Lecture notes

Vittorio Zaccaria

March 5, 2019

Contents

1 Lecture 0 - Specifying an API

1.1 Welcome

- You will learn how to finally deploy your ***internet application*** in the real world
- You will be ***graded with a project assignment*** which will consist of both ***source code*** and a ***running instance of your app on a cloud service***.
  - The final goal of this part is to give a foundation for the construction of modern web applications
  - Almost all of Backend lessons will be hands-on, so ***bring your own device*** and a stable network connection.
  - This course is *not* a deep-dive into:
    - ⋆ Cloud Computing
    - ⋆ Distributed Systems
    - ⋆ Databases (we will take off from what you've already seen in that course)

1.2 Course project

Concerning the project there are a few important points:

- Every project should host its code on a ***private*** Git repository and provide a running application on cloud hosting platform.
- The team is composed of max. 3 students, where one of them will be elected as *team administrator*.

1.3 Web applications and web services

- This part is mostly an overview of the theoretical background behind web applications.
- This is just a recap about topics addressed in the *Information Systems* course.
- Let us start from the basics and see where this fits into the web application scenario.

We start by recalling what the term ***service*** means

- logically represents a business activity with a specified outcome which can be any kind of artifact
- is self-contained. Assuming that someone is offering the service, you dont need anything else to bring the activity to completion.
- is a black box for its consumers. Consumers only know the surface of the service. Otherwise it would result in a too tightly coupled communication.
- may consist of other underlying services.

1.3.1 Without a service oriented architecture

- Clients contain the application logic
- Might be replicated across them for common functionality. Obvious maintainability issues, violation of the DRY principle.
- Changing the structure of databases might imply rewriting the clients.

1.3.2 With a service oriented architecture

- Changing the structure of the database means rewriting only the remote application server.
- Clients are unimpacted by the change.
- In SOA you introduce a ***service layer*** where artifacts are provided by a single service invocation.
- You interact with it through an interface that does not expose implementation details (RPC or other high level language)
- SOA is the main architectural paradigm used to build web application today. Let's see why

1.3.3 Web service

- A Web service is a service built using web standards (we'll see in a few moments what does it mean) just consider:
  - Application layer → browser
  - Service layer → web server
- A web service protocol dictates how HTTP should be used to convey application requests.
- Services are really black boxes. The application knows only about ”resources”

1.3.4 Activity pattern

- Now, a typical pattern of communication between web client and web server is shown here
- ***First two arrows***. The request is tipically initiated by the client (usually, first to get the assets needed to display the page).
- ***Second two arrows***. The client requests the data to render and any other activity (that might also change the state of resources) happens next.

1.4 HTTP based networking

1.4.1 Anatomy

- HTTP is a ***communication protocol***, i.e., a system of rules that specify how a request for a resource operation and the response should be formed (message negotiation and transmission).
- Much of HTTP1.1 standardization was guided by ***a core set of principles and constraints*** that today we call ***REST***.
- You'll see a lot this term in these lectures because we are going to talk about ***REST*** compliant web services (there are other techniques, that you probably will see in other courses such as SOAP).
- Before addressing the most common practices, let's see what are the basic abstractions offered by HTTP.

1.4.2 Resource

- the ***resource*** is the first abstraction we are going to consider
- the ***identifier*** is a text string called uniform resource identifier - URI.
- You can change the resource's state through a request using HTTP verbs.
- a ***representation*** is a textual description of the actual state of the resource (JSON, XML and so on.).

1.4.3 HTTP verbs

- ***Get***:
  - Request a copy of a *resource*
  - This is how the browser requests any HTML page or any other asset
  - The request should have no side-effects (i.e., doesn't change server state). That is why the second one is bad. This is because, there can be several layers of caching in the network.
- ***Post***:
  - Example, chat app, add users to app etc..
  - Generally used to create a *resource* (in this case, a new message)
  - Has ***side-effects***
  - ***Not idempotent*** (i.e., making the same request twice creates two separate, but similar resources)
- ***Put***:
  - Often used to change or completely ***replace an existing resource***.
  - Has ***side-effects***
  - Should be idempotent (e.g., updating a resource twice should result in the same effect to the resource)
- ***Delete***:
  - ***Destroys*** a resource
  - Has side effects
  - Should be idempotent

1.5 Demo D0.0 - using curl to interact with an HTTP service

- Prerequisites: curl, jq
- Remote API: `https://github.com/workforce-data-initiative/skills-api/wiki/API-Overview#introduction`

```
curl -X GET "http://api.dataatwork.org/v1/jobs" -v | jq .
curl -X GET "http://api.dataatwork.org/v1/jobs" | jq .
curl -X GET "http://api.dataatwork.org/v1/jobs?limit=2" | jq .
curl -X GET "http://api.dataatwork.org/v1/jobs/26bc4486dfd0f60b3bb0d8d64e001800/related_jobs" | jq .
curl -X GET "http://api.dataatwork.org/v1/jobs/26bc4486dfd0f60b3bb0d8d64e001800/related_skills" | jq .
curl -X GET /path/to/api/v1/jobs/autocomplete?contains="software"
```

- You do it! Search for the related skills of a baker

```
curl -X GET 'http://api.dataatwork.org/v1/jobs/autocomplete?contains="baker"' | jq .
```

and, choose and UUID and then use relatd skills

1.6 Demo D0.0bis - using the browser to interact with an HTTP service

Use the browser

```
fetch('http://api.dataatwork.org/v1/jobs?limit=2')
  .then(function(response) {
    return response.json();
  })
  .then(function(myJson) {
    console.log(JSON.stringify(myJson));
  });
```

1.7 Demo D0.1 - Using the swagger editor to document an API

1. Load up the Skills API in the swagger editor

2. Describe parameters

3. Describe responses

1.8 Demo D0.2 - Test the API with SwaggerHub

1. Load up the Skills API and try the same commands as above by using the interface

2 Lecture 1 - Javascript

2.1 Demos contained in the presentation

3 Lecture 2 - Implementation

3.1 Demo D2.0 - Generate the server

- Simply load up the bookstore API (only books) into the swagger editor, download and run the server.

3.2 Demo D2.1 - Serve static assets

- Add `serve-static`

```
let app = require('connect')();
/* .... */
let serveStatic = require('serve-static');
app().use(serveStatic(__dirname))
```

- Add example index.html (from 'vz-bookstore-alpha-2019', tag only.book.v0)

- Deploy on github

3.3 Demo D2.2 - Deploy on Heroku

Deploy on Heroku
Important:

- change swagger.yaml ”host” to: `polimi-hyp-vz-demo.herokuapp.com` so that swagger user interface can work.
- change swagger.yaml ”scheme” to `https`
- change the port in the code to `process.env.PORT || 8080`

Then:

1. Install the Heroku command line
2. Create an application with name polimi-hyp-vz-demo (region europe)
3. Connect to github
4. Find the Repo and press manual deploy
5. Press Open App

4 Lecture 3 - Sessions and state

4.1 Demo D3.0 - Add a user with login and logout actions to the OpenAPI spec

4.2 Demo D3.1 - Add cookie-session

4.3 Handson D3.2 - Add a `user/logout` endpoint

4.4 Handson D3.3 - Add a `user/register` endpoint