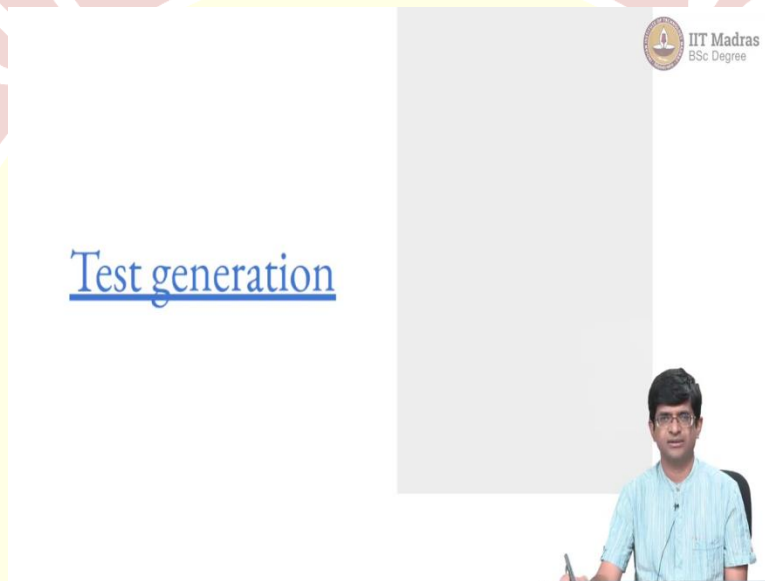


**IIT Madras**  
ONLINE DEGREE

**Modern Application Development-1**  
**Professor Nitin Chandrachoodan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**  
**Test Generation**

Hello, everyone, and welcome to this course on modern application development.

(Refer Slide Time: 00:16)



So, now that we know that tests are important, and what kind of tests need to be done, one question that we can ask is, is there some systematic way to generate these tests?

(Refer Slide Time: 00:27)



### API-based testing

- Application Programming Interface: abstraction for system design
- Standard representations for APIs
  - OpenAPI, Swagger etc.
- Can they also generate test cases?

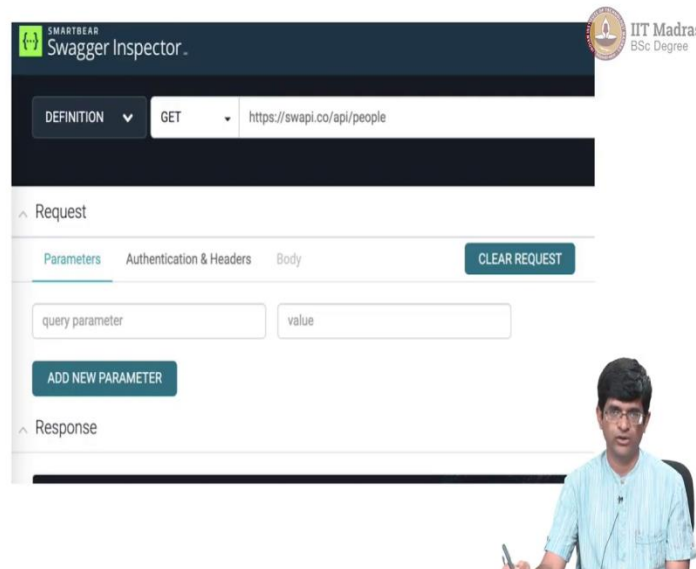


And we are going to look at a few different options as far as that is concerned over here. One approach that is sort of quite popular and is a very important thing to know is what is called API based testing. So, the application program interface is an abstraction that is brought in first system design we already know about restful APIs and know how they are created.

Why they are created? And we have also seen the open API as a specification method that allows to represent APIs. So, the question that we might naturally ask is, if I have an open API specification, or a swagger specification, can I automatically generate test cases based on that, because we already seen that open API, there are ways by which I can use the open API output to, generate some part of the code at least.

So, that the rest of it just needs to be filled in by the developer, but at least part of the code will be generated directly by a script from the open API spec. Can it also generate test cases?

(Refer Slide Time: 01:37)



So, there are ways of doing it. So, for example, swagger, which is the earlier version of open API, and which is where open API evolved from, if you look at the smart bear, the company smart bear software, they have something called swagger inspector, which essentially allows you to, create, and you can give it some kind of an open API endpoint.

And from there, it would be able to generate a set of tests. So, you could, for example, give it what are the query parameters, what should be the authentication, what should be the body, and it would be able to generate tests for you. So, this is very powerful. What it says is open API, why was it useful in the first place, because it helps to documentation of the API.

And secondly, it also helped to the implementation, because some stub code could be generated. Now it is a linear you can go one step further and even generate tests. This is clearly a good idea for the simple reason that you are now trying to have one single source of truth for multiple different parts of your implementation.

(Refer Slide Time: 02:48)



### Use cases

- Import API definition from standard like OpenAPI
- Generate tests for specific endpoints, scenarios
- Record API traffic
- Inject possible problem cases based on known techniques
- Data validation tests



Now, what are the use cases? If you have your API already defined using a standard like open API, then you could import an API definition from there, generate tests for specific endpoints, for different scenarios? You could record API traffic, so the swagger inspector that I showed earlier, they have a lot of variants on this.

Which allow you to do different kinds of tests, record different kinds of data, inject possible problem cases, based on known techniques and how a whole bunch of data validation tests, all of which are sort of known to be good practice. So, the main reason why these become useful is because they are able to capture the wisdom that software developers have got over the years, and make it relatively easy for a newcomer to use those things, use those ideas.

सिद्धिर्भवति कर्मजा

(Refer Slide Time: 03:40)

### Abstract Tests

- Semi-formal verbal description:
  - Make a request to '/' endpoint
  - Ensure that result contains text "Hello world"

```
def test_hello(client):  
    """Verify home page."""  
    rv = client.get('/')  
    assert b'Hello world' in rv
```



So, related to this, I mean, going slightly different away from the idea of an API test, there is something called an abstract test. Now, what exactly is an abstract test? One possible way of thinking about an abstract test is that it is a semi-formal description. What do I mean by semi-formal? Look at these two bullets that I have over here, it says make a request to the / endpoint and ensure that the result contains a text "Hello world."

So, this would be what I would call an abstract test, because it tells me in a reasonably clear way, exactly what I need to do, but it is not cold. This, on the other hand, might be considered an actual test. Because what it would say is I have defined a function test hello, which verifies the contents in the home page. I am going to assume that I have something called a client.get will address this a little bit later when we talk about pytest.

But client.get you can imagine is issuing an HTTP get request to the /endpoint which is a direct translation of this line. And the next thing is assert hello world in rv.data, which is direct translation of this line, the result contains the text "Hello world." So, in this way, these two lines of Python code have implemented these two bullet points, and therefore this code on the right becomes a Python test that implements this abstract test that I have on the left.

So, on the left is an abstract test, what I have on the right is what is called an executable test, it can be run. So, this kind of abstract test might be something generic, I might be able to come up with a set of abstract tests that apply for a large set of examples, but I need to tune them for my specific usage scenario. And that part of it has to be done by writing code, can it be automated, there may be certain circumstances in which part of this can be automated, but at the very least, if I have the abstract tests, then generating the executable tests becomes a little bit easier.

(Refer Slide Time: 06:02)

### Model-based testing

Example: Authenticate user before showing information

- Scenarios:
  - User already logged in - page shown
  - User not yet logged in - redirect to login page
  - Forgot password - after resetting, come back to desired page
- Model:
  - Possible states (logged in, password reset, ...)
  - Possible transitions
  - Generate tests for the possible transitions



A related thing in the context of abstract test is what is called model based testing; an example of a model based testing might be that, you can think of an entire software system as following a certain model of execution. An example would be that let us say I need to authenticate a user before showing them certain information, and there are multiple scenarios over here.

The user may be already logged in, in which case, I just showed them the page, user has not yet logged in, I need to redirect them to the login page, after they log in, I need to then come back to the first step and show them the page. Let us say I forgot the password, now it should redirect them to a reset page. Once they have logged in, finally, I should still come back to the page that I wanted to see in the first place.

So, those are possible scenarios that I might want to implement, which means that each of these scenarios, each of these states of the system where the user is logged in, or we need to reset their



password. That is a different state of the system. There are different possible transitions over here. And I can from that construct some kind of a finite state machine which says, user is initially not logged in, present them with a login page.

They have forgotten their password that goes to another state, then something else they come and then finally they are in the logged in state. Now, given a finite state machine of that sort I know what are the possible transitions between the different states which mean that it may be possible to construct this model and construct test cases automatically, that will test all the possible transitions. So, for example, if I go to the password reset page after that, where do I go next?

Do I come back to the correct page that I wanted, what happens if the password reset failed? What happens if it times out all those are things that I can construct automatic test cases for based on this model of the system that I have. So, model based testing is yet another way by which I can generate tests automatically for a system.

(Refer Slide Time: 08:06)

### Models and Abstract Tests

- Abstract tests apply to generic models
- Create model for system-under-test
- Derive “executable” tests by combining abstract test information with model

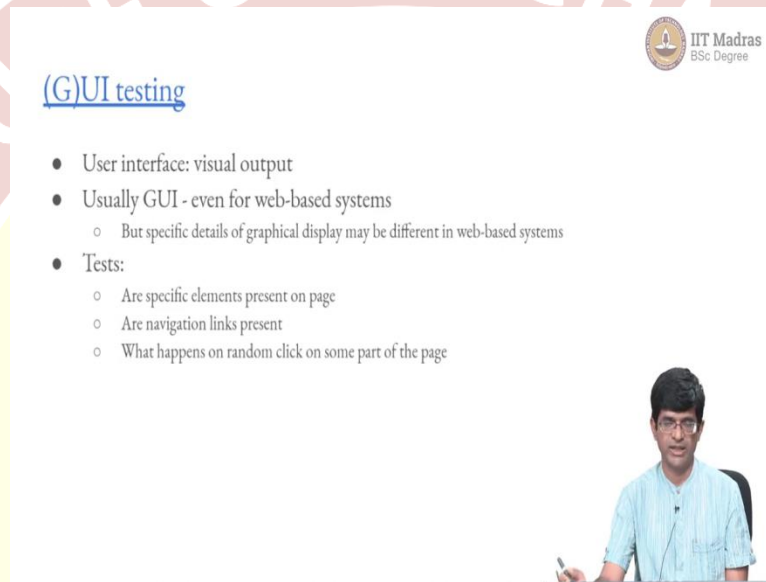


Models are usually combined with these abstract tests. Because abstract is can be applied to generic models, I can sort of talk in general about, user is not logged in, do this kind of thing. But that is not the same as writing code for my specific scenario, because I am finally implementing it using Python flask. Whereas the model might be something more generic, which just says there are multiple states possible for the system.



And the abstract test might simply say I need to check if the user is logged in, whether implemented using Python flask, or Django or PHP Laravel, or Java or ASP .net or anything else of that sort. I should finally be able to translate these kinds of abstract tests in combination with this model, and be able to run similar kinds of tests on all of them. So, that is converting from the model to the executable test.

(Refer Slide Time: 09:05)



(G)UI testing

- User interface: visual output
- Usually GUI - even for web-based systems
  - But specific details of graphical display may be different in web-based systems
- Tests:
  - Are specific elements present on page
  - Are navigation links present
  - What happens on random click on some part of the page

Now, after all of this, we also have this problem of so called user interface testing and the reason I put the G in bracket, so there is because mostly when we talk about user interface, we are talking about graphical user interface, so GUI testing. Now, even for a web based system, we are talking about GUI testing. The one difference in a web based system is because of that separation of concerns, normally in a GUI, I have things like, is there a button at this position on the screen.

Or is there a navigation link on the top left part of the screen. Whereas for a web based system, because of the fact that the styling is handled by CSS and the semantics, or the meaning is handled by the HTML, I may be able to also test in different ways. I might just look whether the navigation links are present in the page, which is good enough, if I am looking at an accessible reader for example.

Or I might be interested in is the navigation link present on the top left corner of the page? So, those are different kinds of tests that need to be run. But I might also want to try out things like,

what happens if I just generate a random click on some part of the page? Will it cause problems? Can it ultimately cause the server to crash? Or can it cause corruption of data somewhere inside the database? Those are my biggest problems ultimately.

(Refer Slide Time: 10:37)

### Browser automation

- Some tests cannot be directly run programmatically
  - Browser is **required**, just requests not sufficient
- Example:
  - IRCTC or SBI website - captcha protected
  - Some user input also required - cannot be completely automated
- Request generation:
  - Python requests library
  - Capybara (ruby), ...
- Direct browser automation:
  - Selenium framework - actually instantiate a browser



And for this, there are certain tests that cannot be directly run completely programmatically, because, you might have come across things like the IRCTC or the SBI banking website, which are usually protected by extra means, such as captcha, and various other things. The reason being, they do not want people to abuse the site, they want to limit the number of people who can access it at a given point.

The problem is external testing then becomes difficult, because unless I can solve the captcha, I cannot proceed beyond a given stage. So, this is where there is, in particular, a framework called Selenium which allows you to directly automate a browser. It is almost as though you open a Chrome or Firefox browser and you can programmatically tell the mouse to go and click on one particular corner of the screen or click on some part of the thing or look for an element and selected and click on it.

So, all that scripting can be done to a large extent, it imitates a browser, not just imitates actually instantiate a browser, so that the behavior is exactly the same as what would be seen by the end

user, who is going to do it manually, it just makes it a bit faster to do. There are a few other things like the Python request library that can be used to automate request generation.

But Selenium goes one step beyond that it puts a browser there so that even things like cookies, and so on are handled exactly the same way that the end user would see it. All of these techniques are there; they are pretty much beyond the scope of what we can do in this course. But they are powerful tools for a person who is developing a large web app.

(Refer Slide Time: 12:22)

### Security testing

- Generate invalid inputs to test app behaviour
- Try to crash server - overload, injection etc.
- Black-box or White-box approaches
- **Fuzzing** or Fuzz-testing:
  - Generate large number of random/semi-random inputs

And finally, the last part of testing is what is called security testing. And the idea here is very simple. I should see whether there are certain behaviors I can invoke that would cause the app to either crash, or even worse, corrupt the internal databases. And the reason I am saying even worse, corrupted internal databases is because if the database gets corrupted without sort of alerting someone.

It means app can continue to run and people will not even know that the data in it is wrong, whereas at least if it crashes, that is called more like a denial of service the nobody is able to use it, but at least you do not get the wrong information. And there are multiple ways of this kind of security testing, we have already talked about security, we have talked about so called SQL injection attacks. Those are the standard tests.

Try various kinds of SQL injection attacks, and they can be automated to a large extent. Here, again, I could do either a black box approach, where I assume that the attacker knows nothing about how it is implemented. Or I try a white box approach where I know how the code is implemented, and then try to see what happens if I generate certain kinds of code.

Now, as an app developer, at least for security testing, you should ideally be looking at the white box approach, because you cannot just rely on a black box and say, there is no way that outsider could guess that I have used this kind of structure to implement it, you have to assume that they have somehow got access to your code as well, even then they should not be able to break the system.

And one related thing is what is called Fuzzing or Fuzz testing, which what it tries to do is generate a large number of random or semi random inputs with the goal of breaking either some input text box or clicking randomly all over the screen, trying to see what happens whether this sim gets overloaded or whether it crashes or whether it takes in wrong data. So, all these are important for point from the point of view of security testing.

