

Lecture notes

Vittorio Zaccaria

March 13, 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 don't 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 typically 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>

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

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

```
1 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

```
1 fetch('http://api.dataatwork.org/v1/jobs?limit=2')
2   .then(function(response) {
3     return response.json();
4   })
5   .then(function(myJson) {
6     console.log(JSON.stringify(myJson));
7   });
```

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

1.9 Handson D0.3 - Devise an OpenAPI spec for users, items, and carts

2 Lecture 1 - Javascript

2.1 Demos contained in the presentation

3 Lecture 2 - Implementation

3.1 Demo D2.0 - Generate the server

- Define the bookstore API as follows:

```
1  swagger: '2.0'
2  info:
3    description: >-
4      This is a simple bookstore server with a book inventory, users and a shopping cart.
5    version: 1.0.0
6    title: Simple Bookstore
7    contact:
8      email: vittorio.zaccaria@polimi.it
9    license:
10     name: Apache-2.0
11     url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
12 host: none.yet.io
13 basePath: /v2
14 tags:
15   - name: book
16     description: Available book
17   - name: cart
18     description: Access to the cart
19   - name: user
20     description: Operations about user
21 schemes:
22   - http
23 paths:
24   /books:
25     get:
26       summary: Books available in the inventory
27       tags:
28         - book
29       description: 'List of books available in the inventory'
30       produces:
31         - application/json
32       parameters:
33         - name: offset
34           in: query
35           description: Pagination offset. Default is 0.
36           type: integer
37         - name: limit
38           in: query
39           description: >-
40             Maximum number of items per page. Default is 20 and cannot exceed
41             500.
42           type: integer
43       responses:
44         '200':
45           description: A collection of Books
46           schema:
47             type: array
48             items:
49               $ref: '#/definitions/Book'
```

```

50     '404':
51         description: Unexpected error
52     /books/{bookId}:
53         get:
54             summary: Find book by ID
55             tags:
56                 - book
57             description: Returns a book
58             operationId: getBookById
59             produces:
60                 - application/json
61             parameters:
62                 - name: bookId
63                   in: path
64                   description: ID of book to return
65                   required: true
66                   type: integer
67                   format: int64
68             responses:
69                 '200':
70                     description: successful operation
71                     schema:
72                         $ref: '#/definitions/Book'
73                 '400':
74                     description: Invalid ID supplied
75                 '404':
76                     description: Book not found
77     definitions:
78         Book:
79             title: Book
80             description: A book for sale in the store
81             type: object
82             required:
83                 - title
84                 - author
85                 - price
86             properties:
87                 id:
88                     type: integer
89                     format: int64
90                 title:
91                     type: string
92                     example: Il deserto dei tartari
93                 author:
94                     type: string
95                     example: Dino Buzzati
96                 price:
97                     $ref: '#/definitions/Amount'
98                 status:
99                     type: string
100                    description: book availability in the inventory
101                    enum:
102                        - available
103                        - out of stock

```

```

104 Amount:
105   type: object
106   description: >
107     Price
108   properties:
109     value:
110       format: double
111       type: number
112       minimum: 0.01
113       maximum: 10000000000000000
114     currency:
115       $ref: '#/definitions/Currency'
116   required:
117     - value
118     - currency
119 Currency:
120   type: string
121   pattern: '^[A-Z]{3,3}$'
122   description: >
123     some description
124   example: eur
125 externalDocs:
126   description: Find out more about Swagger
127   url: 'http://swagger.io'

```

- 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

```

1 let app = require('connect')();
2 /* .... */
3 let serveStatic = require('serve-static');
4 app().use(serveStatic(__dirname) + "/www");

```

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

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
6     <meta name="viewport" content="width=device-width" />
7
8     <title>Book store</title>
9
10    <link rel="stylesheet" href="style.css" />
11    <!--[if lt IE 9]>
12      <script src="//html5shiv.googlecode.com/svn/trunk/html5.js"></script>
13    <![endif]-->
14  </head>
15
16  <body>
17    <h1>Our first server is running!</h1>

```



```

18     <ul></ul>
19 </body>
20 <script>
21     var myList = document.querySelector("ul");
22     fetch("v2/books")
23     .then(function(response) {
24         if (!response.ok) {
25             throw new Error("HTTP error, status = " + response.status);
26         }
27         return response.json();
28     })
29     .then(function(json) {
30         for (var i = 0; i < json.length; i++) {
31             var listItem = document.createElement("li");
32             let { title, author, price } = json[i];
33             listItem.innerHTML = `${title} - ${author} - ${price.value} (${
34                 price.currency
35             })`;
36             myList.appendChild(listItem);
37         }
38     });
39 </script>
40 </html>

```

- 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

```
1 /user/login:
2   post:
3     tags:
4       - user
5     summary: Login
6     description: Login with a form
7     consumes:
8       - application/x-www-form-urlencoded
9     produces:
10      - application/json
11    parameters:
12      - name: username
13        in: formData
14        required: true
15        type: string
16      - name: password
17        in: formData
18        required: true
19        type: string
20    responses:
21      '200':
22        description: succesfull login
23      '404':
24        description: unauthorized
25
26 /cart/{cartId}:
27   get:
28     tags:
29       - cart
30     summary: View the content of the cart
31     produces:
32       - application/json
33     parameters:
34       - name: cartId
35         in: path
36         required: true
37         type: integer
38         format: int64
39     responses:
40       '200':
41         description: succesful operartion
42         schema:
43           $ref: '#/definitions/Cart'
44       '404':
45         description: unauthorized
46
47 definitions:
48   User:
49     title: User
50     description: A user
51     type: object
```

```

52     properties:
53       id:
54         type: integer
55       name:
56         type: string
57       address:
58         type: string
59       creditcard:
60         type: string
61     example:
62       id: 1
63       name: Vittorio
64       address: DEIB
65       creditcard: xyzabc
66
67     Cart:
68       title: Cart
69       description: Order for books
70       type: object
71       properties:
72         total:
73           $ref: '#/definitions/Amount'
74         books:
75           type: array
76           items:
77             $ref: '#/definitions/Book'

```

4.2 Demo D3.1 - Add cookie-session

- Change controllers/User.js

```

1  module.exports.userLoginPOST = function userLoginPOST(req, res, next) {
2    var username = req.swagger.params["username"].value;
3    var password = req.swagger.params["password"].value;
4    if(!req.session.loggedin) {
5      req.session.loggedin = true;
6    } else {
7      req.session.loggedin = !req.session.loggedin;
8    }
9    User.userLoginPOST(username, password)
10     .then(function(response) {
11       utils.writeJson(res, response);
12     })
13     .catch(function(response) {
14       utils.writeJson(res, response);
15     });
16 };

```

- Change controllers/Cart.js

```

1  module.exports.cartCartIdGET = function cartCartIdGET(req, res, next) {
2    var cartId = req.swagger.params["cartId"].value;
3    if (!req.session || !req.session.loggedin) {
4      utils.writeJson(res, { error: "sorry, you must be authorized" }, 404);
5    } else {

```

```
6     Cart.cartCartIdGET(cartId)
7     .then(function(response) {
8         utils.writeJson(res, response);
9     })
10    .catch(function(response) {
11        utils.writeJson(res, response);
12    });
13 }
14 };
```

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

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

5 Lecture 4 - Serving data

5.1 Demo D4.0 Installation

```
1 npm install knex -SE
2 npm install pg
```

5.2 Demo D4.1 Setup data layer

- service/BookService.js module

```
1  let sqlDb;
2
3  exports.booksDbSetup = function(s) {
4    sqlDb = s;
5    console.log("Checking if books table exists");
6    return sqlDb.schema.hasTable("books").then(exists => {
7      if (!exists) {
8        console.log("It doesn't so we create it");
9        return sqlDb.schema.createTable("books", table => {
10          table.increments();
11          table.text("title");
12          table.text("author");
13          table.float("value");
14          table.text("currency");
15          table.enum("status", ["available", "out of stock"]);
16        });
17      } else {
18        console.log("It exists.");
19      }
20    });
21  };
22
23  exports.booksGET = function(offset, limit) {
24    return sqlDb("books")
25      .limit(limit)
26      .offset(offset)
27      .then(data => {
28        return data.map(e => {
29          e.price = { value: e.value, currency: e.currency };
30          return e;
31        });
32      });
33  };
```

- service/DataLayer.js module

```
1  let { booksDbSetup } = require("../BookService");
2
3  const sqlDbFactory = require("knex");
4  let sqlDb = sqlDbFactory({
5    debug: true,
6    client: "pg",
7    connection: process.env.DATABASE_URL,
8    ssl: true
```

```

9   });
10
11  function setupDataLayer() {
12    console.log("Setting up Data Layer");
13    return booksDbSetup(sqlDb);
14  }
15
16  module.exports = { database: sqlDb, setupDataLayer };

```

- index.js

```

1  let { setupDataLayer } = require("./service/DataLayer");
2
3
4  // Initialize the Swagger middleware
5  swaggerTools.initializeMiddleware(swaggerDoc, function(middleware) {
6
7    // ...
8
9    setupDataLayer().then(() => {
10      // Start the server
11      http.createServer(app).listen(serverPort, function() {
12        console.log(
13          "Your server is listening on port %d (http://localhost:%d)",
14          serverPort,
15          serverPort
16        );
17        console.log(
18          "Swagger-ui is available on http://localhost:%d/docs",
19          serverPort
20        );
21      });
22    });
23  });

```

5.3 Demo D4.2 Launch the server

```
1 DATABASE_URL=localhost node index.js
```

Insert some row in the database with PG-Commander and reload data