

The Test SDK: The SDET's Best Friend in Monolith-to-Microservices Migrations

Wayne Ng

wng@uship.com

Abstract

Transitioning from a legacy monolith to a distributed systems architecture presents both opportunities and challenges for the test automation engineer. It's exciting to use new modern test frameworks that are not bound by the confines of the legacy tech stack, but daunting at the same time given how easy it is for your test architecture to quickly become chaotic and unmanageable. How does one unify and standardize automation test architecture and functionality as more and more new services are spun up?

This article outlines uShip's approach to overcoming the hurdles involved in moving to distributed systems, how we developed an internal test software development kit to facilitate writing test automation and the lessons of how things have worked using the test software development kit (SDK) a few years later in our distributed system journey. It will also highlight how the SDK created an extensible strategy that enabled centralized methods of test data creation, standardized core test helpers, and allowed for simple distribution and updates for the engineering teams. Finally, we'll show how creating the SDK allowed engineering to easily roll out new future projects and improvements to our test automation in the ever changing evolution of our software.

Biography

Wayne Ng is currently a Principal Software Engineer in Test at uShip and has a decade of experience in the testing space, ranging from building test automation frameworks to establishing accessibility, chaos testing and monitoring best practices. He is a strong advocate about having a test automation first mindset to enable organizations to deliver value more frequently and more sustainably.

Outside of work, Wayne enjoys exploring nature, photography, eating good food and spending time with his wife and kids.

1 Introduction

Migrating from a monolithic project to a distributed systems architecture sounds great on paper: deploys should go faster for individual components, ownership of code is clearer based on repos and there are opportunities to use new modern technologies. However, some things get lost in the transition because they may come as an architectural afterthought, such as a centralized testing framework familiar to everyone, a fully built-out library of page objects, and a variety of helper functions to facilitate navigating through flows or setting states for application components.

As functionality moves or new functionality is built outside of a monolith, different teams can easily start implementing core testing functionality in various ways or rewrite the same code too many times. For example, if your test needs a helper to set up a test user, no one wants the same user helper implemented 10 different ways, in 10 different places on 10 different teams. That's a lot of code changes for maintenance and updates if something systematically changes.

To prevent all chaos from breaking loose during a monolith to distributed architecture migration, this paper advocates for the planning and creation of a testing SDK as a non-negotiable, day-zero architectural activity, promoting a quality and automation first mindset.

2 The Benefits of the Test SDK

At its core, a testing SDK is a collection of tools and libraries designed to assist engineering in the process of testing their software. Generally, it contains functionality required for writing automated tests that can range from test data creation to fetching secrets or to setting configurations.

DRY (Don't Repeat Yourself) code is a central software development principle that allows for maintainable and scalable architectures. The primary goal of a test SDK is to prevent WET (Write Everything Twice) code since WET code makes updating code a nightmare. Test automation code is more often DAMP (Descriptive and Meaningful Phrases) so that it's easily understandable to see the behaviors the code is testing and sometimes allows for some repetition to make certain aspects of the user flow clear to the test engineer. A testing SDK is a way to help DRY that code out a little by reducing repetitive patterns and duplicate code and logic in favor of modular and referenceable code. In some ways, a testing SDK can be thought of as another manifestation of the factory pattern at an organization level scope - a single abstracted location where common functionality is maintained.

2.1 Centralized Methods For Leveraging Core Functionality

Pre-prepared, static datasets used in testing, also known as canned data, are not ideal when writing automated tests because the datasets need to be reset to a known good state prior to subsequent test runs. This can be problematic if a test fails since data might be in an unknown state prior to re-running the test, which may cascade more test failures. Tests with canned data often are slow and flaky as a result because they may need to run serially to prevent cross-contamination of the sample data store and ensure clean test runs.

In an ideal world, we would generate test data for setup in each test to maximize test isolation and test speed. New services spun off outside of a monolithic application are likely in service to that monolith, at least during the transition period. Tests will then share core business logic that is similar if not identical across the entire application.

For example, in an e-commerce application, you most likely have users and some form of a product, and those users interact with the product in some fashion. But if every microservice or single page application outside of the monolith needs a user and a product, in the spirit of DRY code, we do not want to write the same user or product creation script multiple times, especially if changes need to be made to users or the product.

Wrapping core API functionality in the test SDK solves much of the data creation problem. In doing so, there is a single place that will call that functionality, standardized such that API usages will be centralized. These tools can also be designed with flexibility in mind. If a test engineer needs the test data to be customized to meet their needs, they can easily pass in an override to the function signature to adjust the api call for their needs without the need for additional test code cruft.

```
async function createUser(baseUser: UserModel, userOverwrites: UserOverwrites = {}): Promise<InternalUser> {
  const uuid = uuidv4().replace(/-/g, '');
  // create a deep copy of baseUser to avoid mutating it
  const baseUserCopy = JSON.parse(JSON.stringify(baseUser));
  // merge baseUserCopy with userOverwrites, including nested objects
  const model = {
    ...baseUserCopy,
    ...userOverwrites,
    user: {
      ...baseUserCopy.user,
      ...userOverwrites.user
    },
    userPreferences: {
      ...baseUserCopy.userPreferences,
      ...userOverwrites.userPreferences
    },
    profile: {
      ...baseUserCopy.profile,
      ...userOverwrites.profile
    }
  }
} as UserModel;

// Update model with uuid
model.user.userName = model.user.userName || `${baseUser.user.userType}${uuid}`;
model.user.emailAddress = model.user.emailAddress || `${baseUser.user.userType}${uuid}@node.uship.com`;

const userInfo = await postUser(model);

await patchTestUser(userInfo.userId, {
  EmailVerified: true
});

const internalIds = await getInternalUserInfo(userInfo.userId);

return {
  ...userInfo,
  userInternalId: internalIds.data.userId,
  userGuid: internalIds.data.userGuid,
  isDeleted: internalIds.data.isDeleted
};
}

export async function createShipper(userOverwrites?: UserOverwrites): Promise<InternalUser> {
  const shipper = await createUser(defaultShipper, userOverwrites);
  return shipper;
}
```

Figure 1. Wrapper around user creation function with optional model overrides

Figure 1 above shows an example of how user creation is implemented in uShip's testing SDK. We have a base createUser function that takes in a defined typed userModel based on the types of users uShip has in its system and then a second parameter with optional user overrides to customize test data as needed. Once the desired user model is set up, the JSON payload is sent to the POST /users endpoint to create the user, followed by a subsequent PATCH operation for user property setup and then the user data is returned in the response. The second function in Figure 1 shows a simple implementation of how we use the generic createUser function to create a generic shipper user for reusable data creation that any test can call for setup given all our tests require a shipper.

2.2 Ability to Standardize Test Architecture

Working in a greenfield distributed system allows the opportunity for new technologies to become integrated into the tech stack since there are no legacy constraints if an engineering organization chooses to do so. While engineering teams often have freedom to use whatever tooling they wish, it is highly recommended to limit supported tooling during the distributed architecture migration to keep things maintainable. Standardizing the test framework allows teams to use a great new tool while keeping test patterns and writing styles uniform, so teams can ask one another questions and forge best practices within the supported technology stack. From an architectural perspective, an organization ideally wants to select one test framework. Just as an example in the Javascript space, no architecture group wants to support Playwright, Cypress, WebdriverIO, and Nightwatch all at the same time given they are all tools trying to solve the exact same problem.

As teams adopt core functionality within test frameworks, it's possible to bind the dependencies into the test SDK. While this does marry engineering to that test framework, it also allows QA to push many changes out across the organization in a simple fashion. Tightly coupling libraries into the test SDK comes with various pros and cons, so doing so may or may not be the right choice for organizations. Here are some of the benefits that we've seen as a result of binding a test framework to the test SDK.

```
export const desktopTest = baseTest.extend<DeviceTestFixture>({
  page: async (( page, config ), use) => {
    const internalSdkConstants = await getConstants();
    await page.setExtraHTTPHeaders({
      'CF-Access-Client-ID': internalSdkConstants.cloudflareAccessClientId,
      'CF-Access-Client-Secret': internalSdkConstants.cloudflareAccessClientSecret
    });
    await page.route(`${config.webUrl}`, async (route) => {
      await route.fulfill({
        headers: { 'access-control-allow-origin': '*' }
      });
    });
    await use(page);
  }
});
```

Figure 2: Playwright Fixture that is used across engineering for all end to end tests, regardless of repo

For example, in Figure 2 above as an example, uShip has chosen to adopt Playwright as its primary end-to-end test framework. To standardize our browser instances for the tests, we use Playwright's fixtures to set parameters such as viewport, device type, and browser type. For security reasons, every test in our organization also needs special headers appended to run in the VPC. Instead of having to manually add them to every test in every repo, and updating them when the header secrets are updated, we include them in a base fixture. All our automation then inherits from that fixture in the test SDK.

A second benefit is the ability to roll out new initiatives within our QA organization through the test SDK. As an example, to roll out test automation analytics and metrics as a new project, we bind a single library to handle all labeling required for the metrics in the SDK. We can design it using interfaces so it can be consumed regardless of framework. This avoids having to change syntax between repositories because of implementation differences in the frameworks.

```
export interface AllureReporter {  
  // Include both method signatures to support allure wdio/jest/playwright  
  label?(name: string, value: string): void;  
  addLabel?(name: string, value: string): void;  
}
```

Figure 3: An interface for Allure Reporting in the Test SDK

Figure 3 showcases an example interface for uShip's test metric reporter. Since test metrics are collected across various test frameworks at uShip, to unify metadata tagging, this interface in the test SDK allows us to use the same base library in the tagging function regardless of how the labeling function is implemented.

2.3 Simple Package Distribution for Engineering Teams

Once the library is ready to be packaged up and consumed by teams, the SDK gets published to our internal package repository. In a distributed architecture, some teams may install the testing SDK directly into their repo, or as new projects arise, another option is to have a boilerplate repository spun up that has the test SDK pre-installed as a dev dependency.

As updates are made to the test SDK, we use semantic release to push new versions to the package repository. Semantic release allows new changes to be version-controlled to avoid forcing breaking changes on the consumers of the SDK. This gives time for teams to choose when to update repositories to the latest release. If a team needs to consume new functionality immediately, they can simply just install the latest version of the package (i.e. bump from version 9 to version 10), but if the existing code needs some time to be updated due to a breaking change in the newest version of the package, the team can stay on the current version until refactoring and maintenance can be done on the repository.

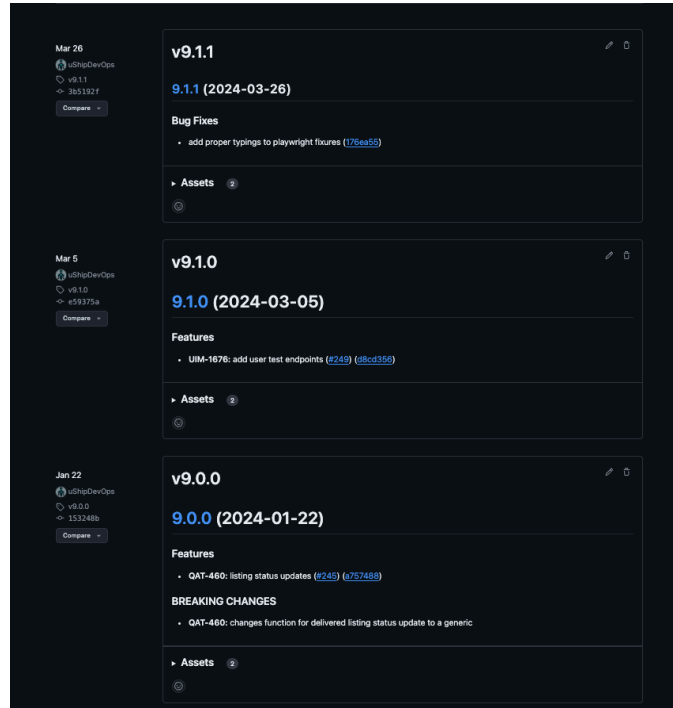


Figure 4: Test SDK Release Versions for a bug fix, feature release and a breaking change

Figure 4 above highlights example release versioning. Semantic release categorizes releases in three buckets: breaking changes (major version bump), new features (minor version bump) and bug fixes (minor version bump). A breaking change to the SDK necessitates changes by the consumer for the SDK to keep working. For example, renaming a function in the SDK would constitute a breaking change since the downstream consumer will need to call the newly named function for tests to keep working. Generally, adding any new functionality to the SDK will create a feature release. An example would be exposing a new endpoint under a namespace. An SDK bug fix would most likely come from a bug report by a consumer where an edge case of functionality is unexpected. For example, while the default function might work, adding an override causes a failure because a property is not respected accordingly and requires some fixes.

3 Tips and Tricks For Writing a Test SDK

A standardized, maintainable and easily distributed test SDK has many benefits as noted above but can be daunting to build from scratch. Below are some best practices for how to begin building and maintaining a test SDK.

3.1 Start Your Test SDK By Exporting Existing Helpers

Most likely, somewhere in your legacy code, there is pre-existing test automation code and associated helper functions or API calls for data setup. When building out the test SDK, start by porting the core helpers needed in many tests, using the legacy code as the template to build out the SDK. Start with a MVP (minimal viable product) SDK and adjust the code as needed to make it flexible for necessary use cases. With version control, if you find a case that doesn't meet the requirements, you can always publish

an update at a later time. If your distributed architecture is in the same programming language, this might be as simple as copy pasting code over to the new test SDK. Otherwise, translating legacy code into another language will require some effort but the core business logic and exposed api endpoints should hopefully already exist.

```
namespace uShip.EndToEndTests.Shared.Factories.Users
{
    public class Broker : User, IShipper, IServiceProvider
    {
        private Broker(UserModel userModel, InternalUserModel internalUserModel)
            : base(userModel, internalUserModel) { }

        public static async Task<Broker> CreateAsync(Action<PostAccountModel> actionToPerformDuringCreation = null, [CallerMemberName] string caller = null)
        {
            var aboutMe = TestHelpers.GetTestName() ?? caller;
            var userModel = await PostUserAsync(GetAccountModel(aboutMe), actionToPerformDuringCreation);
            // new brokers are now automatically held, so we will reactivate them now
            await ActivateUser(userModel.UserId);
            var internalUserModel = await GetInternalUserInfoAsync(userModel);
            return new Broker(userModel, internalUserModel);
        }

        protected static PostAccountModel GetAccountModel(string aboutMe)
        {
            var username = $"Broker{Guid.NewGuid().ToString().Replace("-", "")}";
            return new PostAccountModel
            {
                User = new PostUserModel
                {
                    UserType = "Broker",
                    UserName = username,
                    FirstName = "FirstName",
                    LastName = "LastName",
                    EmailAddress = $"{username}@2e.uship.com",
                    Password = "P@ssw@rd123456",
                    CompanyName = "Test Company",
                    HomePhone = "5125125125",
                    PrimaryAddress = new PostAddressModel
                    {
                        StreetAddress = "123 Test Lane",
                        MajorMunicipality = "Austin",
                        PostalCode = "78701",
                        Country = "US"
                    },
                    AboutMe = aboutMe
                },
                UserPreferences = new PostUserPreferencesModel
                {
                    Currency = "USD",
                    TimeZone = "Central Standard Time",
                    SiteId = "UnitedStates"
                },
                Profile = new PostProfileModel
                {
                    BrokerMcNumber = "123456",
                    EntityType = 5,
                    TrailerTypes = new List<string> { "Reefer", "AirRideVan" },
                    Commodities = new List<string> { "Vehicles", "LTL", "Boats" },
                    Regions = new List<string> { "NA-USA" }
                }
            };
        }
    }
}
```

Figure 5: Existing legacy helper function for user creation (C#)

```

import { UserModel, UserType } from '../src/types/user';

const defaultBroker: UserModel = {
  'user': {
    'userType': UserType.Broker,
    'userName': null,
    'firstName': 'FirstName',
    'lastName': 'LastName',
    'companyName': 'Test Company',
    'emailAddress': null,
    'password': 'P@ssw0rd123456',
    'homePhone': '5125125125',
    'aboutMe': 'ScratchpadTest',
    'primaryAddress': {
      'streetAddress': '123 Test Lane',
      'majorMunicipality': 'Austin',
      'postalCode': '78701',
      'country': 'US',
      'stateProvince': 'TX'
    }
  },
  'userPreferences': {
    'timeZone': 'Central Standard Time',
    'currency': 'USD',
    'siteId': 'UnitedStates'
  },
  'profile': {
    'entityType': 5,
    'trailerTypes': [
      'Reefer',
      'AirRideVan'
    ],
    'commodities': [
      'Vehicles',
      'LTL',
      'Boats'
    ],
    'regions': [
      'NA-USA'
    ],
    'brokerMcNumber': '123456'
  },
  'thirdPartyIdentifier': null,
  'customFields': null
};

export default defaultBroker;

```

```

import axios, { AxiosRequestConfig, AxiosResponse } from 'axios';
import { URL } from 'url';
import { config, getConfig } from '../config';
import { InternalUser } from '../types/internalUser';
import { PayOnTerms } from '../types/payOnTerms';
import { PatchUserBody, User, UserModel } from '../types/user';
import { stringify } from '../utils/stringify';
import { getDefaultHttpConfig } from '../defaultHttpConfig';

export async function postUser(userModel: UserModel): Promise<User> {
  getConfig();
  const httpConfig: AxiosRequestConfig = { ...await getDefaultHttpConfig() };
  httpConfig.headers['X-Mashery-Auth-User-Context'] = config.superUserId;

  const usersUrl = new URL('/v2/markets/1/users', config.baseApiUrl).toString();
  if (config.verbose) console.log('POST Request: ${usersUrl}', httpConfig);
  const userResponse = await axios.post(usersUrl, userModel, httpConfig);
  if (userResponse.status != 201) {
    throw console.error(stringify(userResponse));
  }
  if (config.verbose) console.log(`${userResponse.request.method} Response: ${stringify({
    status: userResponse.status,
    statusText: userResponse.statusText,
    data: userResponse.data
  })}`);

  const userId = (userResponse.headers.location!).split('/').pop();

  const userInfo: User = {
    firstName: userModel.user.firstName,
    lastName: userModel.user.lastName,
    companyName: userModel.user.companyName,
    homePhone: userModel.user.homePhone,
    userName: userModel.user.userName!,
    emailId: parseInt(userId!, 10),
    emailAddress: userModel.user.emailAddress!,
    userAccountModel: {
      userPreferences: userModel.userPreferences,
      profile: userModel.profile,
      thirdPartyIdentifier: userModel.thirdPartyIdentifier,
      customFields: userModel.customFields
    },
    userType: userModel.user.userType,
    password: userModel.user.password
  };
  return userInfo;
}

```

Figure 6: New version of helper function for user creation implemented in the test SDK (Typescript)

Figure 5 and Figure 6 above show an user creation function that has been ported from the legacy monolithic test helpers in C# and rewritten in Typescript in the test SDK (split into 2 files). Most of the logic used in the creation of users remains unchanged, with just some minor adjustments for semantics that have evolved over the 20 years of our system and differences between C# and Typescript. Porting code into a test SDK, whether it's in the same programming language or different, can be done fairly quickly if there are no major updates that need to be made. Especially if your organization allows use of generative AI such as chatGPT, code can be very quickly transcribed if porting languages and allows for rapidly building out the test SDK.

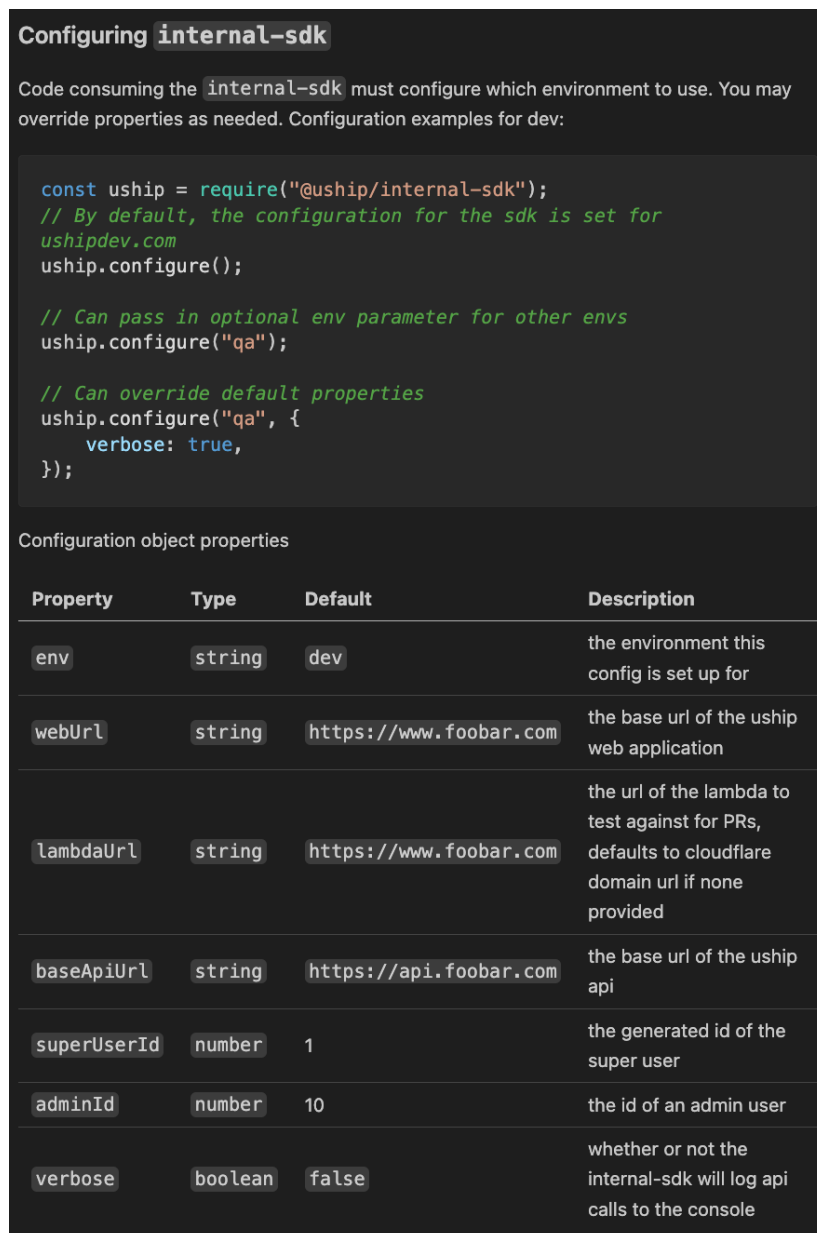
3.2 Limit SDK Contents to Reusable Functionality

To prevent having a massive monolithic test SDK, it is advised to only put functionality in the SDK that will be used by consumers in multiple projects (i.e. more than one). Most likely, this functionality will be core business logic that most teams will need to implement in their tests. For example, almost all companies likely have some concept of a user, so standardizing and centralizing user creation in a test SDK would be a good starting point.

One-off functionality in the SDK is not helpful if no one else intends on consuming it, and thus can just live in the project where it is needed as a traditional helper function. As organizations discover that functionality is no longer a one-off, that would create an opportunity to add the functionality to the SDK and consume the newest version of the SDK. Collaboration between teams is key to finding out when one-off functionality becomes used in multiple places.

3.3 Valid Table of Contents

As new functionality gets added to the SDK, make sure the README or API documentation for your SDK is up to date. Especially as fixes and changes are made, having up-to-date documentation greatly aids understanding for engineers who may not be as familiar with the SDKs. Some plugins will auto-generate a table of contents to speed up documentation. Also, having sample implementations is beneficial so engineers can quickly either copy existing calls or understand how the functions work.



Configuring `internal-sdk`

Code consuming the `internal-sdk` must configure which environment to use. You may override properties as needed. Configuration examples for dev:

```
const uship = require("@uship/internal-sdk");
// By default, the configuration for the sdk is set for
// ushipdev.com
uship.configure();

// Can pass in optional env parameter for other envs
uship.configure("qa");

// Can override default properties
uship.configure("qa", {
  verbose: true,
});
```

Configuration object properties

Property	Type	Default	Description
<code>env</code>	<code>string</code>	<code>dev</code>	the environment this config is set up for
<code>webUrl</code>	<code>string</code>	<code>https://www.foobar.com</code>	the base url of the uship web application
<code>lambdaUrl</code>	<code>string</code>	<code>https://www.foobar.com</code>	the url of the lambda to test against for PRs, defaults to cloudfare domain url if none provided
<code>baseApiUrl</code>	<code>string</code>	<code>https://api.foobar.com</code>	the base url of the uship api
<code>superUserId</code>	<code>number</code>	1	the generated id of the super user
<code>adminId</code>	<code>number</code>	10	the id of an admin user
<code>verbose</code>	<code>boolean</code>	<code>false</code>	whether or not the internal-sdk will log api calls to the console

Figure 7: SDK Documentation for how to configure the SDK

Figure 7 above shows sample documentation in the test SDK README file. This sample function configures uShip's test SDK so that individual repos can point their tests to run against any environment, which can help validate functionality that has environment parity in the system. Ideally, the README

should have a high level description of what the function does, sample code that implements the function and property information for any parameters.

3.4 Strongly Typing Your Test SDK

Using a strongly typed language such as Java or Typescript to build your test SDK is highly recommended. A strongly typed language is one in which types are enforced strictly by the compiler or interpreter so that each variable is bound to a specific data type, and operations on these variables are restricted to what is allowed for their data type. Attempting to perform operations that are not allowed for a particular type, such as adding a string to an integer without explicit conversion, will result in a type error which will help minimize bugs in the test automation code. For example, presume your user creation API is not well documented and a new engineer is unsure whether an override to customize the user takes a string or an integer or some other complex object parameter. While it would be ideal to have API documentation, the strongly typed test SDK would error and return the error code that would help the engineer realize their mistake.

Using a strongly typed language also allows autocomplete when using the test SDK to quickly fill in functionality in tests. Especially with AI tooling like Github Copilot available in some engineering organizations, this can greatly speed up test writing efficiency. It also prevents misuse of the SDK by enforcing the data types, helping data remain close to its production state if modifications need to be made and mirror the end user experience.

3.5 Testing Your Test SDK

The testing SDK should be treated as production code, conforming to your organization's software development best practices. Changes to the SDK should go through the same SDLC process that other code changes follow (code reviews, QA, etc). While much of the testing SDK is designed for convenience to make testing easier for engineers, it should also be designed in a way that doesn't compromise security (i.e. do not directly expose API keys or secrets). It is highly recommended to write test automation (unit, end to end tests) using the functions in the test SDK to ensure they work as expected as part of the SDK's own deployment pipeline.

3.6 Feature Requests

As the business builds new features, likely new functionality will need to get added to the test SDK or changes in business logic cause breaks in the SDK. Having a process to handle necessary new feature requests or bug tickets aids the maintainer of the package to know what priority to work on necessary changes. uShip uses Github Issues to track feature requests and bugs for the test SDK with associated tickets on a Kanban board. This way, changes to the test SDK can go through the engineering organizations normal prioritization and SDLC workflow.

4 The Challenges of the Test SDK

Like many other projects in an engineering organization, the hardest aspect of having a test SDK is its maintenance. Since much of the SDK houses wrappers around common shared functions, making a breaking change to the APIs associated with SDK functionality means remembering to also change the

SDK accordingly. Failure to do so could break future SDK builds or potentially disrupt pipelines in the distributed system.

General maintenance is another challenge if the SDK lacks clear ownership. Since the SDK includes functionality from different engineering domains, feature ownership is shared. Adding a CODEOWNERS file can help enforce code reviews for pull requests and establish ownership. At uShip, the QA group, and specifically SDETs own the test SDK. Even with ownership set for an SDET group, a cadence needs to be established to review dependencies, security, and to apply fixes or updates.

Setting up a bot such as Renovate or Dependabot helps alleviate some maintenance but still requires diligence to merge in all generated pull requests. Proper teardown and disposal of test data after test runs is critical, depending on the data cleanup maturity of your engineering organization, to maintain a manageable sized data store. Work with your data team to determine a strategy for cleaning up the test data prior to implementing the test SDK.

Maintenance on a consumer level across engineering is another challenge. When new major versions are released or old versions need to be deprecated, engineering teams must find bandwidth to update the packages and affected tests. While teams are often content with using versions that meet their needs, security updates or other critical changes may require significant investment to update to the latest version. For example, at uShip, we have some teams still on v.6.0.0 of our test SDK when the latest version of the test SDK is on v.10.0.0, which means several teams are months, if not years, behind the most recent version. This requires the SDET group to continue supporting v.6.0.0 of the SDK until teams can update their code to use v.10.0.0.

5 Conclusion

This paper highlights the importance of having a testing SDK as testing gets increasingly decentralized in a distributed system. By creating a test SDK, a quality team is able to wrap core business APIs in a single maintainable package for test data generation, enforce standardized test architecture to remain scalable as the application grows and simplify distribution as changes to the apis get made. Especially as an organization scales, having a testing SDK reduces the stress of having to build additional test infrastructure in new projects since the core helpers all exist in a single package. While building and maintaining the testing SDK requires effort and resources, the rewards enable a significantly more manageable test architecture that accelerates feature delivery to end-users while empowering QA to confidently write tests.

References

- dbt. "DRY." Analytics Engineering Glossary. <https://docs.getdbt.com/terms/dry>. (accessed May 10, 2024).
- Martin Fowler. "Page Object," martinowler.com, <https://martinfowler.com/bliki/PageObject.html>. (accessed May 10, 2024).
- Microsoft. "Fixtures." Playwright. <https://playwright.dev/docs/test-fixtures>. (accessed May 10, 2024)
- Semantic Release. "Semantic Release," Github, <https://github.com/semantic-release/semantic-release> (accessed May 10, 2024).
- Renovatebot. "Renovate Documentation," Renovate Docs, <https://docs.renovatebot.com/> (accessed May 10, 2024).