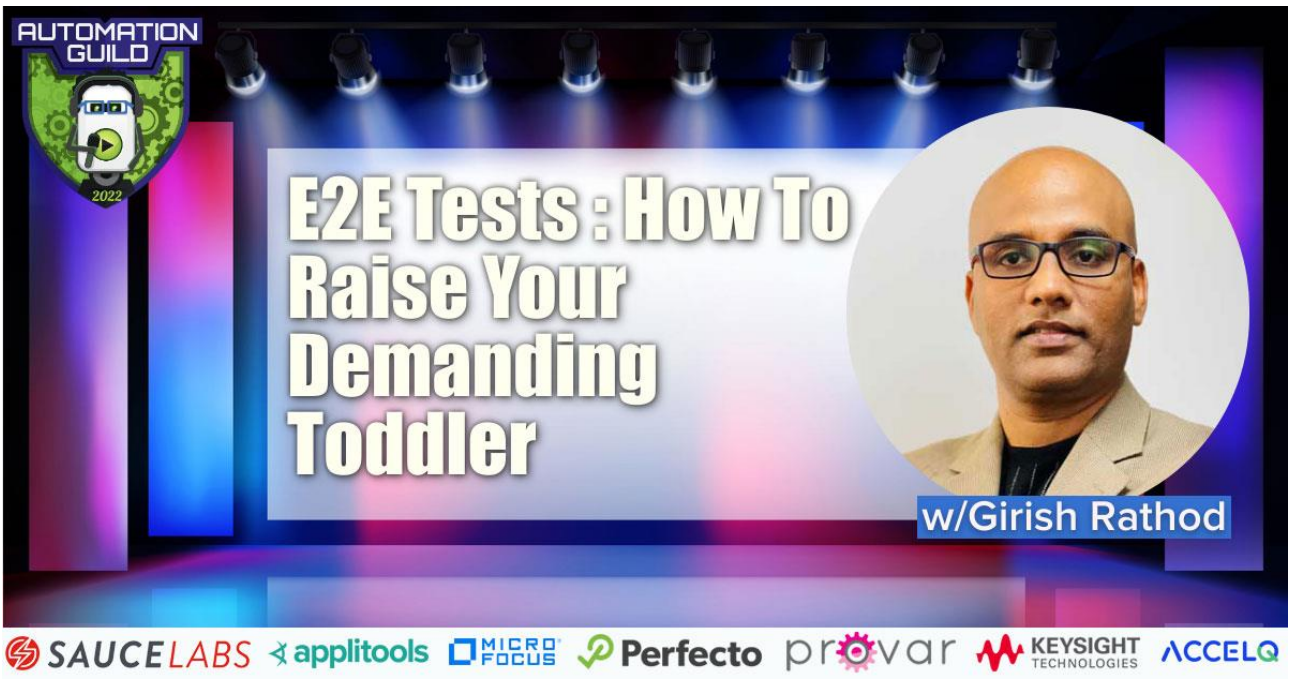

[00:00:01] I'm often told that I'm good with kids and toddlers are my favorite. They're so cute, innocent, and real. All is Hunky Dory, if everything goes as per their been, but as soon as that stops, they break down. Also, you can't leave a toddler unattended. They need constant care and looking after. And if you don't do that, they break down. I hope you're seeing where I'm going with it, don't you? Right. And it is also said that once you raised your first child, the second one onwards, it's much easier. On those lines, I'm going to walk you through our 18 months journey to raise our toddlers. Sorry. To build a robust and scalable end-to-end test system. And my hope is this will help you when you're trying to build or improve on yours.

[00:01:01] Today, I plan to cover not only the usual why, how architecture, and some working code, but also I will share some top learnings that we got from it. And I believe that will greatly help you when you're are trying to build one yourself. Without further ado, let's dive right in.

[00:01:22] Girish Rathod, I work as a Principal Software Engineer at Fidelity Investments. And just to be clear, that's not the video from my childhood. Let's start by understanding the lay of the land, like what were we testing? Overall, there are 2 UI or web-based applications?

[00:01:46] One was internal and other was external facing. The external facing was modernized and it used angular on the front end and used to call web services to get its data from the backend. And the internal ones were still getting upgraded. And they were the majority of them on some legacy frameworks.

[00:02:15] There are six teams who were working across all of the apps and functions. We used to do one release per month. And functionally, this was a donor-advised fund short form DAF, which is nothing but an application which connects donors and nonprofit organizations. And our definition of end-to-end test -- and this might offend some purist -- where any test that tests the application post-deployment. By that, I mean any test that executes against any non-product environments and not the test which executes during the build or build our packaging phase of the apps. And this consisted the end-to-end test so-called end-to-end testing that we have or that we'll be talking about today will consist of

UI and web service test that mimics the action of the end-user and validates most of the end states.

[00:03:25] So what were the main pain points that led us to start our journey of building our own test suite? Own end-to-end test system? So these are the four things that were on the top of the list. The first was lack of stability. The end-to-end testing that we wrote using the old system were breaking every day and that was not because did a bad test. That was because the system that we were using underneath was not efficient or was not optimally built, and that was becoming the new normal, which is kind of scary when people say the tests are failing, but that's OK. The second thing was it was hard to use. So whenever we used to use that to write our test it felt as doing and playing the game of Minesweeper. I don't know if everybody knows about it, but it's basically a game which read your game the ground and then you can run into a mine which is hidden and that's the end of the game. So that's how it felt when used to write the test. It is too big so often. The next tool, it was like kind of one monolithic repo. It was one repo which had web service test, UI test. The code part of it, the libraries, the framework part of it. Everything was packed into one monolithic repo, which was really hard to manage and maintain. And last but not least, the Cucumber framework. I don't have anything against Cucumber framework. I believe the way we were using the Cucumber framework was not optimal. And the major pain point was step definitions, we're not using it in an optimal fashion. And there are some fundamental ways Cucumber framework behaves which had some issues where we were not able to write efficient code. So let's look into one of the example. I explain what I mean. So this is a feature file that we used to write before.

[00:05:55] This is nothing, but there is a big page for account opening and then user has to fill in their personal details, and this test is trying to mimic that flow. But as you can tell, the test is quite verbose. It is doing the operation for each feed. It is also trying to do things like mixing the functionality with the technical aspect of it, talking about index and whatnot. And then there is a data table, which it is trying to get the data to fill into the field, into the page and both the type of environment the data is for both the types of environment which it represents each line of the data table. It is saving the data inside the feature file itself, which can be improved, as you can see. And one important thing that I would want to point out that bond our fingers a lot was step definitions.

[00:07:17] Like, for instance, there is a step definition or here which you can see, and I click and this can be a button or any web element, for instance. But nobody is stopping anyone to write one more, one more step definition, which can basically be doing the same thing, which is I click a button, right, which is theoretically doing the same thing, but then I would duplicate of what it was doing. So similarly, this thing happened for other step definition a lot. And then we ended up having so many step definitions which were doing the same thing, but they were far different, which is which was very hard to catch during the code review. So this led to something which became unmanageable and led us to think about something else.

[00:08:17] **Speaker 1** So one fine day we decided enough is enough and we need something better. So few of us created the case for the next two versions. To build the next version of end-to-end system and socialize it with some key people in our group which included dev managers. They don't help convince management to fund an effort and so that prevent our team of six people was spun off and out of which three were test engineers, two were developers and one was scrum master or product owner. And initial time frame that was given to us to build something was one year. Once we started, however, the journey was bumpy. And we were a new team, so we were getting to know

each other due to there were starting torque, there were endless debates and analysis paralysis that sort of stuff, and the results were very hard to come by. After awhile, though, we started to make some progress. We were starting to agree more than we disagree. But however, there are some external pressures that we came under. There was some disagreement in terms of scope and there is a misalignment with the architecture team due to which management was not happy and they didn't feel confident that we would be able to deliver on what we're working on.

[00:09:54] And there was some re-org, but then soon after we learned from it, we did a pivot and we set a reasonable expectation with the stakeholders and management we aligned with the architecture, met midway, and then started turning out that test system and help people adapt it. Let's take a deeper look into the major components of the system that we had built. So the yellow box. It represents the core, which is nothing but the libraries that anyone can use to test the UI or web services-based applications like, for instance, you could use it to test anything within Fidelity or even Amazon.com, for instance.

[00:10:42] You can think of the blue box as the actual test implementation that uses the libraries to build the test. So if you want to focus a little bit on the yellow part, the core part of it. So the main thing in the core is the test runner, which is the entry point for all the tests, be the UI test or web services test it. It spuns spring boot application, which in turn creates the context, which is to ensure its liquid base, which clears the data locally on its database for the tests. There are other components which are out of the box, which tests can use. These are authenticators of which people can use to do what our customers want.

[00:11:38] There are also the components for the base classes, which contain the boilerplate code for that the test wouldn't have to do things like wire up the driver or select the browser, for that matter. So the test can just extend the UI base and then they should be ready to go with the basic functionality and see similar need that is web services based, which web services test can use. Let's dive a little bit deeper into the end-to-end testing part of it. So as you can see, end-to-end testing part contains the actual test.

[00:12:23] So UI test. Any UI test would extend the UI test base and web services test will extend repetitious test based, as we talked about. Other than that, it uses page object model, and there are some helper classes that are used to map the actual web elements of a page to the page object more than those are all page mappers and also there were helper classes for doing assertions. So those were called page asserters. This gives you an overview of the file structure and how it relates to the core.

[00:13:02] The test data handling was one of the more complex design decisions that we had to make. We had tried with various types of approaches. But we ended up going with the one that I'm sharing here right now, mainly because we didn't want our tests to be directly coupled with the data. That actual application used, so that's the reason we built an intermediate process called ETL, which is nothing but extract, transform, and load. So it gave us the ability to get the data from various data sources. We'd like our DB or record DB or Mongo DB like a web service for an instance, which was maintained by a third party. So it used to get external data from all those sources, massage it and transfer or transform it into the data that test would consume and then destroy in the Test DB. So this was kind of the job which used to run daily and which is to generate the data that test used consume.

[00:14:21] On the execution part of the things we used to generate our Docker image out of the core and the actual test. And that Docker image then used to get uploaded into the cloud for execution via Jenkins. And this is the high-level overview of the things, how they look when all the components come together. So basically, you can see the test which are consuming the data from the Test DB. And also the test execution that we just talked about.

[00:15:02] So while going cloud shopping for your toddler, the typical decisions that you have to make things like, should you buy clothes that are going to fit now and also after six months? Should you buy a combo or should you buy individual tops or bottoms? Should you buy gloves? Should you buy socks? On similar lines? We had to decide on tech stack for our end-to-end system. We knew early on that we want to use Selenium. However, we didn't want to build a wrapper around it ourselves. We knew we had to build a wrapper, but we didn't want to do it ourselves because there are so many frameworks that were available out of the box, which we could use and one of the things that stood out was GEB. The GEB stands for gebish.org. It's open-source, and it comes with its own stacked. The stack is made up of Groovy, which is the language which you write the code in. Obviously, it's wrapper was Selenium and it uses Spock runner, which internally uses JUnit under the benefit of the Spock that we liked that we wanted to use was BDD. It had some skeleton of BDD that it enforces and coming from Selenium are coming from Cucumber world. That was something that we definitely wanted to use.

[00:16:43] On the web service side of things. We wanted rest clients a basic rest client, which would do the lower level functionalities or operations, rest operations like Post, GET, and all those things. And as you are already using Groovy in UI, we didn't want to change the language. So we went with groovy REST Client Libs or Groovy REST Client Library and we used Groovy to write the web services test. If the common area or common text stack for the database, which was Amazon Aurora, which is similar to MySQL, which was hosted in the cloud. For the test data capabilities, for building the test data capabilities we used ETL. And which was using Apache Spark. For the test data executions, we use Jenkins pipeline and the Docker. And for multiple browser support, we selected Selenium Grid four, which was hosted internally. So let's look into the actual test implementation of the system.

[00:18:20] This is a typical UI test. Functionally, it is a donor who is contributing to EFT, which is like four to five pages flow and add enough to submission of the contribution. That is a validation that is happening to ensure that the contribution went and fine. Here are the major blocks of this test, the feature is kind of metadata of this test suite, which has a name, description, module, and layer. So this gets sealed into the test execution part of the tables of the test DB, then there are test cases which have the name of this particular test. And then this is the actual test which takes input as input data. So input data is nothing but a data table over here. And if you look closely, there are two sets of data which are there. So one is the starting data, which is this hardcoded values and that is dynamic data. So this dynamic data is coming from the data service, which is fetching the data from the data generated by ETL that I was showing before, which is getting the donor information, which can be dynamic based on these filter criteria. So once the test gets the data and each of these blocks represents one test execution, as you might guess, and multiple blocks will execute the test multiple times. So you can easily replicate this and change few and just add the variation in this test. So once the test gets the data, which is typically adapted to a map, then that is given when then. Given is typically logging in and then is actually the submission, the submission of contribution which is happening, which is like three to four page flow. As you can see, this can be reused easily. And there are

assertions that are happening once the submission is successful. And on the similar line, this is a typical web service test, which again has a similar format of test features and test case information, which gets recorded with every run. In that way, we can look back into the test executions.

[00:21:15] The actual test in this case, the test data is getting passed into this test using the variable over here. It's typical given when then block given when and then when this functionality grant submission, where the grant is getting submitted, we get the response back response is validated over here just for the status code and actually more detailed validation is happening in this matter. It extends the spec base, which is a base class which has the common functionalities which any other test in this similarity can use. And finally, this is just a peek in the runner class, which is the entry point of the code.

[00:22:17] As you can see, it's a spring boot application which uses Hibernate and Mongo configuration, and the main class is the entry point over here. And all it does is starts another spring boot, starts that spring boot application or starts that runner, which keeps starts the test execution.

[00:22:49] I think it will be interesting to talk about the repos. Initially, we started with just three repos, one repo for UI tests, one repo for web services, and one for the core, which contains the library and all that right. So then eventually we grew into this many numbers of repos that I'm going to walk you through. So as you remember, there were two apps, so we had to split up the UI tests into two repos, one for each app. Web services test, we could keep it in the same repo and the core part we had to split into three different parts as the things grew.

[00:23:34] So one was definitely the core framework, which contains the basic libraries that we just saw. Others were modules, which contains the data objects, the domain types of objects, and third was the ETL part of the code, which loads the data, right? So it has its own repo. Now that we have completed the deep dive into end-to-end test systems. Let's look at some of the valuable learnings that we had while working on it.

[00:24:17] As shown in this graph and also as I touched in the past that we started as a new team. So for any new team starting out on a project, there is a good chance that the efficiency will go down to start with, like us, we were hardly able to deliver anything in first few months. But then with time, things start to get better. People start to get along and then the efficiency increases. So this also is reflected with the study that was made on this topic, where any new team goes through these phases of forming, storming, morning, and performing.

[00:25:04] Like going back to the example of raising a toddler. Both the parents need to be on the same page to be able to raise the toddler in the right way. But it does not start that way, right? Both of them have their own viewpoint. And with time, things start getting better. So keeping this thing in mind while working with the new team or working on a new team is valuable. So next is about M.V.P. basically coming up with the MVP as soon as possible or as early as possible in your development cycle. And they're taking the buy in for that MVP from the stakeholders.

[00:25:51] So this helps keep the stakeholders in loop and maintains the trust and support that that team would need from the management. Also down the line, it will help reduce the pressure and help get the better outcomes. As you can tell from the test pyramid over here, end-to-end tests are always expensive in terms of time and resources. So it's always

a good idea to be cognizant about that fact and focus on quality over quantity of end-to-end testing. Right. And it is going to sound counterintuitive, but I want you to think about it for a moment. When I say don't rely on end-to-end testing to test the product.

[00:26:39] But rely on it to catch the breakages that may happen in the future. So this is one of those things that we didn't plan for it beforehand, and we had to spend a lot of time and resources, which is onboarding. Sometimes we feel that onboarding people to use the end-to-end testing is even more work than building the test itself.

[00:27:03] So basically, to help with it, plan for training and ongoing support beforehand so that you can plan for the resources and time for it and good documentation is the key. So plan to have a live document somewhere in the conference or whatever tool you use for the documentation where it is easily searchable and easily people can go back if they find some discrepancies. It should be easy for them to go back and fix it or update it. One of the valuable learnings that we had was we don't have to be done, done, completely done with MVP for it to open up for consumption.

[00:27:46] And as long as we give a reasonable workaround, you can ask saying that, Hey, this is something which is just to work around placeholder, just do the steps while we work to make this better. That is usually acceptable, that people are okay to or more than happy to use the new stuff that you built. And as far as feedback is concerned, do ask for the feedback. They can note of it, but then don't be too eager to act on it initially. Basically, what might happen is people have their first hearts. But then down the line, they usually change their mind to either agree with whatever you have built or they would refine their suggestions and we have something more to work on. So do act on it eventually if you see a pattern.

[00:28:48] But then in the right way. It's important to ensure that people are not going back to their old bad habits. So that's where the test governance comes in the picture. So test governance can be thought of as a combination of setting up the high standards of development. And in addition to that, electing group of people who can act as the test governors. So these are the group of people who are required approvers in all the PRs that are going into the system. Meet a new test or change the library, and they can help ensure that the high standards are calling development are being followed on the time. So like in case of a toddler, one can just expect them to be on their own all the time.

[00:29:52] They would need constant, low care, and affection. Whenever they ask for your attention, you have to leave whatever you are doing. And go and attend them. In the similar way, end-to-end test needs to be looked after. And that has to be a structured approach of monitoring the test executions or is it leads to constant failures and these failures pile up really fast. So our structured approach was each team was tasked to monitor the test execution for Sprint. And teams used to take up this task in a round robin fashion.

[00:30:44] So that team would be the first set of people who would look into their test failures. They may not be the one who are really fixing it, but they would be the one who would facilitate the addressing of the test. And that has to be done on the basis of priority or that should be the first thing in the list, as in when the test phase.

[00:31:14] This was our approach to monitor and make sure that test execution remains green all the time. And finally, I would like to thank each and every one of you to patiently listening to what I had to share. And I hope you are able to take something out of this and

you all enjoyed as much as I enjoyed putting this together for you all. If you want to stay in touch, these are my contacts and wish you all the very best of luck to raising your own and going, toddler. See you soon. Bye.