Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 10 — January 23, 2019

Patrick Lam *version 1*

REST APIs

REST (Representational State Transfer) is an architectural style widely used in web applications. Architectural styles are SE 464 material, but relevant to Assignment 1, so we'll talk about them briefly here. [I wouldn't ask questions about REST itself on exams, but one could expect questions about testing REST systems.]

Many applications in the world these days are web apps, and as you know, these apps usually have a frontend and a backend. Assignment 1 has you testing both the frontend and the backend of my app. We'll talk about the backend here. For our purposes, it provides an API which allows the client to store data.

Key features of REST:

- client-server;
- stateless; and
- provides uniform interfaces.

Client-server is easy. The client (in our case, the front-end JavaScript code) sends requests to a server. The server is responsible for storing the data. This enables the client to change somewhat independently of the server. For instance, the server might decide to change what database it uses behind the scenes. The client doesn't need to know about that—separation of concerns.

Statelessness helps systems scale better. What this means is that the server isn't responsible for remembering anything about client identities. The server receives requests and processes them independently. The usual concern is about authentication; requests need to carry authentication data with them (usually provided by a separate service). An analogy: because SE is a small program, I try to remember state in my head about each of you between interactions—at least your name. Larger programs need students to provide the necessary state (student ID numbers) with each request; the advisors would then look you up in a database. You can see how my approach doesn't scale to larger sets of students.

Uniform interfaces include resources and verbs. Resources are what the REST service is modifying, e.g. cat pictures. Each resource has a Uniform Resource Identifier. In our se465-flashcards app, it's just an ID number. But it could be more complicated, e.g. `/store/1/employee/2`. Verbs are what to do with the resources. HTTP defines standard operations, including `POST`, `GET`, and `DELETE`.

Further reading:

- Brian Mosigisi. "A Quick Understanding of REST".
  `https://scotch.io/bar-talk/a-quick-understanding-of-rest`

- Wikipedia is an OK reference here: `https://en.wikipedia.org/wiki/Representational_state_transfer`

Testing the REST API

Sending REST requests. At the lowest level, we need to be able to send REST requests. It's easy to send GET requests in the browser by just navigating to a page. You can't quite send other requests with your bare hands. But you can use the browser developer tools in Firefox and Chrome. The syntax can be a bit unwieldy; stackoverflow provides the following example[1]:

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  body: JSON.stringify({
    title: 'foo', body: 'bar', userId: 1
  }),
  headers: {
    'Content-type': 'application/json; charset=UTF-8'
  }
})
.then(res => res.json())
.then(console.log)
```

Or, you can use command-line tools like `cURL` and `wget`. The Firefox dev tools will allow you to copy a request as cURL and run at the command line, possibly after editing.

But really, you want to script this. REST Assured lets you write Java code like this[2]:

```
    @Test public void makeSureThatGoogleIsUp() {
        given().when().get("http://www.google.com").then().statusCode(200);
    }
```

and run it as a JUnit test.

To check the result, one can compare the status code to what's expected. 200 OK is usually a good HTTP code, which APIs should return when everything went well. And there's also a result, which one can reason about:

```
    @Test public void verifyNameStructured() {
        given().when().get("/garage").then().body("name",equalTo("Acme garage"));
    }
```

You can also construct a POST request using the REST Assured API.

Testing Strategy. We discussed testing at the tactical level above, which is all you have to do for Assignment 1. The broader view from this course also includes test suite construction strategies, and we can talk about various kinds of coverage. You could, for instance, require coverage of the complete API. Better yet, you could require API coverage and also require that the tests cover the specified behaviour of the API.

[1] `https://stackoverflow.com/questions/14248296/making-http-requests-using-chrome-developer-tools`

[2] `https://semaphoreci.com/community/tutorials/testing-rest-endpoints-using-rest-assured`