resources. Financial resources are an important consideration, since increasing CI jobs requires more runners, which can significantly impact a company's budget with a large team.

6. Conclusion

Not all companies adhere to writing tests in the development life cycle. However, those that do include, encourage, and want to take advantage of Elixir's concurrency, ecosystem, and community will notice how easy and fast it is to maintain an Elixir test suite. At Remote, even with more than 95,000 tests, the team strives to keep the deployment time within an appropriate range, follow the guidelines and patterns, and not let the developer experience degrade. With this mindset, there's room to have a stellar customer experience, care about the codebase, and the Engineering team at the same time.

[8] https://hexdocs.pm/credo
[9] https://hexdocs.pm/mix_test_watch

References

Leopardi, A. and Matthias, J. (2021). *Testing Elixir: Effective and Robust Testing for Elixir and its Ecosystem*. Pragmatic Bookshelf, 1st edition.

StackOverflow (2022). 2022 Developer Survey. https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted. Accessed: 2025-05-12.

StackOverflow (2023). 2023 Developer Survey. https://survey.stackoverflow.co/2023/#technology-admired-and-desired. Accessed: 2025-05-12.

StackOverflow (2024). 2024 Developer Survey. https://survey.stackoverflow.co/2024/technology#admired-and-desired. Accessed: 2025-05-12.

The Elixir Team (2012). The elixir programming language. https://elixir-lang.org/. Accessed: 2025-05-12.