Universidad Autónoma de Madrid

Departamento de Informática

Computer Systems Project
Assignment 2.2

Roberto Marabini
Alejandro Bellogín

Changelog

Version[1]	Date	Author	Description
1.0	10.10.2022	RM	First version.
1.1	17.11.2022	RM	Code formatting.
1.2	5.12.2022	RM	Renumbering assignment 3 -> 2
1.3	12.12.2022	AB	Translation to English
1.4	29.1.2023	RM	Review before upload document to *Moodle*
1.5	22.2.2023	RM	Added paragraph regarding how to use enviromental variables in vue.js

[1] Version control is made using 2 numbers $X.Y$. Changes in Y denote clarifications, more detailed descriptions of some aspect, or translations. Changes in X denote deeper modifications that either change the provided material or the content of the assignment.

Contents

1 Goal

Next, we will describe how to extend the application designed in the first part of the assignment so that it reads and writes data about people. In a first version, we will use an API we have created for this assignment (`https://my-json-server.typicode.com/rmarabini/people/personas`), and in the final version you will implement your own API using *Django*.

2 Communication with the API

As it was mentioned in the previous section, we will use the API defined in `https://jsonplaceholder.typicode.com/`, that accepts reading requests (GET), creation requests (POST), update requests (PUT), and data removal requests (DELETE). The API returns data in JSON format. We must create methods that communicate with the API, which are in charge of sending requests while managing the produced response.

It is worth mentioning that the requests we send will not modify the list of people in the server database, since this is an example API with no writting capabilities; this inconvenience will be fixed in the last part of the assessment, where we will use *Django* to create our own API.

Next, we will create the asynchronous methods that will connect with the API. To simplify, we will use the Fetch API that incorporates JavaScript natively. The asynchronous methods that we will create will use the async/await commands and will have the following structure:

```
async asyncMethod() {
  try {
    // Get data using await
    const response = await fetch('url');

    // Response in JSON format
    const data = await response.json();
```

```
    // Here we process data
  } catch (error) {
    // In case of error
  }
}
```

Commands `fetch` and `response` are non-blocking, this means that they allow the application to continue running while it connects with the server. The sentence `await` prevents the next line of the function to be executed until functions `fetch` or `response` had finished. You may only use `await` inside functions created with `async`.

Let us now add the methods in the component `App.vue`, more specifically in the object `methods`. Moreover, we will also add the method `mounted`, which will be called when the component is mounted, which in this case happens when the application loads:

```
//src/App.vue
export default {
  name: 'app',
  components: {
    TablaPersonas,
    FormularioPersona,
  },
  data() {
    return {
      personas: [],
    },
    methods: {
        listadoPersonas(){
        // Metodo para obtener un listado de personas
        }
        agregarPersona(persona) {
```

```
        // Metodo para agregar una persona
        },
        eliminarPersona(id) {
        // Metodo para eliminar una persona
        },
        actualizarPersona(id, personaActualizada) {
        // Metodo para actualizar una persona
        },
    },
    mounted() {
    this.listadoPersonas();
    },
}
```

Please note we have removed the array variable assigned to `personas`, as this information will now be collected through the API. In the same way, we have added a method `listadoPersonas` and removed the content of methods `agregarPersona`, `eliminarPersona`, and `actualizarPersona`, since these operations will now use the API. Next, we shall see in detail the code of each method.

2.0.1 GET

This is the method we shall use to obtain people, it will be executed whenever the following component is mounted:

```
//src/App.vue
async listadoPersonas() {
  try {
    const response = await fetch('https://my-json-server.typicode.com/
       ↪ rmarabini/people/personas/');
    this.personas = await response.json();
  } catch (error) {
    console.error(error);
  }
```

```
}
```

2.0.2 POST

This is the method we shall use to create people, by sending data of the person in the request body. As you may observe, we use the method `JSON.stringify` to transform the people array to a `JSON` format. We also use the spread operator "..." to merge the people array with the inserted object:

```
//src/App.vue
async agregarPersona(persona) {
  try {
    const response = await fetch('https://my-json-server.typicode.com/
       ↪ rmarabini/people/personas/', {
      method: 'POST',
      body: JSON.stringify(persona),
      headers: { 'Content-type': 'application/json; charset=UTF-8' },
    });

    const personaCreada = await response.json();
    this.personas = [...this.personas, personaCreada];
  } catch (error) {
    console.error(error);
  }
}
```

2.0.3 PUT

We now create the method we will use to update a person. We send the id of the person we want to update in the URL, following, in this way, the standard `REST`. We also send user data updated in the request body.

```
//src/App.vue
```

```
async actualizarPersona(id, personaActualizada) {
    try {
        const response = await fetch('https://my-json-server.typicode.
            ↪ com/rmarabini/people/personas/'+personaActualizada.id+'/
            ↪ ', {
            method: 'PUT',
            body: JSON.stringify(personaActualizada),
              headers: { 'Content-type': 'application/json; charset=UTF
                  ↪ -8' },
            });

        const personaActualizadaJS = await response.json();
        this.personas = this.personas.map(u => (u.id ===
            ↪ personaActualizada.id ? personaActualizadaJS : u));
    } catch (error) {
    console.error(error);
    }
},
```

2.0.4 DELETE

Finally, we create the method that will remove users:

```
//src/App.vue
async eliminarPersona(persona_id) {
  try {
    await fetch('https://my-json-server.typicode.com/rmarabini/people/
        ↪ personas/'+persona_id+'/', {
      method: "DELETE"
    });

    this.personas= this.personas.filter(u => u.id !== persona_id);
  } catch (error) {
```

```
    console.error(error);
  }
}
```

With this, we would have included all the necessary methods. Observe that, to simplify the application, we only collect the listing of people when the page is loaded, if other users existed in the system that would modify the database while we are working, that information would not be updated in our local copy of the `personas` array until the page is loaded again and function `listadoPersonas` is executed.

3 Application components

Component `TablaUsuarios` that we created in the first part of the assessment is flexible enough to be reused without further changes. Now you may test and run the application in your browser. You should see in your screen a table with the users we have gathered through the API:

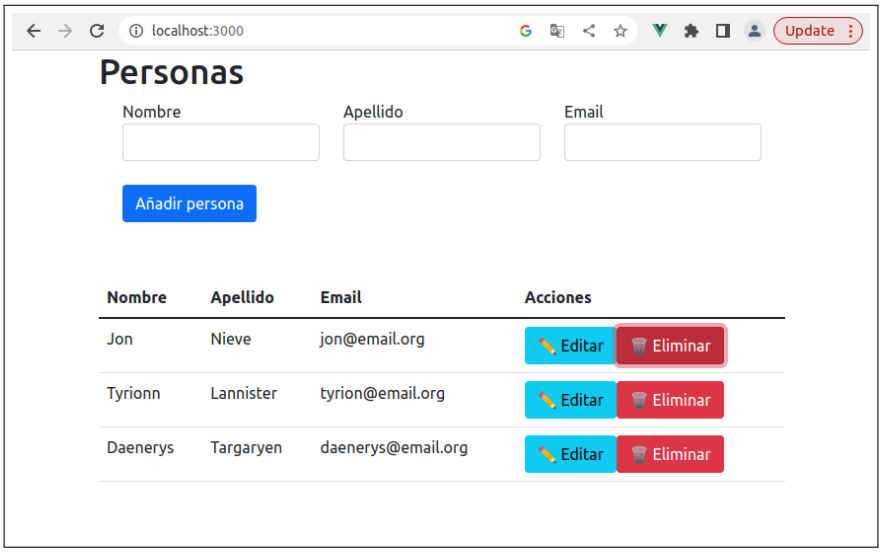

Figure 1: Web page showing a listing with people. Note that the email has changed with respect to the previous version (domain `com` now is `org`) *Vue.js*.

Our application should keep on working normally even though our "fake" API is not going to update or add new people, so every time the page is loaded that information will get lost. We will fix this issue shortly.

4 Building and Deploying the application

Do not forget to deploy the application in *render.com*. If everything was performed correctly, as soon as you upload the code to github the application should be deployed. Our experience with the automatic deployment is not good, and it is often convenient to clear the cache explictly in *render.com* before deployment.

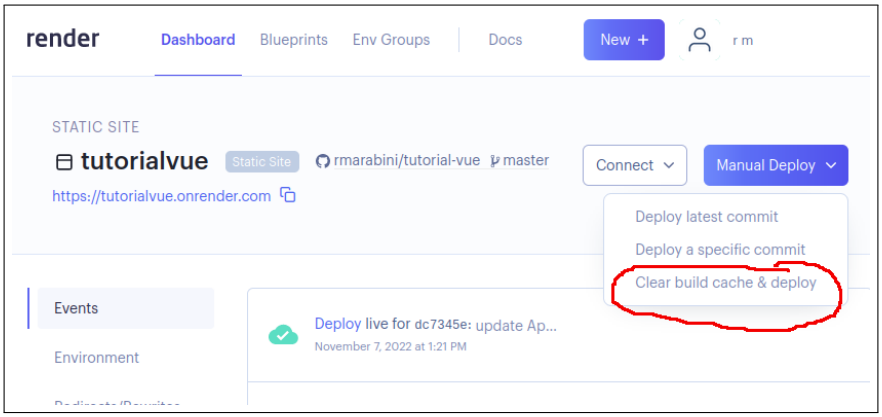

Figure 2: Redeploy application in *render.com* cleaning the cache) *Vue.js*.

5 Creating a REST API using *Django*

Let us create our API using *Django*. We will start by creating a virtual environment of python3 where we will install the same modules we used in the previous assignment. In particular, check the following two modules exist in the `requirements.txt` file, otherwise, add them there:

```
django-cors-headers==3.2.1
djangorestframework==3.13
djangorestframework-simplejwt==4
```

Create a *Django* project called `persona` with an application called `api`. Do not forget to add the applications `api` and `rest_framework` to the `INSTALLED_APPS` variable in `persona/settings.py`.

Our application will use two different servers, one for *Django* and another for *Vue.js*. They will run in different ports and work as two separate domains. We must explictly allow cross-origin resource sharing (CORS) to send HTTP requests from *Vue.js* to *Django* since, for security purposes, this type of requests are disabled by default. `django-cors-headers` is the module that manages how this type of access is (dis)allowed. It is installed as if it was an extra application, so you must add it to the `INSTALLED_APP` variable as `'corsheaders'` and, similarly, you must add it to the `MIDDLEWARE` variable as `'corsheaders.middleware.CorsMiddleware'` at the beginning of such list. Finally, you must create a list with the domains from where you may access the *Django* server (all these changes will be made in file `persona/settings.py`).

```
CORS_ORIGIN_ALLOW_ALL = False
CORS_ORIGIN_WHITELIST = [
    'http://localhost:3000',
]
```

Next, we will create a model for the people in `api/models.py`, whose model will be provided by the following relational schema:

```
persona(id, nombre, apellido, email)
```

Using the `meta` class, make sure all the results are sorted by `id` in an increasing way (lower `id` are shown first). Migrate the changes (`makemigrations`, `migrate`). Now, let us configure a serializer, this will define the content of the `personas` as they will be returned by the API (file `api/serializers.py`):

```
# api/serializers.py}
from .models import Persona
from rest_framework import serializers

class PersonaSerializer(serializers.ModelSerializer):
```

```
  class Meta:
    model = Persona
    # fields = ['id', 'nombre', 'apellido', 'email']
    fields = '__all__'
```

let us now add a view that returns a listing with people

```
#api/views.py

from .models import Persona
from .serializers import PersonaSerializer
from rest_framework import viewsets

class PersonaViewSet(viewsets.ModelViewSet):
  queryset = Persona.objects.all()
  serializer_class = PersonaSerializer
```

And we now add the `urls` to the path of the `viewset` in the API:

```
#persona/urls.py

from django.contrib import admin
from django.urls import path, include

from api import views
from rest_framework import routers

router = routers.DefaultRouter()

# En el router vamos agnadiendo los endpoints a los viewsets
router.register('personas', views.PersonaViewSet)

urlpatterns = [
  path('api/v1/', include(router.urls)),
```

```
  path('admin/', admin.site.urls),
]
```

By default, the viewsets have public permissions, in this way anyone could manage the movies. If you wish to disallow the public access to the API,you may set a default permission in `settings.py`. For example:

```
# persona/settings.py
# in your project you do NOT need to add these lines to settings

REST_FRAMEWORK = {
  'DEFAULT_PERMISSION_CLASSES': [
    'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly',
  ],
}
```

transforms the API in read-only for non-authenticated visitors. Only identified users will be able to access creation, update, and delete actions. In the meantime, leave the API with free access.

Finally, start the *Django* server to check the API works. It will be accessible in the URL `http://localhost:8001/api/v1`.

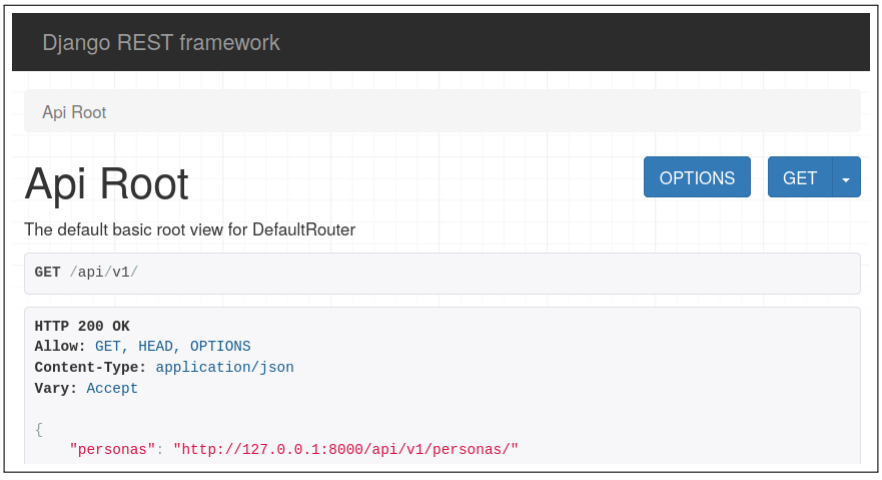

Figure 3: Interface to the API `api` built using *Django*.

6 Consuming the API

The only remaining thing to do is consuming the API `api` from your *Vue.js* application; for this, you will need to change the URL from `https://my-json-server.typicode.com/rmarabini/people/personas` to `http://127.0.0.1:8000/api/v1/personas`. Remember you need to start both servers, *Django* and *Vue.js*.

7 Server deployment in *render.com*

Do not forget to deploy in *render.com* both the updated `persona` project and a second project containing the *Django* REST server. Note that the URL to which the `fetch` command should connect is different if you run the project locally or in *render.com*. In order to create a more robust code this URL should be stored in an environment variable. The use of environment variables in *Vue.js* is described at `https://vitejs.dev/guide/env-and-mode.html`. To summarize, you need to create two files called `.env.development` and `.env.production` (as you can see they are preceded by a dot). When the server is running in development mode (i.e., it is started with the command `npm run dev`) the environment variables of .env.development will be read, otherwise the environment variables of .env.production will be read. Be aware that the variables must start with the preffix `VITE` e.g:

```
.env.development:
    VITE_DJANGOURL=http://127.0.0.1:8001
.env.production:
    VITE_DJANGOURL=https://kahoorclone.onrender.com
```

Finally, the value of these variables can be accessed using the command

```
const myVar = import.meta.env.VITE_DJANGOURL;
```