GUIDE TO TESTING INSURANCE SOFTWARE

Karate Labs

Testing insurance software requires a different approach taking the extra complexity into account. It is a useful guide for teams delivering digital transformation for insurance containing tips and ideas on how to address the challenges and what to focus on for success.

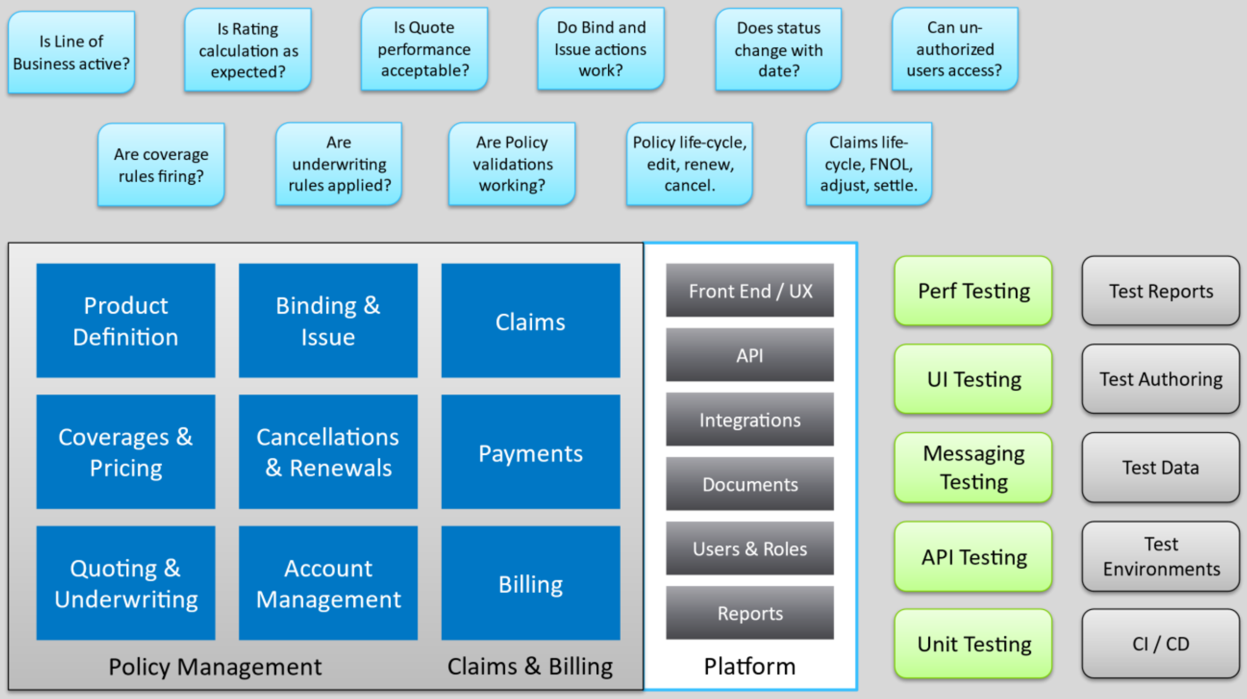
The document contains the following sections:

Why is testing insurance software so hard?

Software testing for the insurance domain requires expertise in not just the enabling technologies and platform, but a deep understanding of the business flows, users and stakeholders.

One of the key characteristics of Insurance is the complexity of business-logic – for example rating and premium calculation, and the different kinds of personas that must be served – e.g. product-owners, rating analysts, developers, underwriters and actuaries. Keeping up to date with industry regulations, staying on top of compliance and keeping the business profitable in a highly competitive environment is a stiff challenge.

Quality can refer to functional accuracy and the smoothness of the end-user experience. A lack of quality can impact the business in multiple ways. For example, if products are priced or quoted incorrectly, this affects the bottom-line. If policyholders are not shown contextually accurate data, the consequences can impact the brand or result in additional legal expenses. If quotes are not available to users immediately, they will promptly move on to other providers.



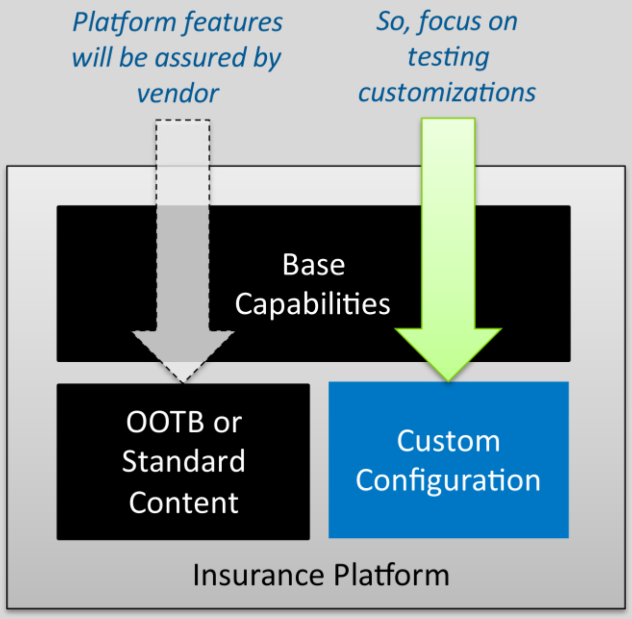
Test Automation is critical for being able to **release customer-benefit faster** and more often – for competitive advantage. Tests elevate confidence in the quality of your releases, and help you keep up with the fast pace of what the industry demands. And a test-automation tool that can support all the aspects shown above is key.

Insurance Testing Challenges

Content and Customizations

The range of Insurance products is vast and there are many different factors that drive their design. For example, rules vary depending on country, state or sub-jurisdiction. There is certainly a need for shared commonalities in the industry which is why standards such as ISO ERC (Electronic Rating Content) and ACORD exist in the US. Yet, there is pressure on insurers to differentiate, and they extend or tweak offerings in multiple ways. Differentiation can be based on cost, and extra clauses depending on the risk-appetite of the insurer. Personalizing the user-experience for specific personas is common.

What this means is that the leading insurance platforms are designed to be highly configurable and customizable. These platforms typically provide extension points that must be implemented as code or via configuration, or a more business friendly or visual way to define custom rules, rates, coverage clauses and calculations. Implementing, testing and taking these customizations to production is what takes up most of the time in any insurance technology project.



On the Insurance provider side, while it is tempting to write a few tests for the Out Of The Box (OOTB) functionality, it should be clear that testing the base platform is the responsibility of the vendor. Teams should focus on writing tests that exercise the parts of the Lines of Business (LOBs) that have been customized. For example, policy clearance before creation, rules, and quotation through issuance.

We have seen teams make the mistake of writing tests for OOTB functionality which should be avoided, as it is unnecessary effort and leads to a false sense of security.

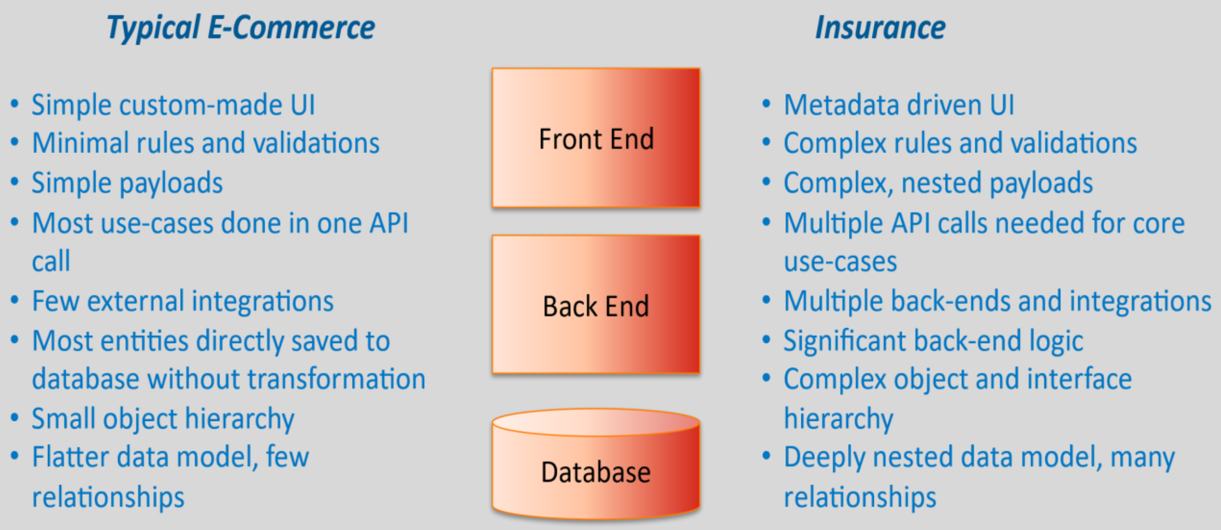
Calculations and Business Rules

The insurance domain has a high degree of math and logic compared to what you expect from a typical e-commerce implementation. This means that test-data creation becomes increasingly more difficult yet critical. Testing teams will need to model different variations and permutations as test-scenarios and encode them for testing.

A simple example is to model 10 coverages, each with 2 terms, where 15 rate barring elements can be collected from the policy applicant. This implies 10 * 2 * 15 = 300 permutations to validate the accuracy of the implementation.

Automation is key or else the team would be spending significant amounts of time going through the motions manually.

Testing needs to be performed in close collaboration with the business and domain experts. Tests should be written or at least reviewable by the business, which is why development approaches such as Behavior Driven Development (BDD) are popular in the pursuit of collaboration. A good testing tool should enable the business, developers and QA to have a shared understanding, instead of having them work in silos.


Typical E-Commerce
Simple custom-made UI
Minimal rules and validations
Simple payloads
Most use-cases done in one API call
Few external integrations
Most entities directly saved to database without transformation
Small object hierarchy
Flatter data model, few relationships
Front End
Back End
Database
Insurance
Metadata driven UI
Complex rules and validations
Complex, nested payloads
Multiple API calls needed for core use-cases
Multiple back-ends and integrations
Significant back-end logic
Complex object and interface hierarchy
Deeply nested data model, many relationships

Metadata Driven User Experience

Handling automation-breaking changes in the UI is one of the hardest problems to solve in web-browser automation.

A characteristic of Insurance solutions is that they are the result of customizations or configuration of a base platform. What this means is that the user interface is increasingly generated from metadata, as opposed to being hand-crafted by front-end developers. This change has significant implications for automated UI testing.

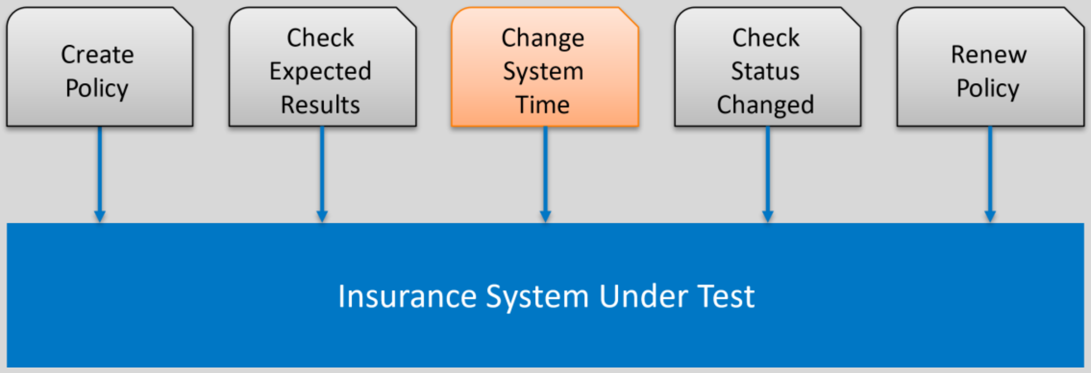
Most test-automation tools have challenges identifying components and performing actions in dynamically generated UIs. For example, when the HTML is dynamically generated, XPath and CSS locator-dependent tools such as Selenium make it harder to write tests, and results are unpredictable at run time.

The solution is to enhance test automation tooling to have access to the metadata, configuration and code that defines the user-interface. API testing tools are well suited to retrieving metadata from the back-end. We have seen other approaches such as converting the configuration to page object models (POMs) as a build-time step. Each approach has pros and cons.

While code-generation can accelerate initial outcomes, it can add overhead and constraints to the development process. For example, code generation may have to be redone every time the front-end changes.

The Effect of Time

Insurance is the selling of written promises, as contracts, referred to as policies. Policies flow through life cycles, with time-based events such as renewals, and ad-hoc events such as policy changes (i.e. contract amendments). It is not practical to wait for months to elapse to test certain end-to-end insurance flows.



This is why insurance platforms support the concept of “Time Travel” testing where in development mode, you can request the server to change the system clock and “fast-forward” to a date in the future. This enables testing all the business-rules and status-changes that apply across the life cycle of a claim or policy.

When the time-travel mechanism is available as an API call, this means that in a single seamless test-flow, you can test a life cycle that normally takes months - in a matter of seconds. This is another reason why API testing is better suited to test Insurance flows, as time-travel is easier to switch on via an API.

Personally Identifiable Information (PII) and Sensitive Data

Digital solutions for insurance have the responsibility of collecting and presenting large amounts of PII.

Tests often reflect production realities. While it is common to create dummy users and addresses for pre-prod environments, configuration details such as the locations of servers and API URLs would invariably be part of tests. This is also sensitive information especially as a lot of test-environments today live in the cloud, in environments such as AWS, Azure or GCP.

This implies that you should hold your test-scripts, test-data and test-configuration within your private network. A test-automation tool which is “local first” is the right choice. Such tools have the additional advantage of fitting naturally within your developer Integrated Development Environment (IDE) and environments such as Git and your CI / CD pipelines.

If you are using a tool that forces you to store test artifacts in “somebody else’s cloud”, this is a security risk that most insurance providers are not willing to take.

PII is also a reason why test-automation approaches that involve taking a snapshot of data from production need to perform a data-masking or obfuscation step before the data can be considered ready for use in pre-prod environments.

Data Size and Permutations

Proper testing of Insurance scenarios requires many scenarios and permutations of input data to be run through the system under test.

One approach is to take a snapshot or “cut” of production data, pre-process it to remove any PII or sensitive information and then use it for regression testing. In theory, if the result is unchanged, the system is working as expected.

This does however run the risk of “preserving” bugs un-intentionally. But, it is a legitimate approach that many teams take to validate platform upgrades, or when the team does not have the time or resources to invest in hand-crafting the test-scenarios and edge-cases needed for comprehensive testing.

Even when tests are hand-crafted, the volume of test data needed for comprehensive testing is high.

Data Prerequisites

Insurance entities stored in the database have multiple dependencies and strict validations. For example, a policy has to be associated with an account, and the account has to exist before a policy can be created. Policy contacts and the underwriter need to be created beforehand.

Tests take two distinct strategies. One is to reset the database with a set of “golden data” before running a test suite. This carefully curated database will have a known set of product lines, accounts, organizations and rules which all tests can depend on.

The other approach is to create all dependent entities as a “setup” step when the test suite is run. Both approaches have pros and cons

Regulatory Changes

Keeping track of changes is an ongoing challenge and teams are constantly playing “catch up” when it comes to test-automation.

The modern developer is burdened by the need for speed and multiple expectations. The responsibilities include requirements, design, architecture, coding, testing, releases, monitoring, observability, production support, hiring, etc.

A testing tool that empowers QA or business users to create tests without hand holding from developers will increase team velocity. API testing can help teams achieve the goal of “in sprint” automation.

Platform Upgrades

When the time comes to upgrade to a new version – this can be the source of a lot of stress. It can be hard to manage insurance platform upgrades while ensuring customizations are not impacted.

Without the safety net of a comprehensive test-automation suite, the team will have to perform manual testing using the UI, which takes time and reduces confidence.

Localization

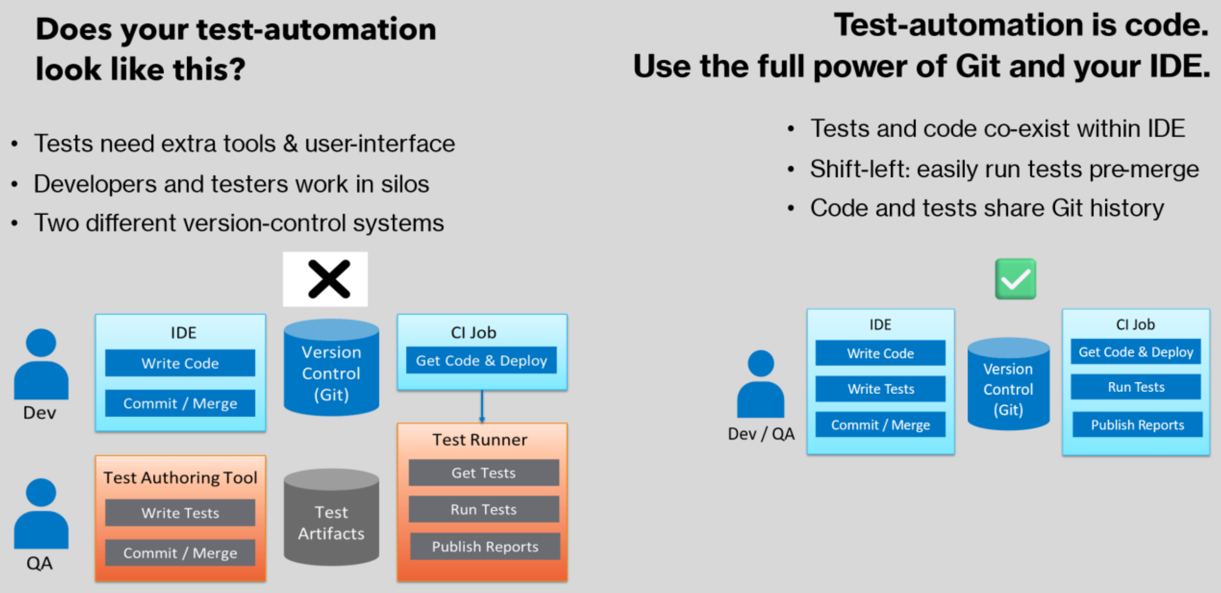
Tests may need to account for currency changes and how monetary amounts are handled and displayed. Validating that the user-interface works in all languages is also important.

Factors to Consider when choosing a Test Automation Tool

Local First

Just like source code, test artifacts (data, secrets and cloud URLs) should remain within the team's private network or cloud.

A test-automation tool should enable developers and QA to collaborate within the IDE and naturally gain the benefits of Git-based version control.



Does your test-automation look like this?

- Tests need extra tools & user-interface
- Developers and testers work in silos
- Two different version-control systems

**Test-automation is code.
Use the full power of Git and your IDE.**

- Tests and code co-exist within IDE
- Shift-left: easily run tests pre-merge
- Code and tests share Git history

Multiple Modes - API, UI and Integration

API testing hits the sweet spot of speed, ease of writing tests and being able to maximize functional coverage while remaining "black box" and "out of process".

That said, there will always be the need to test the UI. The user-interface is the primary channel by which customers interact with the software. If the UI does not work or is not usable, then all bets are off.

We cover the pros and cons of UI vs API testing later in this document. Given the numerous challenges with UI automation, we recommend a strategy where there are fewer UI tests, which only focus on:

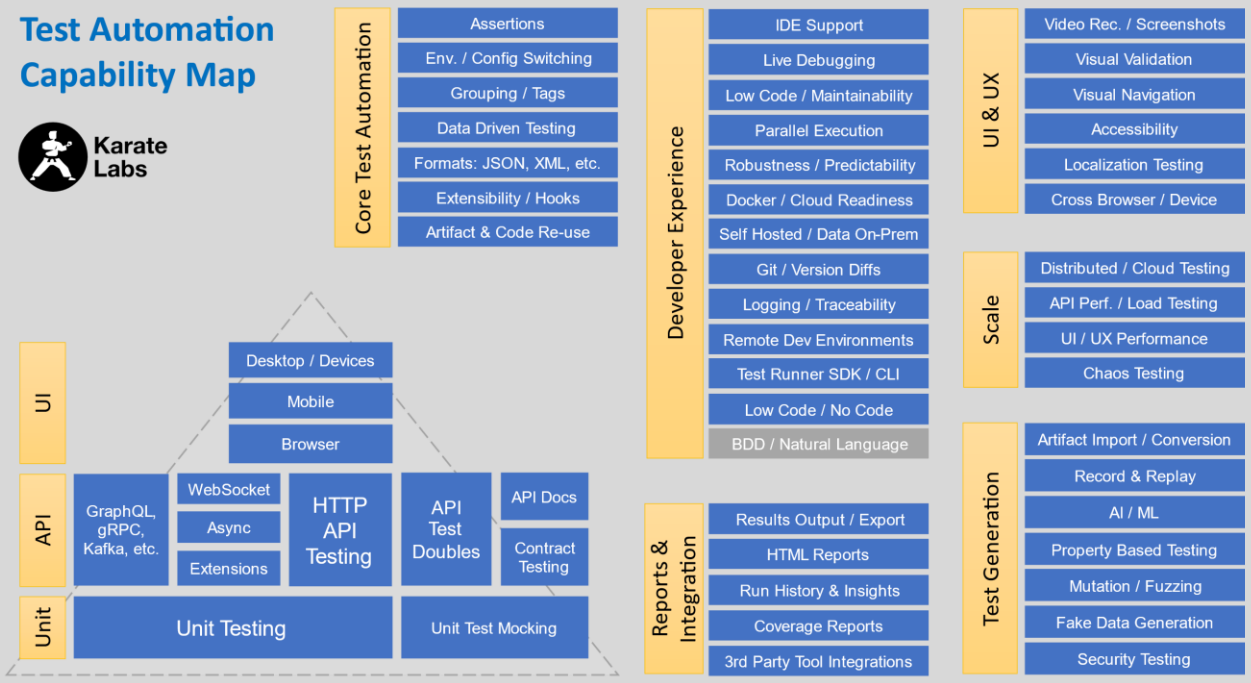
- Usability of widgets and on-screen components
- Whether data can be persisted and retrieved for "happy path" scenarios
- Whether the validations work and show up on the screens correctly.

All data-driven testing of boundary cases and negative edge-cases can be left to API tests. This will ensure that tests run fast, which is a key hygiene factor for any development team. When tests take more time to run, it reduces team morale and decreases the motivation to write more tests.

The ideal test-automation tool should be able to support all the following:

- Unit Testing
- API Testing
- Async Messaging Testing
- UI Testing
- Performance Testing

We present a capability map to increase awareness of all the finer details that should be considered when planning a test-automation strategy.



A point to note is how “busy” the API layer looks in the test-pyramid depiction on the bottom-left. This is because JSON over REST is not the only game in town. SOAP is rare nowadays, but we are seeing more of EDA (Event Driven Architecture) in the enterprise. This means that Webhooks, WebSocket, Kafka and gRPC are becoming common especially to handle inter-application communication requirements.

For example, in insurance flows, you would want to listen for any life-cycle events in a policy management system and use those to trigger business processes in a billing or payments system. A very common pattern is to use an external rating engine.

Investing in a test-automation tool proven to handle the multiple technologies needed for end-to-end testing of the typical insurance platform stack is recommended.

Developer Experience

A test-automation tool that seamlessly integrates into the IDE is the ideal developer experience. Many teams are slowed down when developers and QA work in silos because they are using completely different tools and decoupled workflows.

Tool Integrations

IDE and Git integration are fundamental, but there are a few other DevOps aspects that a test-automation tool needs to support or integrate with. For example:

- Test Reporting Tools
- Test Management Tools
- Test Data Generation Tools
- CI / CD Tools
- Docker
- Cloud Environments: AWS, Azure and GCP

Tools that support these integrations are also likely to be future proof, with indications of a large community and information on best-practices available on the internet. A question to keep in mind is how easy it is to find skills in the test-automation tool and whether a team can quickly get support in case of questions or when stuck.

Parallel Execution

Large test-suites can take a lot of time to execute and it is important to be able to use multiple threads or computing nodes when executing tests. Most test-automation tools do not support the execution of tests in parallel and generating a single aggregated report. This is a very important factor to consider when evaluating tools.

Extensible

It is important for a testing tool to be easily extensible. This is especially important when testing a complex platform or when there are aspects of your environment that require special handling where general-purpose approaches will not cut it.

For example, setting up master data or initializing the database is needed to test complex insurance flows such as policy submission. With a good test-automation tool, you can plug in code that executes SQL queries or restores a database snapshot.

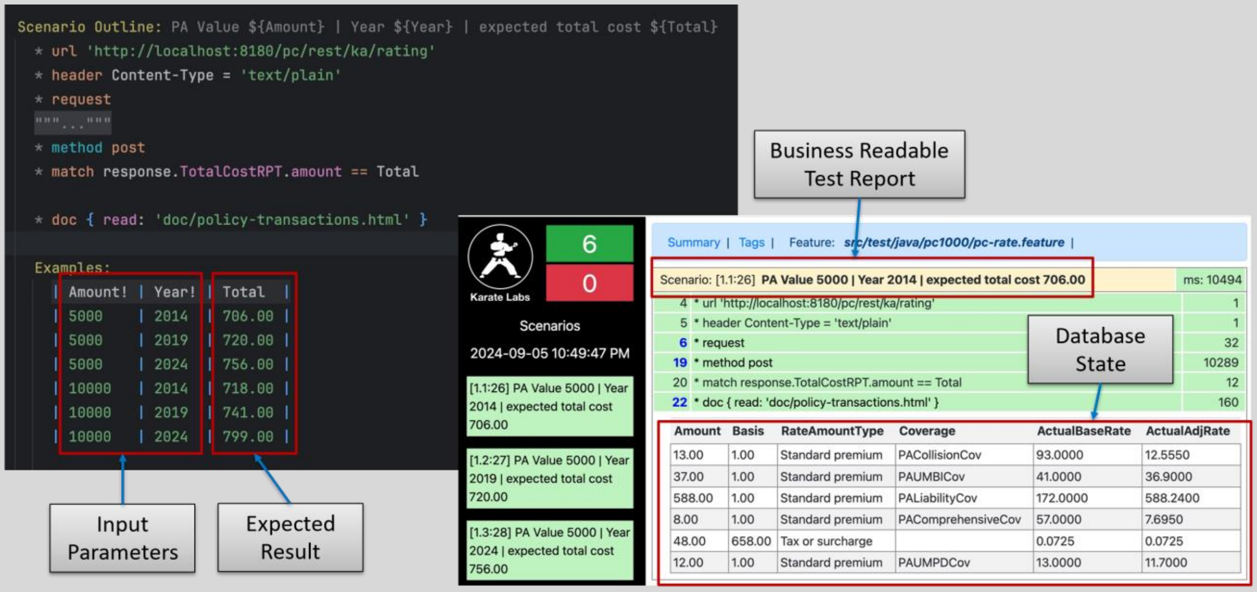
Validating a PDF using visual-testing or image comparison is another example. Integration to a mainframe or other legacy systems using unorthodox protocols can be easily handled with the right kind of tool.

Data Driven

When the core flow is the same, but the input parameters vary to give very different results, the most efficient testing approach is to encode all the scenarios into a tabular data-structure and then loop over the rows to feed to the core flow.

Excel files or Comma Separated Values (CSV) files are common ways to represent tables of data. It becomes easy to change the test-data while keeping the implementation details abstracted away.

Tests end up being business-friendly, and domain-experts are likely to be working in excel sheets to begin with.



```
Scenario Outline: PA Value ${Amount} | Year ${Year} | expected total cost ${Total}
  * url 'http://localhost:8180/pc/rest/ka/rating'
  * header Content-Type = 'text/plain'
  * request
  """..."""
  * method post
  * match response.TotalCostRPT.amount == Total

  * doc { read: 'doc/policy-transactions.html' }

  Examples:
    | Amount! | Year! | Total  |
    | 5000    | 2014  | 706.00 |
    | 5000    | 2019  | 720.00 |
    | 5000    | 2024  | 756.00 |
    | 10000   | 2014  | 718.00 |
    | 10000   | 2019  | 741.00 |
    | 10000   | 2024  | 799.00 |
```

Scenarios

2024-09-05 10:49:47 PM

[1.1:26] PA Value 5000 | Year 2014 | expected total cost 706.00

[1.2:27] PA Value 5000 | Year 2019 | expected total cost 720.00

[1.3:28] PA Value 5000 | Year 2024 | expected total cost 756.00

Feature: src/test/java/pc1000/pc-rate.feature |

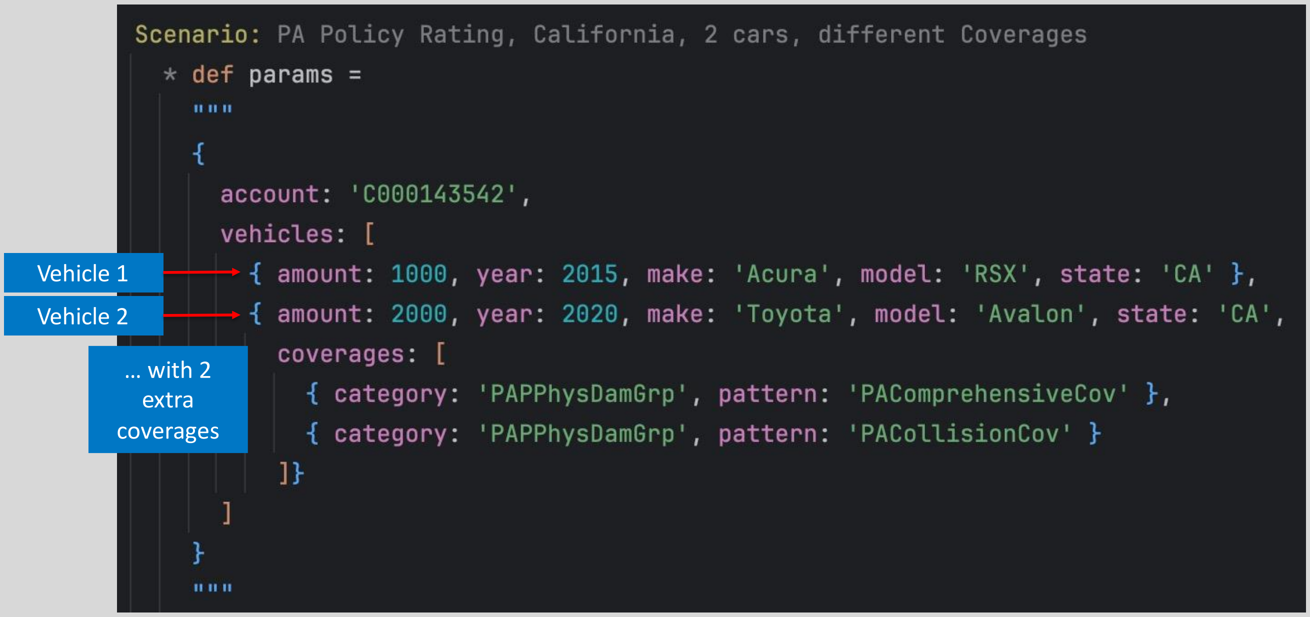
```
Scenario: [1.1:26] PA Value 5000 | Year 2014 | expected total cost 706.00     ms: 10494
 4 * url 'http://localhost:8180/pc/rest/ka/rating'                                1
 5 * header Content-Type = 'text/plain'                                           1
 6 * request                                                                     32
19 * method post                                                              10289
20 * match response.TotalCostRPT.amount == Total                                 12
22 * doc { read: 'doc/policy-transactions.html' }                               160
```

Amount	Basis	RateAmountType	Coverage	ActualBaseRate	ActualAdjRate
13.00	1.00	Standard premium	PACollisionCov	93.0000	12.5550
37.00	1.00	Standard premium	PAUMBICov	41.0000	36.9000
588.00	1.00	Standard premium	PALiabilityCov	172.0000	588.2400
8.00	1.00	Standard premium	PAComprehensiveCov	57.0000	7.6950
48.00	658.00	Tax or surcharge		0.0725	0.0725
12.00	1.00	Standard premium	PAUMPDCov	13.0000	11.7000

The example above shows how a tool that supports data-driven tests results in very readable test scripts and reports. The bottom section on the right is extra data about the policy being shown within the HTML test report. This is achieved by transforming parts of the JSON response, and API tools that have this capability are well worth it. This means that if something fails, you have more information at your fingertips to determine the root cause.

Test data in the world of insurance also tends to be very nested in nature, instead of being "flat" and simple. For example, a personal auto policy can have multiple vehicles and then multiple drivers, which requires multiple "one to many" relationships for test-data in technical terms.

Expressing such data in a flat structure is difficult, but extremely easy to do in JSON which has the advantage of being human-friendly to read and author. In fact most IDEs do this by default. A testing tool that handles JSON arrays makes data-driven testing easier for data that has nested relationships.



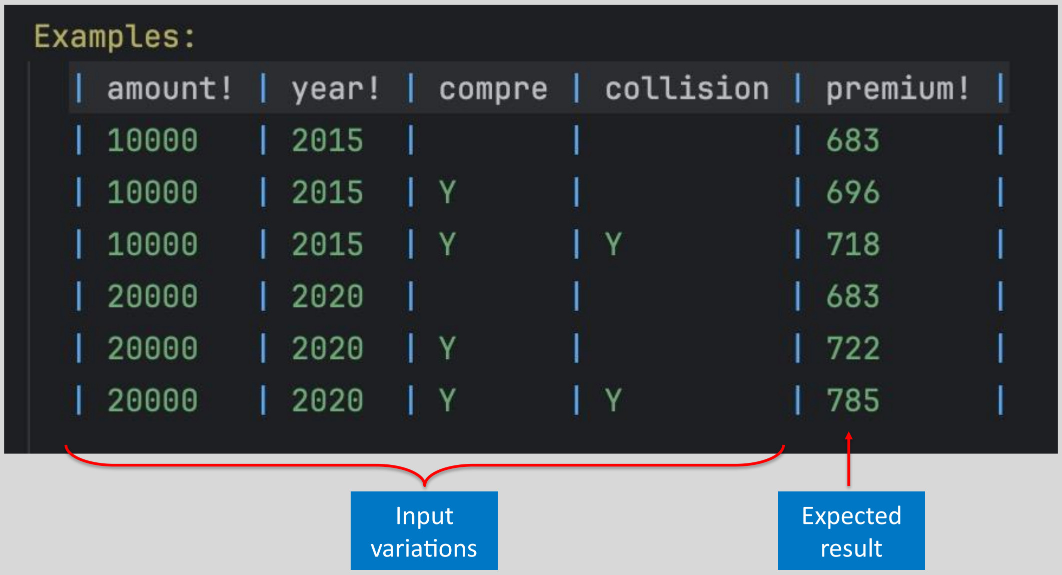
```
Scenario: PA Policy Rating, California, 2 cars, different Coverages
  * def params =
    """
    {
      account: 'C000143542',
      vehicles: [
        { amount: 1000, year: 2015, make: 'Acura', model: 'RSX', state: 'CA' },
        { amount: 2000, year: 2020, make: 'Toyota', model: 'Avalon', state: 'CA',
        coverages: [
          { category: 'PAPPhysDamGrp', pattern: 'PAComprehensiveCov' },
          { category: 'PAPPhysDamGrp', pattern: 'PACollisionCov' }
        ]}
      ]
    }
    """
```

A testing tool which can use dynamically generated data as a data-source is also good to have. Take the example of a function that generates random data or is able to generate the correct dates for a policy depending on when a test is executed.

For insurance, there are many such situations where a team cannot rely on a "hard coded" table of data, but it has to be dynamically created at the time of running a test. Dynamically generating data can avoid the need to rely on production data, and reduce the volume of test data needed to be stored and maintained.

Enable Business and Development to collaborate

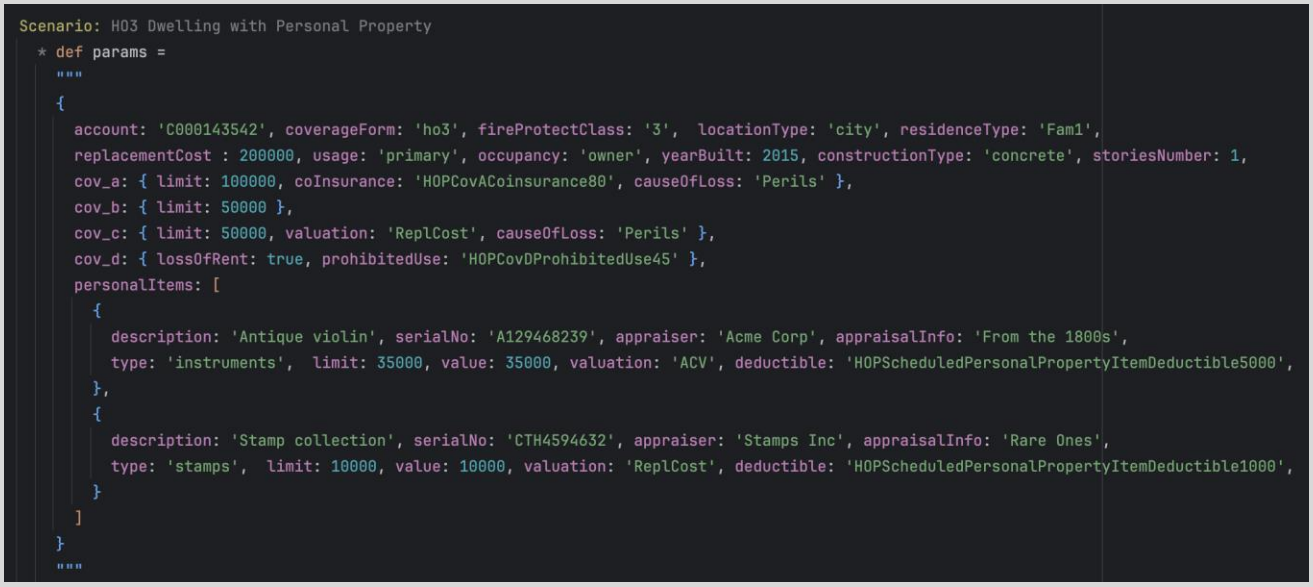
We have already laid out how the insurance domain is business-rule heavy and how much more important it is to bring subject matter experts and the development team together on the same page. Data-driven testing makes it easier for tests to be expressed in tabular form, and aligned with the excel-sheet form that product owners are comfortable with.



```
Examples:
  | amount! | year! | compre | collision | premium! |
  | 10000   | 2015  |        |           | 683      |
  | 10000   | 2015  | Y      |           | 696      |
  | 10000   | 2015  | Y      | Y         | 718      |
  | 20000   | 2020  |        |           | 683      |
  | 20000   | 2020  | Y      |           | 722      |
  | 20000   | 2020  | Y      | Y         | 785      |
```

Flows that involve complex data are hard to represent as test scripts. The creation of a fully fleshed out policy submission is a good example. On the UI, this typically involves multiple pages of data-entry screens, along with validations for mandatory fields and value ranges. This is one reason why UI automation is so complex; the UI script would be filled with a lot of "noise" from the business point of view - such as having to click buttons to navigate to the next page or tab etc.

In contrast, an API test can focus on the pure data of the scenario and the policy creation step can happen in a single API call.



```
Scenario: HO3 Dwelling with Personal Property
  * def params =
    """
    {
      account: 'C000143542', coverageForm: 'ho3', fireProtectClass: '3',  locationType: 'city', residenceType: 'Fam1',
      replacementCost : 200000, usage: 'primary', occupancy: 'owner', yearBuilt: 2015, constructionType: 'concrete', storiesNumber: 1,
      cov_a: { limit: 100000, coInsurance: 'HOPCovACoinsurance80', causeOfLoss: 'Perils' },
      cov_b: { limit: 50000 },
      cov_c: { limit: 50000, valuation: 'ReplCost', causeOfLoss: 'Perils' },
      cov_d: { lossOfRent: true, prohibitedUse: 'HOPCovDProhibitedUse45' },
      personalItems: [
        {
          description: 'Antique violin', serialNo: 'A129468239', appraiser: 'Acme Corp', appraisalInfo: 'From the 1800s',
          type: 'instruments',  limit: 35000, value: 35000, valuation: 'ACV', deductible: 'HOPScheduledPersonalPropertyItemDeductible5000',
        },
        {
          description: 'Stamp collection', serialNo: 'CTH4594632', appraiser: 'Stamps Inc', appraisalInfo: 'Rare Ones',
          type: 'stamps',  limit: 10000, value: 10000, valuation: 'ReplCost', deductible: 'HOPScheduledPersonalPropertyItemDeductible1000',
        }
      ]
    }
    """
```

If business-analysts are not just able to read tests but able to edit them – that unlocks a significant advantage. The team would be living the agile principle of "collective ownership".

There would be less cases where someone is waiting for somebody from some other team to be available for testing.

Which brings us to another aspect of testing – which is exploratory testing. APIs lend themselves well to being called in an ad-hoc manner, What this means is that anyone could spin up a browser and play around with how the APIs are responding.

The previous section on data-driven testing demonstrates how APIs can be "spiced up" via HTML reports to present data in a business-friendly way. If done right, this can make the APIs "browsable". Imagine a scenario where business analysts can use APIs in a self-serve mode and answer questions for themselves such as:

- Is this product line being rated correctly?
- Are all the premium costs calculated correctly?

Re-use API Tests as Performance Tests

From a technical point of view, it is not practical to re-use a UI test-automation suite as a performance test. But API calls are well suited to be used for driving concurrent load on a server. Very few test-automation tools allow for the re-use of API tests as performance or load tests – and this should be a key factor when you evaluate testing tools.

In the domain of insurance, a challenge that insurers face is providing potential customers with a quote in the shortest possible time. It is common knowledge that when it comes to web-interfaces, users tend to get impatient after waiting for a response for more than a second and would likely leave after 10 seconds.

Ensuring that core back-end calculations take the shortest amount of time even under periods of high load (e.g. during weather-related events or natural disasters) should be a key area of focus.

API vs UI Testing

This has been covered in detail in our white-paper: Navigating the Brave New World of API Testing. A summary is shown below.

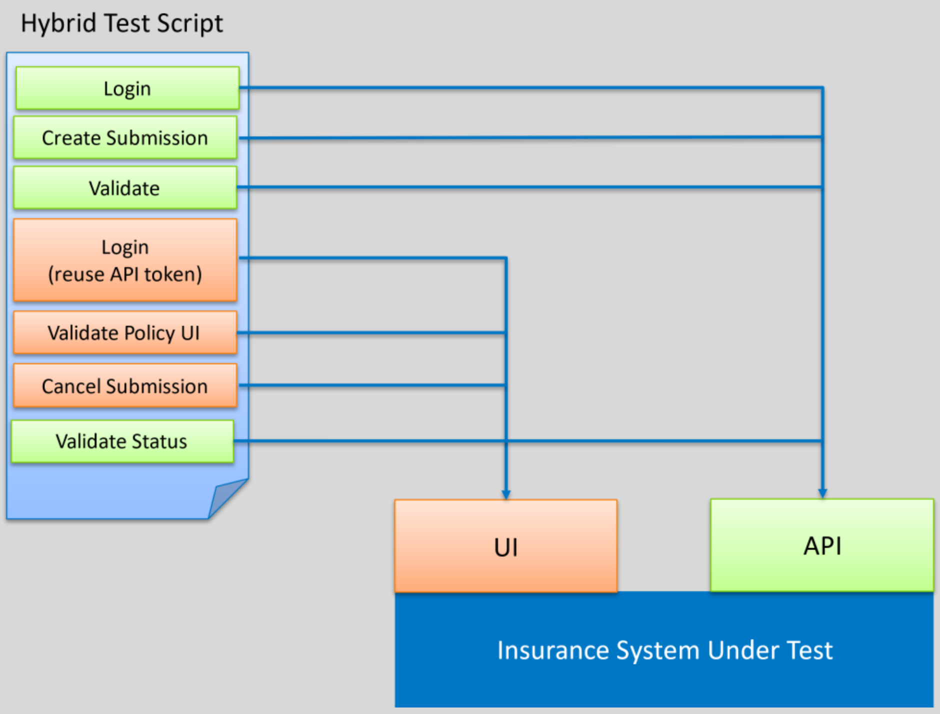
	UI Testing	API Testing
Complexity	High	Low
Speed	Very Slow	High, can be run in parallel
Resource Requirements	Very High	Minimal
Stability	Flaky	Very Stable
Data Coverage	Hard to achieve	Easier to achieve, higher confidence
Mocking / Isolation	Hard or near impossible	Easy
Architecture Coverage	Only the 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

Teams are recommended to reduce the percentage of UI automation where possible and lean towards API testing.

Hybrid Testing

We are certainly not saying that UI automation should be zero. In fact, there are various scenarios where mixing API and UI calls in the same test-script make sense.

For example, since it is so time-consuming and unreliable to enter all the policy details via the UI, a team can choose to use the API to push data into the database and then use the UI to validate that the data is rendered correctly. This is a smart way to reduce the time for testing while still addressing the need to validate the basic functionality of the user-interface.

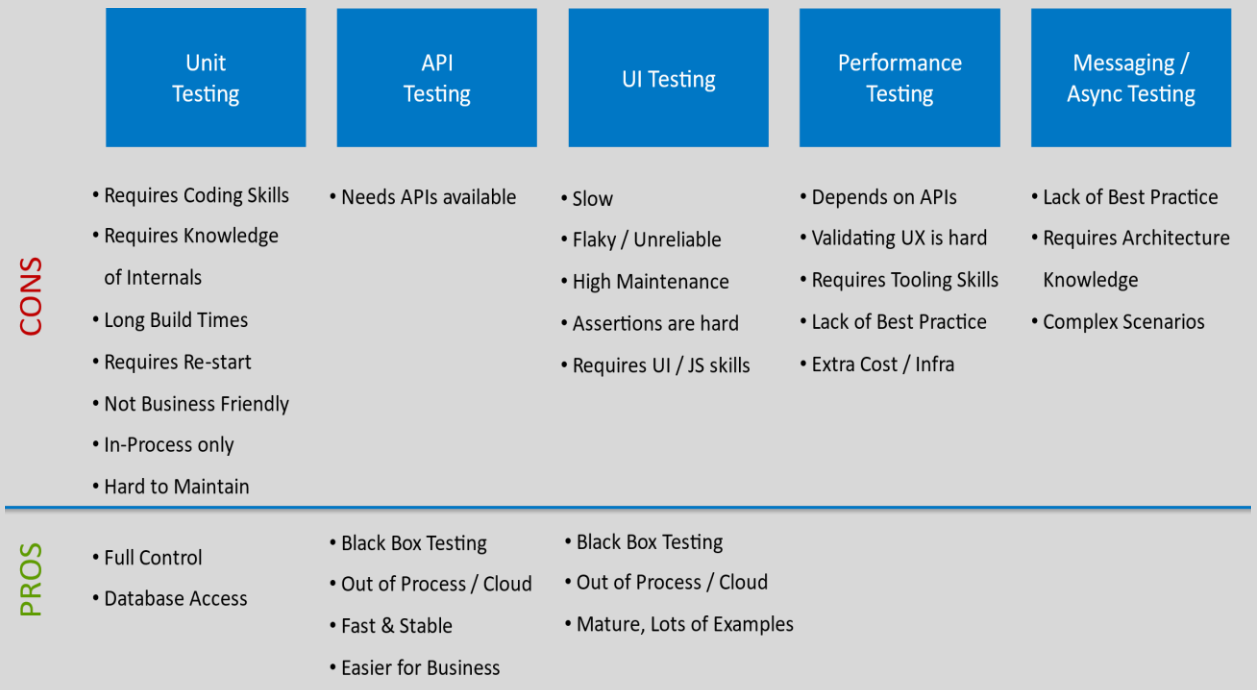


A nice benefit of using the API for a sign-in is that the auth token can be reused for the UI flow. Typically, this is a simple matter of dropping a cookie onto the page. What this means is that UI tests can skip the time consuming and potentially flaky step of signing in and entering the username and password, submitting the login form, and being redirected to the dashboard page.

API vs other kinds of testing

Digital transformation based on an insurance platform is a significant undertaking and involves multiple technologies, ranging from backend code to front-end and communication between other internal or third-party systems.

The various kinds of testing you can do are summarized below.



	Unit Testing	API Testing	UI Testing	Performance Testing	Messaging / Async Testing
CONS	• Requires Coding Skills • Requires Knowledge of Internals • Long Build Times • Requires Re-start • Not Business Friendly • In-Process only • Hard to Maintain	• Needs APIs available	• Slow • Flaky / Unreliable • High Maintenance • Assertions are hard • Requires UI / JS skills	• Depends on APIs • Validating UX is hard • Requires Tooling Skills • Lack of Best Practice • Extra Cost / Infra	• Lack of Best Practice • Requires Architecture Knowledge • Complex Scenarios
PROS	• Full Control • Database Access	• Black Box Testing • Out of Process / Cloud • Fast & Stable • Easier for Business	• Black Box Testing • Out of Process / Cloud • Mature, Lots of Examples		

Especially when customization of the base-platform via code is involved, the focus has traditionally been on writing unit-tests to validate lines-of-business and to provide a code-coverage figure that is tracked.

Most teams then focus on the UI because of multiple reasons. One is that it is very natural to think about the UI as the only way to interact with the functionality. However, awareness of API testing and its benefits is on the rise. But still there has been such a lot of investment in UI automation in the past that we find teams being slow to switch.

API testing has multiple advantages, but it does depend on having production-ready APIs available, which may not be the case for all projects. We discuss a solution to address this later in the next section.

API testing can be well worth it because of recent technology advances such as Docker that enable a team to run the entire stack locally on a developer laptop. In fact, we see a trend where teams realize that API tests can be as fast as unit tests – which means they can even reduce the number of unit tests in favor of API tests.

API tests cover all layers of the implementation, including the back-end logic that unit-tests tend to focus on. They can indeed replace unit-tests if done right.

Karate Labs API Testing Adapter

After years of working in a variety of domains such as Income Tax and Insurance, we noticed common challenges and evolved patterns to address them.

First let us talk about the challenges.

Challenges

Non-Availability of APIs

Since insurance platforms involve a heavy degree of customization and configuration over the base platform, the custom product lines are not automatically exposed as REST endpoints. There are code-generation and configuration steps that need to be performed first. This takes time and is a blocker for the QA team because they must wait for these APIs to be made available.

APIs are "chatty"

Take the example of a policy submission. It is known to typically involve a lot of steps – starting with making sure the user account is created or available. A submission will be in draft mode as all the information such as coverages, clauses and questions are put in place. Finally, a submission passes validation and can be rated, quoted, bound and issued.

The APIs for the insurance platform would reflect this long-running flow by default. They may have an option to perform a "composite" API call which adds complexity of another kind because of the need to massage the scenario data into the JSON format expected by the "bulk" API.

In extreme cases, the APIs may be in "async" mode – where an initial call is made and then the caller has to wait for a callback (typically webhook) or must poll until the result is available. This is common for business processes that take a lot of time.

APIs not used in production

This is becoming rarer nowadays since we are moving towards an API-first era. But for example, some insurance platforms may only have SOAP endpoints exposed because of legacy reasons, or because that is how the inter-platform communication is set up to work.

The other case is that the insurance provider is using the UI 100% to run their business, by providing agent-facing or claim-servicing portals.

In these cases, the team may not be comfortable putting time into standing up APIs that are useful only for the testing team.

Solution

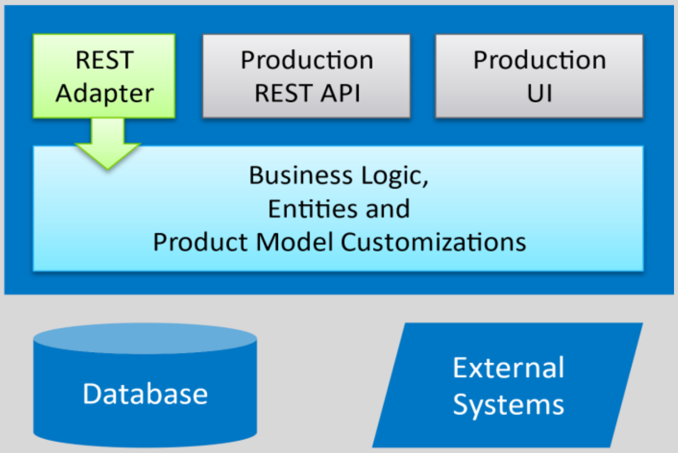
If there are available REST APIs used in production, the next section covers testing these entry points in a “traditional” sense. But for teams that want to reap the benefits of API testing, we have a unique solution described below.

The solution is to expose a special REST endpoint on the server under test. This endpoint is intended for use only in “development” or “pre-prod” mode and will not be available in production deployment.

This REST endpoint will provide a flexible way to read and write data but honoring the contracts and business-rules. In other words, this is not a “brute-force” approach of using SQL on the database directly. This means that all customizations and configuration changes made to the base platform can be examined and exercised.

A visual representation of this solution is shown below.

- Embedded into Server
- Part of Platform REST framework
- Low footprint, zero overhead
- Return data from any entity
- Call any server-side code
- Re-use server-side helper utilities
- Script complex data-set up or builders
- Edit test without re-starting server
- Edit test without re-compiling code
- Full control like in Unit Tests
- The developer experience of API Tests
- Run on remote server, out-of-process



This innovative solution combines the best of both worlds – the fine-grained control that one gets from Unit Testing – but with the out-of-process developer-experience of API Testing.

We know of many teams that must wait minutes and even hours for the server to rebuild after small changes to code. Since unit-tests run “in process” the time taken to iterate while building a test is similar.

Most meaningful unit tests depend on a running database and back-end. Running a unit test means waiting for the whole cycle of starting the database and the server-side coming up. This is a huge productivity killer for teams that just want to write a comprehensive suite of tests.

The embedded adapter approach means:

- The team can iterate on writing complex functional tests without having to restart the server for every small change made in the test script.
- Data can be set up in any combination needed, along with currency amounts and numbers in true data-driven fashion.

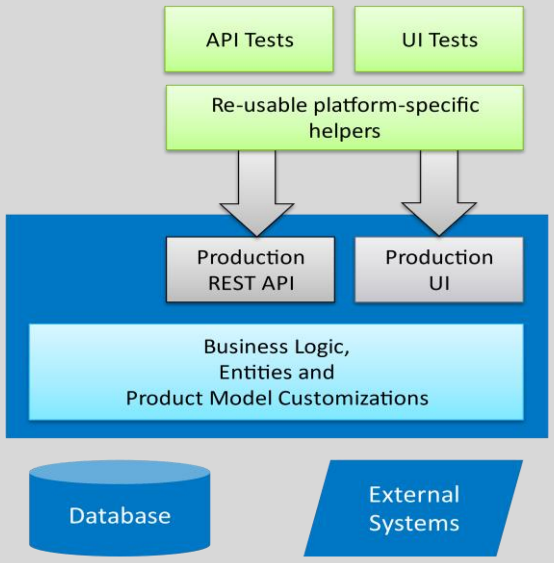
API Testing Accelerators

A defining feature of a good test automation framework is how easy it is to customize for targeting some kinds of situations or patterns more efficiently.

For users of insurance platforms, it is common to see the following patterns:

- Helpers for creating API payloads – common patterns include headers, authorization tokens and how pagination is handled.
- Helpers for data setup – many scenarios depend on some "master data" to exist, for example a policy cannot be created without an account, producer and underwriter.
- Helpers for date conversions and calculations - there is a need to calculate date ranges and time-period spans. Dealing with currency for monetary amounts and handling rounding and arithmetic precision is key. Being able to handle the concept of a "Big-Decimal" is important.
- UI Helpers – UI automation can be tricky when dealing with metadata-driven HTML where reliable locators are not available. Custom helpers can be built to traverse the UI using some other kind of locator strategy, for example using the product model metadata that can be retrieved via the API or back-end. Being able to do hybrid API and UI testing in the same script unlocks more efficient testing.
- Open API and Swagger support – A good API automation tool will be able to ingest a spec and accelerate the creation of API calls to end points within that spec. Since the payloads can be quite complex in the insurance domain, this can significantly help the team write API tests faster. Most insurance platforms support the auto-generation of Open API specification files from the available API endpoints, and this can be used to great effect.

Karate offers all the above functionality to users of insurance platforms.


API Tests
UI Tests
Re-usable platform-specific helpers
Production REST API
Production UI
Business Logic, Entities and Product Model Customizations
Database
External Systems

info@karatelabs.io

www.karatelabs.io

https://github.com/karatelabs