Universidad Autónoma de Madrid

Departamento de Informática

Proyecto de Sistemas Informáticos
Práctica - 2.2

Roberto Marabini
Alejandro Bellogín
Jose A. Macías
Alejandro Moya

Registro de Cambios

Versión[1]	**Fecha**	**Autor**	**Descripción**
3.0	13.01.2025	AB	Actualización sintaxis a Composition API, script setup.
3.1	17.02.2025	JAMI	Actualización de librerías Vue.js y otros aspectos sobre Neon.
4.0	01.09.2025	AB	Adaptación para el curso 2025-26
4.1	30.11.2025	RM	Revisado por RM
4.2	09.02.2026	AM	Modificación del texto con aclaraciones menores

[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 cómo 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 un API ya existente, que acepta peticiones de lectura (GET), creación (POST), actualización (PUT) y borrado de datos (DELETE), aunque los cambios no son persistentes entre llamadas. 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:

Listado 1:

```
const asyncMethod => async () => {
  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` no son bloqueantes, esto es, permiten que la aplicación siga ejecutándose 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`. Además, también agregaremos el método `mounted`, que se ejecutará cuando se monte el componente, que en este caso ocurre al cargar la aplicación:

Listado 2:

```
import { ref, onMounted } from 'vue';

defineOptions({
  name: 'app',
});

const personas = ref([]);

const listadoPersonas = async () => {
  // Metodo para obtener un listado de personas
};

const agregarPersona = async (persona) => {
  // Metodo para agregar una persona
```

```
};

const eliminarPersona = async (persona_id) => {
  // Metodo para eliminar una persona
};

const actualizarPersona = async (id, personaActualizada) => {
  // Metodo para actualizar una persona
};

// Fetch data when the component is mounted
onMounted(() => {
  listadoPersonas();
});
```

Código fuente: 

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.

2.1. GET

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

Listado 3:

```
  // Metodo para obtener un listado de personas
  try {
    const response = await fetch('https://my-json-server.typicode.com/
       ↪ rmarabini/people/personas/');
```

```
    personas.value = await response.json();
  } catch (error) {
    console.error(error);
  }
};
```

2.2. 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 objeto persona a formato `JSON`. También usamos el operador spread "`...`" de propagación para unir el array de personas con el objeto que hemos insertado:

Listado 4:

```
  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();
    personas.value = [...personas.value, personaCreada];
  } catch (error) {
    console.error(error);
  }
};
```

2.3. 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.

Listado 5:

```
// Metodo para actualizar una persona
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();
    personas.value = personas.value.map(u => (u.id ===
       ↪ personaActualizada.id ? personaActualizadaJS : u));
} catch (error) {
  console.error(error);
}
};
```

2.4. DELETE

Finalmente crearemos el método encargado de eliminar usuarios:

Listado 6:

```
// Metodo para eliminar una persona
try {
```

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

    personas.value= personas.value.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`.

Código fuente: 

3. Componentes de la aplicación

El componente `TablaPersonas` 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:

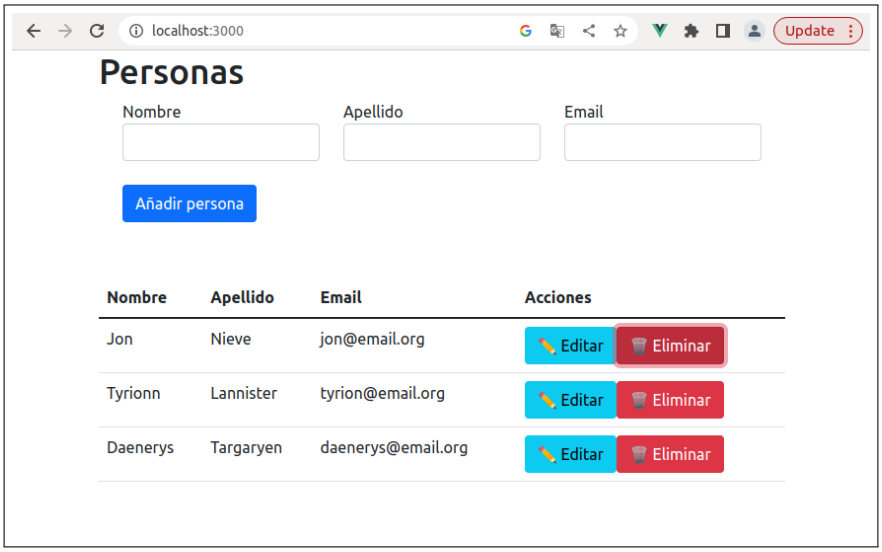

Figura 1: Página web mostrando un listado de personas. Nótese que el email ha cambiado con respecto a la versión anterior (el dominio `com` pasa a ser `org`) – *Vue.js*.

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 caché explícitamente en *Render.com* antes de desplegar.

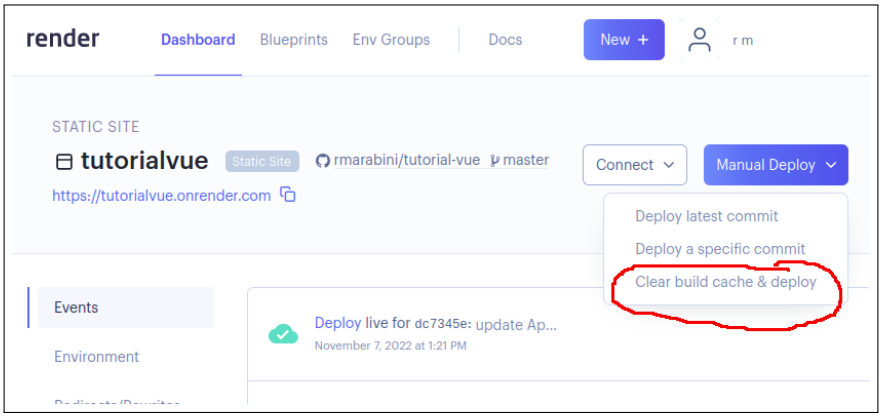

Figura 2: Re-despliegue de la aplicación en *Render.com* limpiando la cache – *Vue.js*.

5. Testing de la versión dinámica usando API de solo lectura

Antes de pasar a la fase del API de Django, ejecuta y comprueba que funcionan correctamente los mismos tests que en el apartado anterior (ya que, aunque la aplicación Vue es dinámica, los cambios no son persistentes), es decir, los que empiezan por `static`.

6. Creación de un API REST usando *Django*

Vamos a crear nuestra API usando *Django*. Comenzaremos creando un entorno virtual de python3.11 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==4.3.1
djangorestframework==3.14.0
```

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`. Recuerda que debes configurar el proyecto con una base de datos de PostgreSQL.

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 módulo 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` como `'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:5173',
]
```

Seguidamente creamos 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 los registros de las personas. 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 a 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`.

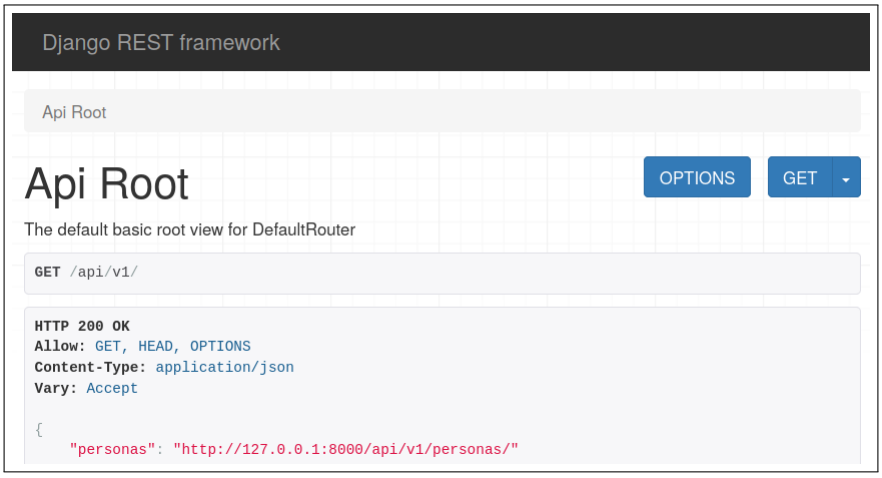

Figura 3: Interfaz al API `api` construida usando *Django*. Esta página está disponible en http://host/api/v1:port (donde "host" será el ordenador donde se ejecuta *Django*) y "port" el puerto donde se está ejecutando *Django*.

7. Consumir el API

El siguiente paso es consumir el API desde vuestra aplicación *Vue.js*, para lo cual tendréis que cambiar el URL de `https://my-json-server.typicode.com/rmarabini/people/personas` al URL donde esté desplegado *Django* (e.g. `http://localhost:8001/api/v1/personas`). Recordad que tenéis que arrancar ambos servidores, el de *Django* y el de *Vue.js*.

Cuando hayáis realizado este cambio con éxito, la aplicación seguirá funcionando correctamente, con la diferencia de que los cambios entre peticiones serán persistentes porque vuestra API en Django así lo permite.

8. Variables de sesión (*Vue.js*)

Las variables de sesión son un mecanismo ampliamente utilizado para dotar al protocolo HTTP de variables que persistan entre peticiones y así mantener el estado de una sesión. En esta praćtica vamos a incluir variables de sesión a partir del módulo

`pinia`[2].

Una vez instalado, debemos incluirlo en la aplicación. Para ello debemos importar el módulo, crear una variable `pinia` y declarar su uso en la aplicación antes de montarla:

Listado 7:

```
import { createApp } from 'vue'
import App from './App.vue'
import { createPinia } from 'pinia'

const myapp = createApp(App)
const pinia = createPinia()

myapp.use(pinia)
myapp.mount('#app')
```

Código fuente: 

En esta práctica vamos a incluir un contador del número de personas almacenadas. Para ello, creamos una carpeta `src/stores` en el directorio de la aplicación e incluimos el siguiente archivo `counter.js`.

Listado 8:

```
import { defineStore } from 'pinia';
import { ref, computed } from 'vue';

const COUNTER_LOCAL_STORAGE_KEY = 'localCounter';

const getCount = () => {
  const storedCounter = localStorage.getItem(
    ↪ COUNTER_LOCAL_STORAGE_KEY);
  return storedCounter ? JSON.parse(storedCounter) : 0;
};
```

[2]Si hiciera falta, se puede instalar usando la siguiente instrucción: `npm install pinia` .

```
export const useCounterStore = defineStore('counter', () => {
  const count = ref(getCount());

  const singleCount = computed(() => count.value);

  const increment = () => {
    count.value++;
    localStorage.setItem(COUNTER_LOCAL_STORAGE_KEY, JSON.stringify(
        ↪ count.value));
  };

  return {
    count,
    singleCount,
    increment
  };
});

if (localStorage.getItem('state')) {
  pinia.state.value = JSON.parse(localStorage.getItem('state'));
}
```

Código fuente: 

El siguiente paso es hacer accesible la variable de sesión ("store") a la aplicación, para ello hay que incluir varias directivas en `App.vue`.

```
<!-- App.vue -->
<script setup>
...
import { useCounterStore } from '@/stores/counter';

const store = useCounterStore();
```

```
...
const agregarPersona = async (persona) => {
    try {
  ...
      store.increment();
    } catch (error) {
      console.error(error);
    }
  };
   ...
</script>
```

Finalmente, agregamos en el cuerpo de la aplicación principal el código HTML para visualizar el contador de visitas:

```
<!-- App.vue ->>
<template>
...
 <div>
    <p>Count is {{ store.count }}</p>
  </div>
</template>
...
```

Ten en cuenta que, al ser una variable de sesión, este contador sólo considera las personas que se han insertado desde el navegador actual, no el número total de personas que hay en el sistema. Tampoco tiene en cuenta, porque no está implementado, aunque tendría sentido, decrementar el contador cada vez que se elimina una persona, ya que sólo se actualiza cuando se invoca el método relacionado con agregar una persona.

9. Navegación usando Vue Router

Las aplicaciones desarrolladas con *Vue.js* suelen seguir el paradigma de las *Single Page Applications* (SPA). En una SPA, la aplicación se carga inicialmente como una única página HTML y la navegación entre distintas vistas se realiza de forma dinámica, sin necesidad de recargar la página ni de realizar nuevas peticiones completas al servidor.

Este comportamiento se basa en la **reactividad** de *Vue.js*: cuando el usuario interactúa con un enlace de navegación, el contenido mostrado cambia automáticamente en función de la ruta activa, manteniéndose la misma página web subyacente. De este modo se consigue una experiencia de usuario más fluida y rápida que en aplicaciones web tradicionales.

Para gestionar esta navegación en aplicaciones *Vue.js* se utiliza un *router*. En concreto, `vue-router` es la biblioteca oficial de enrutamiento del framework y permite asociar distintas rutas (URLs) con componentes de la aplicación, de forma que cada ruta muestre una vista diferente.

Durante la creación del proyecto en la primera parte de la práctica, `vue-router` fue preinstalado automáticamente por la herramienta de creación del proyecto. Sin embargo, con el objetivo de simplificar el desarrollo inicial, la configuración de las rutas se dejó comentada y no se utilizó en las secciones anteriores.

En esta sección se retoma dicha configuración y se explica cómo activar y utilizar `vue-router` para implementar una aplicación sencilla con tres páginas, permitiendo la navegación entre ellas sin necesidad de comunicarse con el back-end.

9.1. Analizando los ficheros creados

Vamos a comentar las principales diferencias entre los ficheros creados por defecto cuando se añade y cuando no se añade el router.

`main.js`

Si recordáis, las aplicaciones *Vue.js* se instancian en el fichero `main.js`. Ahora podéis ver que, además de instanciarse la aplicación, se añade el módulo router

(`app.use(router)`). Esta parte la comentamos al principio de la práctica, ahora es el momento de descomentarlo para poder hacer uso de esta funcionalidad.

Listado 9: Contenido del fichero `main.js`, las líneas relacionadas con el router se marcan con una cadena de #.

```
//src/main.js
import { createApp } from 'vue'
import { createPinia } from 'pinia'

import App from './App.vue'
import router from './router' ################
import './assets/main.css'

const app = createApp(App)
app.user(createPinia())
app.use(router) ###############
app.mount('#app')
```

index.js

Este fichero es similar a `urls.py` de *Django* y contiene un "mapeo" entre URLs y ficheros `vuejs`. El que viene por defecto es bastante complejo, borrarlo y copiar el código siguiente:

```
//src/router/index.js
import { createRouter, createWebHistory } from 'vue-router'
import HomeView from '../views/HomeView.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
```

```
      name: 'home',
      component: HomeView
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('../views/AboutView.vue')
    },
    {
      path: '/faq',
      name: 'faq',
      component: () => import('../views/FaqView.vue')
    },
  ]
})

export default router
```

donde se crean tres mapeos para las URLs `/` (la única página realizada hasta ahora), `/about` y `/faq` (las dos nuevas componentes que se explican en la siguiente sección).

9.2. Creando las componentes

Todavía no hemos creado los componentes `about` y `faq`. Crea (o sobreescribe) dos ficheros llamados `AboutView.vue` y `FaqView.vue` en el directorio `views` con el contenido:

Listado 10: Template para crear los ficheros `AboutView.vue` y `FaqView.vue`. Cambia XXX por ABOUT o FAQ para distinguir ambas páginas (luego deberías incluir contenido real, pero por ahora, es suficiente con esto).

```
<template>
    <!-- cambia XXX por lo que corresponda -->
    <h1>Hi! I am page XXX! </h1>
```

```
</template>

<script>
    export default {}
</script>
```

Para hacer uso de las vistas recién definidas, hay que modificar el fichero que se carga por defecto (`App.vue`) e incluir en la aplicación que se carga por defecto `HomeView.vue` el contenido previo de nuestra aplicación de personas.

Listado 11: Contenido del fichero HomeView.vue (antiguo fichero App.vue).

```
<template>
  <div id="app" class="container">
    <div class="row">
      <div class="col-md-12">
        <h1>Personas</h1>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <formulario-persona @add-persona="agregarPersona" />
        <tabla-personas
          :personas="personas"
          @delete-persona="eliminarPersona"
          @actualizar-persona="actualizarPersona"
        />
      </div>
    </div>
  <p> Count is {{ store.count }}</p>
  </div>
</template>

<script>
  import TablaPersonas from '@/components/TablaPersonas.vue'
  import FormularioPersona from '@/components/FormularioPersona.vue'
  ...
```

Listado 12: **Nuevo** contenido del fichero App.vue. Los componentes asociados a cada URL se introducirán entre los tags `<router-view></router-view>`.

```
<template>
  <div>
      <h1>Usando Router</h1>
      <router-view></router-view>
  </div>
</template>

<script setup>
import { RouterLink, RouterView } from 'vue-router'
</script>
```

Ya puedes arrancar el servidor y conectarte a las URLs: `http://localhost:5173/`, `http://localhost:5173/about` y `http://localhost:5173/faq`.

9.3. Creando Links

Finalmente te puede hacer falta crear links para saltar de una página a otra. Si usas la syntaxis tradicional `<a href=‘‘...’’>link</a>` el navegador se conectará al servidor que **no es lo que deseas** puesto que lo que se busca es que el navegador gestione el paso de una página a otra sin que intervenga el servidor. En particular, el navegador va a estar siempre en la misma página pero va a ir **inyectando** diferentes componentes en cada parte.

Añade las siguientes dos líneas de código al fichero `App.vue`:

Listado 13: Añadiendo links a una página *Vue.js*.

```
<a class="button" ><router-link to="/">Click to access Home</
    ↪ router-link></a>

<a class="button" ><router-link to="/about">Click to access About<
    ↪ /router-link></a>

<a class="button" ><router-link to="/faq">Click to access FAQ</
    ↪ router-link></a>
```

Ahora aparecerán sendos links en la página principal que te permitirán acceder a los componentes `About` y `Faq`, junto con otro para volver a visualizar la página original con la tabla de personas.

10. Testing de la versión dinámica basada en API REST

Antes de pasar a la fase de despliegue, ejecuta y comprueba que funcionan correctamente los tests que asumen que los cambios son persistentes, es decir, aquellos que empiezan por *dynamic*.

11. Despliegue de los servidores en *Render.com*

No os olvidéis desplegar en *Render.com* tanto el proyecto `persona` como un segundo proyecto que contenga el servidor REST que acabáis de crear usando *Django*. Recuerda además que, para que la conexión con la base de datos siga funcionando, habrá que configurar una BD en *Neon* para asegurar la persistencia de los datos, realizando, para ello, los cambios y redirecciones correspondientes. Hay que tener en cuenta además la posibilidad de poblar esta BD en la nube.

Con el objetivo de tener un código más general y que necesite el mínimo número de cambios para su despliegue, se puede incluir estos comandos en `settings.py` que

utilizan una variable de entorno definida en Render:

```
RENDER_EXTERNAL_HOSTNAME = os.environ.get('RENDER_EXTERNAL_HOSTNAME')
if RENDER_EXTERNAL_HOSTNAME:
    ALLOWED_HOSTS.append(RENDER_EXTERNAL_HOSTNAME)
```

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://localhost:8001
.env.production:
    VITE_DJANGOURL=https://my-json-server.typicode.com
```

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

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

Recuerda lanzar los **tests de Cypress** preparados para la versión dinámica de la aplicación tal y como viste en el enunciado anterior.