Universidad Autónoma de Madrid

Departamento de Informática

Proyecto de Sistemas Informáticos
Práctica - 2.2

Roberto Marabini
Alejandro Bellogín

Registro de Cambios

Versión[1]	**Fecha**	**Autor**	**Descripción**
1.0	10.10.2022	RM	Primera versión.
1.1	17.11.2022	RM	Reformateo del código.
1.2	5.12.2022	RM	Renumerar práctica 3 ->2
1.3	12.12.2022	AB	Traducción al inglés
1.4	29.1.2023	RM	Revisión previa a subir el documento a *Moodle*
1.5	22.2.2023	RM	Añadido un párrafo explicando como se gestionan las variables de entorno en vue.js

[1] La asignación de versiones se realizan mediante 2 números $X.Y$. Cambios en Y indican aclaraciones, descripciones más detalladas de algún punto o traducciones. Cambios en X indican modificaciones más profundas que o bien varían el material suministrado o el contenido de la práctica.

Índice

1. Objetivo

A continuación se describe como ampliar la aplicación diseñada en la primera parte de la práctica para que lea y escriba los datos sobre las personas. En una primera versión usaremos un API que hemos creado para esta práctica (`https://my-json-server.typicode.com/rmarabini/people/personas`) y en la versión final implementaréis el API usando *Django*.

2. Comunicación con la API

Como se comentó en la sección previa usaremos el API existente en `https://jsonplaceholder.typicode.com/`, que acepta peticiones de lectura (GET), creación (POST), actualización (PUT) y borrado de datos (DELETE). La API devuelve datos en formato JSON. Debemos crear los métodos que se comuniquen con la API, encargados de enviar las peticiones a la misma y también de gestionar la respuesta obtenida.

Conviene comentar que las peticiones que enviemos no modificarán la lista de personas en la base de datos del servidor, ya que se trata de una API de ejemplo sin posibilidades de escritura, este inconveniente se solventará en la última parte de la práctica en la que usaremos *Django* para crear nuestra propia API.

A continuación vamos a crear los métodos asíncronos que se conectarán con la API. Por simplificar, usaremos la API Fetch que incorpora JavaScript de forma nativa. Los métodos asíncronos que crearemos usarán las sentencias async/await y tendrán esta estructura:

```
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
  }
}
```

Los comandos `fetch` y `response` son no bloqueantes, esto es permiten que la aplicación siga ejecutandose mientras contacta con el servidor. La sentencia `await` impide que se ejecute la línea siguiente de la función hasta que las funciones `fetch` o `response` no hayan terminado. Solo puedes usar `await` dentro de funciones creadas con `async`.

Ahora agregaremos los métodos en el componente `App.vue`, concretamente en el objeto `methods` del mismo. Además, también agregaremos el método `mounted`, que se ejecutará cuando se monte el componente, que en este caso ocurre al cargar la aplicación:

```
//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();
    },
}
```

Nótese que hemos suprimido el array que asignábamos a la variable `personas` puesto que esta información la vamos a conseguir a través del API. Igualmente hemos añadido el método `listadoPersonas` y borrado el contenido de los métodos `agregarPersona`, `eliminarPersona` y `actualizarPersona` puesto que se accederá al API para realizar estas operaciones. A continuación vamos a ver en detalle el código de cada método.

2.0.1. GET

Este es el método que usaremos para obtener personas, que se ejecutará cuando se monte el componente:

```
//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

Este es el método que usaremos para crear personas, enviando los datos de la persona en el cuerpo de la petición. Tal y como ves, usamos el método `JSON.stringify` para transformar el array de personas a formato `JSON`. También usamos el operador spread "..." de propagación para unir el array de personas con el objeto que hemos insertado:

```
//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

Ahora crearemos el método utilizado para actualizar una persona. Enviamos el id de la persona que queremos actualizar en la URL, siguiendo así el estándar `REST`. Enviamos también los datos del usuario actualizado en el cuerpo de la petición.

```
//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

Finalmente crearemos el método encargado de eliminar usuarios:

```
//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);
  }
}
```

Y con esto, ya habríamos agregado todos los métodos necesarios. Nótese que para simplificar la aplicación sólo nos traemos el listado de personas al cargar la página, si existieran otros usuarios del sistema que modificaran la base de datos mientras estamos trabajando esa información no se actualizaría en nuestra copia local del array de `personas` hasta que la página no se vuelva a cargar y se ejecute la función `listadoPersonas`.

3. Componentes de la aplicación

El componente `TablaUsuarios` que creamos en la primera parte de la práctica es suficientemente flexible como para ser reutilizado sin necesidad de cambios. Ahora ya puedes probar a ejecutar la aplicación en tu navegador. Deberías ver por pantalla una tabla con los usuarios que hemos obtenido desde la API:

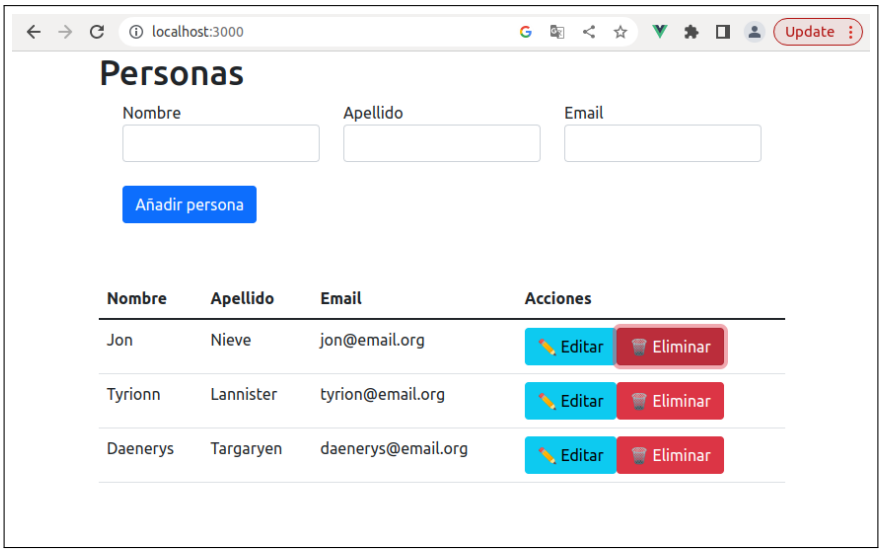

Figura 1: Página web mostrando un listado de personas. Nótese que el email ha cambiado con respecto a la versión anterior (el dominio `com` pasa a ser `org`) *Vue.js*.

La aplicación debería seguir funcionando con normalidad aunque nuestro "fake" API no va a actualizar o agregar nuevas personas así que cada vez que se recarge la página esa información se pierde. En breve resolveremos ese problema.

4. Build y Deploy de la aplicación

No os olvidéis de desplegar la aplicación en *render.com*. Si todo se ha realizado correctamente tan pronto como subáis el código a github la aplicación debe desplegarse. Nuestra experiencia con el despliege automático no es buena y muchas veces es conveniente limpiar el cache explícitamente en *render.com* antes de desplegar.

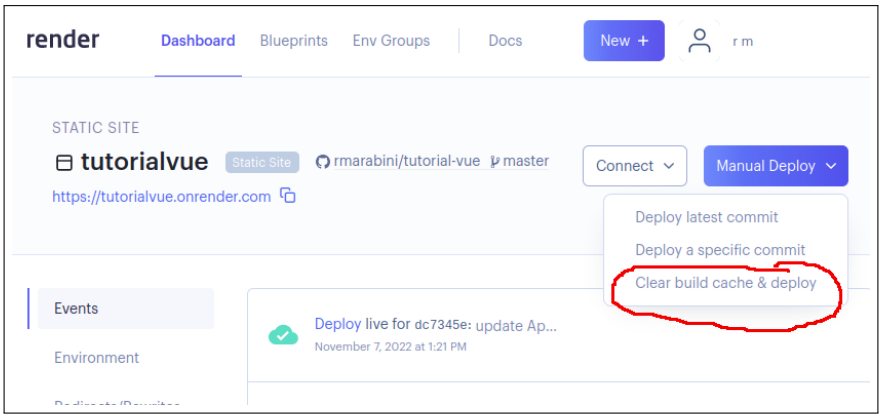

Figura 2: Re-despliegue de la aplicación en *render.com* limpiando la cache) *Vue.js*.

5. Creación de un API REST usando *Django*

Vamos a crear nuestra API usando *Django*. Comenzaremos creando un entorno virtual de python3 en el que instalaremos los mismos módulos usados en la práctica anterior. En particular comprueba que estos dos módulos están en el fichero `requirements.txt`, en caso contrario añádelos:

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

Crea un proyecto de *Django* llamado `persona` con una aplicación llamada `api`. No olvides añadir las aplicaciones `api` y `rest_framework` a la variable `INSTALLED_APPS` en `persona/settings.py`.

Nuestra aplicación utilizará dos servidores distintos, uno para *Django* y otro para *Vue.js*. Se ejecutarán en diferentes puertos y funcionarán como dos dominios separados. Debemos permitir explícitamente el intercambio de recursos de origen cruzado (CORS) para enviar solicitudes HTTP desde *Vue.js* a *Django* pues, por motivos de seguridad, este tipo de peticiones está desabilitado por defecto. `django-cors-headers` es el modulo que se encarga de permitir este tipo de accesos. Se instala como si fuera una aplicación extra por lo que debes añadirlo a la variable `INSTALLED_APP` co-

mo `'corsheaders'` e igualmente tienes que añadirlo a la variable `MIDDLEWARE` como `'corsheaders.middleware.CorsMiddleware'` al principio de dicha lista. Finalmente debes crear una lista de los dominios desde los cuales se puede acceder al servidor de *Django* (todos estos cambios se realizarán en el fichero `persona/settings.py`).

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

Seguidamente creemos un modelo para las personas, en `api/models.py`, cuyo modelo venga dado por el esquema relacional siguiente:

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

Usando la clase `meta` asegúrate de que los resultados estén ordenados por el `id` de forma creciente (`id` menores en primer lugar). Migra los cambios (`makemigrations`, `migrate`). Ahora vamos a configurar un serializador, éste definirá el contenido de las `personas` tal como las devolverá la API (fichero `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__'
```

añadimos una vista que devuelva el listado de personas

```
#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
```

Y ahora añadimos a las `urls` la ruta de la `viewset` en la 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),
]
```

Por defecto las viewsets tienen permisos públicos, así que cualquiera podría manejar las películas. Si deseáis impedir el acceso público al API podéis poner un permiso por defecto en `settings.py`. Por ejemplo:

```
# 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',
  ],
}
```

convierte el API en de sólo lectura para visitantes no autenticados. Sólo los usuarios identificados podrán acceder a las acciones de creación, modificación y borrado. Por el momento dejad el API con acceso libre.

Finalmente, arrancad el servidor *Django* para comprobar que el API funciona. El API estará accesible en la dirección `http://localhost:8001/api/v1`.

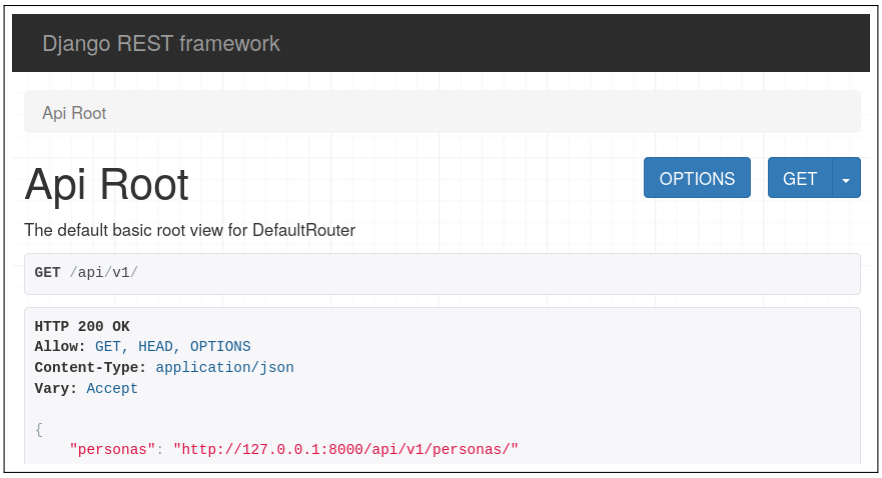

Figura 3: interfaz al API `api` construida usando *Django*.

6. Consumir el API

Ya sólo queda consumir el API `api` desde vuestra aplicación *Vue.js* para lo cual tendréis que cambiar el URL de `https://my-json-server.typicode.com/rmarabini/people/personas` a `http://127.0.0.1:8000/api/v1/personas`. Recordad que tenéis que arrancar ambos servidores el de *Django* y el de *Vue.js*.

7. Despliegue de los servidores en *render.com*

No os olvidéis de desplegar en *render.com* tanto el proyecto `persona` como un segundo proyecto que contenga el servidor REST que acabáis de crear usando *Django*. Para crear un código más robusto la dirección a la que debe conectarse el comando fetch debería estar guardada en una variable de entorno siendo distinta cuando la aplicación se ejecuta localmente o en *render.com*. El uso de variables de entorno en *Vue.js* está descrito en `https://vitejs.dev/guide/env-and-mode.html`. En resumen, tenéis que crear dos ficheros llamados `.env.development` y `.env.production` (como veis van precedidos por un punto) cuando el servidor funcione en modo de desarrollo (esto es se arranque con el comando `npm run dev`) se leerán las variables de entorno de `.env.development`, en caso contrario se leen las variables de entorno de `.env.production`. Nótese que las variables deben empezar con el prefijo VITE por ejemplo:

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

Finalmente, el valor de estas variables se puede optener con el comando:

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