Mock servers are used to help solve many different testing problems. They can help you control what inputs the UI gets, and they can be used to help drive better designs and to better understand how an API works.

During development stages, when no backend is yet developed, we can create Mock server, so it simulates already working backend. We can set up what backend would give us in response and what request should have (which headers, body). So when already working backend is released, we would just switch the URI of Mock server to backend URI and run our checks.

**Different approaches to testing with mocks**

* **We are Producers of an API.** When we are working in this way, we are looking at how to verify that an API works the way it should. One way to test this is at the integration level where you send a request and see what you get back. This kind of testing is great because it lets you see that all the different pieces are working well together. However, it also means that there are many points of failure. If something goes wrong in the network, or if there is a bug in the database query, or if there is a problem in formatting the response, the test will fail. This can make it harder to figure out where the issue is. It can also make each test take some time to run

Although you should always have some integration tests, it is often more efficient to break up the different parts of the service and test them on their own. So, instead of just doing one call that sends a request through the network, does work on the server, and then sends back a response, you could test the different pieces of it separately. You could create tests on the server that verify that the SQL queries work correctly, tests that check that the server can handle different kinds of API requests, and tests that check that everything gets properly packaged up again.

* **We are Customers of an API**. This might be an API that another team at our company has made, or an API that provides information from another company altogether. In this scenario, the API is a black box to you, and you are not able to test the underlying pieces of it. The testing that you do in this situation is often more targeted at making sure that the API gives you the information that you need in the format you need it. These kinds of tests are less about helping you find where bugs are in the API and more about ensuring that any changes to the API won't break your downstream workflows.

In this realm, a full integration test would be doing something like clicking in a UI and ensuring that things go correctly through the system and that the UI responds correctly. There is still further test decomposition that can happen here as well. We could have a set of tests that checks whether the API continues to give back the responses we want, and then another set of tests that checks that the UI works correctly with those responses.

As testers, we often approach UI test automation in a very foolish way. We act as if it is a good way to test the functionality of a system. The reality is, UI automation is a very slow and error-prone way to check the functionality of a system. It is much better to check low-level functionality in a unit test. However, this does not mean that UI automation is not valuable. It really is. It's just that it needs to be used to test the UI. We shouldn't be trying to verify business logic or database behavior or stuff like that in UI automation. We should be trying to verify UI behavior. UI tests are for testing the UI. OK, I won't rant about this anymore, but it leads nicely into why mocking is such an important part of a good API testing strategy.