



Universitat
Oberta
de Catalunya

M4.257 - Herramientas HTML y CSS

PEC 1. Desarrollo de una web

Aníbal Santos Gómez

ÍNDICE: DOCUMENTACIÓN.

- 1. Introducción y objetivos.**
- 2. Configuración del entorno de desarrollo.**
- 3. Diseño de la arquitectura front-end.**
- 4. Desarrollo.**
- 5. Publicación.**

1. Introducción y objetivos.

A. Introducción.

Para el desarrollo de una web moderna, necesitamos herramientas modernas que nos permitan fácilmente el desarrollo de la misma. El principal objetivo de este desarrollo es centrarnos en la construcción de una web y perder el mínimo tiempo posible en otras actividades.

Para ello se eligen un conglomerado de herramientas que nos facilitarán, el desarrollo en local, el pase a producción, el control de versiones, la maquetación, adaptación a diferentes navegadores incluyendo versiones antiguas, y otra serie de herramientas que se justifican continuación:

- Editor de código: **Visual Studio Code**.

Se utilizará **VS Code** como editor de código, ya que nos permite instalar plugins que nos permiten utilizar snippets para agilizar nuestra forma de programar. Además es totalmente personalizable en función del proyecto que se vaya a realizar, en cuestión de herramientas. Actualmente se han instalado plugins como auto close tag, rename tag, css formatter, highlight matching tag, JS snippets, Vetur, Vue VS Code extension, o Prettier.

- Control de versiones y repositorio: **Git y Github**.

Para el control de versiones utilizaremos **Git** y para almacenar nuestro repositorio en la nube utilizaremos el servicio que nos brinde **Github**.

- Gestor de paquetes: **Npm**.

Como gestor de paquetes de Node.js utilizaremos **Npm**, que contiene un sin fin de dependencias desarrolladas por otros programadores que nos facilitaran la configuración de los diversos paquetes que utilizaremos.

- Module bundler: **Parcel**.

Para empaquetar todo nuestro proyecto utilizaremos **Parcel** con una serie de plugins para poder configurar **Babel** con **Vue**.

- Preprocesadores de código: **Babel y PostCSS**

Utilizaremos **Babel** como polyfill para convertir nuestro código moderno en ES6 a código JavaScript que pueda ser interpretado por navegadores antiguos, y **PostCSS** junto con **Autoprefixer** para convertir nuestro CSS moderno en CSS interpretable por navegadores antiguos.

- Framework frontend: **VueJS**.

Aunque los requisitos de esta práctica no exigen la utilización de un framework front-end. Se elige Vue JS para agilizar y ordenar la estructura de datos que van a ser renderizados y así facilitar el manejo del código HTML y CSS. Además configuraremos el enrutado dinámico para que las estructuras sean homogéneas, el tiempo de carga entre página y página no exista y otro tipo de características que se irán desarrollando a lo largo del resto del semestre.

- Framework UI: **Bulma**.

Bulma como framework UI, nos facilita el maquetar algunas partes del proyecto, por lo tanto tiempo ganado en maquetación responsiva, es tiempo invertido en programación y refactorización. Además utilizaremos **FontAwesome** para utilizar algunos de los iconos vectorizados.

- Publicación: **Netlify**.

Por último para el despliegue se utilizará **Netlify**, que nos permite vincular nuestra cuenta de **Github**, y el repositorio en cuestión, y ejecutar la compilación en nuestro servidor en cuestión de un comando.

B. Objetivos.

- Crear un boilerplate reutilizable, configurable y modular.
- Aprender a configurar un entorno 100% moderno, pensando en la arquitectura frontend.
- Crear una aplicación web responsive, visible en todos los dispositivos y navegadores.
- Aprender librerías nuevas como Vue JS o Bulma.

2. Configuración del entorno de desarrollo.

A continuación se detalla como configuraremos nuestro entorno de desarrollo mediante Parcel y el resto de herramientas.

2.1. Node y npm.

En primer lugar deberemos tener instalado **npm**, para ello necesitaremos instalar Node js si no lo tenemos instalado.

Existen muchas maneras de instalar Node js, en este caso, lo hemos realizado mediante Node Version Manager (en Ubuntu 19.10) ejecutado mediante bash, para ello podremos consultar la documentación sobre cómo instalarlo aquí:

<https://github.com/nvm-sh/nvm>

Básicamente ejecutaremos en terminal el siguiente comando:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

Y para instalar Node, una vez instalado NVM, utilizaremos el siguiente comando sustituyendo <version> por la versión que necesitemos:

```
nvm install <version>
```

Comprobaremos que npm ha sido instalado:

```
npm -v
```

A continuación usaremos el siguiente comando para empezar a crear nuestro proyecto, e iremos completando las instrucciones que nos van apareciendo por consola. Se nos generará un package.json para ver la configuración que tenemos:

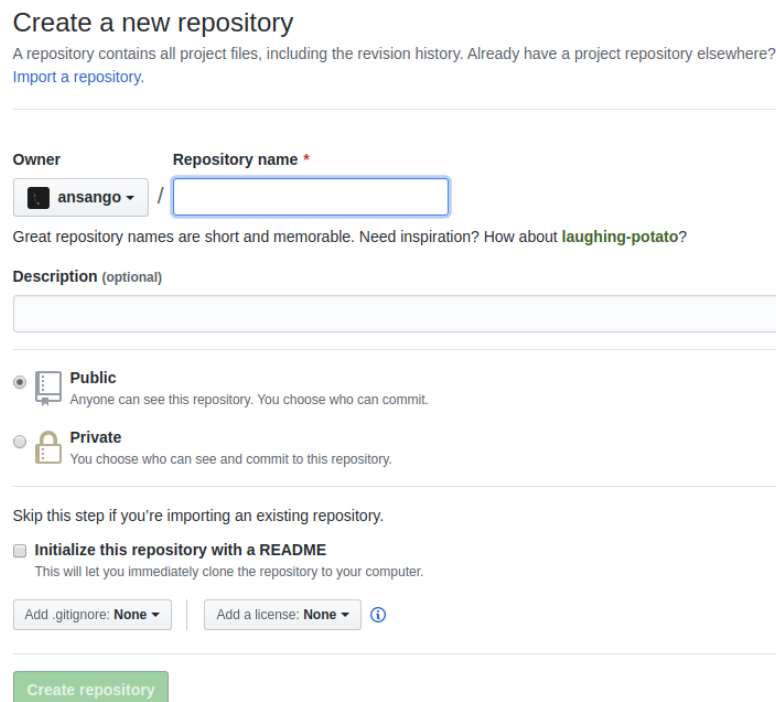
```
npm init
```

2.2. Configuración de Git y Github.

En segundo lugar instalaremos **Git** y configuraremos nuestro repositorio en **Github**, para ello ejecutaremos en nuestro terminal lo siguiente (ubuntu-debian):

```
sudo apt install git
```

Para configurar nuestro repositorio remoto, accedemos a **Github**, y crearemos un nuevo repositorio:



The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and provides a brief explanation of what a repository is, with a link to 'Import a repository'. Below this, there are two main input fields: 'Owner' (a dropdown menu showing 'ansango') and 'Repository name' (a text input field). A hint below these fields suggests short and memorable names, with an example 'laughing-potato'. There is also a 'Description (optional)' text area. Underneath, there are two radio button options for repository visibility: 'Public' (selected) and 'Private'. Below these, there is a checkbox for 'Initialize this repository with a README'. At the bottom, there are two dropdown menus for 'Add .gitignore' (set to 'None') and 'Add a license' (set to 'None'), followed by a 'Create repository' button.

Después en terminal introducimos, en el directorio donde queremos sincronizar nuestro proyecto con el repositorio remoto, lo siguiente:

1. `echo "# a" >> README.md` (nos genera un readme markdown para el proyecto),
2. `git init`, iniciamos el control de versiones,
3. `git add *`, añadimos todos los archivos no registrados,
4. `git commit -m 'first commit'`, hacemos nuestro primer commit,
5. `git remote add origin https://github.com/ansango/a.git`, añadimos nuestro proyecto al repo remoto,
6. `git push -u origin master`, y subimos los cambios.

Así se hace de forma básica un push (a partir del tercer paso) desde un repo local a uno remoto. Esta operación la realizaremos siempre que tengamos cambios o que queramos subir nuestro código al repositorio remoto. La idea sería crear ramas mediante git branch para ir realizando nuestras tareas y luego mergearlas a la rama master. También sería una buena opción configurar git flow, para desarrollar features en la rama desarrollo y por último cuando están funcionando correctamente los cambios mergearlos a master, así utilizaríamos releases. Pero en este caso siendo un proyecto de estudio, no ha sido necesario complicarse tanto.

En este momento ya tendríamos configurado nuestro repo local con el remoto, y para comprobar el estado de git podemos utilizar git status.

Además crearemos un archivo de configuración de git “.gitignore”, que evitará que subamos dependencias no deseadas, e introducimos lo siguiente:

```
node_modules/
```

2.3. Instalación y configuración de Parcel.

Para instalar parcel, utilizaremos npm y el siguiente comando:

```
npm install -g parcel-bundler
```

Para poder iniciar Parcel necesitaremos crear dos archivos básicos que sirven como punto de entrada al compilador:

```
index.html
```

```
index.js
```

Deberemos importar el index.js en nuestro html, y ejecutando:

Parcel index.html, se arrancaría parcel en un servidor local.

A continuación configuraremos lo que será nuestro entorno de desarrollo y nuestro entorno de producción, mediante los siguientes scripts en nuestro package.json:

```
"scripts": {  
  "dev": "npm-run-all clean parcel:dev",  
  "build": "npm-run-all clean parcel:build",  
  "parcel:dev": "parcel public/index.html --open",  
  "parcel:build": "parcel build public/index.html",  
  "watch": "parcel watch index.html",  
  "clean": "rimraf dist .cache .cache-loader"  
},
```

Para ejecutar nuestro entorno de desarrollo ejecutaremos:

```
npm run dev
```

Para compilar y ejecutar nuestro entorno de producción:

```
npm run build
```

Esto nos genera un /dist, que son nuestros archivos compilados.

2.4. Configuración e instalación de Babel y PostCSS.

Parcel nos trae por defecto instalado Babel, que configuraremos creando un archivo en la raíz del directorio, “.babelrc”, donde añadiremos lo siguiente:

```
{  
  "presets": [  
    "@babel/preset-env"  
  ]  
}
```

A continuación instalaremos crearemos un archivo de configuración para los navegadores “.browserlistrc”, donde introduciremos lo siguiente:

last 4 version

> 2%

not dead

IE 10

Esto nos indica las versiones de los navegadores que convertirán los preprocesadores de código.

Por último instalaremos Autoprefixer y PostCSS mediante:

```
npm install --save-dev autoprefixer.
```

Crearemos un archivo de configuración llamado “.postcssrc”:

```
{  
  "plugins": {  
    "autoprefixer": true  
  }  
}
```

Donde activaremos el plugin de Autoprefixer.

2.5. Instalación y configuración de Vue JS.

En quinto lugar vamos a instalar y configurar Vue JS con Parcel. Esta tarea entraña cierta complejidad puesto que lo normal es utilizar Webpack como Module Bundler para Vue, además de que Vue cuenta con un CLI que nos facilita toda la configuración.

A continuación se explica como configurarlo con Parcel así como reorganizar las dependencias que tendrá de partida nuestro proyecto, para poder organizar mejor los componentes de Vue que se crearán.

Para ello seguiremos estos pasos:

1. Copiaremos en nuestro index.html lo siguiente:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="icon" href="favicon.ico">
  <title>Front-end Roadmap!</title>
</head>
<body>
  <div id="app"></div>
  <script src="../src/main.js"></script>
</body>
</html>
```

2. Crearemos un archivo llamado App.vue, que será la entrada a Vue, con la siguiente estructura:

```
<template>
  <div id="app">
    <div class="container">
    </div>
  </div>
</template>

<script>
export default {
  name: "App",
};
</script>
```

3. Cambiaremos el nombre de index.js a main.js e introducimos lo siguiente:

```
import 'babel-polyfill';
import Vue from 'vue'
import App from './App'

Vue.config.productionTip = false

new Vue({
  el: '#app',
  render: h => h(App)
}).$mount('#app')
```

Con esto tendremos configurado Vue en parcel, pero realizaremos las siguientes modificaciones e instalaciones para dejar listo nuestro proyecto, a fin de poder empezar a programar de una forma más ordenada.

1. Instalamos `@vue/component-compiler-utils`, `vue-template-compiler`, `@vue/babel-helper-vue-jsx-merge-props`, `@vue/babel-preset-jsx`, `vue-hot-reload-api`. Todo esto nos ejecutará la compilación interna de Vue y la conversión mediante Babel.
2. Añadiremos en nuestro package.json:

```
"alias": {
  "vue": "./node_modules/vue/dist/vue.common.js"
}
```

Que nos permitirá ejecutar la compilación a producción correctamente.

3. Y ejecutamos `npm run dev`, que nos compilara todo el proyecto y lanzará la aplicación de Vue en local.

2.6. Instalación y configuración de Vue Router.

Para finalizar la configuración inicial de Vue, se decide instalar Vue Router para poder configurar el enrutamiento dinámico. Una vez instalado reorganizamos las dependencias para seguir un estándar y poder mantener nuestro proyecto.

1. Instalamos Vue Router:

```
npm install vue-router
```

2. Creamos los siguientes directorios y archivos en la raíz del proyecto (y movemos el los creados), en este orden:

- public
 - index.html
 - favicon.ico
- src
 - assets
 - * cualquier imagen (jpg, svg...)
 - components
 - HelloWorld.vue
 - router
 - index.js
 - store
 - index.js
 - views
 - Home.vue
 - App.vue
 - main.js
- .babelrc
- .browserslistrc
- .gitignore
- .postcssrc
- package-lock.json
- package.json

3. A continuación en main.js introducimos lo siguiente:

```
import 'babel-polyfill';
import Vue from 'vue'
import App from './App'
import router from './router'

Vue.config.productionTip = false

new Vue({
  el: '#app',
  router,
  render: h => h(App)
}).$mount('#app')
```

4. En router/index.js introducimos:

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Home from './views/Home.vue'

Vue.use(VueRouter)

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
]

const router = new VueRouter({
  mode: 'history',
  routes
})

export default router
```

Así quedaría nuestro proyecto con la instalación de Vue y Vue Router, de una forma eficiente y estandarizada.

2.7. Instalación y configuración de Bulma y FontAwesome.

Para instalar Bulma y FontAwesome seguimos los siguientes comando en terminal:

```
npm install bulma,  
npm install font-awesome
```

Y para añadirlo a nuestro proyecto, añadimos en main.js lo siguiente:

```
import 'babel-polyfill';  
import Vue from 'vue'  
import Bulma from 'bulma';  
import FontAwesome from '@fortawesome/fontawesome-free/css/all.css'  
import App from './App'  
import router from './router'
```

```
Vue.config.productionTip = false
```

```
new Vue({  
  el: '#app',  
  router,  
  Bulma,  
  FontAwesome,  
  render: h => h(App)  
}).$mount('#app')
```

Así queda renderizado en Vue, Bulma y FontAwesome y podemos utilizarlo en nuestro proyecto sin tener que añadir links o scripts en el html de entrada, dejando nuestra configuración limpia.

2.8. Configuración de Netlify y sistema de publicación en producción.

Por último vamos a configurar Netlify como sistema de despliegue, antes de nada deberemos crear un archivo en la raíz llamado "netlify.toml", con lo siguiente:

```
[[redirects]]
  from = "/*"
  to = "/index.html"
  status = 200
```

Esto nos permitirá utilizar el sistema de enrutado dinámico de vue router. De otra manera cada vez que metiesemos un parámetro no registrado en la url, Netlify sería el encargado de mostrarnos un 404, mientras que de esta manera podemos crear nuestra propia página de Not Found.

A continuación comitemos a nuestro repositorio remoto y subimos los cambios realizados hasta ahora.

Es hora de configurar nuestro sistema de despliegue:

1. Abrimos netlify.com, iniciamos sesión, con Github, si tenemos cuenta en en la misma, y le damos al botón de añadir "New site from Git".
2. Seleccionamos nuestro proveedor, en este caso Github, y seleccionamos el repositorio que queremos añadir.
3. Seleccionaremos la rama master para el deploy.
4. Como comandos para la compilación, deberemos seleccionar aquellos que tengamos configurados en nuestro package.json, en este caso npm run build.
5. Escribimos nuestro directorio de publicación, el que se genera al hacer la compilación. En este caso dist.
6. Por último le damos al botón de "Deploy site".

7. Cuando finalice el proceso se nos mostrará un enlace a la url generada, en este caso <https://anibalsantos.netlify.com>
8. Para volver a hacer un deploy simplemente tendremos que commitear y pushear nuestros cambios de nuestro repo local al repo remoto.

3. Diseño de la arquitectura front-end.

Detalladas y explicadas las herramientas y la configuración inicial del entorno de desarrollo y de producción, así como el sistema de despliegue, podemos proceder a plantear el siguiente punto en el desarrollo de nuestra aplicación, la arquitectura.

Partiremos de los siguientes requisitos de la página web:

- Página de portada, que deberá enlazar a las páginas de categoría
- Dos páginas de categoría, que deberán enlazar a las páginas de detalle.
- Una página de detalle por cada categoría que deberá incluir enlaces a otras páginas de detalle de la misma categoría.

En función de la arquitectura inicial que hemos realizado configurando Vue partiremos de lo siguiente:

- public
 - index.html
 - favicon.ico
- src
 - assets
 - * cualquier imagen (jpg, svg...)
 - components
 - HelloWorld.vue
 - router
 - index.js
 - store
 - index.js
 - views
 - Home.vue
 - App.vue
 - main.js

Por lo que rediseñaremos de otra forma los requisitos para poder realizar el proyecto de una forma más dinámica y que sea menos costoso de mantener.

En primer lugar tendremos las siguientes páginas que corresponden a la dependencia views:

- Home: esta será nuestra página de inicio. Que será única, y conducirá a la página de detalle.
- Not Found: esta será nuestra página de error. Que será única y conducirá a la página de inicio.
- Road: esta será nuestra página de categoría. Esta página será dinámica, tendremos tantas páginas de categoría según queramos, porque almacenamos los datos de cada categoría en `store/index.js`, en forma de objetos, para renderizarlos y poder manejar la ampliación de la misma. Teniendo siempre la misma estructura. Además hemos configurado así el router para que vaya componiendo las rutas en función de los nombres definidos en los objetos. Así, si introducimos en el store más categorías, se ampliarán sin tener que realizar más componentes con maquetación, algo que sería más laborioso.
- Tool: esta será nuestra página de detalle. Que será igual que la anterior, generada dinámicamente, en nuestro data esto será un array de objetos que será una propiedad de categoría, por lo que cada objeto del array será una subcategoría y podremos tener las que queramos porque se renderizan de forma dinámica.

Al crear este tipo de páginas dinámicas, se ha modificado el funcional de esta tarea, para aún siendo una vista en sí misma independiente la vista de detalle, generarla de forma dinámica seguida a la de categoría, para mejorar la interactividad de la aplicación, en primer lugar y en segundo para mejorar la experiencia de usuario.

Así mismo, dentro de componentes se incluirá una barra de navegación, presente en todas las vistas y un footer también presente en todas las vistas.

4. Desarrollo.

Diseñada la arquitectura es el momento del desarrollo. En primer lugar vamos a definir como será el router de la aplicación y el almacén de datos.

Nuestro router se definirá de la siguiente manera:

```
const routes = [  
  {  
    path: "/",  
    name: "home",  
    props: true,  
    component: () => import("../views/Home.vue")  
  },  
  {  
    path: "/road/:slug",  
    name: "road",  
    props: true,  
    component: () => import("../views/Road.vue"),  
    children: [  
      {  
        path: ":toolSlug",  
        name: "tool",  
        props: true,  
        component: () => import("../views/Tool.vue")  
      }  
    ],  
    {  
      path: "/404",  
      alias: "*",  
      name: "notFound",  
      component: () => import("../views/NotFound.vue")  
    }  
  ]  
];
```

Definimos las páginas de Home, Categoría (Road), Detalle (Tools) y 404.

Cogemos como parámetros para generar las rutas el mismo almacenado en el store. Nuestro store tendrá toda la información que renderizamos, además de las rutas, que son una propiedad más. A continuación se muestra un fragmento:

```
roadmap: [  
  {  
    id: 1,  
    name: "Basics",  
    slug: "basics",  
    image: svgImage[4],  
    Tools: [  
      id: 1,  
      name: "HTML",  
      slug: "html",  
      video: "https://www.youtube.com/embed/pQN-pnXPaVg",  
      icon: "fab fa-html5",  
    ]  
  }  
]
```

Este sería un pequeño ejemplo de como vamos construyendo los datos. Todos llevarán un id, un nombre, una dirección, una imagen o video o una url.

El principal problema que se ha encontrado a la hora de renderizar este tipo de datos, es que al almacenar la ruta de una imagen como cadena de texto:

```
image: "../assets/imagen.svg"
```

Y renderizarla en el componente .vue, normalmente suele utilizarse un alias que en parcel no es definible pero sí en Webpack. Por lo que hemos optado por importar todas las imágenes de un formato como un objeto y parsearlas, para poder utilizar ya el parámetro dinámico de la imagen ya compilada y poder utilizarla en nuestro almacén, del siguiente modo:

```
// store/index.js
import svg from "../assets/*.svg";
const svgImage = Object.values(svg);

{
  id: 1,
  name: "Basics",
  slug: "basics",
  image: svgImage[4],
  resume:
}
```

```
// home.vue
```

```
<div class="container" v-for="road in roadmap" :key="road.id">
  
</div>
```

A nivel de componente en vue, se sigue la estructura de template que es la siguiente:

```
<template>
</template>

<script>
</script>

<style scoped>
</style>
```

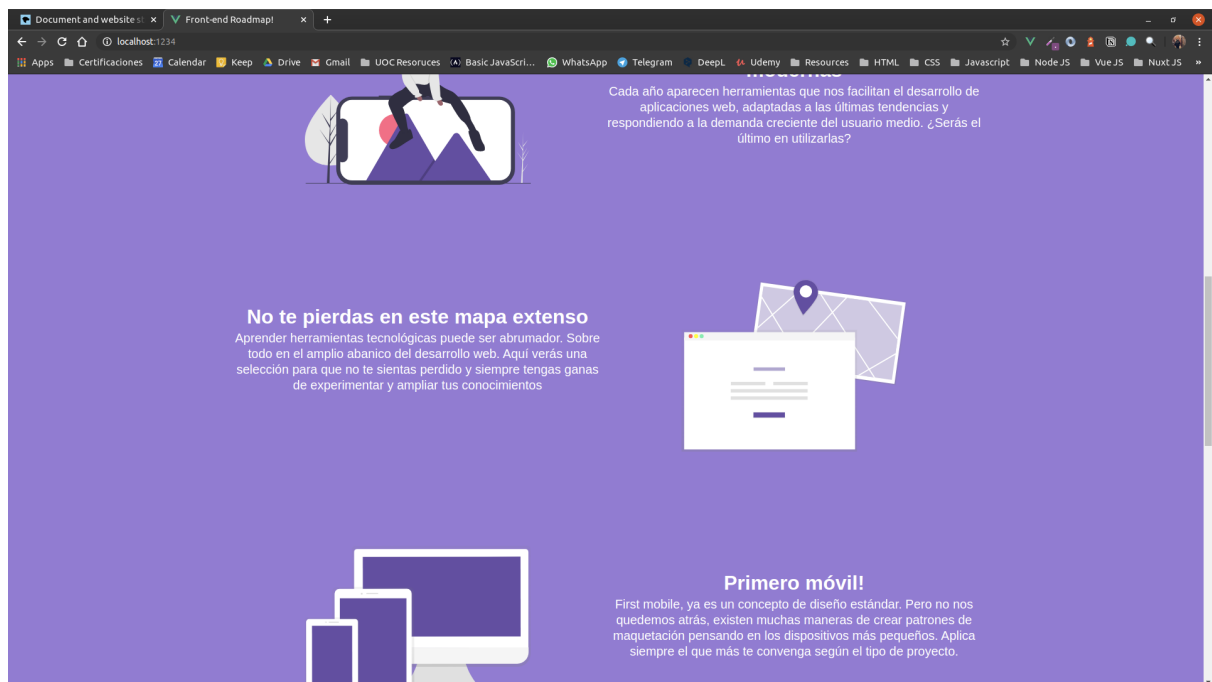
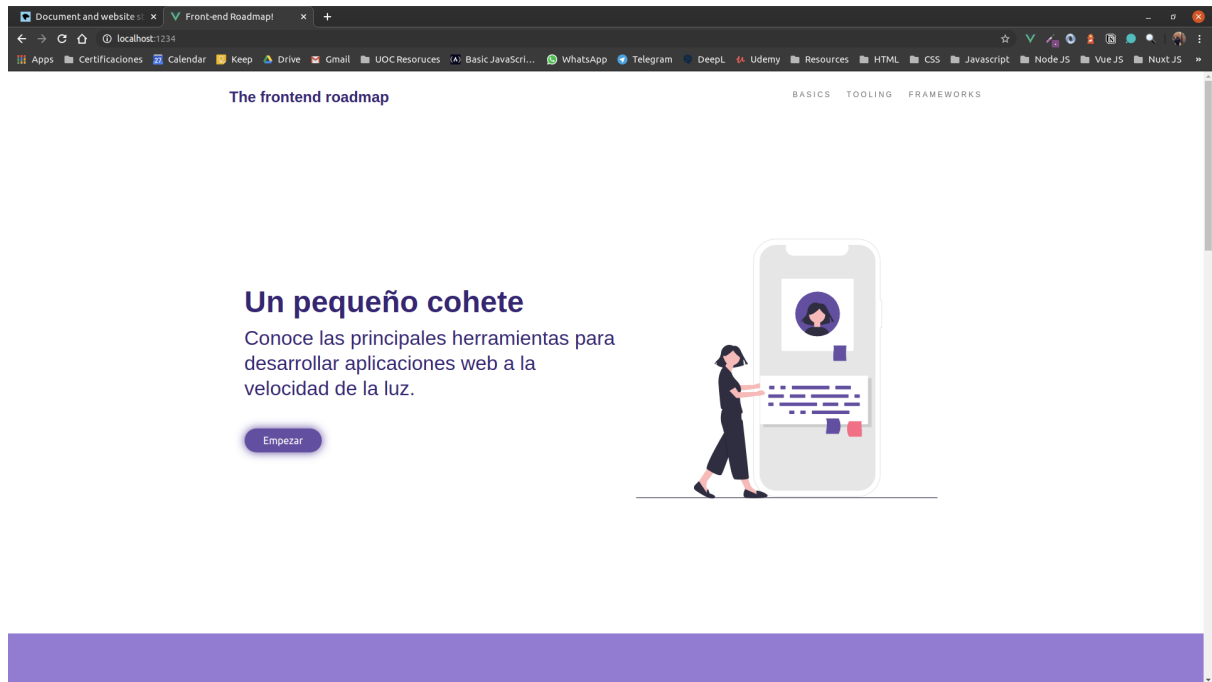
Cada etiqueta hace referencia a una parte de programación web, template es la parte html, script la parte de js, y style la parte de css. Como norma general aplicamos el atributo scoped en style para aislar por componentes la maquetación, aunque existe una excepción y es que en App.vue no se aplica, para poder generar estilos que van a ser utilizados por el resto de componentes, como clases de colores, para textos, botones, etc...

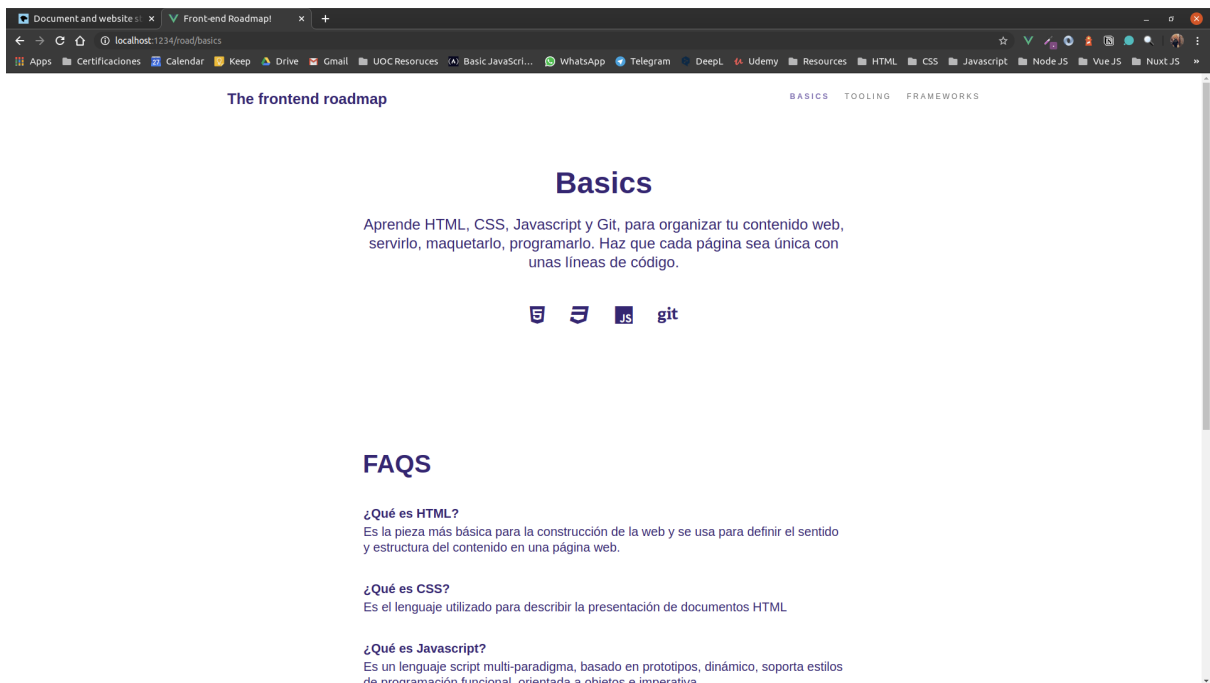
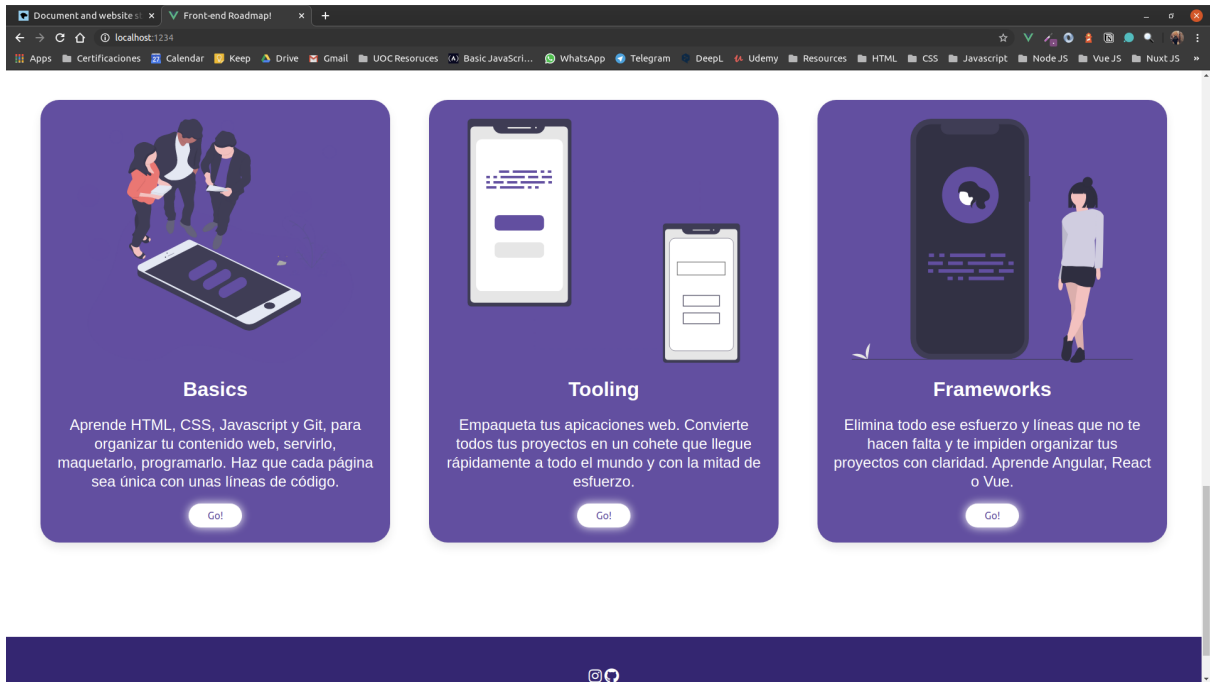
- Maquetación: respecto a la maquetación se siguen los siguientes parámetros de diseño:
 - Bulma: utilizaremos Bulma para la estructuración de algunos bloques de contenido como las secciones, tarjetas y barras de navegación, pie de página y modales para los videos.
 - Clases genéricas: para crear los colores primarios, secundarios, fondos, y botones.
 - First mobile ambiguo: se sigue un patrón de diseño first mobile, pero pensando en que en algunos casos no es necesario reagrupar este contenido en desktop. Así hemos establecido media queries para editar en bloque contenidos que cambian en móvil o escritorio. Siempre utilizando flexbox para no tener problemas con Internet Explorer, ya que Bulma en algunos puntos de este desarrollo crea conflicto entre clases en este navegador.
- Funcionalidad: en este apartado sobre todo se utiliza la funcionalidad otorgada por vue router para crear animaciones de scroll de una parte a otra de la web, almacenando en un objeto las coordenadas de posición y ejecutando mediante la función scrollBehavior como se dirigirá al usuario a puntos de interés.

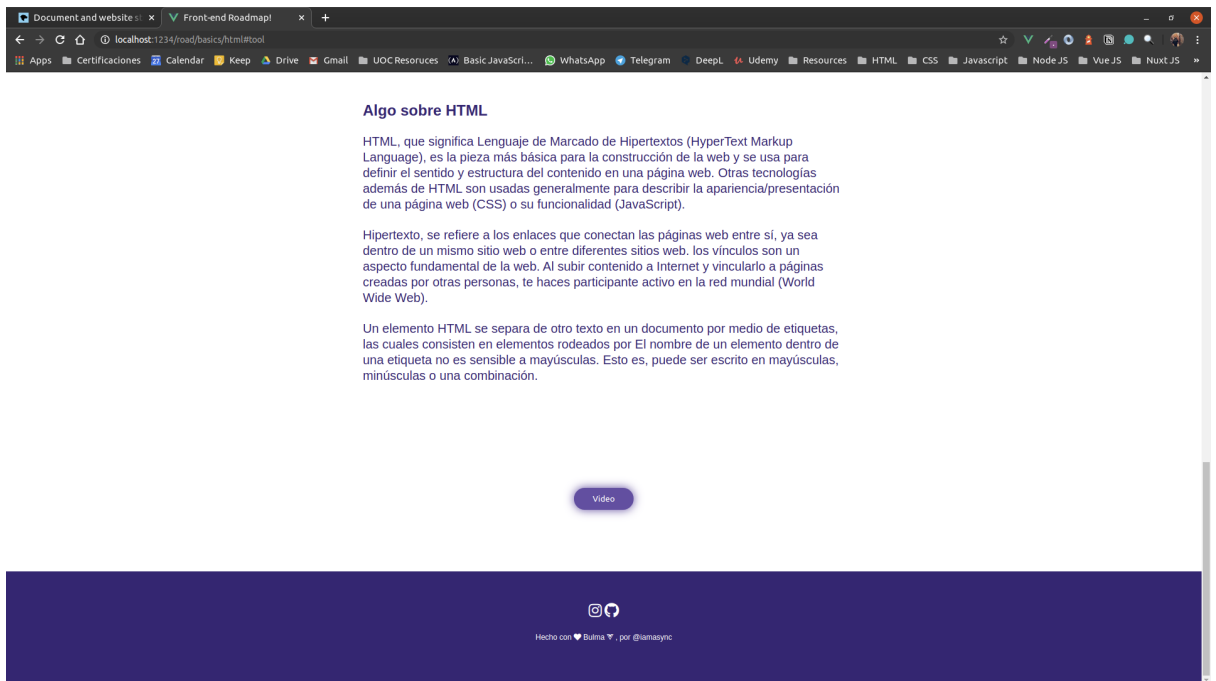
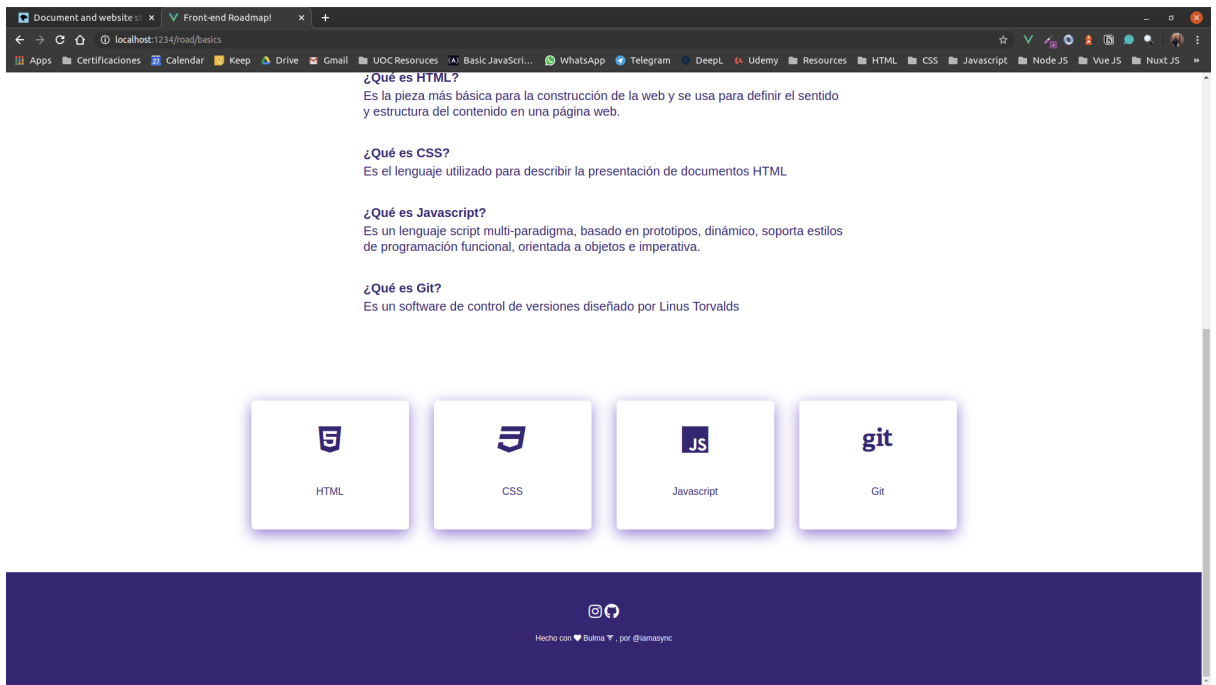
De esta parte es donde deducimos la supresión de una nueva página de detalle y se integra de forma orgánica después de categoría, (al clickar en una de las tarjetas que se encuentran en cada categoría).

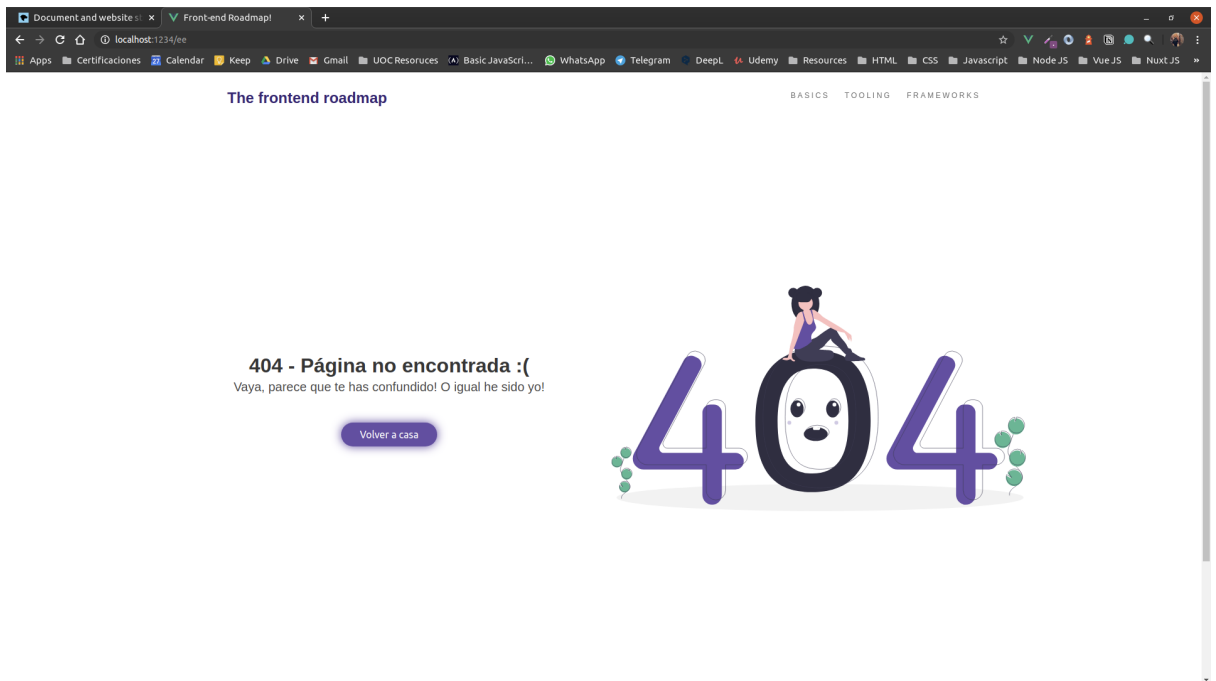
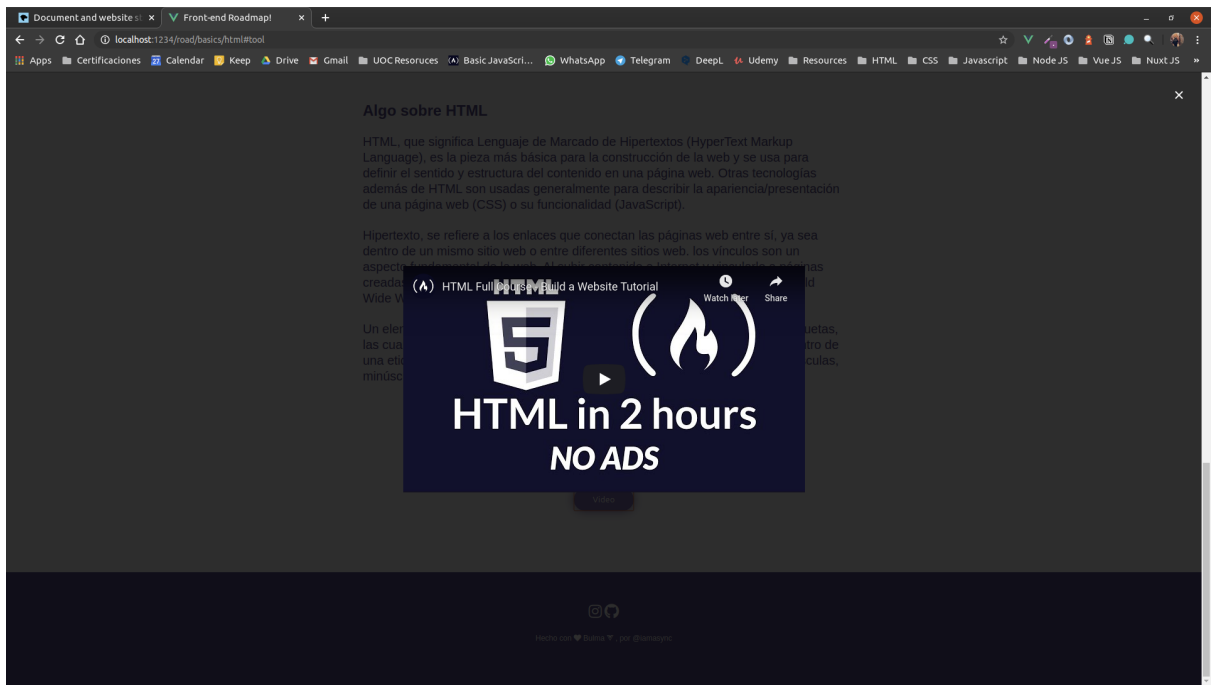
Por último se muestran a continuación capturas de pantalla del resultado final, que podrá ser ampliado en siguientes prácticas:

- Desktop:

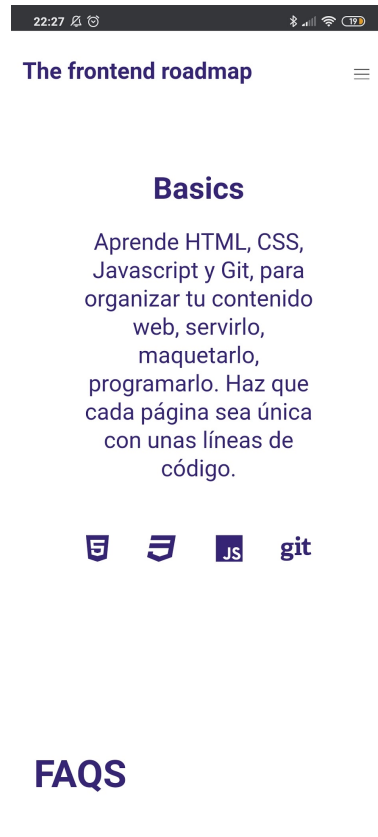


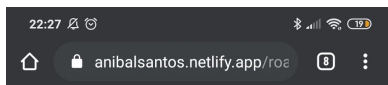






- Móvil:





FAQS

¿Qué es HTML?

Es la pieza más básica para la construcción de la web y se usa para definir el sentido y estructura del contenido en una página web.

¿Qué es CSS?

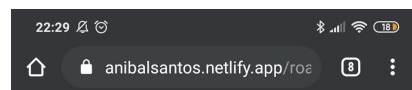
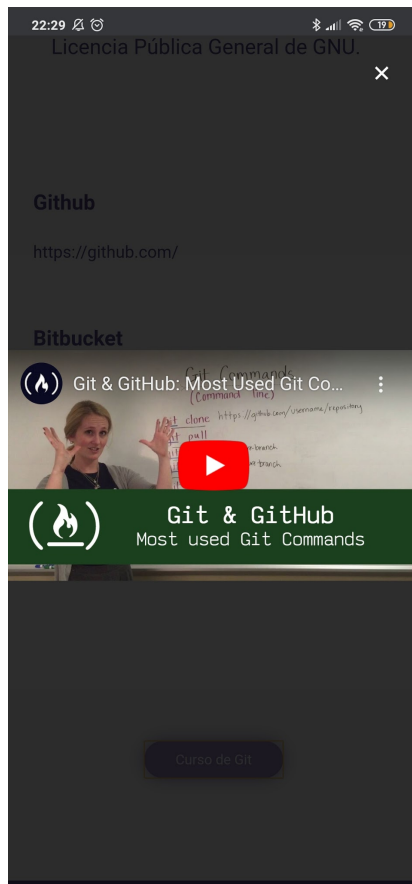
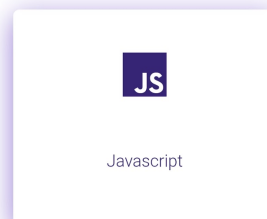
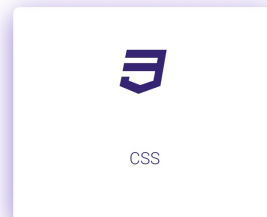
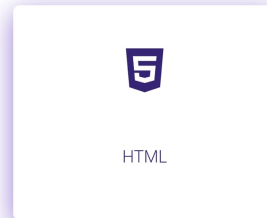
Es el lenguaje utilizado para describir la presentación de documentos HTML

¿Qué es Javascript?

Es un lenguaje script multi-paradigma, basado en prototipos, dinámico, soporta estilos de programación funcional, orientada a objetos e imperativa.

¿Qué es Git?

Es un software de control de versiones diseñado por Linus Torvalds



The frontend roadmap



404 - Página no encontrada :(

Vaya, parece que te has confundido! O igual he sido yo!

Volver a casa



5. Publicación.

Finalmente, después de realizar todo el proceso de desarrollo y testeo correspondiente, realizamos el push para publicar en master nuestros últimos cambios. Estos serán compilados y publicados en Netlify en la siguiente dirección:

<https://anibalsantos.netlify.com>

El repositorio público se encuentra alojado en la siguiente dirección de Github:

<https://github.com/ansango/PEC-1---Desarrollo-de-una-web>

Y el boilerplate vacío se ha guardado en un repositorio a parte para empezar a construir otros proyectos de una forma rápida y quitando el esfuerzo de configuración inicial de todas las herramientas. Se encuentra en el siguiente repositorio en Github:

<https://github.com/ansango/Vue-Parcel-Bulma>