



Universitat
Oberta
de Catalunya

M4.258 - Herramientas HTML y CSS II

PEC 3

Aníbal Santos Gómez

ÍNDICE: DOCUMENTACIÓN.

- 1. Objetivos.**
- 2. Herramientas utilizadas.**
- 3. Metodologías aplicadas.**
- 4. Desarrollo.**
- 5. Publicación.**

1. Objetivos.

- Diseñar y ejecutar un pequeño sitio web responsive. Partiendo de un boilerplate.
- Escoger criterios de desarrollo y arquitectura reutilizable CSS. Utilizar el lenguaje de preprocesado de estilos Sass.
- Instalar y configurar correctamente Tailwind dentro del entorno de UOC Boilerplate.
- Incorporar React.js como librería JS para agilizar la construcción de una UI atómica y aligerar la escritura de Sass.
- Utilizar y configurar herramientas para la mejora y estilo de código, como ES Lint y Prettier.
- Publicar el repositorio en GitHub y realizar un deployment en Netlify.
- Generar una documentación práctica que siga la elección de criterios, proceso de desarrollo y el proceso de despliegue.

2. Herramientas utilizadas.

Para el desarrollo de una aplicación 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 que ofrecerá una Landing Page o página de producto (en este caso un grupo de música), readaptada de la práctica anterior utilizando la atomización de TailwindCSS.

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, la generación de código con un control de calidad 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, Prettier, Styling plugin, ESLint, Prettier, Babel Javascript, Quokka.js.

- 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 facilitarán la configuración de los diversos paquetes que utilizaremos.

- Module bundler: **Parcel**.

Para empaquetar todo nuestro proyecto utilizaremos **Parcel** ya que es por defecto el module bundler que se encuentra definido en el package.json de UOC boilerplate. Además crearemos un archivo de configuración (.parcelrc) para poder utilizar los plugins de Parcel.

- Preprocesadores de código: **PostCSS y Sass**.

PostCSS es una herramienta de desarrollo de software que utiliza complementos basados en JavaScript para automatizar las operaciones de rutina de CSS.

Sass que nos permitirá extender funcionalidades CSS que no están contenidas de forma nativa, como funciones, variables... y que nos permitirán programar nuestras hojas de estilos de una manera mucho más ágil.

- Dependencias: **Babel**, **React.js**, **React Icons**, **ESLint**, **Prettier**, **TailwindCSS**, **Headlessui**

Babel, es un transcompilador de JavaScript gratuito y de código abierto que se utiliza principalmente para convertir el código ECMAScript 2015+ (ES6+) en una versión de JavaScript compatible con versiones anteriores que puede ser ejecutada por motores de JavaScript más antiguos.

React.js, es una biblioteca Javascript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones en una sola página. Mantenido por Facebook y la comunidad de software libre.

React Icons, librería que utiliza importaciones ES6 que permiten incluir una colección de iconos, (Font Awesome. Material...), pero sólo importando aquellos que necesitemos como si fueran componentes.

ESLint, es una herramienta de análisis de código estático para identificar patrones problemáticos encontrados en el código JavaScript.

Prettier, es un formateador de código con opinión. Impone un estilo consistente analizando su código y reimprimiéndolo con sus propias reglas que tienen en cuenta la longitud máxima de la línea, envolviendo el código cuando es necesario.

TailwindCSS, es un framework CSS de utilidades, que se componen directamente sobre las etiquetas html.

Headlessui, librería de UI que funciona junto con TailwindCSS en React o Vue.

- 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.

3. Metodologías utilizadas.

Para el desarrollo de esta aplicación web orientada a componentes e interfaces de usuario, hemos utilizado como metodologías CSS, dos metodologías que se complementan muy bien y nos permitirán reducir nuestro código css de una manera recursiva y sustancial.

Las metodologías elegidas son:

- **Mobile First:** es la forma que utilizaremos para construir el diseño, partiendo siempre desde pantallas pequeñas a pantallas más grandes. Por defecto construiremos orientado a dispositivos móviles y abordaremos el resto de comportamientos en función del ancho de la pantalla para definir comportamientos para tablet y desktop.
- **ITCSS:** como base para el desarrollo se escoge este tipo de metodología o arquitectura, que nos permite establecer un pequeño estándar a desarrollar para nuestra aplicación.

Esta metodología nos ayudará a definir determinados tipos de selectores, clases y configuraciones en función, en primer lugar, de la magnitud o alcance, la especificidad y según la claridad respecto a la abstracción.

- **OOCSS:** como arquitectura complementaria hemos elegido OOCSS, para poder realizar abstracción de ciertos elementos que se repiten a lo largo de nuestro proyecto y que tienen una base común, pero que de ninguna manera son ajustes, herramientas, configuraciones genéricas o elementos nativos HTML.

Esta selección junto con el uso de React.js y TailwindCSS nos permite atomizar los elementos que compondrán nuestras interfaces de usuario y el mantenimiento del mismo.

Cabe destacar que dado que la intención es atomizar lo máximo posible los elementos y utilizarlos en la mayor parte de los casos, hemos aligerado la estructura de ITCSS al máximo posible, puesto que ya no necesitaremos tantos settings en Sass o helpers, puesto que disponemos de los mismos en TailwindCSS.

Comenzaremos construyendo en primer lugar orientado a componentes atomizados, desde botones, banners, títulos, párrafos o cualquier otro elemento que podamos reutilizar dentro de nuestra UI.

Seguiremos planteando, como ya veníamos haciendo en anteriores prácticas la construcción desde dispositivos móviles, a otros dispositivos más grandes.

Utilizaremos ITCSS para crear capas jerárquicas que nos aislaran según la especificidad la reutilización de bloques css.

Pero sobre todo nos centraremos en lo anteriormente comentado. Construir de menos a más, y de pequeño a grande en todas las dimensiones, componentes y pantallas.

A continuación se detallan las dependencias que componen la arquitectura mencionada en estos los puntos anteriores. Para llevar a cabo la misma, hemos partido de la configuración de dependencias del boilerplate assets/styles. Los ficheros/dependencias a tener en cuenta por impacto de mayor a menor son:

- main.scss: este archivo carga todos los módulos de la arquitectura ITCSS. En él procedemos a importar cada módulo de dicha arquitectura.

```
@tailwind base;
```

```
@tailwind components;
```

```
@import
```

```
url("https://fonts.googleapis.com/css2?family=Fira+Code:wght@300;400;700&family=Montserrat:wght@300;400;700&display=swap");
```

```
@import
```

```
url("https://fonts.googleapis.com/css2?family=Permanent+Marker&display=swap");
```

```
@import "elements/index";
```

```
@import "objects/index";
```

```
@import "components/index";
```

```
@tailwind utilities;
```

- Librería de utilidades de TailwindCSS: importaremos en primer lugar el base y los components.
- Fuentes: en segundo lugar las fuentes que vayamos a utilizar en el proyecto.
- ITCSS en SASS: en este caso hemos reducido nuestro ITCSS a elementos, objetos y componentes.
 - Elements: en este módulo daremos estilo base a los diferentes elementos html por defecto. En este caso lo hemos utilizado únicamente para modificar los diferentes tipos de títulos html a nivel global.
 - Objects: en este módulo realizaremos la arquitectura OOCSS, el patrón de objetos que utilizamos básicamente se compone de vista, contenedor título, contenedor notas, contenedor tarjetas y not found.
 - Components: esta dependencia se compone de un índice que va importando todos los módulos css relativos a los componentes utilizados en React. Como los componentes en React son reutilizables, este código también.

Tenemos en este caso, componentes como, banner, card, code, footer, gallery, hero, navbar, parallax, quote, table.

- ¿Qué diferencias hay entre el enfoque de tipo CSS semántico (el que usaste en las otras PEC) y el CSS de utilidades? ¿Cómo afectó esto a tu proceso de desarrollo? ¿Y a tu código?

El CSS funcional (a veces denominado CSS atómico) es la práctica de utilizar clases de utilidad pequeñas, inmutables y con nombres explícitos para construir componentes. Una variedad de bibliotecas de clases listas para usar, como Tailwind CSS, han surgido para hacer que empezar con CSS funcional sea lo más fácil posible.

El CSS Semántico nos permite abstraer ciertos patrones repetitivos dependiendo del patrón elegido.

En nuestro desarrollo hemos podido mezclar ambos conceptos, ya que partiamos de una base sólida que separaba por capas (ITCSS) y atomizaba por objetos, pero no lograba apuntar a la libertad de utilidades en css que nos permite empezar a programar interfaces de 0 obviando muchas otras cuestiones.

Siguiendo lo ya comentado, de lo más pequeño a lo más grande, fusionar ITCSS y El CSS funcional nos ayuda a poder manejar, reutilizar y estructurar todas nuestras interfaces de una manera más eficiente y mantenible.

- ¿Qué diferencias encontraste entre usar una librería de componentes y una librería de utilidades?

A groso modo las librerías de componentes css como Bootstrap nos “atan” de alguna manera a un patrón contemplado por un equipo de desarrolladores. Por muy customizable que sea, no es nuestro desarrollo, lo cual nos obliga muchas veces a tener que rehacer el trabajo ya realizado.

En contraposición una librería de utilidades nos permite organizar a nuestro gusto la forma de construir UI, podemos atomizar cualquier cosa, no solo elementos html, sino bloques, componentes, vistas, etc... Esto hace que si pensamos en como va a ser la interfaz, o si bien tenemos un diseño ya realizado de base, abstraer en pequeños elementos resulta sencillo y a la vez rápido para construir, ya que no tenemos que preocuparnos de todos los selectores css.

Al final lo que nos interesa es priorizar y ser eficientes en lo que respecta al tiempo de las entregas con el cliente.

- ¿Qué componentes decidiste extraer y por qué?

En esta práctica he extraído todos mis componentes. Posiblemente he obviado algunos sencillos por cuestión de reutilización, como algunos botones. Pero he podido atomizar títulos, banners, estructuras como vistas, tablas, tarjetas, barras de navegación y footer.

La cuestión de porqué hacerlo responde a una cuestión de mantenimiento, legibilidad de código y arquitectura.

Como comenté finalmente en esta práctica, utilizar librerías como Tailwind y React, nos permiten reutilizar todo si lo atomizamos. Pensemos por ejemplo en un Título (h1), que se incluye dentro de cada página o vista, este a su vez puede o no reutilizarse en un banner, puede estar alineado en dicho banner en el centro, o a la derecha, según la vista. Es necesario atomizar cada elemento, para customizar en situaciones cambiantes.

De ahí que haya decidido juntar ambas librerías, que junto con Sass nos permiten enfocarnos a cada componente y mantener un scope por componente irrompible.

4. Desarrollo:

Para explicar el desarrollo completo de este proyecto, hemos dividido esta sección en las siguientes fases, que a continuación resumen el desarrollo del mismo:

1. Descargar boilerplate.

En primer lugar descargamos el boilerplate facilitado para esta práctica. En mi caso realicé previamente un fork del boilerplate de la UOC a mi cuenta de usuario en **Github**.

Podemos descargarlo haciendo git clone en cualquier dependencia de nuestro equipo que deseemos mediante terminal, introduciendo lo siguiente:

```
git clone https://github.com/ansango/uoc-boilerplate
```

o

```
git clone https://github.com/uoc-advanced-html-css/uoc-boilerplate
```

2. Instalación y configuración de dependencias.

En primer lugar instalaremos el proyecto mediante el siguiente comando:

```
npm i
```

Ya tendremos instaladas las dependencias base del boilerplate instaladas. A continuación instalaremos y configuraremos las dependencias **Babel**, **React.js**, **React Icons**, **ESLint**, **Prettier**, **TailwindCSS**, **Headlessui**.

Vamos a instalar **React** en primer lugar, para ello, desde terminal introduciremos el siguiente comando:

```
npm i react react-dom react-router-dom
```

Con esto hemos instalado React, React DOM y React Router, para poder trabajar con rutas. Antes de instanciar React y empezar a montar la aplicación necesitaremos instalar **Babel** y su preset para poder trabajar con **JSX**, además, necesitaremos utilizar un plugin para **Parcel**, que nos transforme las urls de las imágenes que importamos como objetos al proyecto, de otra manera no podremos renderizar imágenes de forma dinámica ni importarlas en nuestros archivos de Javascript.

Primero instalaremos **Babel** y sus plugins:

```
npm install --save-dev @babel/core @babel/preset-env @babel/preset-react  
@parcel/transformer-sass
```

A continuación crearemos un archivo de configuración de **Babel**, **.babelrc**:

```
{  
  "presets": ["@babel/preset-react", "@babel/preset-env"],  
  "plugins": ["@babel/plugin-transform-react-jsx"]  
}
```

Después necesitaremos añadir un plugin a Parcel para la importación de imágenes. Crearemos un archivo **.parcelrc**:

```
{  
  "extends": "@parcel/config-default",  
  "transformers": {  
    "*.{png,jpg,jpeg)": ["@parcel/transformer-raw"]  
  }  
}
```

Ahora ya podemos importar React en el main.js de nuestra aplicación:

```
import React from "react";  
import ReactDOM from "react-dom";  
import App from "./App";  
ReactDOM.render(<App />, document.getElementById("root"));
```

Nuestro package.json debería de contener además de las dependencias mencionadas anteriormente, las siguientes:

```
"devDependencies": {  
  "@babel/core": "^7.13.10",  
  "@babel/preset-env": "^7.13.15",  
  "@babel/preset-react": "^7.13.13",  
  "@parcel/transformer-image": "^2.0.0-nightly.2274",  
  "@parcel/transformer-sass": "^2.0.0-beta.2",  
},  
"dependencies": {  
  "react": "^17.0.2",  
  "react-dom": "^17.0.2",  
  "react-router-dom": "^5.2.0"  
}
```

A continuación vamos a instalar TailwindCSS en nuestro proyecto:

```
npm install -D tailwindcss@latest postcss@latest autoprefixer@latest
```

Configuramos nuestro archivo .postcssrc con lo siguiente:

```
{  
  "plugins": {  
    "autoprefixer": {},  
    "postcss-preset-env": {},  
    "tailwindcss": {}  
  }  
}
```

Posteriormente ejecutaremos en la terminal el siguiente comando, que nos generará un archivo de configuración de TailwindCSS (tailwind.config.js), en la raíz de nuestro proyecto:

```
npx tailwindcss init
```

```
module.exports = {  
  purge: [],  
  darkMode: false, // or 'media' or 'class'  
  theme: {  
    extend: {}  
  },  
  variants: {},  
  plugins: []  
}
```

En nuestro archivo main.scss incluiremos:

```
@tailwind base;  
@tailwind components;
```

[resto de archivos scss]

```
@tailwind utilities;
```

Eliminaremos de index.html la referencia al main.scss e incluiremos la misma en el main.js de nuestra aplicación:

```
import React from "react";
import ReactDOM from "react-dom";
import "../styles/main.scss";
import App from "./app";
ReactDOM.render(<App />, document.getElementById("root"));
```

Se ha intentado configurar el purge en tailwind.config.js, pero al realizar el dist para producción no compila correctamente los archivos y elimina los mismos de TailwindCSS, la documentación oficial provee únicamente soluciones para Create React App basada en Webpack y Nextjs, por lo tanto dejamos sin configurar el mismo, ya que se han intentado implementar diversas soluciones con Parcel sin éxito. Esto puede deberse también a que la versión de Parcel aún no es estable y estamos usando una versión nightly.

Comentar que el tiempo de carga sin purge es algo costoso, y afecta negativamente a la performance. Por lo tanto no debería de utilizarse en producción.

Para nuestra práctica nos sirve como aprendizaje para poder configurar diversas herramientas que no tienen tanta documentación para entrelazarlas entre sí. Entiendo que posteriormente se podría buscar una solución más conveniente, pero puesto que en la anterior práctica ya partíamos de React, era conveniente reutilizar el trabajo ya realizado.

A continuación instalamos **React Icons**, sustituyendo así a FontAwesome, y pudiendo acceder a cualquier librería de iconos, como Material Design, etc...

Para ello ejecutamos el siguiente comando:

```
npm install react-icons --save
```

Para importar cualquier ícono de esta librería y utilizarlo, simplemente hacemos lo siguiente en cualquiera de nuestros módulos de .js:

```
import { FaHeart, FaTwitter } from "react-icons/fa";
```

Y para utilizarlo simplemente lo tratamos como un componente:

```
<FaHeart />
```

Para ver más ejemplos tenemos la documentación:

<https://react-icons.github.io/react-icons/>

Siguiendo el proceso de instalación, continuamos con HeadlessUI, que nos permitirá tratar temas como transiciones y/o componentes no realizados en TailwindCSS con lo cual solventar funcionalidades js para poder activar determinadas funcionalidades.

Para ello ejecutamos:

```
npm install @headlessui/react
```

Para ver ejemplos de uso:

<https://headlessui.dev/>

Nuestro paquete de dependencias debería de parecerse a esto:

```
"devDependencies": {  
    "@babel/core": "^7.13.10",  
    "@babel/preset-env": "^7.13.15",  
    "@babel/preset-react": "^7.13.13",  
    "@parcel/transformer-image": "^2.0.0-nightly.2274",  
    "@parcel/transformer-sass": "^2.0.0-nightly.652",  
    "autoprefixer": "^10.2.3",  
    "npm-run-all": "^4.1.5",  
    "parcel": "^2.0.0-nightly.566",  
    "postcss-preset-env": "^6.7.0",  
    "prettier": "^2.2.1",  
    "rimraf": "^3.0.2",  
    "sass": "^1.32.5",  
    "sharp": "^0.27.1",  
    "tailwindcss": "^2.1.2"  
},
```

```
"dependencies": {  
    "@headlessui/react": "^1.2.0",  
    "react": "^17.0.2",  
    "react-dom": "^17.0.2",  
    "react-icons": "^4.2.0",  
    "react-router-dom": "^5.2.0"  
}  
}
```

Por último vamos a instalar y configurar ESLint y Prettier, dos dependencias que nos ayudarán a escribir mejor código en JS (ESLint) y formatearlo (Prettier) a un estándar por proyecto. Esto es muy útil para aprender buenas prácticas y tener una misma escritura cuando el proyecto es desarrollado por un equipo.

Para instalar **ESLint** y **Prettier**:

```
npm i --save-dev eslint eslint-config-prettier eslint-plugin-import  
eslint-plugin-jsx-a11y eslint-plugin-prettier eslint-plugin-react  
eslint-plugin-react-hooks prettier babel-eslint
```

Con esto instalamos todo las dependencias necesarias para lintear un proyecto moderno en React (Hooks, imports), con Babel y Prettier.

Ahora tenemos que configurar ESLint, creamos en la raíz del proyecto un archivo de configuración, `.eslintrc.json`:

```
{  
  "env": {  
    "browser": true,  
    "es6": true,  
    "node": true  
  },  
  "extends": [  
    "plugin:react/recommended",  
    "plugin:jsx-a11y/recommended",  
    "plugin:import/recommended",  
    "plugin:prettier/recommended"  
  ]  
}
```

```

],
"plugins": ["react-hooks"],
"globals": {
  "Atomics": "readonly",
  "SharedArrayBuffer": "readonly"
},
"parser": "babel-eslint",
"parserOptions": {
  "ecmaVersion": 2018,
  "sourceType": "module"
},
"rules": {
  "react-hooks/rules-of-hooks": "error",
  "react-hooks/exhaustive-deps": "warn"
},
"settings": {
  "react": {
    "version": "detect"
  }
}
}

```

A continuación hacemos lo mismo para Prettier, creamos otro archivo de configuración, prettierrc.json:

```
{
  "semi": true,
  "singleQuote": false
}
```

Nuestro paquete de dependencias debería de parecerse en última instancia a esto:

```

"devDependencies": {
  "@babel/core": "^7.13.10",
  "@babel/preset-env": "^7.13.15",
  "@babel/preset-react": "^7.13.13",
  "@parcel/transformer-image": "^2.0.0-nightly.2274",

```

```

    "@parcel/transformer-sass": "^2.0.0-nightly.652",
    "autoprefixer": "^10.2.3",
    "babel-eslint": "^10.1.0",
    "eslint": "^7.25.0",
    "eslint-config-prettier": "^8.2.0",
    "eslint-plugin-import": "^2.22.1",
    "eslint-plugin-jsx-a11y": "^6.4.1",
    "eslint-plugin-prettier": "^3.4.0",
    "eslint-plugin-react": "^7.23.2",
    "eslint-plugin-react-hooks": "^4.2.0",
    "npm-run-all": "^4.1.5",
    "parcel": "^2.0.0-nightly.566",
    "postcss-preset-env": "^6.7.0",
    "prettier": "^2.2.1",
    "rimraf": "^3.0.2",
    "sass": "^1.32.5",
    "sharp": "^0.27.1",
    "tailwindcss": "^2.1.2"
  },
  "dependencies": {
    "@headlessui/react": "^1.2.0",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-icons": "^4.2.0",
    "react-router-dom": "^5.2.0"
  }
}

```

Ya tenemos instaladas y configuradas todas las dependencias. Ahora nos queda obligar a la compilación a utilizar linter y e interrumpir el build cuando existan fallos y que no se despliegue la aplicación en producción. En el package.json los scripts deberían de quedar de la siguiente manera:

```

"scripts": {
  "parcel:serve": "parcel serve src/index.html -p 8080 --open",

```

```

    "parcel:build": "parcel build src/index.html --public-url ./ --dist-dir dist
--no-source-maps --no-cache",
    "clean": "rimraf dist .cache .cache-loader .parcel-cache",
    "dev": "npm-run-all clean lint parcel:serve",
    "build": "npm-run-all clean lint parcel:build",
    "lint": "eslint src"
},

```

3. Entorno de desarrollo y configuración.

Una vez instaladas y configuradas las dependencias procederemos a abrir nuestro proyecto con el IDE que queramos, en este caso se utilizará VSCode.

Dentro de la amalgama de plugins y extensiones que pueden añadirse a VSCode, hemos elegido los siguientes, que nos facilitarán la escritura de código:

- Better Comments: Comentarios coloreados de todo tipo. Personalmente he instalado este plugin para poder documentar de una manera mucho más llamativa toda la documentación y hacerla menos pesada.
- Colorize: para destacar los colores seleccionados en CSS. Es complicado memorizar colores hexadecimales, por lo que es útil tener un selector gráfico de los mismos.
- Mithril Emmet: son snippets inteligentes para crear código de manera rápida en HTML, CSS y Javascript.
- Prettier - Code formatter, nos permite formatear e indentar todo nuestro código, tanto html, js, css. Es instalable como dependencia, pero por el momento no hemos considerado esta opción y hemos preferido adaptarlo en el plugin de nuestro IDE.
- ES7 React/Redux/GraphQL/React-Native snippets, extensiones simples para React, Redux y Graphql en JS/TS con sintaxis ES7.

- ESLint: La extensión utiliza la biblioteca ESLint instalada en la carpeta abierta del espacio de trabajo. Si la carpeta no proporciona una, la extensión busca una versión de instalación global.
- Tailwind CSS IntelliSense: mejora la experiencia de desarrollo de Tailwind proporcionando a los usuarios de Visual Studio Code funciones avanzadas como el autocompletado, el resaltado de sintaxis y el linting.

4. Desarrollo.

Para abordar el desarrollo propiamente dicho lo primero que haremos es iniciar git en nuestro proyecto con el siguiente comando:

```
git init
```

Después añadiremos todos los cambios y haremos commit de los mismos:

```
git add *
git commit -m "first commit"
```

Cambiaremos de rama:

```
git branch -M main
```

Accederemos a Github y crearemos un nuevo repositorio remoto e introduciremos por consola:

```
git remote add origin https://github.com/ansango/band-uoc-tailwind/
```

Por último haremos push del commit establecido:

```
git push -u origin main
```

Una vez creado nuestro repositorio remoto y configurado git podremos empezar a escribir código. Para abordar correctamente este proyecto hemos establecido las siguientes fases:

- HTML: en primer lugar vaciamos el contenido que tengamos en nuestra etiqueta <body> y lo sustituimos por lo siguiente:

```
<body>
  <div id="root"></div>
  <script src=".//assets/scripts/main.js"></script>
  <!-- do not remove this line! -->
</body>
```

Esto importará en último lugar el script que carga react y cogerá el div con id root que renderizará nuestra aplicación React.

- Arquitectura CSS:

- Módulos ITCSS: aquí llevaremos la arquitectura ITCSS explicada en el apartado metodología, que nos ayudará a ir creando las clases selectoras de más reutilizable a menos.
- Objetos OOCSS: estableceremos la abstracción de objetos y la creación de componentes.
- Atomización de componentes con TailwindCSS y Sass: hemos hecho toda la abstracción de componentes mediante clases atomizadas en sass, utilizando @apply y agregando dentro de Sass las jerarquías correspondientes. Esto nos da una super estructura muy mantenible. Un ejemplo de componente mantenable y reutilizable utilizado en Sass, Tailwind y React sería el siguiente:

// SASS:

```
/* -----
#TICKET
\----- */
```

```
.ticket {
  @apply mb-5;
  & > div {
    @apply flex justify-between p-4 border border-gray rounded-md;
```

```

    }
    & p {
        @apply mb-4;
    }
    & button {
        @apply bg-primary rounded-md px-3 py-2 text-white;
    }
    & span {
        @apply text-primary;
    }
}

```

// REACT

```

const Ticket = (props) => {
    let appoint = props.appoint;
    let date = `${appoint.date.month} ${appoint.date.day}`;
    let dayWeek = `${appoint.date.dayWeek}`;
    let country = `${appoint.location[1].region}`;
    let time = `${appoint.hour}`;
    let place = `${appoint.location[0].place}`;
    let event = `${place} | ${time}`;
    return (
        <div className="ticket">
            <div>
                <div>
                    <p>{date}</p>
                    <p>{event}</p>
                    <p>{country}</p>
                    <button>Tickets</button>
                </div>
                <span>
                    <FaTicketAlt size={20} />
                </span>
            </div>
        </div>
    );
}
```

```
};

Ticket.propTypes = {
    appoint: PropTypes.object.isRequired,
};

export default Ticket;
```

- Javascript: desde el main.js empezaremos a estructurar nuestra arquitectura en React. En nuestro caso, hemos creado una dependencia “App” que contiene un index.js que exporta por defecto el componente App que se renderiza en main.js:

```
import React from "react";
import ReactDOM from "react-dom";
import "../styles/main.scss";
import App from "./App";
```

```
ReactDOM.render(<App />, document.getElementById("root"));
```

A partir de aquí vamos haciendo code splitting y fragmentando en dependencias Components, Data y Views. La dependencia Components contendrá así mismo otras que albergarán todos los componentes reutilizables de nuestra UI.

La dependencia Views contendrá las vistas que se renderizan en el Router de nuestra aplicación, así podremos delegar el trabajo en vistas y componentes y estas podrán reutilizar dichos componentes con parámetros dinámicos. Esto nos ahorra escribir mucho código html.

Por último Data contiene todos los datos que se utilizarán en componentes y vistas, estos podrían venir de una base de datos, es simplemente un mockeo de datos.

Hemos mejorado la estructura del router para hacer lazy loading agnóstico, y atomizado cada componente y cada vista en la medida de lo posible. De manera que el código es simplemente declarativo a cada paso. Podemos

parametrizar Títulos de página, Tarjetas, Banners, Vistas y un sin fin de componentes.

La versatilidad de React y Tailwind en su conjunto es infinita. Construir UI es sumamente organizado y más sencillo, además de mantenible. Todo esto unido a la legibilidad de Sass, hace que el código pueda evolucionar sin sobresaltos por problemas de arquitectura.

5. Compilación para producción.

Después de haber realizado la fase de codificación, llevaremos a cabo nuestro primer despliegue de la aplicación. Para ello debemos verificar que todo ha ido bien compilando el proyecto. Ejecutaremos en la raíz del proyecto el siguiente comando:

npm run build

Este comando nos generará en la raíz del proyecto, una dependencia llamada **dist**, donde podremos ver toda la compilación. Podremos ver que todo funciona abriendo el index.html en cualquier navegador. Esta dependencia está lista para ser servida en un servidor.

Cabe añadir que utilizando React Router y la configuración que tenemos por defecto (BrowserRouter), si abrimos el index.html en local, el cambio entre vistas no funcionará, ya que BrowserRouter está preparado para un entorno de webservice. Si queremos ver que todo funciona correctamente incluyendo el Router, tendremos que cambiar el tipo de router por HashRouter o Memory Router. Esto se debería hacer en App/index.js de la siguiente manera:

cambiamos:

```
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";
```

Por:

```
import { HashRouter as Router, Switch, Route } from "react-router-dom";
```

O:

```
import { MemoryRouter as Router, Switch, Route } from "react-router-dom";
```

Volvemos a ejecutar **npm run build**, y ahora si, en dist/index.html podremos navegar entre rutas sin problema.

6. Publicación.

Por último una vez compilado el proyecto y viendo que el proceso ha ido correctamente, vamos a desplegarlo en un servidor.

Para ello hemos utilizado **Netlify**, que nos ayudará a automatizar el sistema de despliegues de sitios estáticos.

Antes de pushear los cambios en el repositorio, añadiremos un pequeño archivo de configuración para **Netlify**, que nos corregirá el error por defecto al navegar en rutas no existentes en la aplicación (404 de Netlify por nuestra vista 404). Eso sucede en cualquier SPA. Por lo que haremos lo siguiente:

- Crearemos un archivo “netlify.toml” en la raiz del proyecto.
- Copiamos y pegamos lo siguiente:

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

Por aclarar esta parte, comentar que en el componente App (App/index.js), donde llamamos al router, tenemos un componente para todas aquellas rutas que no sean las que están definidas (devuelve un 404 al uso):

```
const NotFound = React.lazy(() => import("./Views/404/index"));
```

...

```
<Route path="/" render={() => <NotFound />}></Route>
```

A continuación deberemos hacer push de los últimos cambios. Para configurar **Netlify**, seguiremos los siguientes pasos:

1. Crearemos un nuevo sitio seleccionando el botón: “**New site from git**”
2. En “**Continuous Deployment**” seleccionaremos **Github**.
3. De la lista de repositorios seleccionaremos en que creamos anteriormente.
4. Seleccionamos la rama **main**.
5. Escribimos como comando de build: **npm run build**.
6. Escribimos como directorio de publicación: **dist/**
7. Y por último seleccionamos el botón “**Deploy Site**”

5. Publicación.

Como comentamos en el último subpunto del punto anterior, después del despliegue tendremos accesible nuestra aplicación en una url real.

En este caso hemos generado la siguiente en **Netlify** desde el repositorio de **Github**:

<https://github.com/ansango/band-uoc-tailwind/>

<https://band-uoc-tailwind.netlify.app/>

El resultado final es el siguiente:

!undefined

X

band
tour

A BEAT BAND



!undefined

We are not undefined an electronic collective from Berlin, Germany. We make noise and beautiful things. For bookings or any other information you can send us an email to:

info@notundefined.com

Streaming

- Spotify
- Apple
- youtube

Next date

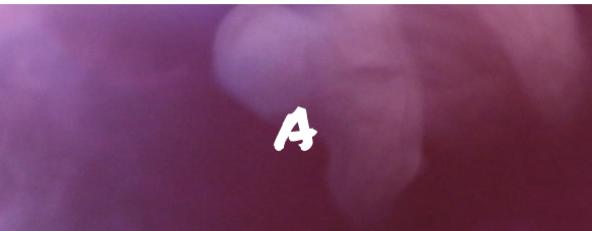
Kindl-Bühne Wuhlheide, Berlin, Germany

Saturday 02 September 2021

Checkout all new dates!

!undefined

band
tour



!undefined

X

band
tour



HELLO WE ARE...

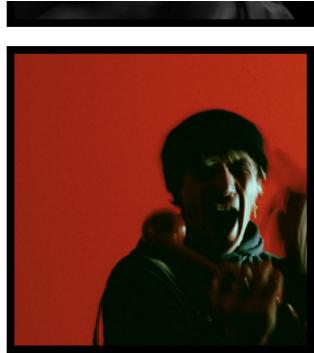


"Not Undefined is generous with moments like this tracks gather steam and reveal fresh layers as they move forward and they're what make the record a captivating listen. "

—Joe Colly, Pitchfork

"Fusing elements of free jazz, breakbeat, acid house, dub, ambient, and more, the Not Undefined first full-length album is simultaneously comforting and destabilizing."

—Nathan Smith, The Guardian



!undefined

X

band
tour



WE WERE BORN IN BERLIN, WE MAKE NOISE, WE LIKE NOISE, WE CAN'T STOP MAKING NOISE. SOMETIMES WE DO QUIETER THINGS, BUT ONLY WHEN WE HAVE A HANGOVER ON FRIDAYS.

COVID HAS KEPT US FROM TOURING, BUT WHEN IT'S OVER, YOU CAN FIND

Jun 2 Thu



Charles Krug Winery | 19:30

St. Helena, CA, United States

[Tickets](#)

St. Helena, CA, United States

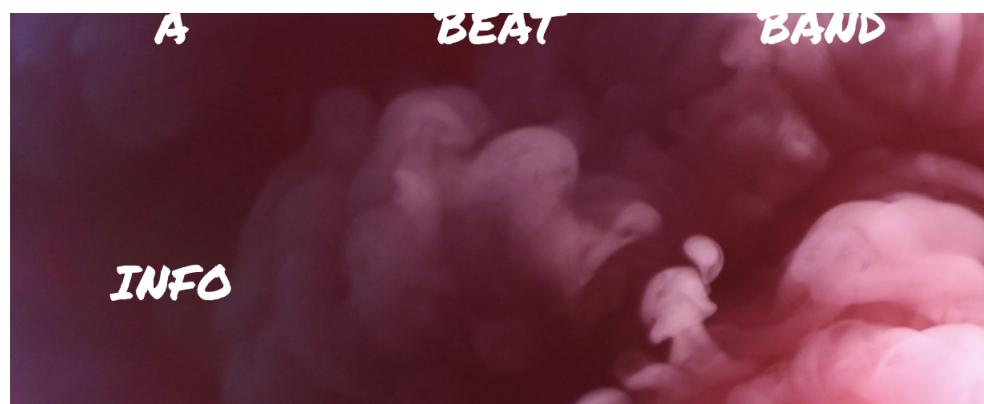
[Tickets](#)



!undefined

band tour

A BEAT BAND



Streaming

- Spotify
- Apple
- youtube

Next date

Kindl-Bühne Wuhlheide, Berlin,
Germany

Saturday 02 September 2021

[Checkout all new dates!](#)

!undefined

We are not undefined an electronic collective from Berlin, Germany.
We make noise and beautiful things. For bookings or any other
information you can send us an email to:

info@notundefined.com

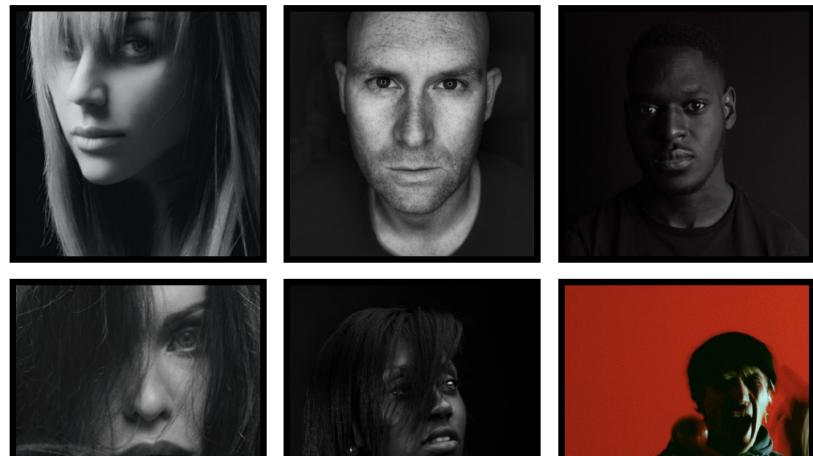
! undefined

band tour

HELLO WE ARE...

"Not Undefined is generous with moments like this tracks gather steam and reveal fresh layers as they move forward and they're what make the record a captivating listen."
—**Joe Colly, Pitchfork**

"Fusing elements of free jazz, breakbeat, acid house, dub, ambient, and more, the Not ! Undefined first full-length album is



MAKING NOISE. SOMETIMES WE DO QUIETER THINGS, BUT ONLY WHEN WE HAVE A HANGOVER ON FRIDAYS.

COVID HAS KEPT US FROM TOURING, BUT WHEN IT'S OVER, YOU CAN FIND US IN BASEMENTS, GARAGES, CHURCHES, ABANDONED CHURCHES, DISCOTHEQUES, COUNTRY BARS, POSH PLACES, PUNK PLACES AND WHEREVER THERE'S A PLUG AND WIFI.

WE LIKE BOOKS, IPA PUNK BEER, BIKES, SKATEBOARDING AND BUSTING TYMPANI.

WE HOPE TO SEE YOU SOON. XXX

Streaming

-  Spotify
-  Apple
-  youtube

Next date

Kindl-Bühne Wuhlheide, Berlin, Germany

Saturday 02 September 2021

[Checkout all new dates!](#)

!undefined

We are not undefined an electronic collective from Berlin, Germany. We make noise and beautiful things. For bookings or any other information you can send us an email to:

info@notundefined.com

!undefined

[band](#) [tour](#)



Jun 2 Thu

Charles Krug Winery | 19:30

St. Helena, CA, United States

[Tickets](#)



Jun 2 Thu

Charles Krug Winery | 19:30

St. Helena, CA, United States

[Tickets](#)



Jun 2 Thu

Charles Krug Winery | 19:30

St. Helena, CA, United States

[Tickets](#)



Jun 2 Thu

Charles Krug Winery | 19:30

St. Helena, CA, United States

[Tickets](#)

	Jun 2 Thu	Charles Krug Winery 19:30	St. Helena, CA, United States	Tickets
	Jun 2 Thu	Charles Krug Winery 19:30	St. Helena, CA, United States	Tickets
	Jun 2 Thu	Charles Krug Winery 19:30	St. Helena, CA, United States	Tickets

