[00:00:00] **Ankit Pangasa** Hello, everyone. My name is Ankit Pangasa. I now work as a Senior lead with Adobe Systems. I build products. And when I say I build products, I mean, I discuss, design, develop, and test them. And testing manually is a thing of the past now and automation is the key to success. But is this automation sufficient? Or do we need faster automation? I guess since almost all of us work in Agile Methodologies, the answer is faster automation.

[00:00:33] So today, I discuss an automation design that eases your life by enabling you to write and check automation faster. So when we created this design, there are four key requirements that we wanted this design to abide by. It should be fast, it should be reusable, it should be extendible, and anyone should be able to easily extend and maintain it. I'll be using IntelliJ in my demo, and that demo is built using TestNG, and rest assured. The application used in the demo is a rest API. However, you can easily adapt this solution for the validation of soap APIs or even standalone applications. For the demo. We are using this Create Booking API that takes in details such as firstname, lastname, price, dates, and any special requirements to create a booking for the user.

[00:01:40] Now, let's discuss the conventional approach of writing test. We generally use functional programing during test automation. In this approach, each test is an individual independent entity in case if you are automating an API? The parameters and headers are defined in that test method itself, and all the tests are bound as a single unit in a single test class.

[00:02:06] However, there might be some utility or helper functions which are present outside of that class, but this is a quite simple design to start writing the rest. All you need to do is create a test class if it does not present, add a new test method and then you're just done. You can just get started. Let's take a look at the conventional approach in action. I have created a few positive and negative test cases for illustration. You can find this test in booking creation conventional approach class. As you can see here, we create a JSON for the test parameters. All these test parameters, including the first name, last name, price, dates, deposit paid, and additional needs are defined inline.

[00:03:10] The JSON is then passed on to the rest assured class for executing the API call. We then extract the response from this API and validate it against the expected output. So here we are, taking a look at one of the exception tests. We can also find a test for a positive case. Now, let's understand the pros and cons of this approach. To start with, this is a very convenient approach. This is the simplest of ever designs one can think of. So it is very easy to understand. But the downside of this approach is every time you add a new test, you need massive code reviews. Why? Because even a new test is an additional of and lines of code. Another downside is say, if the design changes because of any reason requiring you to change some inputs or outputs you need code change does require a review of almost all the test cases that you have written so far. And we all know how much we love code reviews.

[00:04:35] In fact, that times because of these code reviews, people fail to check their codes. And moreover, when you're working on tight deadlines, which is more than often the case code development takes an upper hand on writing automation, and thus the automation becomes a step child eventually, getting neglected and non-maintained or enhanced for new features. So here are the five pillars or considerations for a new design. First, we need to manage the data and code separately. Next is we want to take modular approach and break the big problems into smaller modular byte libraries.

[00:05:21] Another pillar is we want to have highly extendible design so that the automation can be easily extended for additional feature additions. Another key feature is the code should be easy to maintain so that even new members in the team can extend automations. And last but not least, we should follow coding practices so that any new member can understand or modify the existing code if required. Since data-driven testing is one of our key requirements, I would like to briefly discuss it. In data-driven testing, we managed the code and data separately so that take test data forwarded to the application and the test and then received output from the application under test. In the meantime, or sequentially, after receiving the data from that application under test, it can then take the expected output from the application.

[00:06:35] And finally, it compares both the expected and actual results to mark the test as passed or failed. So this is how our design looks like for the same API that we discussed before, we have four major modules. The first one is test package that holds all the test classes to better organize this test, we can club all the tests for a specific functionality in a single package. So like here, we have club all the booking creation tests in the booking creation package.

[00:07:11] For the more, we usually run the positive test more frequently than a negative test. For example, you might want to run a positive test every hour an exhaustive suite twice a day. So that's why we have created separate classes for positive and negative cases so that tests can run or can be easily controlled using these test classes in the testNG.xml files. You may say that the same can be achieved using test groups, etc. So, yeah, you can use them too. But we felt that this provides better test management. So this was our choice. And then we have model packages that would just be the backbone of the complete design. This package further has packages for input parameters, output parameters, test configurations, authentication parameters, if applicable, and then we have the utility of helper functions, which provide the test for specific functions to execute small and big, repetitive tasks. Finally, we have the test data that we place in the JSON package.

[00:08:27] As you can see, both the exception and the successful test are maintained in separate JSON files here. Now, let us deep dive into the code. Let's start with the JSON. As you can see, all of the tests are managed as JSON objects. Each test object has a config object that holds the test case id which helps us by debugging a test and the test description that briefly defines the purpose of the test. Then we have the input parameters. Since we have a JSON body booking resource, we have placed an exact replica of the same in the input parameters. If we would have got headers in the API, we can create another object headers in the input parameters and keep them as key-value players in there. And then finally, we have our output parameters. The output params have an object of a positive case.

[00:09:54] Since the API returns the created results. We hold the complete results value here for validation. Let's take a look at the exception test case JSON as well. So the exception test case JSON looks the same, but the exception JSON has the status code and error that should match with the exception and error returned by the server.

[00:10:26] Let's now take a look at our model package. Let's start with the test cases class. Each test cases class holds a list of test case object. That class has methods to retrieve these objects as required. Let's now open the test case class. Each test class has a composition of three classes, the test case config class. the case input parameters class, and the test case output parameters class. Furthermore, now let's take a look at the test config class. So we can see that we have the same two variables, id, and description, which we saw in the JSON present here. And then there are input parameters. As we saw, we had booking resource JSON in the input parameters.

[00:11:37] So we have the same here as well. Similarly, if we would have had headers in this API, you would have seen a representational class object for them as well. Now, moving into the booking resource, we can see all the input fields represented here. Nothing offered that in the future. If we want to add the email id as well for a user while booking all that we would need to do is extend this booking resource class and we will be done. Similarly, the booking date was in itself an object. So we have a class for booking dates as well. Now moving on to the output parameters.

[00:12:36] As we discussed, the output parameters can hold either a positive case or an exception case, so we have both objects over here. For every test, one of them would be null. Let's now take a look at the positive case. In the positive case, we hold the booking resource object, which is the same object that we were using in the import resource. So no extra coding is required here. Taking a look at the exception test case, we can see that we have created variables for status code and the error message here. Now, let's take a look at the test class booking creation success test. First thing that you can notice, we have just used 58 lines of code to automate the same 10 test cases that we're taking more than 300 lines.

[00:13:45] And the good part here is even if the test cases grow further for this functionality, the lines of code are not going to change. If you are still asking why. So that's because the test cases are going to be added in the JSON and for execution of those test cases, you're not going to add any single line of code. They are just read automatically and executed automatically. So we start our test by learning them. So let's take a look at this load test cases method. As you can see, we use faster xml to load the test cases using JSON property annotations. Using the data provider method. We then start executing the test method for all our test cases.

[00:14:45] In case if you're debugging the test, you would not want to run the whole suite. So we have a provision to run the specific test case here. So all you need to do is uncomment this code and comment out this part. So as you can see here, first, we validated the status code and then all the variable fields. Similarly, for the booking creation exception test we check for the status code and the error message. The key advantage of this design is faster automation checkings actually because of faster automation writing and reviews. This is easy to extend for new functionalities. A major advantage that we had with this design was when we decided to support another tech stack other than what we were offering for our product. Since that JSON was holding all our automation tests cases.

[00:16:04] Our tests cases were just right there. All that we needed to create was a design and we were up and running with all the test cases in just a few days. Here are some important links. The code is available at GitHub for quick reference and adoption, and I'm available on LinkedIn, GitHub, and Facebook to discuss this further.

[00:16:29] I hope you enjoyed attending the session as much as I enjoyed delivering it. In case if you want to adopt our design? Prototype is available at the GitHub location to quickly get started. However, if you are wondering if the design would fit for your use cases, we can make it work together.

[00:16:47] Thank you, and I'm looking forward to a lot of new connections to expand this design further. I'll end with a quote, many ideas grow better when transplanted into another mind than in the one where they sprang up. Thank you.