Using test automation frameworks to speed your DevOps delivery

WHITEPAPER

ZEPHYR
Real-Time Test Management

Today's fast-moving software delivery environment emphasizes pushing value to production as quickly as possible. "Value to production" implicitly requires high quality—low quality is of little value to anyone!—which means teams need to focus on being able to specify, build, test, and deploy software effectively and quickly. Acceptance and regression testing have often historically been manually intensive efforts, resulting in a slower pace of production releases.

Test automation is a boon to organizations when it's approached with reasonable expectations for skills, time, and problems it's able to solve.

Table of Contents

Types of Automation

Many different types of automation exist: scripting for repeatable tasks, batch files, orchestration, the entire realm of DevOps, etc. This paper focuses on test automation. Moreover, we're focusing on one particular type of automation: functional or acceptance test automation.

That said, it's helpful to understand the basic types of test automation.

UNIT TESTS

Automated unit tests are coded verifications that validate one specific behavior in a small section of the system. Unit tests focus on code that has no external dependencies: no calls to the database, no web services, etc. Unit tests are blisteringly fast because of this narrow focus and lack of dependencies on external systems/services.

Unit tests focus on ensuring all paths through code are properly validated. Think of a payroll algorithm responsible for computing wages for hourly workers. The algorithm would take in number of hours worked and the hourly rate, and would return the wages for the period. Such an algorithm would need to handle a number of different situations such as:

- Standard time (0-40 hours)
- Overtime (greater than 40 hours to whatever the company's max hours per period)
- Error handling (negative hours, negative wages, over max hours)

Such cases can be easily validated via unit tests using widely adopted tools such as NUnit, JUnit, RSpec, etc.

Unit tests are generally written by developers. The test cases are best based on interaction with testers and potentially the business analyst, often during Three Amigos conversations.

INTEGRATION / SERVICE TESTS

Integration tests validate behaviors between components, and are most often written by developers. These can involve checking behaviors for web services, database calls, or other API interactions. Integration tests are much slower than unit tests because they need to handle significant amounts of "ceremony" to stand up connections, handle authentications, as well as deal with service and network latency.

Integration tests generally should avoid granular validations (those are best left to unit tests) and should instead focus on more significant validations. For example, continuing with the payroll example from the unit test section above, a good integration test would be: when the payroll service is invoked with valid data, do I get a correct response. The integration test should not cycle through all test cases for validating the payroll algorithm—that's the responsibility of the unit tests!

FUNCTIONAL TESTS

Functional tests validate a slice of system functionality: can I create a new contact? Can I run payroll? Am I prevented from editing an employee's pay data when I'm not authorized to do that action? Functional tests often run via the User Interface, which means they are much slower than integration tests, and dramatically slower than unit tests.

Because of their slow and brittle nature, functional tests must be focused on a few high-value scenarios. Functional tests shouldn't handle validation or low-level actions.

OTHER TEST AUTOMATION

Test automation is a broad domain and includes many other types, including:

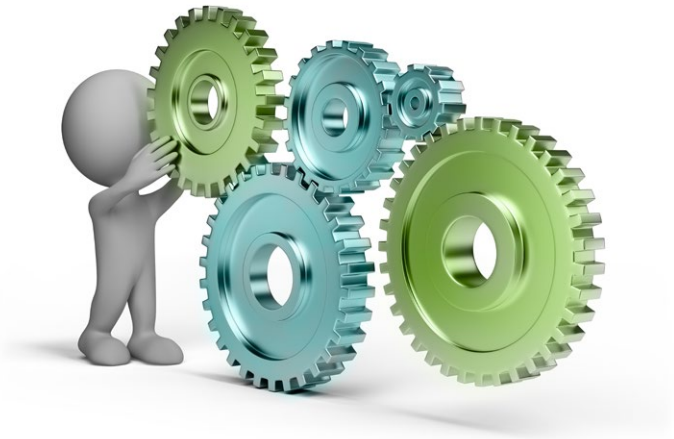
- Performance
- Load
- Security
- Accessibility
- Production monitoring

All these are valuable automation types, and most can leverage some form of framework support; however, this paper won't address them.

UNDERSTANDING EFFECTIVE TEST COVERAGE

Close coordination among the delivery team helps ensure the proper testing is happening. Moreover, that coordination ensures there's little or no duplication of test coverage. It's especially critical to reduce duplication so that more expensive tests (functional, e.g.) can focus on specific concerns. Returning to the payroll example, a good distribution of tests might look something similar to this:

- **Unit Tests:** Cover cases for standard time, over time, boundary tests, zero and negative inputs
- **Service/Integration Tests:** Check one positive and negative case to ensure proper value and error states are returned. May also validate security-related aspects.
- **Functional Tests:** One happy and one sad path to validate the system returns a proper value and error message at the UI level



Value of Automation

Test automation requires a significant investment in teams' skills, tooling, and infrastructure. There are also changes required to how a team designs, builds, and delivers their software. Moreover, there is definitely a cost in velocity for the teams: it takes time to write those automated tests.

With that in mind, a common question is "Why should we spend the time to automate appropriate tests?"

REDUCE ERRORS IN SPECIFICATIONS/ UNDERSTANDING

Test automation is often as much about the discussions that happen before any work is done as the automation itself. Testing should be part of the discussion all the way to initial programming phases up through the actual construction of the software. These discussions ensure the delivery team clearly understands what the Stakeholders/Product Owners are looking for in the system behavior.

Again, returning to the payroll discussion: developers and testers might ask the PO to clarify how the system should handle conditions like inputs of 200 hours worked per week. BAs might as how employee salary changes are handled. These discussions drive the entire team's clarity of understanding, and significantly lower the amount of rework involved later.

Liz Keogh, one of the earliest adopters of Behavior Driven Development, emphasizes the importance of conversations: "Having conversations is more important than documenting them. Documenting conversations is more important than automating them."

MITIGATE RISK

Most Stakeholders and Product Owners rarely know a thing about Object Relational Mappers, n-tier architectures, or distributed caching systems. What the business folks in an organization *do* know about is managing risk. Risk of not delivering on time. Risk of building something that doesn't meet user needs. Risk of cost overruns due to massive rework required by buggy software.

Test automation can help Stakeholders and POs better manage risk by addressing many of their concerns early in the process. Moreover, the cost of change to a system is greatly reduced by test automation, so it's much easier when a bug fix or feature enhancement needs to be pushed to production.

DevOps: Automating Your Delivery Pipeline

The sections above speak to test automation; however, "automation" is frequently used in the context of automating a project's deployment pipeline. Often this is referred to as "DevOps," or Developer Operations. Continuous Integration, Continuous Deployment, and Continuous Delivery are all facets of this overarching domain: leveraging automation and tools to quickly build, test, deploy, re-test, and promote software changes through environments to eventually deploy in to production.

ENVIRONMENT MANAGEMENT

Managing different environments during a software project quickly becomes a significant effort, often requiring one or more team members dedicated full-time to that effort.

DevOps automation doesn't eliminate the need for skilled infrastructure staff; however, automation can make those staff much more effective by reducing manual effort, making processes repeatable and stable. Spending time up front building infrastructure and environment tooling means it's possible to stand up, provision, and deploy to new environments at the push of a metaphorical button. Complexity and differences between environments are mitigated as teams are able to build in configuration scripts instead of manually performing those same tasks.

TEST DATA

"Managing our complex test data is my most favorite part of this job!" said no one, ever.

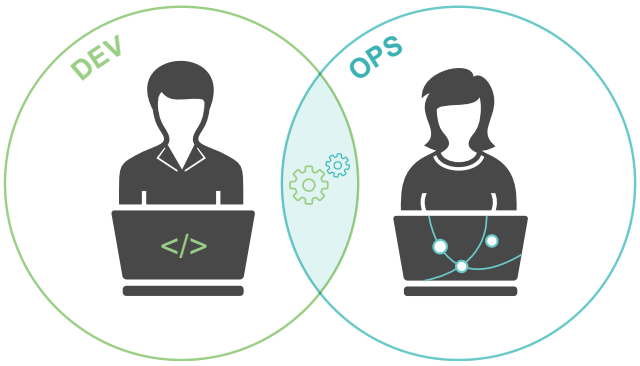
Good test data is critical to any serious delivery program. Real-world data is hard to model, difficult to build, and hard to store as an asset.

Automation, particularly around DevOps, can be a great help in letting teams move past the drudge work of test data management. DevOps orchestration tools like Jenkins, Travis CI, Team City, etc. can be configured to handle extremely complex workflows around test execution. A typical set of actions might look like:

- Team starts a test cycle, initiates test preparation job on the build server
- Test data assets are pulled from source control
- System is built, automated tests are run, and system deployed to appropriate environment
- Test data is provisioned in environment
- Post-deployment automated tests are run (system integration, functional, performance, etc.)
- Team performs manual testing such as exploratory, accessibility, etc.
- Post-testing job is executed on build server to archive assets, create reports, etc.
- Jobs might also update test data based on new learnings, and archive that back to source control

TEST MANAGEMENT

Managing testing efforts can be an extremely burdensome, frustrating exercise. We've already discussed handling different environments, but that's only a small part of what's involved in a large test effort. Deployment of code and assets can be very time-consuming and error-prone. Standing up testing assets such as data, scenarios, tooling, etc. Is also a tedious task. Additionally there's understanding of what areas of the system are at risk, changed, or well-tested.



Automation isn't a panacea for all test management work, but it does greatly ease planning, executing, and reporting on testing efforts. Good automation eases that burden and makes the entire testing process much smoother and less frustrating—leaving the teams to focus their energy on testing.

Automation Frameworks Defined

Test automation code must be treated with the same discipline and approaches as the systems it's used to test. Applying solid design and craftsmanship principles to the automation code ensures teams get the best value from test automation as the project is initially delivered. Well-built automation code is also an extraordinary value benefit over the lifetime of the system, as it's much simpler to continue executing, updating, and enhancing the system with solid backing automated tests.

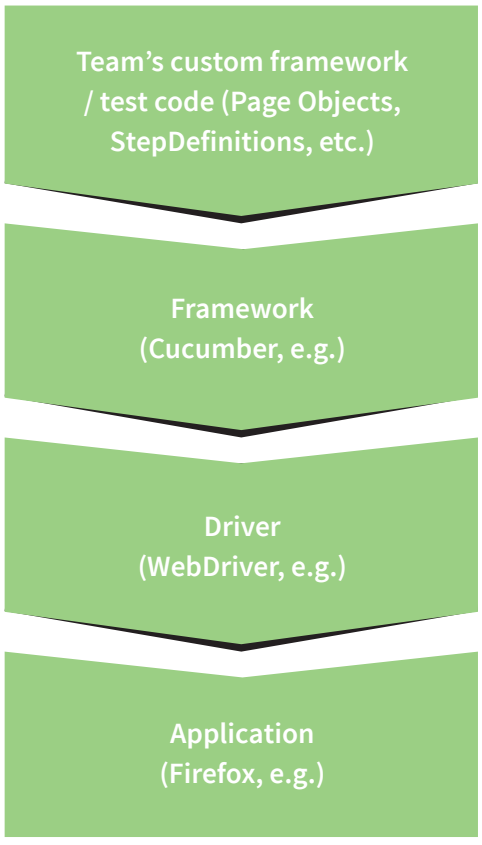
Frameworks are a critical part of any test automation effort. Frameworks provide reusable components for teams to effectively create automation tests specific to their needs.

FRAMEWORKS DEFINED

"Framework" is a term used for many different concepts in software development. This paper uses framework in the context of a high-level layer abstracting low-level details of interactions with the actual UI technology. Proper abstraction is critical for usability and long-term effectiveness of any test automation suite.

The image below shows how different parts of an automation suite work together with each other. Starting at the bottom we have the following components:

- **Application:** This is the actual UI technology being tested, such as a web browser, native iOS application, or WPF desktop application.
- **Driver:** The driver is the lowest-level component. It knows how to interact with the application's specific UI. For example, Selenium WebDriver has different drivers which know how to manipulate Chrome, Firefox, and Microsoft Edge.
- **Framework:** Frameworks like Robot or Cucumber enable teams to write code which focuses on the business problem being tested, versus the specific UI technology. Teams write expressive tests that don't focus on UI implementation details. This also enables, in some cases, the same test to be reused across different web browsers, mobile applications, etc.
- **Custom Code:** This is code specific to the teams' needs and may include abstractions for interacting with page or view-level objects, communicating to web services, checking the database, etc.



OPEN SOURCE SOFTWARE VS. COMMERCIAL

In addition to custom frameworks built specific to teams' needs, many different automation frameworks exist in the open source and commercial domains. Automation solutions may use tooling from both domains to solve the teams' unique problems. Thoughtful discussion needs to guide the team to the right selection of tooling. Some factors that may influence framework selection include:

- **Cost of Adoption:** Commercial frameworks have an explicit price tag tag. Open source has no explicit price tag, but it's far from free due to the cost of learning and implementing it. (Which commercial frameworks have as well!)
- **Industry Adoption Footprint:** A high adoption rate means you'll be able to find thriving communities where you can reach out for troubleshooting, support, and education. Open source generally has a significantly higher adoption rate across the industry. Commercial tools, less so.
- **Integrated Systems:** Commercial frameworks are often part of a larger tool suite. This makes it easy to get an entire end-to-end solution in place for building, executing, and reporting on your automated test suites. Open source software is nearly always a set of different components which teams must integrate and manage. Integrated systems are often less flexible, trading ease of change and power for simplicity in starting and maintaining.
- **Support:** Commercial frameworks are backed by a professional support organization. There's an e-mail address or phone number you can reach out to when you need help. Open Source frameworks have no direct support; however, many professional consultants and companies offer highly skilled third-party support.

WHY USE FRAMEWORKS?

Selecting, implementing, and updating test automation frameworks takes time and effort. Your team *will* see a loss of delivery velocity as you move through the process. So why take on the additional burden?

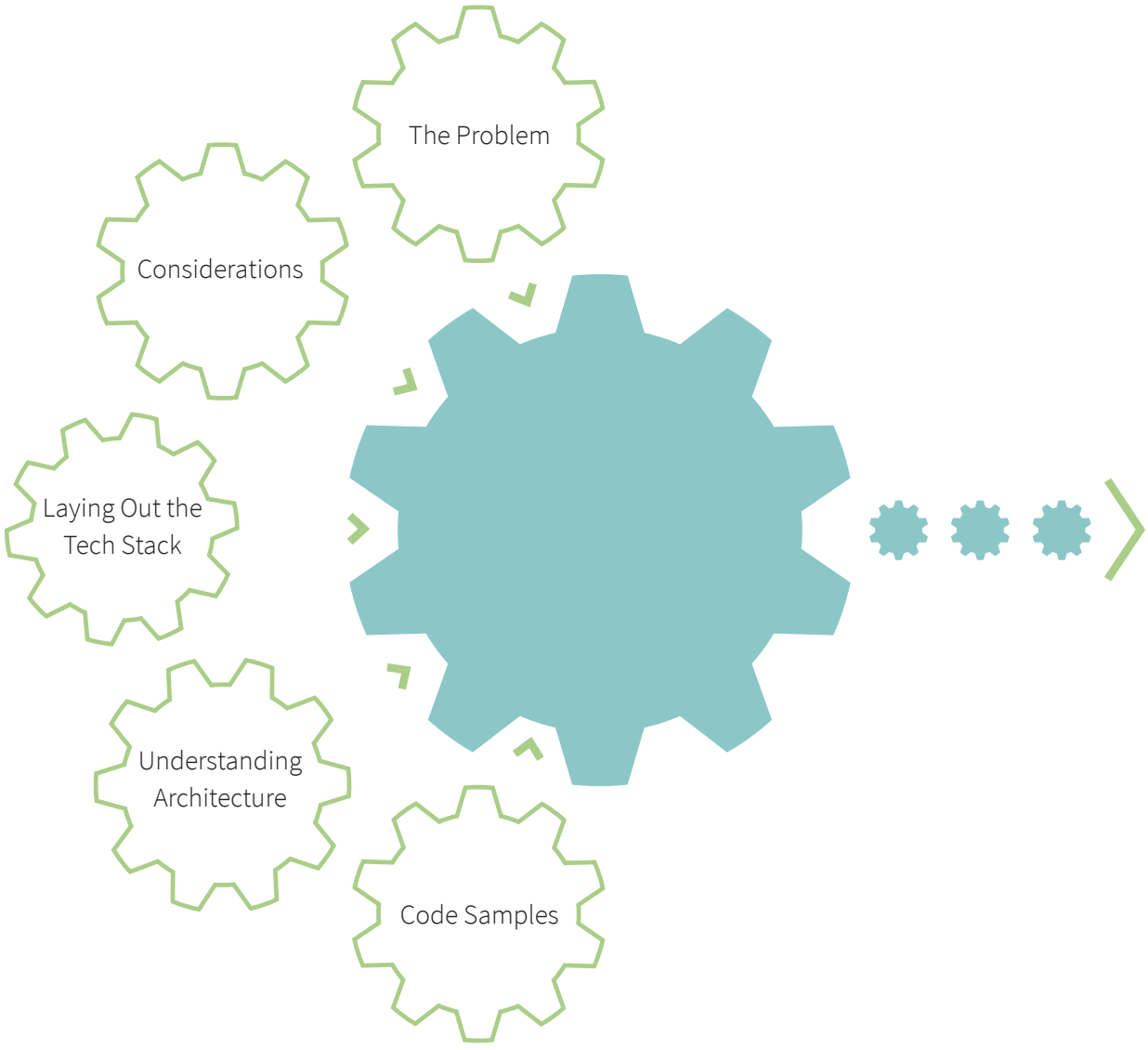
Value. Plain and simple.

Test automation frameworks bring a tremendous amount of value to software delivery, both on the communication side as well as the technical side.

- **Prevent defects versus find defects.** Investing time with a language-based framework like Cucumber means having more detailed conversations earlier in the delivery lifecycle—long before code is actually written. Conversations around acceptance criteria help drive out clarity of functionality, identify specific test cases, and shine a light on edge cases that might not have been identified until much later. Prevention versus detection means eliminating all waste and rework associated with those potential defects or missed functionality.

- **Focus on user value, not technology.** Abstracting away much of the technical detail of how to manipulate a UI means the team can focus on high value: solving problems for the users. Teams worry less about "How do I handle the async delays on this field" and more of "What happens for this payroll if someone changed rates?"
- **Faster creation of tests**. Abstracting away the details of UI interaction lets teams write automation suites much faster. Frameworks enable teams to build blocks of common functionality, then reuse those blocks for future test. Think of actions like logging in to a system, creating an order, editing a user, etc. This concept of reusability can't be overstated enough!
- **Maintainability.** Properly implemented frameworks can dramatically cut test suite maintenance costs. The abstractions used to build suites, coupled with well-constructed designs, mean teams are going to one place, versus numerous places, to fix or extend test cases. (Obviously this requires skilled, knowledgeable team members. Bad code is still bad code. Frameworks don't magically solve that.)

Practical Example: Building a Real World Framework



Author Bio.

As VP of Engineering, Hamesh is responsible for the worldwide engineering function of Zephyr and brings over 18 years of engineering, mobile, and networking technology experience to this role. Prior to Zephyr, Hamesh held numerous lead engineering roles at Asurion, InnoPath Software, Sonus Networks, and Cisco Systems. Hamesh holds an MS in Computer Science from Texas A&M University.

Outside of work, Hamesh enjoys golfing, playing tennis, and spending time with his two daughters.

HAMESH CHAWLA
VP of Engineering
Zephyr

hamesh.chawla@getzephyr.com

About Zephyr

Zephyr is a leading provider of on-demand, real-time enterprise test management solutions, offering innovative applications and unparalleled, metrics based visibility via real time dashboards into the quality and status of software projects.

The feature rich products addresses today's dynamic and global needs across a variety of industries including finance, healthcare, mobile, IT services, and enterprise software.

Zephyr's 10,000+ global customers experience improved productivity, faster time to market and dramatic cost savings.

ZEPHYR
Real-Time Test Management

Contact Zephyr Today!
sales@getzephyr.com
www.getzephyr.com
+1-510-400-8656