NAVIGATING THE BRAVE NEW WORLD OF API TESTING

PETER THOMAS

TABLE OF CONTENTS

Should You Be Doing More API Testing?

API Testing can solve specific problems that trouble the world of test-automation. It is widely discussed that teams struggle with the following:

- Flaky tests
- Keeping up with the pace of development "in-sprint"
- Maintainability
- Functional Regression Coverage
- Performance Testing

Because of their nature, API test-automation suites are well suited to combat these challenges.

This document will deep dive into why and how you can do more with API testing. You will get a sense of how APIs play a central part in modern-day software-architecture and how we as an industry got to where we are now.

This document also includes handy infographics that you can print-out and use as reference.

Why is API Testing not discussed more?

To put this another way, why does most of the discussion revolve around UI test-automation? There are many reasons for this.

API Testing is relatively new, and the discussion has been driven by tools that have been around for decades such as Selenium and UFT.

There are also interesting psychological effects at play. It is much easier for us to discuss things that we can "touch and feel". The user-interface is naturally how teams start thinking about testing the system from a user-acceptance point of view. Things that are hidden away behind the scenes tend to be overlooked or even forgotten in the pursuit of quality-assurance.

There are also far more tool vendors for UI automation, and especially with the no-code or "codeless" variants, there is a perception that UI automation can be easier and more effective. The number of variations possible such as Visual Testing, Accessibility Testing, Exploratory Testing etc. tends to steer any discussion on test-automation towards the user-interface.

There is also a misconception that API Testing is:

- much harder than UI automation.
- requires more programming knowledge.
- and does not achieve enough functional coverage.

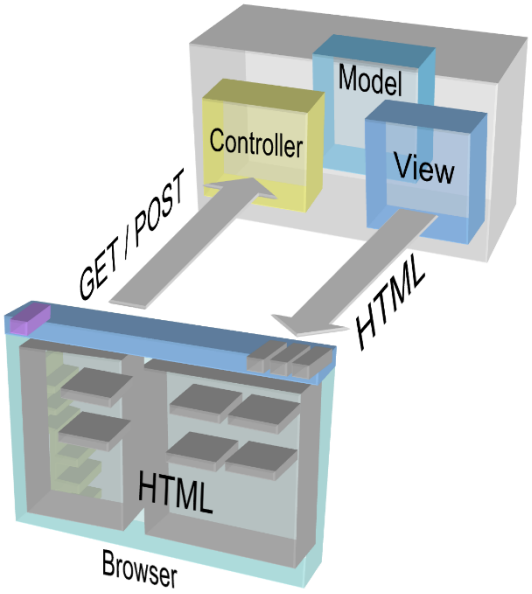
While it is true that UI automation is important and, in many cases, indispensable, we will explore why API testing can supplement or even fulfil the bulk of your regression test strategy. API testing proves to be far easier than UI automation, providing a higher ROI (Return on Investment) when the right tooling is used. When the inherent complexity of UI automation is taken out of the equation, automation can proceed at a fast clip, with teams able to achieve "in-sprint" test-automation.

Why is API testing more relevant today?

The rise of SPA (Single page Applications)

If you were working in tech in the early 2000s, you may remember technologies such as Apache Struts, JSP, PHP, Perl / CGI-BIN and ASP / ASP.NET. That's right, most web-applications in that era were rendered server-side. Server-side patterns such as MVC (Model View Controller) were thoroughly discussed and widely used.

The defining characteristic is that HTML is returned from the server. Data submissions were handled typically via an HTML form POST.

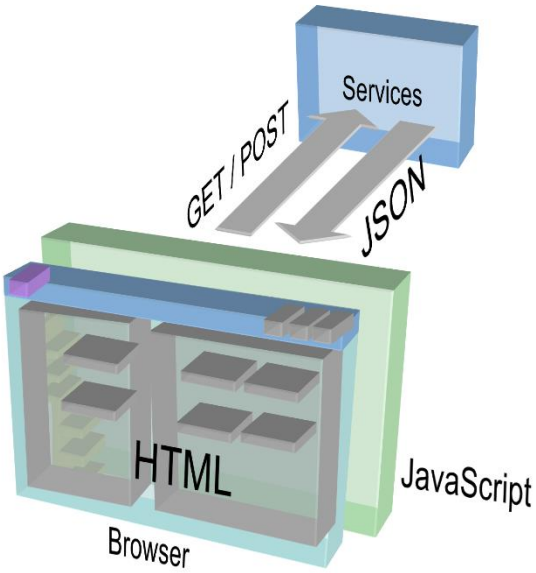


The default user-experience was that when the user clicked a button or a link, the entire page would refresh.

This all changed when things like Google Mail and Google Maps appeared, and the word AJAX (Asynchronous JavaScript and XML) became mainstream. Users started to expect and demand that web-applications be "rich" and dynamic. Partial-refreshing of chunks of HTML on the page became the norm.

The pattern in these web-applications is that JavaScript running in the browser (the front-end) makes HTTP calls to the server and retrieves data. The front-end has the additional responsibility to update the HTML in the browser based on the responses from the server.

While XML was the norm initially, over time, JSON became the data-format of choice for communicating with the server. This is because JavaScript is the native scripting language in the browser, and JSON is a sub-set of JavaScript. It is much easier to manipulate JSON from the front-end - and the simple and effective combination of REST and JSON on the back-end was born. The death of SOAP (which was completely based on XML) also happened because of this reason. The natural and most effective combination for writing front-ends is HTML / CSS and JS. But let us focus on the JS part for a moment.



This pattern of having JavaScript running in the browser and generating the bulk of HTML that the user interacts with - led to the rise of frameworks like AngularJS and React. The old days where jQuery was just used to "spice up" HTML are gone. Now things are reversed, with all the heavy lifting done by JavaScript. Most of this JavaScript is loaded in one-shot when the first "page" of the application is requested by the user. This is what a Single Page Application is (SPA).

Today, SPAs drive the majority of end-user user experiences in the cloud. Think, shopping-carts, e-mail, video-streaming, banking, and other common consumer use-cases. This is a result of the popularity of front-end frameworks such as AngularJS, React and Vue, and the enthusiasm, sheer size, and momentum of the JavaScript ecosystem.

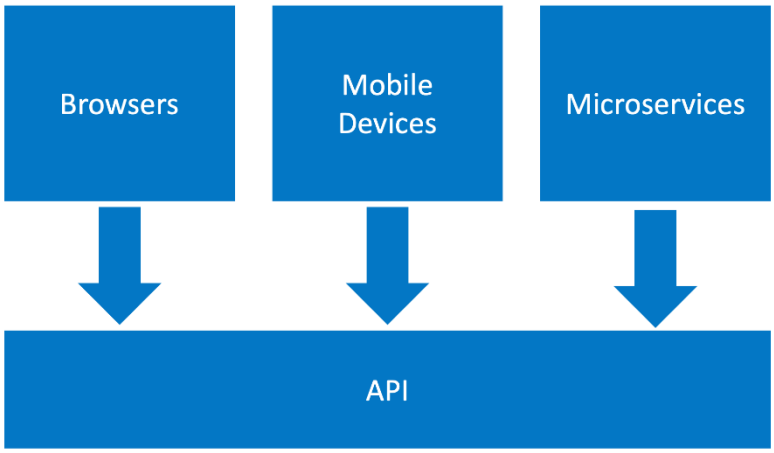
SPAs typically make REST or GraphQL calls to a back-end service. This has become established as a standard architecture pattern and led to the increasing demand for APIs. It has also led to the typical product-development team being divided into two distinct profiles of developers: Front-End and Back-End.

In tech-driven organizations around the world, leadership, training, development and hiring for product-development has organized itself around these core skills. Of course, so has test-automation.

Mobile apps

After SPAs, the dominant mode of delivering end-user experiences to globally distributed users is via a mobile app. Here again there is a thriving developer community targeting the two camps of iOS and Android stewarded by Apple and Google respectively. Ecosystem technologies such as Swift, Kotlin, React Native and Flutter evolve at a fast pace.

Here too, the most logical architecture is to have these mobile front-ends communicate to a back-end server via REST APIs. The great strength of basing your architecture on APIs is that they can support multiple consumer types such as web-apps, mobiles, and even other devices such as IoT nodes.



In the typical enterprise, APIs are a re-usable building-block that multiple other business-units can invoke and many digital-transformation initiatives centre around the concept of microservices. It is very common to have teams structured around the API services that they own, with roles such as API Product Owner becoming more prevalent.

As we will make clear in the next section, the API layer emerges as the enduring bed-rock of your end-user experience and your architecture.

APIs endure, Front-Ends Don't.

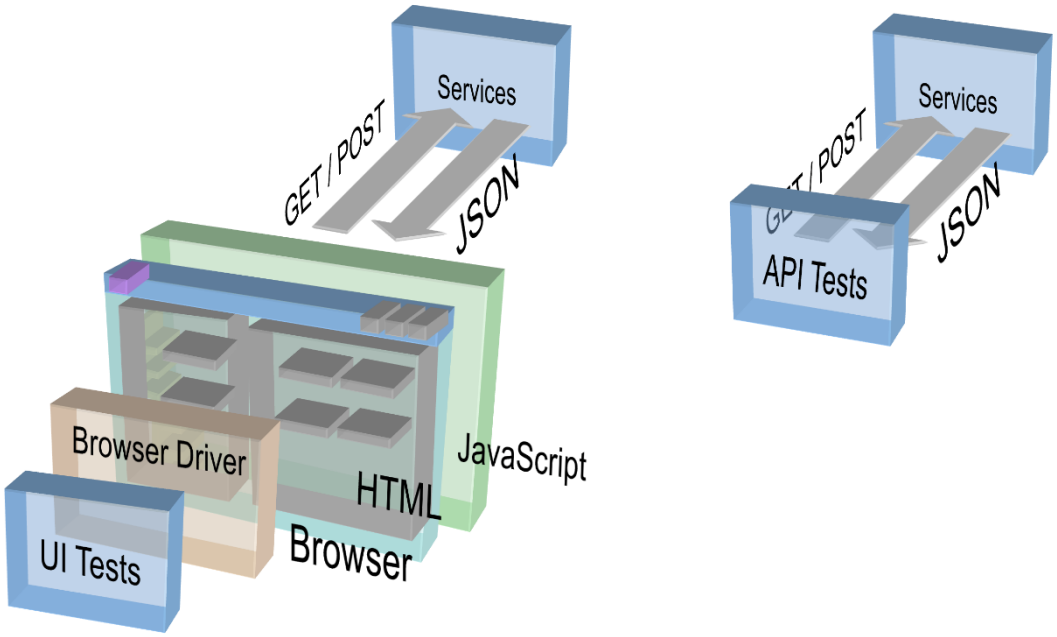
It is widely acknowledged that front-end technologies, suddenly pop into existence, but fade to obscurity as soon as a newer, shinier alternative emerges. This is the reason for a lot of memes and ridicule on the internet, and the term "JavaScript fatigue" comes up in conversation for good reason. Teams are constantly faced with the pressure of keeping up to date with the latest and greatest in the land of JavaScript. The pain of having to migrate from one front-end framework to another is well known. Teams that do not - will suffer the consequences of having to worry about support and the possible lack of security-fixes. Hiring new talent and employee-retention becomes increasingly difficult if the team is on an outdated technology stack.

Why is this relevant to this discussion? You may have heard the saying, "Nobody ever got fired for buying IBM". Architects and technology leaders bet on APIs or a microservice architecture because besides the benefits outlined in previous sections, APIs are a "safe" choice. The technology you choose for an API is very likely going to stand the test of time.

APIs are simple and tend to live on for long periods, because they can be easily versioned (which means multiple versions in production at the same time) or designed to be backwards-compatible, where payload changes will not impact older consumers and vice-versa. This is yet another reason for APIs being so pervasive and making it even more important to factor API testing into your automation strategy.

API vs. UI Test Automation

The diagram below shows where API testing fits in an architecture and how it compares with the typical test-automation set-up for a SPA (Single Page Application).



Note how significantly complex the UI automation stack is and the greater number of moving parts. This is the reason why:

- UI Automation is many times slower than API tests.
- UI Automation is flaky as there are many more failure modes.
- UI Automation is considerably harder to do correctly because assertions must be performed on unstructured output (HTML, images, and video etc.).
- UI Automation requires considerably more computing resources and skills.

We will go into detail in the next section and expand the categories of comparison.

We summarize the comparison in a table at the end.

The cognitive load on the person doing UI test automation is very high. One of the biggest challenges is waiting for elements on the page to load or refresh and this can be surprisingly hard to get right. For example, for tests written in JavaScript, knowledge of "promises" and how to do "async" and "await" is essential, and these are advanced concepts that can even trip-up experienced developers. Asynchronous code and JavaScript "callbacks" make it harder (or near impossible) to debug. Understanding the user-flow when reading such test-automation code becomes more of a challenge.

Here are the concepts one needs to master to do effective UI test automation in a browser:

- Locating Elements / Assertions
    - HTML / DOM
    - JavaScript
    - CSS
    - XPath
- Browser Driver installation and management
- Cross Browser Compatibility
- Waiting for pages or elements to load
- Handling multiple Browser Tabs / Windows
    - Switching tabs or windows
    - Window state, Maximize, Minimize and Restore
    - Handling other apps that may get in the way
- IFrames
- Popups / Modal Dialogs
- Screen Resolution
    - Responsiveness
    - Device emulation
- User Input modes
    - Scrolling
    - Mouse / Drag / Drop
    - Touch / Swipe / Trackpad
- Auth / Handling Cookies
- Handling Captchas
- File Upload
- Accessibility
- Visual Testing
- Running tests in the cloud
    - Headless Testing
    - Managing a Grid
    - Using Docker
    - Using a device farm

Contrast the above with the concepts one needs to know for API testing:

- HTTP
  - Methods
  - Headers / Cookies
- JSON or XML
- JsonPath or JS for assertions
- Authentication (OAuth, etc)

Keep in mind that API tests can run on any platform and don't require a display, so they are "headless" by default. This means that you don't need complicated setups to run tests in parallel. A normal machine will do – since tests can be assigned to multiple threads.

API Tests are Faster.

UI tests are notoriously slower to execute because of the resource-requirements, and the sheer time it takes to spin up a browser and login to an application. The number of aspects that need to be tested (e.g. visibility, buttons, links, dialogs) besides just testing core business functionality tends to add many steps to the typical test script.

APIs are designed for machine-to-machine communications. An API request has a very singular, pointed objective, typically to get or search for data or to execute a business transaction. Because an API call is the sending and receiving of pure data, it will be faster than the equivalent rendered in a UI.

API Tests do not require special infra setup.

The following are well known challenges with UI Automation:

- Hard to run in parallel.
- Requires more hardware.
- Hard to run in the cloud.
- Requires a browser installed.
- Only one browser can run at a time (on one node).
- OS and hardware specific variations need to be tested.

The complexity of the aforementioned factors is the reason why users leverage solutions such as SauceLabs, BrowserStack and LambdaTest.

UI Automation test-reports also ideally need screenshots, video-recording, DOM snapshots etc. All this adds to the resource requirements for running and reporting. It is normal for enterprises to sink a lot of time and effort into setting up grids or clusters to manage the execution of UI test-automation suites.

In contrast, API tests can run on a normal developer workstation on multiple threads. The typical build server can handle 50 to 100 threads with ease.

API Tests are stable.

UI Test automation is widely considered to be flaky. It takes a great amount of effort, discipline, and commitment to keep the CI build "green" when large suites of UI automation are involved. It is all too common to see teams give up and ignore failing tests in UI tests.

As detailed above, the number of moving-parts in web-browser automation is the main reason. Any slight change in network speed or DNS lookup when JS is loaded can cause test-suites to fail.

But the main reason is that simple changes to the underlying HTML and JS can cause the selectors that automated-tests rely on to break. In other words, changes to the application by developers that did not cause functional regressions can still break test-automation.

This is a widely discussed curse of UI automation and the reason why the goal of "in-sprint" automation for such tests is rarely achieved. Many approaches have evolved to just address this problem, such as "self-healing", using AI to determine locators, and visual navigation and assertions.

But nothing can beat the raw simplicity of an API test where you simply make a call, get and receive pure data and validate.

API Tests provide better Data Coverage

This is a subtle, and under-appreciated point. User Interfaces are designed for human-consumption, and this implies two things:

- The data pertaining to the business-transaction in play will be scattered across multiple pages and HTML elements. It is harder to perform assertions and many teams would just do some cursory checks and move on.
- A lot of data which is not relevant to the user such as audit trails, identifiers and flags will not be surfaced in the UI.

APIs are more "white box" than UI tests and typically contain a "future-proof" payload of all data related to the business-transaction. Since an API needs to service multiple consumers (browsers, mobiles, etc) and even future consumers, it is typically designed to surface all data that is relevant or even potentially relevant.

This means that if you test an API and run data-assertions on the entire payload, your confidence in the functional-coverage will be high.

APIs are easier to Mock or Isolate.

This is an obvious one, and User Interfaces are quite "monolithic". Even though the underlying technologies such as React, Vue etc. have a reputation for modularity and re-use, when you run an end-to-end test, you have to string together all your screens into one long flow.

In other words, if the objective of a particular test-automation script is to test just screen "C", the test-flow typically needs to perform a login and pass through screen A and B before you can finally start testing C.

APIs on the other hand, are designed to be composed into flows at run-time. This means that:

- You can directly call an API as long as you know the data that should be sent.
- If the API depends on a call to some other API, you can mock it.

Mocking a call to a third-party API is a very effective technique which teams don't use enough as it is perceived to be hard and not worth the extra effort. It enables you to focus your testing effort on the code you "own" and it saves time as calls to a mock can be faster and in-memory, typically in the order of a few milliseconds per call. A good API testing tool can make the business of using API mocks easy.

API Testing provides more Architecture Coverage.

User Interfaces are just one way to surface functionality to end-users. Testing a UI will not directly solve for other integration points such as

- Databases
- Message Queues
- Kafka
- Batch jobs and reports
- Command Line Interface (CLI)
- Asynchronous notifications
- Email
- APIs
- WebHooks
- Websockets
- gRPC
- Chatbots

On the other hand, APIs are a fundamental way to communicate between computing devices. A framework that is designed to be general-purpose and support multiple protocols can support not just HTTP and REST, but all the protocols listed above.

Since the focus is on preparing and validating data payloads (typically JSON), an API testing framework needs to treat these aspects as core and make it easy. Most UI frameworks are designed to focus only on UI automation. While some have added API testing as an after-thought, connecting other kinds of integrations is hard or just not practical.

In a good API testing tool, adding more protocols is just a matter of pluggable adapters and conversion of data-formats to and from JSON. Teams can even add support for custom or proprietary protocols.

As we will see later, performance testing is also far easier to achieve with an API testing approach.

APIs can better simulate end-user workflows.

Ideally, it should be easy to test a sequence of end-user actions. This is typically complicated because UI navigation involves multiple human-oriented patterns. For example:

- Login screen
- Captchas
- Buttons, Links
- Confirmation Dialogs
- Scrolling
- Expanding collapsed sections on the screen
- Images, Animations and Videos

Because there are so many possible paths that a human-user can take, front-ends are complex, and hence the testing of such interfaces is a very complex undertaking. While it is possible, it just involves more blood, sweat and tears.

Conversely, calling a set of APIs in sequence, while passing data and state around, is simple and can exercise most of the functional "surface area" in fewer steps.

Scenario Variations are simpler in API tests.

A single UI can have multiple things that can "go wrong" from the perspective of an end-user. For example:

- Some JS did not load
- The page is slow
- Some elements not visible because of bad design or CSS
- Button not enabled
- Multiple buttons clicked before action could complete
- Data hidden unless user scrolled
- Actual functionality broken

Whereas APIs are designed for machine-to-machine communication, and there is typically only one way to call a given API to perform meaningful work. In other words, it is very simple to determine if an API is working correctly - just invoke it and validate the response.

To put this another way, a good API test answers the important question: "is the back-end functionality correct" with a minimum of fuss. This is the highest priority for any test-automation endeavour and teams are well advised to put API testing in place before ascending further up the fabled testing pyramid.

API Tests can be re-used as performance tests.

It is not practical to re-use a UI automation suite as a performance test-suite. This is because of the challenges in provisioning multiple web-browser instances and the hardware and networking requirements this poses.

API tests are well suited to be re-used as performance tests. They can be invoked using multiple threads even on a single node and can be effective at determining if the server can handle concurrent load and stress for extended periods.

The ease of mocking APIs also comes in handy here. You can perform a performance test while still ensuring you don't send traffic to 3rd party APIs you depend on.

There are very few tools that allow you to re-use functional tests as a full-fledged performance tests. This should be an important factor to consider when you evaluate a test-automation tool.

API Tests make Test Data Management easier.

End user flows through a set of screens tend to obfuscate the actual data relevant to the business use-case. While possible, it is a lot harder to tease out the equivalence partitions and boundary cases, when planning a UI test script.

APIs focus purely on the data that is needed to achieve a business transaction. Teams that focus on API testing find it easier to achieve and validate that the test suite achieves coverage from a functionality and payload schema point of view.

API tests make Data Driven and Parameterized tests easier.

Due to the sheer time taken for a UI test and the previous point where it is hard to de-couple the pure business-data from a UI test, it is much harder to loop over or parameterize a UI test. While it is possible, teams end up having to fight issues such as flakiness or waiting for pages to re-load.

Validating that the data has been entered into the system correctly is also non-trivial and involves having to inspect HTML and scraping data out of it. Because API tests are near perfect representation of the data needed to achieve a business-transaction, it is very straightforward to loop such tests over a data-source such as a database table, excel-sheet or CSV file. It is possible to even run such data-driven tests in parallel.

API Tests are easier for non-programmers and domain-experts.

While UI tests approach the need of test-automation from an end-user perspective, writing a typical UI test-automation test is a technically challenging task. This may sound counter-intuitive, but it is because of the underlying complexity: HTML, JS, CSS, Locators and XPath, the severe headache of "waits" and the contortions required to assert unstructured data embedded in HTML.

While no-code tools attempt to bridge this gap, it is well-known that such approaches come up short when it comes to data-driven tests and script-ability needed in an enterprise context. Such tools cannot handle integrating with a database or external async communication such as message-queues or Kafka.

As API tests focus on the data required for the business-transaction, they are easy to read and understand by domain experts. Many teams have found that product-owners and business-analysts were able to contribute to API tests.

A good testing tool would be able to abstract the minimal details of the HTTP method, URL, and headers so that only the scenario data is surfaced within the test-script.

This also has the effect of making API tests readable, and hence more-maintainable.

API Tests are more effective as living documentation.

This is a consequence of some of the points already covered such as simplicity, a focus on data and core-business logic – but we feel that this is under-appreciated and deserves to be called out separately.

The inherent complexity in UI automation means that scripts typically have to incorporate what a human user can do, navigating, clicking, scrolling etc. and then attempt to scrape data out of unstructured HTML to do assertions. All this while having to fight against slow-loading elements, using "waits" or "async / await".

What this means is that even a simple business transaction such as "create order" will end up as a long script filled with clicks, waits and HTML DOM operations. But an API test can just focus on the data payloads required to perform a business-transaction.

This makes an API test extremely readable by design. You will be able to eyeball not just the business operation, but the data that was sent and / or received. No other steps or visual concerns will "clutter" the test-script. Tests are strikingly concise. A good API test automation tool will include a capture of the actual HTTP traffic in the HTML report of a test run. This HTML can be easily shared with project stakeholders.

Thus, API tests end up being far more effective as serving as "living documentation" than UI tests.

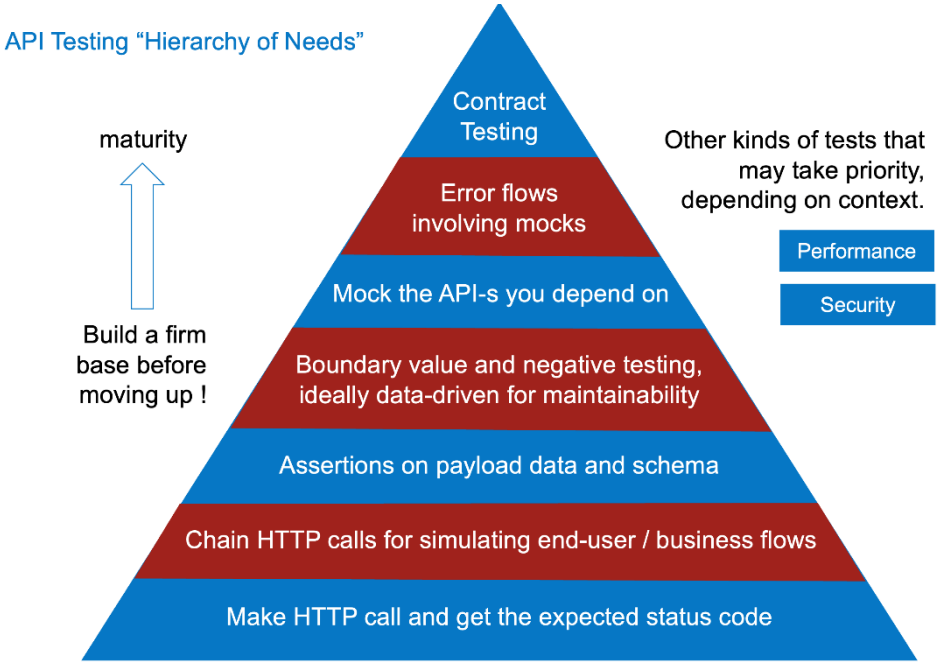
UI vs API Test Automation Summarized

	UI Testing	API Testing
Complexity	High	Low
Speed	Very slow	High, can be run in parallel
Resource Requirements	Very High	Minimal
Stability	Flaky	Very Stable
Data Coverage	Hard to achieve	Simple to achieve, high confidence
Mocking / Isolation	Hard or near impossible	Easy
Architecture Coverage	Only the UI layer	Can easily involve other layers
Simulating End-user Workflows	Hard	Easy
Variations	Many, and hard to cover	Simpler, focused on business-logic
Server Performance Testing	Not possible	Can be re-used effectively
Test Data Management	Hard	Easy
Dynamic / Data-Driven Testing	Hard	Easy
Programming skill required	High	Minimal, easy for non-programmers
Value as living-documentation	Low	High

API Testing - Hierarchy of Needs

Any team that is into API Testing may find this a handy guide to asking questions about the health of test automation. Are you actually checking that the data values that are being returned by the server are as expected? If you aren't - stop everything and fix that right away!

The order and priorities of the various layers below can indeed vary across teams - for example, it is debatable whether negative tests should be more fundamental and appear lower down.



The term "Contract Testing" is not well understood, and each test-automation tool vendor tends to have a different spin on the subject. For a more detailed discussion of the subject, please refer to this article.

API Mocks

Clearly, you don't need mocks if you don't depend on any external service. You may not need mocks if the services you depend on:

- are completely owned by you,
- never change,
- or have readily available test environments.

The classic situation where you need mocks is when the API or service you depend on is owned by a different team, and in extreme cases is even beyond the boundaries of your top-level organization.

You will need mocks in cases where a test-deployment of the service is not available or is so unreliable that it slows you down. Not having a "test environment" that is strong enough to handle performance testing is a common problem. And another extreme case is when the service you depend on is being developed at the same

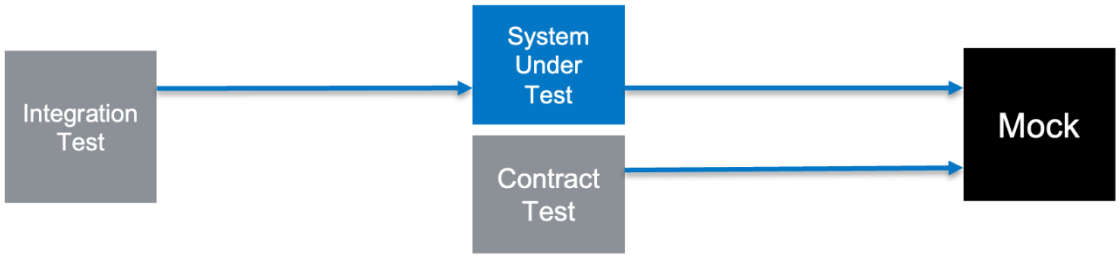
time - which means you have only two options - either wait for a version to appear and be deployed - or use some kind of mock.

When you evaluate an API mocking tool (some vendors refer to this as "Service Virtualization") make sure that it can help you craft "high fidelity" mocks, else they will be very limited in their use.

	Low Fidelity Mock	Medium / High Fidelity Mock	Real Service
Security	Not supported.	Typically not supported, but some scenarios can be scripted with extra effort.	Handles Authentication and Authorization.
Performance	Cannot handle concurrent requests.	Can handle concurrent requests. May not be as fast as the real thing.	Can handle concurrent requests and scale linearly when load is increased.
Request Payload Validation	Limited schema validation (or not supported)	Schema validation and business-rules can be scripted with extra effort.	Schema validation and business rules in place.
State	Not supported, always returns canned responses.	Uses an in-memory data-structure to manage or lookup previous requests. Can detect duplicates and return errors. Data will be lost at the end of the test.	Uses a persistent data-store. No constraints on the amount of data you can use.

With a good mocking strategy in place, you will be able to run your whole test-automation CI pipeline without needing to include 3rd party services into your deployments. This can be a huge competitive advantage for your team.

When you use an additional set of tests that play the role of contract-tests, this means that you can continuously trust the mock to be a valid replica of the real system that you ultimately depend on.

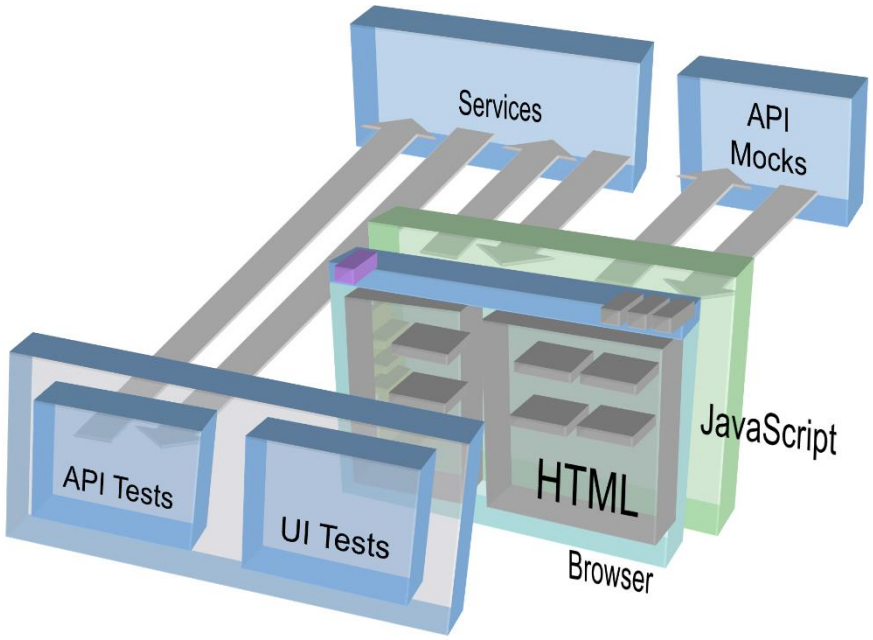


Hybrid API and UI Tests

We have discussed at length on how APIs can be the core, if not full focus of test-automation. That said, there are many applications where the user-interface is key and the primary-means by which end-users experience business-benefit. API testing cannot fully replace UI testing in such cases, and approaches such as Exploratory Testing, Context-Driven Testing are of value.

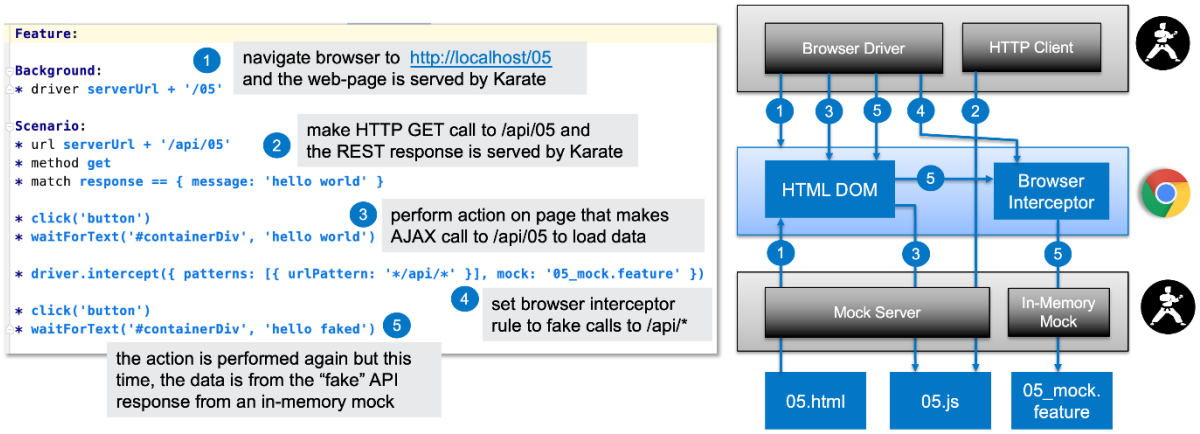
When it comes to regression test-automation suites, teams that have achieved maturity in core API testing can aspire to perform hybrid-testing, where a test-suite can switch between calling APIs or driving a UI even within the same test-script. This has number of advantages:

- Fewer tests needed to exercise more coverage, especially of architecture layers, and reduces test-time.
- A hybrid approach can speed up tests significantly. For example:
    - Use an auth token from an API call instead of typing in the username and password in a login screen and save almost 5 seconds per UI test.
    - Set up the database state via an API call and then perform the UI test instead of having a time-consuming setup phase where multiple transactions are entered via the UI
- API mocks can be carefully leveraged to fake data needed to test some extreme negative scenarios. For example, to test what happens if we transfer large amounts of money when a 3rd party API is involved.
- A UI test run at the same time an API performance test is on can provide insights into how the end-user-experience is affected under load.



As shown above, when you use a suitably capable test-automation framework, it is possible to involve API calls, fake API responses (via mocks) and UI automation into a single flow of a test automation script.

Here is a real example of Karate being used to achieve this. Note how the Domain Specific Language (DSL) keeps things simple, even though there is a significant level of architecture complexity behind the scenes.



Hybrid Tests for Databases

This is called out as a separate section because databases are very common for data-persistence. Like how Hybrid API and UI tests make sense, the same advantages hold good for tests that include the database.

These tests are more "white-box" in nature than standard e2e integration tests, because they assume a couple of things:

- Your tests have direct connectivity to the database, and not just the external entry points (API & UI)
- Your tests have knowledge of the data-schema in your database and what is expected.

Patterns that make sense are the following:

- Setting up initial data before commencing an e2e test. This is easier to do nowadays because of technology advances such as Docker and Cloud Development Environments such as GitHub Codespaces. Spinning up containers or nodes to hold test-data is easy and can be done quickly, which makes this a viable test-strategy. The opposite of this is a team having to share the same instance of pre-prod test-data. This means the team is constrained, because they have to write tests that have to account for the fact that anyone can modify the database during each test.
- Checking the database after an API or UI transaction to ensure a "write" or "update" has happened correctly. There can be data elements that are not surfaced in the API or UI but need to be tested because of business importance or risk. For example, audit data related to GDPR and compliance.

Beyond HTTP – Async and Event Driven Systems

Beyond REST and GraphQL, there are more channels of communication, especially within the enterprise firewall. Examples are WebSocket, gRPC, Kafka, message-queues such as ActiveMQ etc. Even within the world of HTTP, WebHooks are a common pattern and need to be tested.

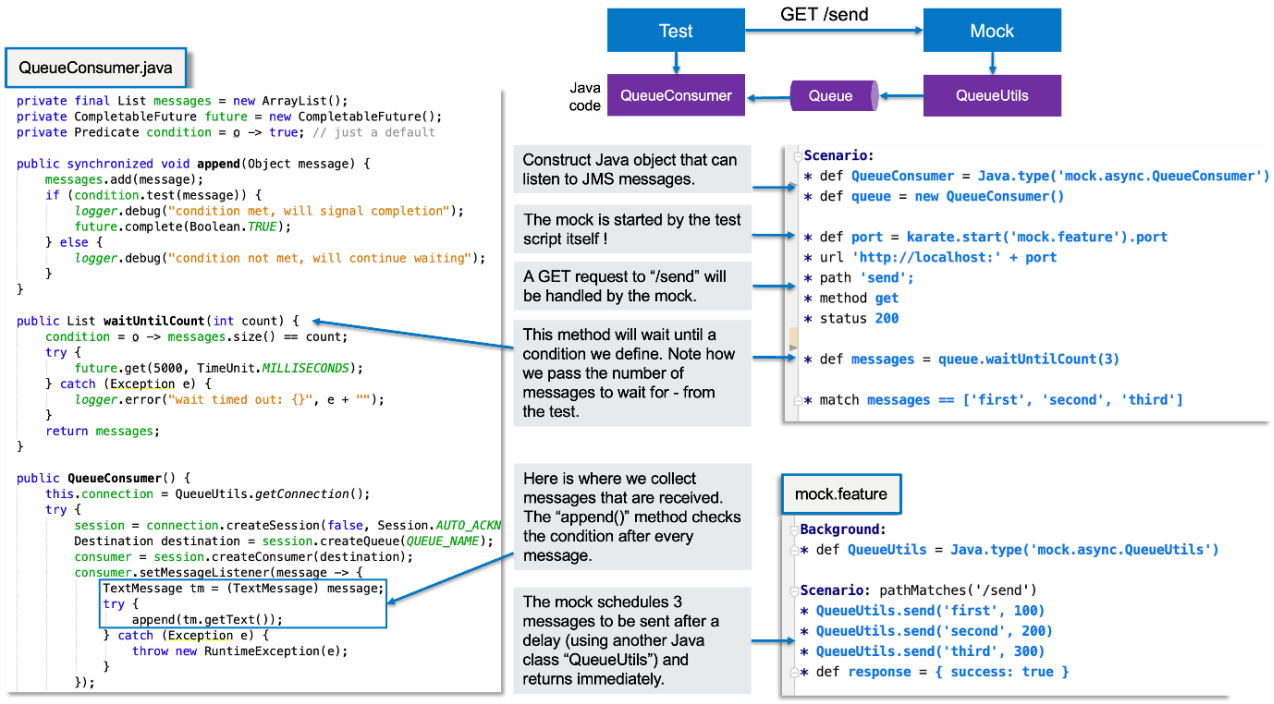
Such communication modes are also increasingly common within cloud and distributed deployments on AWS, Azure, GCP etc.

Payload-formats or schemas can be types other than JSON, for example ProtoBuf, Thrift or custom binary-serialization. Fundamentally the concept of message-passing is the same as REST, but in fire-and-forget fashion (async).

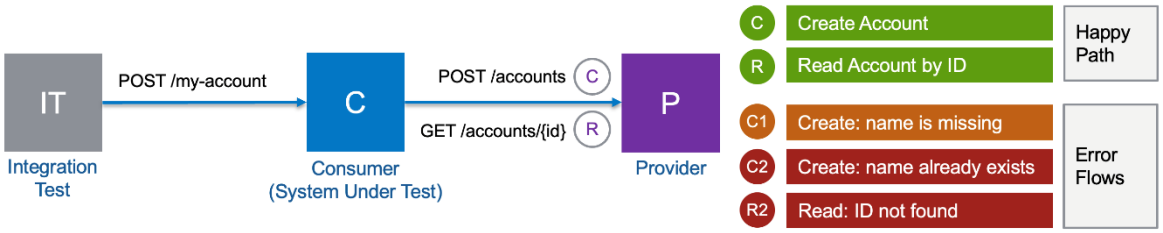
Factoring these non-HTTP communication channels into an end-to-end test-automation strategy is important because these channels can be as foundational as APIs, and augment the end-user-experience.

Asynchronous flows require more thought than simple request-response flows like REST. But they are easy to implement with a test-framework that supports asynchronous "waits". Being able to integrate the communication-channel directly via glue-code is also key.

Here is an example of a very complex flow that involves an API mock (that can work as a WebHook call-back) and a message-queue. Note again how the test is simple, can leverage the power of Java where it makes sense (one-time glue-code) – and the script (on the right) is very readable and concise.



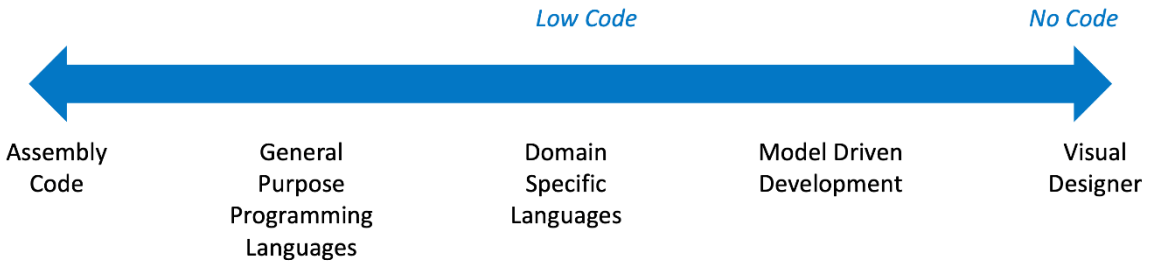
Happy Paths and Negative Scenarios



Good API tests focus on the "happy paths" and then the "negative" scenarios as shown in the simplified diagram above. A test automation tool that supports data-driven tests and parametrization of tests is critical to be able to do this and thus achieve high functional coverage of your code.

Low-Code vs No-Code

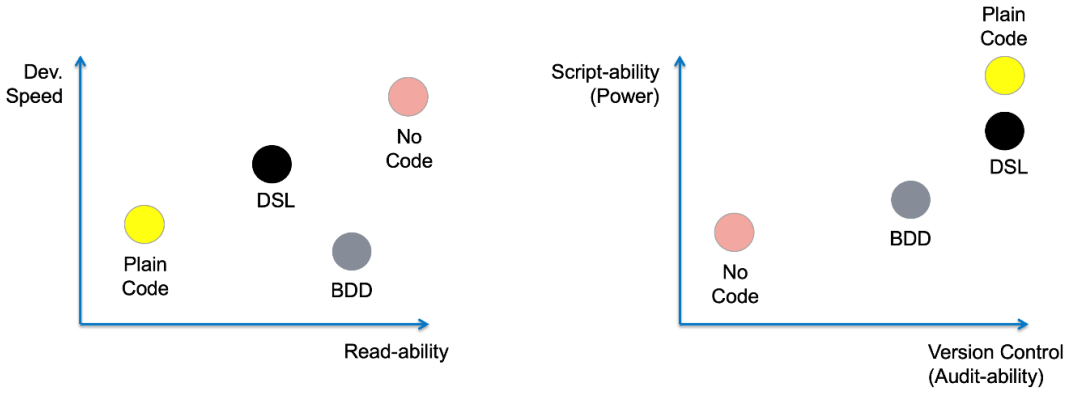
Some things are best expressed as code. Development approaches lie on a spectrum with assembly-code on one end and visual-design on the other. There are clear trade-offs and one needs to be aware of the implications when choosing a test-automation tool.



The aspects you should consider when evaluating a no-code solution are the following.

- How easy is it to re-use code?
  - Can you loop over some actions?
  - How do you pass parameters?
- How are data-driven tests supported?
- Can you easily switch environments?
- Can you run "headless" or in the cloud for CI?
  - Does this require a separate tool to be installed?
  - Do you need extra licenses for each CI node?
  - Can you run tests in parallel?
- Can you write code directly when needed?
- Are the assertions powerful enough?
- Can you step-through & debug?
- Can you version-control, diff and see history for your tests?
- How maintainable are large tests?

As the graphic below shows, there are distinct trade-offs. A low-code DSL (Domain Specific Language) strikes a balance between ease of writing, power & flexibility, and version-control-friendliness.

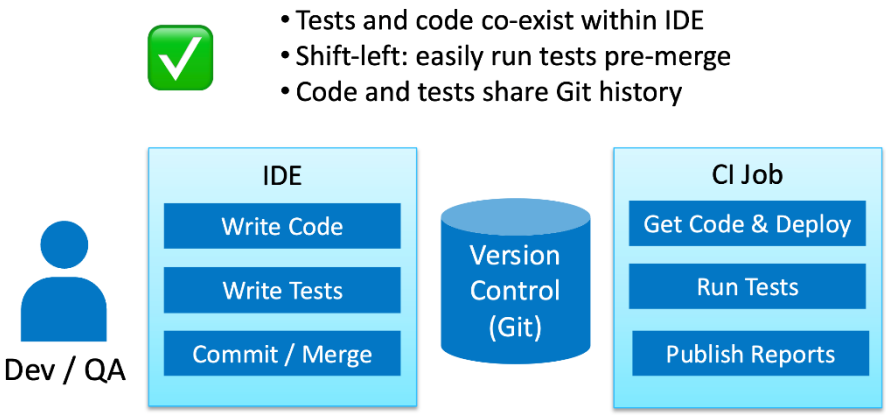


Test Automation should be a first-class citizen of your development pipeline

Just like some things are best expressed as code, some development workflows are most effective when your code is managed in an industry-standard version-control tool such as Git.

Unfortunately, many test automation tools force the team to break out of their IDE and state-of-flow into another user-interface. Some SaaS tools require your test data and artifacts to reside in the cloud which is not an option for enterprises with strict data-security standards. The main issue with such a set-up is that test-automation is disconnected from your primary version-control and you will not get the collaborative benefits that teams naturally enjoy with Git.

The ideal state is shown below. Developers and QA have equal access to write, run and maintain tests.



For further reading on this topic:

- The Test-Automation Capability Map
- Is test-automation a first-class citizen of your development pipeline?

How do I convince my development team to embrace API tests?

We added this section to address a frequently asked question when some organizations start transitioning their teams to do more API tests. The modern developer has a lot of responsibilities, including hiring, mentoring, code-reviews, dev-ops, observability, and on-call support. It is not surprising that when there is an ask to put API test-automation in place, the knee-jerk reaction is that "we already do unit-testing" and API testing is a "waste of time".

Developers just need to realize that with the right tool, API testing enables the QA teams to independently write effective functional-tests without any handholding. There are things that unit-testing cannot do, and an API regression test-suite does wonders for the velocity and confidence of the whole team. We have created this handy reference that can help you make the case for API testing in your organization and facilitate a discussion with developers who may need convincing.

	Unit Testing	API Testing
Can QA contribute	No, typically the responsibility of developers and requires knowledge of the internals (white-box)	QA and developers can equally contribute and share the load. This is a great way to shift-left.
Are business flows tested	Focuses only on slices of the system in isolation	Tests the system end-to-end and simulates real-life usage by end-users.
Tests network communication and schemas	Unit tests typically run "in-process" and do not make network calls	HTTP and network interactions and payload schemas are validated. You get architectural coverage.
Can be re-used as system performance tests	No, since tests are designed to be fine-grained and isolated	Yes, and this is a compelling reason to prioritize API tests.
Useful as living-documentation	Unit tests are low-level and not designed to be readable as specifications	API tests have a reputation for being readable, focused on the scenario data and end-user workflow. Many teams rely on the HTML reports for understanding what the system does.

Origin of Karate

Peter Thomas (creator of the Karate test-automation framework) was part of the API platform leadership at a Fortune 500 company where his team was responsible for a set of 15 services. It was early December 2016, and Peter was in the process of troubleshooting one of the automated tests that his team was working on, and he was trying to solve an issue which had been slowing down the team – which was that a particular test for the core accounting services system was unstable. The test would randomly fail, and this was blocking a production release - because it was not clear if there was a problem with the test or if there was a genuine defect. The test that Peter was looking at was implemented in the Java programming language and it also depended on a framework created in-house, which had evolved over a period of at least 3 to 4 years.

Peter was troubled. Despite his many years of Java programming experience, it was very hard to understand what the test was doing. The test depended on code contained in multiple files scattered across the workspace, and this only made matters worse. It was clear that many programmers had attempted to fix this test over the years, and Peter started to think hard about whether there was a better way to express web-service functional tests.

This gave birth to Karate. What started as a powerful, scriptable framework combining API and UI test automation, is adopted as a best-practice today - in teams around the world and is used by 42 of the Fortune 500 companies.

In our continued mission to democratise testing, we decided to leave our day jobs and work on Karate full time. We incorporated Karate Labs in Nov 2021 as a for profit company that will ensure a balance between the need to improve the open-source code and building premium paid for features. Paid features will give us the ability to hire more developers to work on both versions of the product, to continue releasing more features and make test-automation easy and fun for business stakeholders, product-owners, developers & QA specialists.

- Introduction to Karate in 4-minutes.
- How Karate elevates developer experience.

Customer Blogs

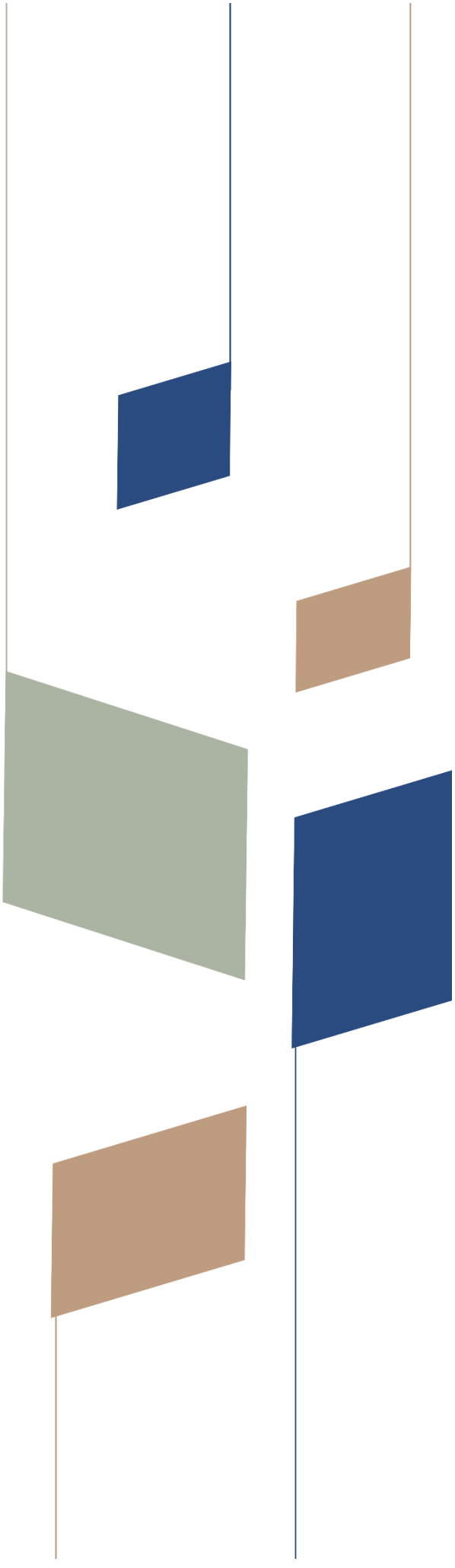
- Walmart Global Tech Blog : KAFKA Automation using KARATE
- Expedia Group Technology : Karate: 5 reasons why you should try it
- Tech at Trivago : Automation-First Approach Using the Karate API Testing Framework
- Oktana : API Testing with Karate Framework
- Globant : Karate API Testing

Customer Videos

- Adobe : Karate, the black belt of HTTP API testing?
- SAP : How the SAP Commerce Cloud team found their Automation Mojo in Karate?
- FIS Global : A Non-Programmer's journey to test-automation with Karate.
- Oracle : A Developer's Journey with Karate.
- Trivago : API vs. UI Testing.

Get started.

- https://github.com/karatelabs
- Official Visual Studio Code Extension
- Official IntelliJ Plugin



info@karatelabs.io

www.karatelabs.io

https://github.com/karatelabs