Lecture notes

Vittorio Zaccaria

January 27, 2020

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 $\rightarrow$ browser
  - Service layer $\rightarrow$ 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

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:

```yaml
swagger: '2.0'
info:
  description: >-
    This is a simple bookstore server with a book inventory, users and a shopping cart.
  version: 1.0.0
  title: Simple Bookstore
  contact:
    email: vittorio.zaccaria@polimi.it
  license:
    name: Apache-2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
host: none.yet.io
basePath: /v2
tags:
  - name: book
    description: Available book
  - name: cart
    description: Access to the cart
  - name: user
    description: Operations about user
schemes:
  - http
paths:
  /books:
    get:
      summary: Books available in the inventory
      tags:
        - book
      description: 'List of books available in the inventory'
      produces:
        - application/json
      parameters:
        - name: offset
          in: query
          description: Pagination offset. Default is 0.
          type: integer
        - name: limit
          in: query
          description: >-
            Maximum number of items per page. Default is 20 and cannot exceed
            500.
          type: integer
      responses:
        '200':
          description: A collection of Books
          schema:
            type: array
            items:
              $ref: '#/definitions/Book'
        '404':
          description: Unexpected error
  /books/{bookId}:
    get:
      summary: Find book by ID
      tags:
        - book
      description: Returns a book
      operationId: getBookById
      produces:
        - application/json
      parameters:
        - name: bookId
          in: path
          description: ID of book to return
          required: true
          type: integer
          format: int64
      responses:
        '200':
          description: successful operation
          schema:
            $ref: '#/definitions/Book'
        '400':
          description: Invalid ID supplied
        '404':
          description: Book not found
definitions:
  Book:
    title: Book
    description: A book for sale in the store
    type: object
    required:
      - title
      - author
      - price
    properties:
      id:
        type: integer
        format: int64
      title:
        type: string
        example: Il deserto dei tartari
      author:
        type: string
        example: Dino Buzzati
      price:
        $ref: '#/definitions/Amount'
      status:
        type: string
        description: book availability in the inventory
        enum:
          - available
          - out of stock
  Amount:
    type: object
    description: >
       Price
    properties:
      value:
        format: double
        type: number
        minimum: 0.01
        maximum: 1000000000000000
      currency:
        $ref: '#/definitions/Currency'
    required:
      - value
      - currency
  Currency:
    type: string
    pattern: '^[A-Z]{3,3}$'
    description: >
      some description
    example: eur
externalDocs:
  description: Find out more about Swagger
  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`

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

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

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
    <meta name="viewport" content="width=device-width" />

    <title>Book store</title>

    <link rel="stylesheet" href="style.css" />
    <!--[if lt IE 9]>
      <script src="//html5shiv.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
  </head>

  <body>
    <h1>Our first server is running!</h1>
    <ul></ul>
  </body>
  <script>
    var myList = document.querySelector("ul");
    fetch("v2/books")
      .then(function(response) {
        if (!response.ok) {
          throw new Error("HTTP error, status = " + response.status);
        }
        return response.json();
      })
      .then(function(json) {
        for (var i = 0; i < json.length; i++) {
          var listItem = document.createElement("li");
          let { title, author, price } = json[i];
          listItem.innerHTML = `${title} - ${author} - ${price.value} (${
            price.currency
          })`;
          myList.appendChild(listItem);
        }
      });
  </script>
</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

```yaml
/user/login:
  post:
    tags:
      - user
    summary: Login
    description: Login with a form
    consumes:
      - application/x-www-form-urlencoded
    produces:
      - application/json
    parameters:
      - name: username
        in: formData
        required: true
        type: string
      - name: password
        in: formData
        required: true
        type: string
    responses:
      '200':
        description: succesfull login
      '404':
        description: unauthorized

/cart/{cartId}:
    get:
      tags:
        - cart
      summary: View the content of the cart
      produces:
        - application/json
      parameters:
        - name: cartId
          in: path
          required: true
          type: integer
          format: int64
      responses:
        '200':
          description: succesful operartion
          schema:
            $ref: '#/definitions/Cart'
        '404':
          description: unauthorized

definitions:
  User:
    title: User
    description: A user
    type: object
    properties:
      id:
        type: integer
      name:
        type: string
      address:
        type: string
      creditcard:
        type: string
    example:
      id: 1
      name: Vittorio
      address: DEIB
      creditcard: xyzabc

  Cart:
    title: Cart
    description: Order for books
    type:  object
    properties:
        total:
          $ref: '#/definitions/Amount'
        books:
          type: array
          items:
            $ref: '#/definitions/Book'
```

4.2 *Demo D3.1 - Add cookie-session*

- Change `controllers/User.js`

```
module.exports.userLoginPOST = function userLoginPOST(req, res, next) {
  var username = req.swagger.params["username"].value;
  var password = req.swagger.params["password"].value;
  if(!req.session.loggedin) {
      req.session.loggedin = true;
  } else {
     req.session.loggedin = !req.session.loggedin;
  }
  User.userLoginPOST(username, password)
    .then(function(response) {
      utils.writeJson(res, response);
    })
    .catch(function(response) {
      utils.writeJson(res, response);
    });
};
```

- Change `controllers/Cart.js`

```
module.exports.cartCartIdGET = function cartCartIdGET(req, res, next) {
  var cartId = req.swagger.params["cartId"].value;
  if (!req.session || !req.session.loggedin) {
    utils.writeJson(res, { error: "sorry, you must be authorized" }, 404);
  } else {
    Cart.cartCartIdGET(cartId)
      .then(function(response) {
        utils.writeJson(res, response);
      })
      .catch(function(response) {
        utils.writeJson(res, response);
      });
  }
};
```

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

```
npm install knex -SE
npm install pg
```

5.2 Demo D4.1 Setup data layer

- `service/BookService.js` module

```
  let sqlDb;

  exports.booksDbSetup = function(s) {
    sqlDb = s;
    console.log("Checking if books table exists");
    return sqlDb.schema.hasTable("books").then(exists => {
      if (!exists) {
        console.log("It doesn't so we create it");
        return sqlDb.schema.createTable("books", table => {
          table.increments();
          table.text("title");
          table.text("author");
          table.float("value");
          table.text("currency");
          table.enum("status", ["available", "out of stock"]);
        });
      } else {
        console.log("It exists.");
      }
    });
  };

exports.booksGET = function(offset, limit) {
  return sqlDb("books")
    .limit(limit)
    .offset(offset)
    .then(data => {
      return data.map(e => {
        e.price = { value: e.value, currency: e.currency };
        return e;
      });
    });
};
```

- `service/DataLayer.js` module

```
let { booksDbSetup } = require("./BookService");

const sqlDbFactory = require("knex");
let sqlDb = sqlDbFactory({
  debug: true,
  client: "pg",
  connection: process.env.DATABASE_URL,
  ssl: true
});

function setupDataLayer() {
  console.log("Setting up Data Layer");
  return booksDbSetup(sqlDb);
}

module.exports = { database: sqlDb, setupDataLayer };
```

- `index.js`

```
let { setupDataLayer } = require("./service/DataLayer");


// Initialize the Swagger middleware
swaggerTools.initializeMiddleware(swaggerDoc, function(middleware) {

  // ...

  setupDataLayer().then(() => {
    // Start the server
    http.createServer(app).listen(serverPort, function() {
      console.log(
        "Your server is listening on port %d (http://localhost:%d)",
        serverPort,
        serverPort
      );
      console.log(
        "Swagger-ui is available on http://localhost:%d/docs",
        serverPort
      );
    });
  });
});
```

5.3 *Demo D4.2 Launch the server*

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

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