DOCUMENTACIÓN TECNOLOGIA MULTIMEDIA

Web Monumentos de España

ALUMNOS:

- Adrián Bennasar Polzin – 41 54 23 28 G
- Álvaro Bueno López – 45 18 87 64 C

PROFESOR: Antoni Bibiloni Coll.

ASIGNATURA: 21755 – Tecnología Multimedia

CURSO: 2021-2022

Índice

1. Introducción ..... Pág. 3
2. Servicio de hosting ..... Pág. 4
3. Estructura de los archivos ..... Pág. 5
4. Fichero JSON ..... Pág. 6
5. Proceso de maquetación y diseño gráfico Pág.7
6. Estructura de la web ..... Pág. 8
   6.1. Página principal ..... Pág. 8
      6.1.1. Funcionalidad y código relacionado ..... Pág. 8
      6.1.2. Código JavaScript para las funcionalidades ..... Pág. 11
   6.2. Página de monumento específico ..... Pág. 25
      6.2.1. Funcionalidad y código relacionado ..... Pág. 25
      6.2.2. Código JavaScript para las funcionalidades ..... Pág. 26
   6.3. Página de rutas ..... Pág. 36
      6.3.1. Funcionalidad y código relacionado ..... Pág. 36
      6.3.2. Código JavaScript para las funcionalidades ..... Pág. 38
7. Web-responsive. Evaluación de la web en varios dispositivos ..... Pág. 50
8. Bootstrap ..... Pág. 64
9. Plantillas/Temas utilizados ..... Pág. 65
10. Librerías utilizadas ..... Pág. 66
11. APIs utilizadas ..... Pág. 67
12. Información de otro grupo utilizada ..... Pág. 68
13. Web Semántica ..... Pág. 69
14. Auditoría de la página web ..... Pág. 80
15. Opinión sobre la práctica. ..... Pág. 87
16. Autoevaluación. ..... Pág. 88
17. Conclusiones ..... Pág. 89

1. Introducción

URL DE LA PRÁCTICA:

https://practicatecmmpc.netlify.app/

REPOSITORIO GITHUB:

https://github.com/AlvaroBueno99/practicaTecMMPC

El tema de la página web es monumentos de España. La web muestra información de los monumentos más importantes de España. Esta información es principalmente el nombre del monumento, año de construcción, localización, historia relacionada, imágenes, etc, es decir todo lo relacionado con la cultura. Cada monumento va acompañado de un vídeo.

Esta información está almacenada en un fichero JSON almacenado en nuestro servidor hosting.

La web se divide en tres páginas: la página principal, la página de monumento específico y la página de rutas que se explicarán en detalle en los siguientes apartados.

Se trata de una página destinada a informar a los usuarios que acceden a ella.

2. Servicio de hosting

El servicio de hosting que hemos elegido es Netlify. En este servicio hemos creado un proyecto y lo hemos enlazado a un repositorio GitHub dedicado a esta práctica. De esta manera la web utiliza el código de este repositorio y se actualiza de manera dinámica cada vez que cambiamos el código del repositorio (Ilustración 1).

Nos hemos decidido por este servicio de hosting debido a sus buenas críticas y a las facilidades que nos da al permitir esa sincronización con GitHub

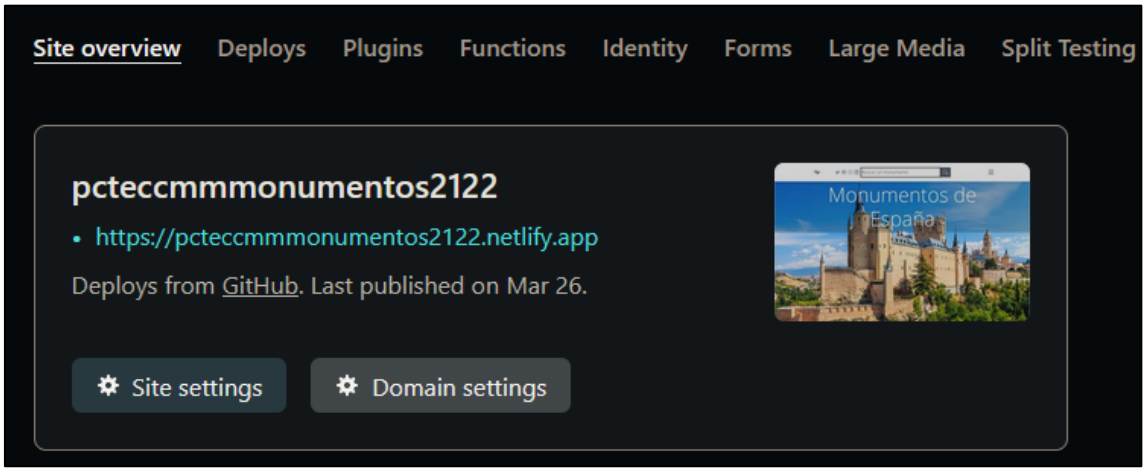


Ilustración 1: Página host con la web creada

Hemos escrito un fichero .TOML para desactivar el bloqueo de peticiones que se realiza por defecto por CORS (Ilustración 2).

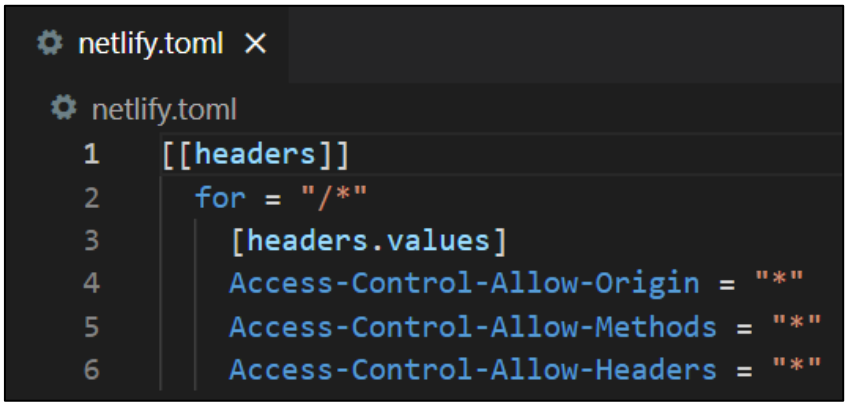


Ilustración 2: Fichero .TOML creado

3. Estructura de los archivos

En el nivel principal del directorio tenemos los ficheros .HTML, el fichero netlify.TOML, el fichero .htaccess que permite reescribir la URL de las páginas para eliminar su extensión y el fichero .gitignore para evitar subir al repositorio GitHub aquellos ficheros o carpetas que no nos interesen. Por otra parte, tenemos las siguientes carpetas, una para cada tipo de archivo utilizado:

- css
- img
- js
- json
- videos

También tenemos las siguientes carpetas relacionadas con Bootstrap:

- bootstrap-5.1.3-dist
- boostrap-icons

Además, en el nivel principal también tenemos un favicon.ico que se corresponde con el icono de la página web que aparece en la pestaña del navegador.

4. Fichero JSON

El fichero JSON, llamado “monumentos.json”, consiste en un array identificado por la palabra clave *monumentos* y en su interior se encuentran varios objetos (uno por cada comunidad autónoma) y a su vez cada comunidad consiste en un array que contiene todos los monumentos de esta.

Todas las propiedades tienen el mismo nombre que en la página schema.org.

Las propiedades de cada monumento son las siguientes:

- Name: es el nombre oficial del monumento en el idioma español.
- Identifier: identificador único del monumento. Sirve para programar funcionalidades en Javascript.
- Image: un array con 3 imágenes del monumento. Se ha intentado que cada una muestre un ángulo diferente de la estructura.
- Description: descripción general del monumento. Incluye solo la información más importante.
- Latitude: latitud del monumento. Está relacionado con su posición geográfica.
- Longitude: longitud del monumento. Está relacionado con su posición geográfica.
- Address: dirección oficial del monumento en el idioma español. Obtenida de la página Google Maps.
- yearBuilt: año en que se construyó el monumento. En algunos casos no se ha puesto el año exacto en que se empezó a construir el monumento, sino un valor medio entre cuándo se empezó a construir y cuando se terminó.
- Video: un array que contiene 2 vídeos, cada uno codificado de una manera diferente con la intención de que el mayor número de navegadores posibles puedan procesar al menos uno de los dos vídeos.

No hemos implementado un JSON de comentarios ya que los hemos implementado utilizando *localStorage* debido a que nuestro servidor de hosting no permite el uso de PHP.

5. Proceso de maquetación y diseño gráfico.

Para realizar el proceso de maquetación y diseño gráfico de la web, es decir, hacer una primera versión de la web a nivel visual para hacernos una idea de cómo será antes de pasar a programarla con HTML y CSS, hemos utilizado las siguientes herramientas gratuitas:

- Mockplus
- Mogups

La siguiente imagen muestra un ejemplo de maquetación que se realizó al principio de la práctica:

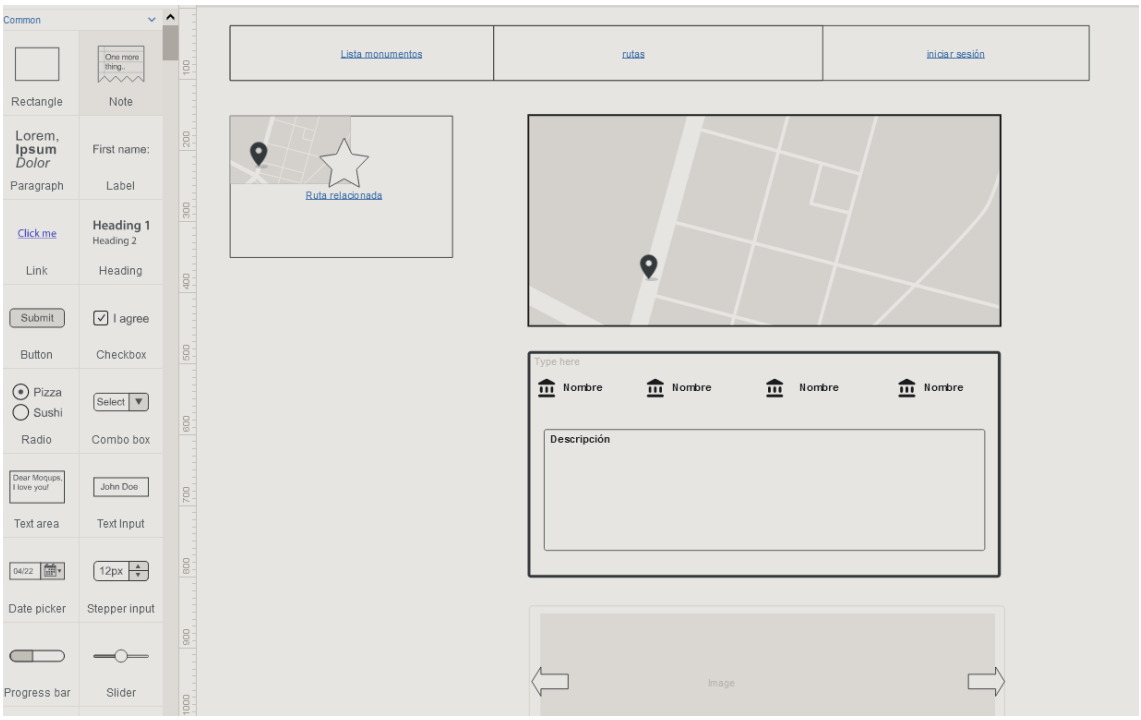


Ilustración 3: Esquema de la página realizado con la aplicación Mogups

6. Estructura de la web

6.1. Página principal

6.1.1. Funcionalidad y código relacionado

La página principal consiste principalmente en exponer varios monumentos. Para construir esta página, no hemos utilizado ninguna plantilla completa, sino que hemos ido construyéndola de manera modular cogiendo varios elementos de varios ejemplos de internet, así como de la documentación oficial de Bootstrap. Por otra parte, utilizamos un tema de CSS, que consiste en un fichero CSS que modifica la apariencia por defecto de todos los módulos de Bootstrap, de esta manera conseguimos una apariencia agradable sin tener que diseñarla nosotros desde cero.

El primer elemento de la página principal consiste en la barra de navegación. Esta barra de navegación es compartida por todas las páginas de nuestra web, por lo tanto, solo la explicaremos una vez, en esta sección.

La barra de navegación tiene la característica de que es fija, es decir, cuando el usuario hace *scroll* verticalmente, la barra de navegación se mantiene en la parte superior. Esta barra contiene los enlaces a las otras páginas que forman la web.

Por otra parte, contiene una barra de búsqueda, donde el usuario puede introducir el monumento en concreto que desea visualizar en la sección de cartas. Esta barra cuenta con la función de autocompletado para mejorar la experiencia del usuario.

Finalmente, a la derecha del todo, contiene los iconos a las redes sociales de nuestra entidad, que son SVG. Además, el logo de la barra de navegación también es una imagen SVG.

Después de esto, lo primero que se visualiza al visitar la página es un *slider* que presenta la temática de la práctica y muestra varias imágenes de monumentos. En la parte superior del *slider* está el título principal de la página. A través de HTML y CSS, le hemos puesto una animación de manera que realiza un cambio de color horizontalmente con un intervalo de tiempo que podemos asignar a nuestro gusto. Los otros títulos están programados de la misma manera.

Cuando el usuario hace *scroll* hacia abajo, lo primero que se ve son dos gráficos dinámicos que muestran información general sobre los monumentos/comunidades. Estos gráficos se han implementado con la siguiente librería:

- *Highcharts*

El primer gráfico es un gráfico de tipo tarta. La información que muestra consiste en la distribución de monumentos en cada comunidad. Dedica un trozo del diagrama de tarta a cada comunidad, junto con el porcentaje de monumentos que contiene sobre el total de monumentos del JSON. Viene con algunas animaciones, como las siguientes:

- Al cargar el gráfico, realiza una animación circular de manera que se va formando el gráfico progresivamente en sentido horario.
- Al pasar por encima de una sección con el cursor, se resalta esta y se reduce intensidad del resto.
- Al hacer click en una sección, separa esta de las 2 secciones que tiene a sus lados y también la mueve hacia el exterior de la tarta.

El segundo gráfico es de tipo columna. Enseña la antigüedad media de los monumentos de cada comunidad. Cada comunidad tiene una columna asociada.

Al igual que el gráfico de tarta, viene con algunas animaciones:

- Al cargar el gráfico, realiza una animación de manera que las barra aparecen del eje X y van creciendo en el eje vertical progresivamente.
- Al pasar el cursor por encima de una columna, aparece un rectángulo resaltado alrededor de esta, así como un cuadro de texto con el nombre de la comunidad y la antigüedad media de esta.

Ambos gráficos tienen un botón en la esquina superior derecha, que al hacer click en él hace aparecer una tabla desplegable debajo de los gráficos, ofreciendo una manera diferente de presentar la misma información al usuario.

Cabe destacar que al hacer *scroll* aparece en la zona inferior derecha un botón con un icono SVG que permite al usuario volver a la parte superior de la página haciendo click sobre este.

Inmediatamente después de los gráficos, ya se presenta la información de los monumentos a través de un catálogo formado por *cards* de Bootstrap.

Las cartas están diseñadas de la siguiente manera:

Cuando el usuario pasa con el cursor por encima de una carta, se aplican los siguientes cambios a la apariencia de la carta:

- Se realiza un *zoom* en la imagen de la carta.
- Se realiza un *zoom* de la carta entera.
- Se resaltan los bordes con un sombreado de color azul claro.

Contienen un botón que permite acceder a la página del monumento en concreto que la carta contiene. También tienen un botón de favoritos, representado por el icono de un corazón, que permite al usuario poner y/o quitar monumentos de un conjunto de favoritos.

A la izquierda de la sección de las cartas, hay el menú lateral. Este menú contiene los botones que permiten al usuario realizar la gran mayoría de filtros de la página principal.

Para filtrar por comunidad, es decir, ver solo las cartas de una comunidad en concreto, el usuario debe utilizar el menú desplegable que contiene un botón por comunidad.

Para filtrar por nombre y antigüedad, el usuario debe utilizar los botones de las *checkbox* y puede elegir el orden ascendente o descendente.

Contiene un botón para los filtros generales y otro para el filtro de favoritos.

En la parte más inferior de la página se encuentra el *footer*. De la misma manera que con la barra de navegación, solo explicaremos el *footer* en esta sección, ya que es compartido de manera estática por todas las páginas de la web.

Este *footer* también contiene los iconos de las redes sociales que son SVG, al igual que la barra de navegación.

Por otra parte, muestra la siguiente información:

- Una descripción general de la página.
- Links de interés relacionados con la práctica, como un link a Bootstrap.

- Información típica de contacto (ficticia).

Las funcionalidades de esta página son:

- Buscar un monumento con la *Search Bar*:
- Filtrar monumentos por comunidad autónoma.
- Ordenar monumentos por nombre o por antigüedad.
- Ordenar monumentos de forma ascendente o descendente, de manera combinada con el filtro anterior.
- Hacer clic en el botón de un monumento para visitar la página de ese monumento en concreto.
- Añadir o quitar un monumento del conjunto de favoritos.
- Mostrar o esconder las tablas desplegables de los gráficos dinámicos.

Cabe destacar que la información semántica relacionada con el *footer* y el *navbar* se genera de forma estática en el <head> de index.html

6.1.2. Código JavaScript para las funcionalidades

- **getJSONFile():**

```
async function getJSONFile() {
  // lo guarda en la memoria principal (RAM)
  const response = await fetch("./json/monumentos.json");
  const { monumentos } = await response.json();
  dataJSON = monumentos;
}
```

Ilustración 4: Función getJSONFile()

Esta función aparece en varios ficheros JavaScript por lo que solo la explicaremos una única vez.

El propósito de esta función es obtener los datos del fichero .json que contiene toda la información relacionada con los monumentos y guardarla en una constante en memoria principal para poder acceder a los datos a través de esta constante desde otras funciones del fichero.

Se ha utilizado la función '*fetch*' que es una de las maneras en que JavaScript permite la obtención de datos de un JSON.

- **generateJSONId(monumento):**
  Es la función con la que se genera la información semántica de la página principal relacionada con los monumentos del catálogo de forma dinámica. Se explica con más exactitud en el apartado 13.

- **generateCard(monumento):**

```
function generateCard(monumento) {
  let añoAjustado = (ajustarAño(`${monumento.yearBuilt}`));
  const card =/* HTML */

    `
    <div itemscope itemtype="https://schema.org/LandmarksOrHistoricalBuildings" class="col-auto m-4">
    <div class="card overflow zoom hoverCard" id="card-${monumento.identifier}" style="width: 18rem;">
      <img class="card-img-top" itemprop="image" src="${monumento.image[0]}" alt="Card image cap" />
      <div class="card-body align-items-center">
      <div id="monumentProperties">
        <h5 itemprop="name" class="card-title">${monumento.name}</h5>
        <div id="direccion">
        <p itemprop="address" class="card-text" id="mainCardText">${monumento.address}</p>
        </div>
        <div id="año">
        <div itemscope itemtype="https://schema.org/Accommodation">
          <p itemprop="yearBuilt" class="card-text" id="mainCardText">Año de construcción: ${añoAjustado}</p>
        </div>
        </div>
        </div>
        <a href="/monumento.html?identifier=${monumento.identifier}" class="btn btn-primary" id="cardBtn">Ver monumento</a>
        <button type="button" data-id="${monumento.identifier}" class="favorite-btn btn btn-outline-info ${monumento.isFavorite
          ? " is-favorite" : "" }" onclick="resultsDelegation(event)">
          ${monumento.isFavorite ? "♥" : "♡"}
        </button>
      </div>
    </div>
  </div>
    `;

  return card;
}
```

Ilustración 5: Función generateCard(monumento)

Esta función se encarga de generar dinámicamente las *cards* de la página principal. Recibe un monumento como parámetro, de manera que sea cual sea el monumento, podemos acceder a los valores de sus propiedades mediante la notación *${monumento.propiedad}*.

Primero de todo, se ajusta el año de construcción del monumento ya que si fue construido en la época a.C, el valor de *yearBuilt* es negativo. Con este ajuste, en lugar de aparecer el año con valor negativo en la carta, aparece con el formato a.C, en positivo.

La estructura del código HTML se coloca entre 2 caracteres *backtick* y se guarda en una constante llamada "*card*". Los valores que deben cambiar dinámicamente se calculan con la notación ${} mencionada anteriormente.

Finalmente, la función devuelve esta constante que contiene el código HTML con valores dinámicos preparado para ser escrito en el documento .html correspondiente.

- **ajustarAño(año):**

```
function ajustarAño(año) {
  if (año < 0) {
    año = año * -1;
    return año + " a.C"
  }
  return año;
}
```

Ilustración 6: Función ajustarAño(año)

Se encarga de ajustar el año de monumentos que fueron construidos en la época a.C para que el año de construcción aparezca con el formato a.C en lugar de con valor negativo.

- **getQueryParams():**

```
function getQueryParams() {
  const search = new URLSearchParams(window.location.search);
  return search.get("term") || "";
}
```

Ilustración 7: Función getQueryParams()

El propósito de la función es guardar en una constante el valor de la URL en un momento dado. A partir de esta constante, se pueden obtener partes específicas de esta URL, como el valor de "*term*" en concreto, que después es utilizado en otras funciones.

Hace uso de la función nativa *URLSearchParams* con parámetro window.location.search.

- **setHighchartsOptions():**

```
function setHighchartsOptions() {
  // Radialize the colors
  Highcharts.setOptions({
    colors: Highcharts.map(Highcharts.getOptions().colors, function (color) {
      return {
        radialGradient: {
          cx: 0.5,
          cy: 0.3,
          r: 0.7,
        },
        stops: [
          [0, color],
          [1, Highcharts.color(color).brighten(-0.3).get("rgb")], // darken
        ],
      };
    }),
  });
}
```

Ilustración 8: Función setHighchartsOptions()

Es la función encargada de establecer los valores de variables compartidas importantes de la librería *Highcharts* que son utilizados por todos los gráficos que se utilicen. Utiliza la función propietaria de *Highcharts setOptions*().

- **generatePieChart():**

Imágenes de la función disponibles en Drive.

Esta función se utiliza para generar el gráfico de tarta de la página principal con los valores obtenidos de manera dinámica. Esto significa que, si un valor del JSON es cambiado, la próxima vez que se cargue el gráfico, este valor estará actualizado en este, ya que se obtiene a través de código JavaScript.

En el array jsonArray se almacenan los valores del JSON, que han sido obtenidos a través de una iteración sobre este con la funciones nativas *variable.forEach* y *Object.keys*. Una vez rellenado este array, se pasa como valor a la función *Highcharts.chart* para que el gráfico utilice los valores contenidos en este array. Los datos se dejan formateados como la librería pide, es decir, en formato de objeto JavaScript.

La función *Highcharts.chart* es propietaria de *Highcharts* y se utiliza en todos los gráficos que se quieran construir, por lo tanto, solo la explicamos una vez:

Consiste en la función principal de la librería donde para cada gráfico se indican tanto los valores que se quieren utilizar como los diferentes parámetros relacionados con el formato y la estética del gráfico.

- **generateColumnChart():**

  Imágenes de la función disponibles en Drive.

  Esta función es la misma idea que la anterior, pero se encarga de generar un gráfico de columnas, por lo que cambian los parámetros propios de este tipo de gráfico respecto del gráfico de tarta.

- **Filter(displayFavorites=false, search=false):**

  Imágenes de la función disponibles en Drive.

  Esta función contiene el código que implementa 4 de las funcionalidades relacionadas con filtrar información en la página principal:

  - **Favoritos:** se obtiene el conjunto de monumentos favoritos almacenado en *localStorage* a través de la función nativa *localStoarage.getItem(key)* y se formatea con la función nativa

JSON.parse para guardarlo en una variable JavaScript llamada "favorites".

- **Seleccionar comunidad específica:** se obtiene el valor del elemento *Select* que el usuario puede elegir haciendo click en cualquiera de las opciones posibles, que consisten en las diferentes comunidades. Este valor se obtiene a través de la función nativa document.getElementById("id").value.

- **Filtrar por nombre o por antigüedad:** a través de la siguiente función nativa:

  document.querySelector('input[name="filter"]:checked').value

  se obtiene el valor del filtro seleccionado en todo momento por el usuario.

- **Ordenar de manera ascendente o descendente:** a través de la siguiente función nativa:

  document.querySelector('input[name="order"]:checked').value;

  se obtiene el valor del elemento checkbox seleccionado en todo momento por el usuario.

  Una vez obtenidos todos los valores, se procede de la siguiente manera:

  Primero se itera sobre el fichero JSON, concretamente la variable "dataJSON", y se añaden al array "monumentsArray" solo los monumentos que coinciden tanto con el valor del Select como con el conjunto de monumentos favoritos obtenido.

Una vez “monumentsArray” se ha rellenado de esta manera, se comprueba si el usuario ha seleccionado el elemento *searchBar*, por lo tanto, se utiliza la función *_.filter* propietaria de la librería lodash pasándole “monumentsArray” como parámetro para filtrar sobre este array con el valor “*term*” perteneciente a la URL de la página que se establece dinámicamente al introducir texto en la *SearchBar*.

Finalmente, se utiliza la función _.orderBy también propietaria de la librería lodash, pasándole como parámetro “monumentsArray” y los valores del filtro que permite las opciones ‘nombre’ y ‘antigüedad’ y la checkbox que permite las opciones ‘ascendente’ y ‘descendente’, guardados en las variables “filter” y “order”, respectivamente.

Al mismo tiempo que se ha filtrado completamente este array de monumentos, ya se llama a función *generateCard(monumento)* dentro de _.orderBy para generar el código de las cartas de la página principal, que serán únicamente cartas de monumentos que han coincidido con el conjunto de filtros establecidos por el usuario, y este código se almacena en una variable que hemos llamado “*cards*”. Al mismo tiempo, también se llama a la función *generateJSONld* para generar la información semántica relativa a todas las cartas de monumentos que se muestran.

Finalmente, cuando se termina con la iteración y esta variable “cards” ya contiene todo el código de los monumentos que han coincido con los filtros, se utiliza la función nativa document.getElementById(“id”) para acceder al elemento del fichero .html que contendrá estas cartas, y se introducen ahí a través de la función nativa *elemento.innerHTML*.

- **setSearch(id):**

```
function setSearch(id) {
  term = document
    .getElementById(id)
    .value.trim()
    .toLowerCase()
    .normalize("NFD")
    .replace(/[\u0300-\u036f]/g, "");
  filter(false, true);
  location.href = "#filaTituloCatalogo"; // sitúa al usuario en la sección de cartas.
}
```

Ilustración 9: Función setSerach(id)

Es la función que permite que el usuario busque un monumento a través de la search bar de la página principal.

Se encarga de establecer el valor de la variable global "term" mediante la propiedad .value de la función getElementById("id") a la cual le pasamos por parámetro el id de la search bar, obteniendo así el texto que el usuario ha escrito en esta. Este texto almacenado en la variable "term" se formatea para pasarlo a letras minúsculas mediante la función *.toLowerCase()* y quitar las tildes mediante la función *.normalize("NFD").replace(value),* de esta manera escriba lo que escriba el usuario, se pasará a un formato estándar de minúsculas sin tildes para que el código siempre reconozca el texto y se pueda comparar fácilmente con los nombres de monumentos encontrados en nuestro fichero JSON.

A continuación, se llama a nuestra función de filtrado *"filter(boolean, boolean)"* con el segundo parámetro establecido a **true**, de esta manera la función *filter* sabe que ha sido llamada desde esta función y cuando genera las cartas de los monumentos tiene en cuenta el texto introducido en la seach bar, al cual puede acceder con la variable "term" porque es global de este fichero, y entonces solo mostrará monumentos cuyo nombre coincida (o tenga ciertas letras en común) con ese texto.

Finalmente, para que sea más cómodo para el usuario, esta función utiliza la propiedad nativa *location.href* para situar al usuario directamente en la sección del catálogo de cartas, para que pueda ver inmediatamente las cartas que coinciden con el filtro sin necesidad de tener que hacer scroll vertical.

- **saveIntoDB():**

```
function saveIntoDB(monument) {
  const monuments = this.getFromDB();

  monuments.push(monument.identifier);

  // Add the new array into the localstorage
  localStorage.setItem("monuments", JSON.stringify(monuments));
}
```

Ilustración 10: Función saveIntoDB()

Esta función está relacionada con la funcionalidad de monumentos favoritos, y es la base que permite añadir un monumento al conjunto ya existente de monumentos almacenado en *localStorage*.

Primero se guarda en una constante(array) el conjunto de monumentos almacenados en *localStorage*, a través de otra función nuestra llamada *getFromDB*().

A continuación, se introduce el elemento pasado por parámetro en este array mediante la función nativa array.*push.* En este caso en concreto, introducimos solo la propiedad 'identifier' del monumento pasado por parámetro, ya que nos basta con el identificador para gestionar el conjunto de monumentos.

Finalmente se sobrescribe el ítem correspondiente de *localStorage* con este array actualizado que contiene el nuevo valor. Para indicar qué key de *localStorage* queremos actualizar, se pasa como parámetro a la función nativa *localStorage.setItem(key, array).* Este array es formateado

con la función nativa *JSON.stringify* para ser almacenado correctamente en *localStorage*.

- **removeFromDB(id):**

```
function removeFromDB(id) {
  const monuments = this.getFromDB();

  // Loop
  monuments.forEach((monumentID, index) => {
    if (id === monumentID) {
      monuments.splice(index, 1);
    }
  });
  // Set the array into local storage
  localStorage.setItem("monuments", JSON.stringify(monuments));
}
```

Ilustración 11: Función removeFromDB(id)

Es la misma idea que la función anterior, pero en este caso se elimina del conjunto de monumentos almacenados en *localStorage* el monumento cuyo identificador coincide con el identificador pasado como parámetro a esta función.

- **getFromDB():**

```
function getFromDB() {
  let monuments;
  // Check from localstorage

  if (localStorage.getItem("monuments") === null) {
    monuments = [];
  } else {
    monuments = JSON.parse(localStorage.getItem("monuments"));
  }
  return monuments;
}
```

Ilustración 12: Función getFromDB()

Función utilizada tanto en removeFromDB() como en saveIntoDB(). Simplemente se encarga de devolver el array de monumentos

almacenados en *localStorage* para que aquellas otras 2 funciones lo almacenen en una variable y apliquen código sobre ella.

- **resultsDelegation():**

```
function resultsDelegation(e) {
  e.stopPropagation();
  e.preventDefault();

  icon = document.querySelector(
    `#card-${e.target.dataset.id} > div > button > i`
  );

  // When favourite btn is clicked
  if (e.target.classList.contains("favorite-btn")) {
    if (e.target.classList.contains("is-favorite")) {
      removeFromDB(e.target.dataset.id);
      // Remove the class
      e.target.classList.remove("is-favorite");
      e.target.textContent = "♡";
    } else {
      // Add the class
      e.target.classList.add("is-favorite");
      e.target.textContent = "♥";
      // Get Info
      const cardBody = e.target.parentElement;

      const monumentInfo = {
        identifier: e.target.dataset.id,
      };

      // Add into the storage
      saveIntoDB(monumentInfo);
    }
  }
}
```

Ilustración 13: Función resultsDelegation(e)

Función encargada de modificar la clase de las cartas de la página principal para que de manera dinámica se clasifiquen las cartas en dos conjuntos, las favoritas y las no favoritas. Esto se hace de manera que las favoritas pertenecen a la clase “is-favorite” y el resto no.

Para actualizar de manera dinámica qué cartas pertenecen a la clase “is-favorite”, se configura con JavaScript el botón con el icono del corazón, de manera que a través del evento lanzado al hacer click en este botón se accede al elemento sobre el que actúa el botón mediante la propiedad nativa *e.target* y al mismo tiempo se accede al conjunto de clases de la carta con la propiedad nativa *element.classList*.

Si la clase “is-favorite” pertenece al conjunto se elimina y viceversa, de esta manera cada vez que se hace click en el botón si la carta era de un monumento favorito, dejará de serlo, y si no era favorita, pasará a serlo cuando el usuario ha hecho click en el botón.

Esto implica que cuando se lanza el evento de click en el botón, si la carta contenía la clase ‘is-favorite’, se elimina de *localStorage* el monumento asociado a esta carta mediante nuestra función *removeFromDB*(). Lo mismo ocurre en el caso contrario, en que el monumento asociado a la carta se añade a *localStorage* mediante nuestra función saveIntoDB().

- **autocomplete(inp):**

Función encargada de mostrar las opciones de autocompletado al escribir en la barra de búsqueda. Esta se utiliza en varios ficheros JavaScript por lo que solo se mencionará una única vez.

La función actúa de la siguiente manera: cuando el usuario escribe en la barra de búsqueda se genera un div que contiene tantos div como opciones posibles existen para autocompletar el texto que se ha introducido. De esta forma el usuario puede seleccionar la opción deseada haciendo click sobre ella, autocompletando el texto que había introducido inicialmente.

Esta función de autocompletado se basa en la disponible en la página oficial de W3Schools pero con una pequeña modificación. La función original recibe por parámetro el *input* (elemento que se verá afectado por

esta función, en este caso la barra de búsqueda) y un array que contiene todos los elementos que conformarán las diferentes opciones que se ofrecerán para autocompletar el texto introducido. En este caso se ha modificado de forma que el array de opciones se determina dentro de la propia función mediante la función "*completeMonumentsArray()*", la cual rellena un array con todos los monumentos contenidos en el JSON, siendo estos las posibles opciones de autocompletado. Además, también se ha comentado un fragmento del código que impedía poder ejecutar la búsqueda pulsando la tecla "*enter*".

- **completeMonumentsArray():**

```
function completeMonumentsArray() {
  var monumentsArray = [];

  dataJSON.forEach((comunidades) => {
    Object.keys(comunidades).forEach((nombreComunidad) => {
      comunidades[nombreComunidad].forEach((monumento) => {
        monumentsArray.push(
          `${monumento.name}`,
        )
      });
    });
  });
  return monumentsArray;
}
```

Ilustración 14: Función completeMonumentsArray()

Esta función se utiliza para rellenar un array con el nombre de todos los monumentos disponibles en el JSON.

- **irArriba():**

```
function irArriba(){
  $('.ir-arriba').click(function(){ $('body,html').animate({ scrollTop:'0px' },100); });
  $(window).scroll(function(){
    if($(this).scrollTop() > 0){ $('.ir-arriba').slideDown(600); }else{ $('.ir-arriba').slideUp(600); }
  });
}
```

Ilustración 15: Función irArriba()

Esta función utiliza la librería de JQuery para que, al pulsar un botón que aparece en la parte inferior derecha de la pantalla al hacer *scroll*, se suba automáticamente a la parte superior de la página web.

- **Window.onload = async function():**

```
window.onload = async function () {
  // ¿Lo tiene que hacer el usuario, o tiene que salir "inmediatamente" cuando cargue la pagina?
  // dentro del onload cuando hay que acceder al DOM "instantaneamente".
  await getJSONFile();

  pruebaObtencionDatosJSON();

  generateColumChart();
  setHighchartsOptions();
  generatePieChart();

  term = getQueryParams();

  filter(false, true);

  autocomplete(document.getElementById("searchBarMain"));

  irArriba();
};
```

Ilustración 16: Función Window.onload = async function():

Función que solo se ejecuta una vez la ventana del navegador se ha cargado. Se encarga de construir la página, ya que realiza llamadas a todas las otras funciones del fichero script.js que contienen cada una el código referente a un aspecto en concreto.

6.2. Página de monumento específico

6.2.1. Funcionalidad y código relacionado

La página del monumento en específico muestra información solo del monumento concreto seleccionado en la página principal.

Se compone principalmente de cinco elementos:

- Un *slider* que muestra tres fotos del monumento

- Una *card* con texto informativo sobre el monumento, especificando datos como el nombre, la dirección o la posición geográfica concreta, además de una descripción general del monumento. Esta información va acompañada de diferentes iconos SVG.

- Un mapa que muestra la localización del monumento. Este mapa está implementado con la API *Leaflet*. El mapa contiene dos botones en la esquina superior izquierda que permiten que el usuario aumente o disminuya el zoom sobre el mapa. Al hacer click sobre el mapa aparece un mensaje de la página que recomienda permitir la geolocalización y a continuación, en caso de que el permiso no haya sido previamente dado, lo solicita. En caso de que el usuario acepte los permisos de ubicación se muestra su posición geográfica con un marcador verde utilizando la API Geolocation.

  Al hacer click sobre el marcador de la localización del mapa, aparece un cuadro de texto con el nombre del monumento.

- Un vídeo general que enseña el monumento. Los vídeos están todos dentro de etiquetas <video> donde ofrecemos dos contenedores diferentes: MP4 y WebM que a su vez están codificados con los siguientes códecs:
  - Códecs de vídeo:
    - MP4: H.264
    - WebM: VP8

- Códecs de audio:
  - MP4: MP3 / ACC
  - WebM: Vorbis

Tanto las conversiones a formato vídeo como los diferentes códecs se ha realizado con el siguiente conversor.

Cabe destacar que se ha hecho la combinación de códecs de audio MP3 y ACC para el contenedor MP4 comprobando antes que son válidos para los principales navegadores utilizando la página Can I Use

- En la parte izquierda de la página encontramos un botón que redirige directamente a la página de la ruta de la comunidad a la que pertenece el monumento.

- Paneles de tiempo:
  - Temperatura: muestra la temperatura y el tiempo en la zona del monumento. Se actualiza cada 10 minutos.
  - Humedad: es el mismo concepto que el otro panel, pero referente a la humedad.

    Ambos paneles están implementados con la API OpenWeather.

6.2.2. Código JavaScript para las funcionalidades

- **getJSONFile():**

  Es la misma función que aparece en otros ficheros .js y permite obtener los datos necesarios del fichero monumentos.json para la implementación de la página rutas.html. Se explica con más exactitud en el apartado 6.1.2.

- **getQueryParams():**

  Es la misma función que aparece en otros ficheros .js y permite modificar los parámetros de la URL necesarios para la implementación de la página rutas.html. Se explica con más exactitud en el apartado 6.1.2.

- **generateJSONId(monumento):**

  Es la función con la que se genera la información semántica del monumento de la página en concreto de forma dinámica. Se explica con más exactitud en el apartado 13.

- **generateSlider(monumento):**

```
function generateSlider(monumento) {
  const infoSlider =/* HTML */

    `
    <div  id="carouselMonumento" class="carousel slide" data-bs-ride="carousel">
    <div class="carousel-indicators">
      <button type="button" data-bs-target="#carouselMonumento" data-bs-slide-to="0" class="active" aria-current="true"
        aria-label="Slide 1"></button>
      <button type="button" data-bs-target="#carouselMonumento" data-bs-slide-to="1" aria-label="Slide 2"></button>
      <button type="button" data-bs-target="#carouselMonumento" data-bs-slide-to="2" aria-label="Slide 3"></button>
    </div>
    <div itemscope itemtype="https://schema.org/LandmarksOrHistoricalBuildings" class="carousel-inner">
      <div class="carousel-item active">
        <img itemprop="image" src="${monumento.image[0]}" id="slide1">
      </div>
      <div class="carousel-item">
        <img itemprop="image" src="${monumento.image[1]}" id="slide2">
      </div>
      <div class="carousel-item">
        <img itemprop="image" src="${monumento.image[2]}" id="slide3">
      </div>
      <button class="carousel-control-prev" type="button" data-bs-target="#carouselMonumento" data-bs-slide="prev">
        <span class="carousel-control-prev-icon" aria-hidden="true"></span>
        <span class="visually-hidden">Previous</span>
      </button>
      <button class="carousel-control-next" type="button" data-bs-target="#carouselMonumento" data-bs-slide="next">
        <span class="carousel-control-next-icon" aria-hidden="true"></span>
        <span class="visually-hidden">Next</span>
      </button>
    </div>
  </div>
  `;
  return infoSlider;
}
```

Ilustración 17: Función generateSlider(monumento)

Esta función se encarga de generar dinámicamente el *slider* de la página monumento.html. En la constante “infoSlider” se almacena el código de la estructura HTML base del *slider*, sin embargo, los valores de los elementos se generan dinámicamente dependiendo del monumento que se introduce por parámetro.

Para generar los valores dinámicamente, se utiliza la notación *${monumento.propiedad}*. De esta manera, con una única función podemos generar el *slider* para cualquier monumento.

Las imágenes contenidas en el slider se obtienen con *${monumento.image[i]}*. En el fichero JSON cada monumento tiene un array con 3 imágenes, por lo tanto, con esta notación colocamos en el *slider* cada una de las 3 imágenes del array.

Finalmente se ejecuta un *return* de la variable “infoSlider”.

- **generateInformationCard(monumento):**

```
function generateInformationCard(monumento) {
  let antiguedad = (new Date().getFullYear() - `${monumento.yearBuilt}`);
  let latitud = +(Math.round(`${monumento.latitude}` + "e+5")  + "e-5");
  let longitud = +(Math.round(`${monumento.longitude}` + "e+5")  + "e-5");
  const infoCard =
    /* HTML */

    `
  <div class="card-header">

      <p class="datosCartaMon"><img src="img/iconos/landmark.svg" class="iconoPagMonum" alt=""><br> ${monumento.name}</p>
      <p class="datosCartaMon"><img src="img/iconos/calendar.svg" class="iconoPagMonum" alt=""> <br>${antiguedad} años de <br> antigüedad</p>
      <p class="datosCartaMon"><img src="img/iconos/place.svg" class="iconoPagMonum" alt=""> <br>${monumento.address}</p>
      <p class=" datosCartaMon"> <img src="img/iconos/map.svg" class="iconoPagMonum" alt=""> <br>
        (${latitud} / ${longitud})
        <meta content=${monumento.latitude} />
        <meta content=${monumento.longitude} />
        </p>
  </div>
  <div class="card-body" id="monumentCard">
      <h1 class="card-title">${monumento.name}</h1>
      <p class="card-text" id="monumentCardText"> ${monumento.description}
      </p>

  </div>
  `;

  return infoCard;
}
```

Ilustración 18: Función generateInformationCard(monumento)

Función para generar de manera dinámica la carta de información de cada monumento en la página monumento.html.

De manera similar a la función anterior, se almacena el código de la estructura base HTML en una constante y los valores de los elementos se generan con JavaScript en lugar de ser estáticos. De nuevo, se recibe un monumento y se accede a las propiedades de este mediante la notación ${monumento.propiedad}.

La mayoría de valores que se muestran son los que ya estaban establecidos en el fichero JSON, sin embargo, para calcular la antigüedad del monumento, que no está almacenada en el fichero JSON, lo que se

hace es restar el año de construcción (*monumento.yearBuilt*) a la fecha actual, la cual se obtiene mediante la función nativa *Date().getFullYear()*.

Finalmente se ejecuta un return de la variable “infoCard”.

- **generateMap(monumento):**

  Imágenes de la función disponibles en Drive.

  Función para generar de manera dinámica un mapa que contiene un marcador para el monumento de la página del monumento donde se encuentra el usuario y un marcador para la posición actual del usuario en el momento de cargar la página. El marcador del monumento consiste en un .svg de color azul, mientras que el del usuario es el mismo, pero de color verde.

  El mapa se genera con la API Leaflet. El mapa se carga con el enfoque centrado en la posición del monumento, por lo que, si el usuario se encuentra lejos de este, no verá el marcador verde a no ser que mueva el mapa.

  Los marcadores se generan con la función *L.marker* de Leaflet, que por una parte recibe como parámetro la latitud y longitud del monumento para el marcador azul, y por otra parte, recibe *position.coords.latitude* y *position.coord.longitude*, que son los valores relacionados con la API *Geolocation*, para el marcador verde. Los marcadores se añaden al mapa base mediante la función *addTo(map)* de Leafelt.

  La posición del marcador azul se calcula dinámicamente para cualquier monumento a través de la notación ${monumento.propiedad}, de manera similar a otras funciones. Concretamente, se utiliza *${monumento.latitud}* y *${monumento.longitud}* para obtener las coordenadas del monumento a través de estas 2 propiedades que tiene establecidas en el fichero JSON.

A este marcador azul se le añade un popUp al hacer click en él, mediante la función *bindPopup* de Leaflet. Este popup es rellenado dinámicamente para cualquier monumento mediante el código *${monumento.name}*

La posición del marcador verde se calcula mediante la API *Geolocation*, nativa de HTML. En el momento en el que el usuario hace click o arrastra sobre el mapa, la página muestra un mensaje recomendando activar los permisos de ubicación para mejorar la experiencia y a continuación se solicitan dichos permisos. En caso de que el usuario los acepte se muestra la ubicación con el marcador verde. En caso contrario no se muestra nada. Estos permisos se mantienen de una página de monumento en concreto a otra. El mensaje de la página web se mostrará una única vez por sesión en la página web y para ello se implementa con *SessionStorage*.

- **generateVideo(monumento):**

```
function generateVideo(monumento) {
  const infoVideo = /* HTML */

    `
      <video controls poster="${monumento.image[0]}" id="vid">
        <source src="${monumento.video[0]}" type="video/webm" />
        <source src="${monumento.video[1]}" type="video/mp4" />
        Su navegador no soporta los formatos de video.
      </video>
    `;

  return infoVideo;
}
```

Ilustración 19: Función generateVideo(monumento)

Esta función es la encargada de generar dinámicamente para cada monumento su vídeo correspondiente. Se hace uso de la API *Media*, nativa de HTML. Por lo tanto, se utiliza el tag <video> con el atributo src.

En valor del atributo src está establecido de manera dinámica con el valor ${monumento.video[i]}, de esta manera, con un único fichero monumento.html, el usuario puede ir a la página de cualquier monumento en concreto y se mostrará el vídeo de ese vídeo y no otro, ya que en la variable monumento estará guardado el vídeo correspondiente.

En el atributo type indicamos el contenedor de los vídeos. En nuestro caso, hemos codificado para cada monumento una versión de su vídeo con el contenedor WebM y otra versión con el contenedor MP4. En caso de que el navegador no pudiera cargar ninguno de los 2 vídeos, mostramos el mensaje “Su navegador no soporta los formatos de vídeo” al usuario.

- **getWeather(monumento):**

```
function getWeather(monumento) {
  let lat = `${monumento.latitude}`;
  let long = `${monumento.longitude}`;
  let API_KEY = "bd1dd16fb46e8746f274b42dd40b7008";

  let baseURL = `https://api.openweathermap.org/data/2.5/onecall?lat=

  $.get(baseURL, function (res) {
    let data = res.current;
    let temp = Math.floor(data.temp - 273);
    let condition = data.weather[0].description;
    let icon = data.weather[0].icon;
    let iconURL = `http://openweathermap.org/img/wn/${icon}@2x.png`;
    let humidity = data.humidity;

    $("#temp-main").html(`${temp}°C`);
    $("#condition").html(condition);
    $("#weatherIcon").attr("src", iconURL);
    $("#humidity").html(`${humidity} %`);

    $("#tituloTemperatura").html(`Tiempo en ${monumento.name}`);
    $("#tituloHumedad").html(`Humedad en ${monumento.name}`);
  });
}
```

Ilustración 20: Función getWeather(monumento)

Con esta función generamos los 2 paneles de tiempo de la página monumento.html. Para generar estos paneles se ha utilizado la API *OpenWeatherMap*.

Para obtener los valores de temperatura y humedad que se muestran en estos paneles, se llama a la función *$.get* apuntando al *endpoint* que la API exige, indicando el valor de latitud y longitud para que la API sepa de que coordenadas debe obtener la temperatura y humedad y al mismo tiempo le pasamos a este *endpoint* el valor de nuestra API_KEY.
De este *endpoint* se obtiene un JSON completo con mucha información sobre el tiempo que almacenamos en la variable local “data”, pero nosotros en esta función filtramos la información para quedarnos solo con lo que nos interesa:

- La temperatura se obtiene mediante la propiedad *data.temp*.
- El texto informativo sobre el tiempo se obtiene con la propiedad data.weather[0].description.
- El identificador del icono relacionado con el tiempo se obtiene a través de la propiedad data.weather[0].icon y se almacena en la variable “icon”. Para obtener el dibujo del icono, se pasa este identificador de manera dinámica al endpoint correspondiente, por lo tanto, la URL del icono queda de la siguiente manera:

  http://openweathermap.org/img/wn/${icon}@2x.png

- La humedad se obtiene mediante data.humidity.

Una vez hemos obtenido todos los valores necesarios, se utilizan al acceder a los elementos de la página monumento.html relacionados con el tiempo a través de su identificador y rellenar su contenido mediante la función *$("id").html(content)* de la librería JQUERY. Por ejemplo, para colocar la temperatura en el elemento HTML correspondiente, se utiliza la línea de código *$("#temp-main").html(`${temp}°C`);*

- **submitSearch():**

```
function submitSearch(e) {
  e.preventDefault();
  const term = document
    .getElementById("searchBarMonum")
    .value.trim()
    .toLowerCase()
    .normalize("NFD")
    .replace(/[\u0300-\u036f]/g, "");
  window.location.href = `/?term=${term}#filaTituloCatalogo`;
}
```

Ilustración 21: Función submitSearch()

La función recibe por parámetro el evento lanzado al ejecutar el botón de la searchBar de la página monumento.html y utilizando el mismo código que el ya explicado en la función *setSearch(id)* del fichero .js relacionado con la página principal, permite al usuario utilizar la barra de búsqueda pero en este caso redirige al usuario a la página principal, ya que esta search bar está en la página monumento.html (como se puede ver, el identificador pasado a la función *.getElementByID* es "searchBar**Monum**") mientras que el catálogo de cartas se encuentra en la página principal.

- **autocomplete(inp):**

Esta función aparece en otros archivos .js y se encarga del autocompletado de texto de la barra de búsqueda. Se explica con más exactitud en el apartado 6.1.2.

- **completeMonumentsArray():**

Esta función aparece en otros archivos .js y se encarga de rellenar un array con todos los monumentos contenidos en el JSON. Se explica con más exactitud en el apartado 6.1.2.

- **Window.onload = asunc function():**

```
window.onload = async function () {
  await getJSONFile();
  const query = getQueryParams();
  let monument = null;
  detailsData.forEach((comunidades) => {
    Object.keys(comunidades).forEach((nombreComunidad) => {
      comunidades[nombreComunidad].forEach((monumento) => {
        if (monumento.identifier === query) {
          monument = monumento;
          comunidad = nombreComunidad;
        }
      });
    });
  });
  if (monument == null) {
    window.location.href = "/";
  }
  generateJSONld(monument);

  getWeather(monument);

  const infoCard = document.getElementById("cartaInformacion");
  infoCard.innerHTML = generateInformationCard(monument);

  const infoSlider = document.getElementById("carouselImagenes");
  infoSlider.innerHTML = generateSlider(monument);

  generateMap(monument);

  const infoVideo = document.getElementById("contVideo");
  infoVideo.innerHTML = generateVideo(monument);

  document.getElementById(
    "botonRutaRelacionada"
  ).href = `rutas.html?comunidad=${comunidad}`;

  autocomplete(document.getElementById("searchBarMonum"));
};
```

Ilustración 22: Window.onload = asunc function():

Al igual que en script.js, es la función que se ejecuta cuando la página del navegador ha terminado de cargar. Toda la página monumento.html se construye desde aquí, ya que está función llama a todas las otras funciones de este fichero que en conjunto generan el código JavaScript que a su vez genera el código HTML y el resultado es la renderización de la página .html en el ordenador del usuario.

Para que las funciones a las que llama reciban el monumento correspondiente a la página donde el usuario accede, se obtiene el

nombre de este monumento de la URL con la función *getQueryParams()* y se almacena en la constante “query”. Entonces, con este valor, se itera sobre el JSON buscando el objeto cuyo nombre coincide con este valor y de esta manera se guarda en la variable “monument” el monumento correcto.

Este es el monumento que se pasa como parámetro a todas las funciones, por lo tanto, con un único fichero monumento.html, se permite que el usuario visualice una versión de esta página diferente para cada monumento donde los contenidos se habrán generado dinámicamente para el monumento adecuado.

De esta forma, pasándole el monumento como parámetro a la función *generateJSONld* se genera la información semántica sobre ese monumento en concreto.

Al mismo tiempo, dentro de esta misma iteración, se obtiene la comunidad a la que pertenece el monumento y de esta manera se configura con la propiedad *.href* el enlace a donde redirige el botón de esta misma página que lleva a la página que permite visualizar la ruta relacionada con el monumento. Para que sea dinámico, a este enlace se le pasa el valor de la comunidad obtenida mediante la notación *${comunidad}.*

6.3. Página de rutas

6.3.1. Funcionalidad y código relacionado

En esta página se muestra un mapa de gran tamaño que contiene marcadores con los diferentes monumentos que contiene una comunidad en concreto (seleccionada por el usuario, siendo Andalucía la comunidad por defecto) y muestra una ruta trazada entre estos. También se muestran con marcadores rojos los restaurantes destacados que se obtienen del JSON externo del grupo de gastronomía de España. El proceso de obtención y su tratamiento se explica más adelante. En la parte izquierda de la pantalla encontramos un total de 17 botones; uno por comunidad, que permiten cambiar la ruta mostrada por el mapa. Cabe destacar que cuando se reduce el tamaño de la pantalla, estos botones se concentran en un *dropdown* para que sea mucho más fácil para el usuario seleccionar otra ruta.

En las rutas se muestra un marcador para cada monumento y también un marcador para la localización en que el usuario se encuentra cuando abre la página. Este mapa está hecho también con la API *Leaflet*, que permite mostrar la localización de los monumentos, y la *HTML Geolocation API*, que permite mostrar la ubicación del usuario en el momento en que carga la página. No obstante, antes de calcular su posición, en el momento en el que el usuario hace click o arrastra sobre el mapa, la página muestra un mensaje recomendando activar los permisos de ubicación para mejorar la experiencia y a continuación se solicitan dichos permisos al igual que en la página de monumento en concreto. En este caso el mensaje de la página también se implementa con *SessionStorage.*

Al hacer click en el marcador de localización de un monumento, aparece un cuadro de texto con el nombre del monumento que actúa de enlace a la página del monumento en concreto.

Por otra parte, también utilizamos un motor de rutas para dibujar la ruta entre los marcadores de los monumentos. Este motor de rutas se combina con la función L.Routing.control de *Leaflet* de manera que en el parámetro “router” se coloca un enlace al motor de rutas utilizado.

En nuestro caso utilizamos dos motores de rutas:

- *Mapbox*: es una API que a su vez ofrece un motor de rutas.
- *GraphHopper*: otra API que a su vez ofrece un motor de rutas a través de un fichero JavaScript llamado “lrm-graphhopper.js” que se puede encontrar en la estructura de ficheros dentro de su respectiva carpeta.

Utilizamos dos motores diferentes porque con el plan gratuito ofrecen cálculos limitados y, por tanto, como el código es idéntico en ambos casos debido a que solo cambia la referencia del motor de ruta, si se diera el caso poco probable de que se agotan los cálculos un día que se requieran más, se pasará al otro motor de búsqueda, por lo que no guardan ninguna relación entre ellos.

Para mostrar los marcadores de los restaurantes se obtiene su localización geográfica y su nombre del JSON externo de otro grupo. En este JSON no se proporciona la comunidad específica en la que está situada cada restaurante por lo que, para dividirlos por comunidades, hemos tenido que utilizar el plugin Esri Leaflet Geocoder el cual nos permite utilizar una función que proporciona la dirección exacta del lugar. En la siguiente sección se puede observar el código que se ha utilizado para llevar a cabo esta labor.

El otro elemento importante de la página de rutas es la sección donde los usuarios pueden escribir comentarios. Justo debajo del mapa de rutas, se encuentra una caja de comentarios que mostrará los comentarios que se hayan realizado sobre la ruta de la comunidad seleccionada en cada momento, es decir, cada comunidad tendrá su *feed* de comentarios. De esta forma, inicialmente, al no haber ningún comentario, se incita al usuario a que ponga uno con un mensaje de forma que cuando lo pone se muestra en la *feed* de esa comunidad. Además, junto a este mensaje se muestra una imagen SVG decorativa.

Cabe destacar que hemos implementado los comentarios utilizando *localStorage* ya que nuestro host no permite el uso de PHP. En la siguiente sección se explica más detalladamente cómo se han implementado los comentarios.

6.3.2. Código JavaScript para las funcionalidades

- **getJSONFile():**

  Es la misma función que aparece en otros ficheros .js y permite obtener los datos necesarios del fichero monumentos.json para la implementación de la página rutas.html. Se explica con más exactitud en el apartado 6.1.2.

- **getExternJSONFile():**

```
async function getExternJSONFile() {
  const response = await fetch("https://gastronomiaesp.000webhostapp.com/JSON/cocineros.json");
  const { cocineros } = await response.json();
  dataRestaurantes = cocineros;
}
```

Ilustración 23: Fetch del JSON externo y almacenamiento en variable dataRestaurantes

Esta función permite acceder a los datos del JSON externo del grupo de gastronomía de España. De esta forma se guarda en una constante para que pueda ser accedido desde otras funciones y tratado según sea necesario.

- **getQueryParams():**

  Es la misma función que aparece en otros ficheros .js y permite modificar los parámetros de la URL necesarios para la implementación de la página rutas.html. Se explica con más exactitud en el apartado 6.1.2.

- **generateJSONId(monumento):**

  Es la función con la que se genera la información semántica de la página de rutas de forma dinámica. Se explica con más exactitud en el apartado 13.

- **switchRouteMap() y switchRouteMapB(Value):**

```
function switchRouteMap() {
  const select = document.getElementById("selectComunidad");
  window.location.href = `/rutas.html?comunidad=${select.value}`;
}

function switchRouteMapB(value) {
  window.location.href = `/rutas.html?comunidad=${value}`;
}
```

Ilustración 24: Funciones switchRouteMap() y switchRouteMapB(value)

Son las funciones que se llaman cuando el usuario cambia la comunidad de la cual quiere ver su mapa de ruta. Cuando la pantalla es de tamaño suficientemente grande, se utiliza una botonera lateral donde cada botón llama a la función *switchRouteMapB(value).*

Cuando la pantalla es de tamaño reducido, se utiliza un select como menú lateral, ya que contiene una botonera desplegable que encaja mejor para un tamaño menor, como puede ser el de un móvil, y este elemento al ser clickado llama a *switchRouteMap()* que detecta el valor seleccionado en cada momento mediante *select.value*.

Estas funciones se encargan de redirigir a la página rutas.html actualizando el valor del parámetro “comunidad” de la URL, para que se muestre el contenido de la comunidad que el usuario quiere cuando hace click en el botón con el nombre de esa comunidad.

Por lo tanto, las funciones hacen uso de *window.location.href*, pasando como valor la página /rutas.html, añadiendo *“?comunidad=${select.value}”* y *“?comunidad=${value}”* respectivamente al final de la URL de la página, para que dinámicamente se muestre siempre la ruta y los comentarios de la comunidad correcta.

- **getRouteMap():**

  Imágenes de la función disponibles en Drive.

  Esta función genera dinámicamente el mapa de ruta de cada comunidad en la página rutas.html.

  Lo primero que se hace es obtener el nombre de la comunidad seleccionada en el elemento html que tiene como identificador "selectComunidad". Se establece que este elemento por defecto tenga seleccionada la comunidad "Andalucía", mediante la propiedad *select.value* dentro de un condicional que comprueba con *!select.value* si ya hay algún valor seleccionado.

  Este mapa, al igual que el mapa de la página monumento.html, se genera utilizando la API Leaflet. Se construye la base del mapa con la función L.tileLayer y a partir de aquí se añaden los marcadores de todos los monumentos que forman la ruta de una comunidad en concreto.

  En primer lugar, se accede al JSON externo del grupo de gastronomía de España y se itera en este obteniendo la latitud, la longitud y el nombre de los restaurantes. Mientras se itera, se llama a la función *reverse().latlng* del plugin Esri Leaflet Geocode pasándole por parámetro las coordenadas obtenidas del JSON. De esta forma se consigue la dirección concreta de esas coordenadas en formato String, el cual se formatea para obtener finalmente el nombre de la comunidad autónoma a la que pertenece dicho lugar. Una vez descubierta la comunidad autónoma se compara con la comunidad autónoma que se está visualizando actualmente. En caso de ser iguales se almacenan las coordenadas y el nombre de los restaurantes que coincidan en 3 arrays distintos destinados a cada uno de esos datos. Posteriormente se hace un recorrido de esos arrays situando los marcadores rojos en las coordenadas correspondientes y asignando el nombre del restaurante a un *popup* ligado al marcador. Cabe destacar que todo ello se hace en un bloque *try-catch* debido a que en

ocasiones salta una excepción del tipo TIME OUT en el proceso de obtención de su JSON, lo que provocaba que no se ejecutara el código posterior al error e impedía que se mostrara el mapa con nuestros monumentos y la ruta, así como los comentarios. De esta forma, en caso de que se produzca algún error con el JSON externo, no afectará a la visualización de nuestros datos internos ya que se lleva a cabo control de errores. Además, en ocasiones también se produce algún error procedente del plugin de Esri Leaflet que también se recoge dentro de este bloque.

El siguiente paso es generar un marcador por cada monumento de la comunidad. Entonces, se realiza un bucle mediante *forEach* sobre dataRutas, que contiene los datos del fichero JSON, y en este bucle se rellena un array llamado markers, que contendrá todos los marcadores, de manera que para cada monumento de la comunidad correcta en el JSON, se llama a la función nativa *array.push* para introducir en este array el elemento generado al llamar a la función *L.latLng* de Leaflet con la latitud y longitud de cada monumento pasadas como parámetro. También se llama a la función *generateJSONld* pasándole el monumento de la iteración por parámetro para generar la información semántica sobre ese monumento.

Cuando se realiza la primera iteración, se llama a la función *map.setView* para establecer el foco del mapa sobre el primer monumento que se añade al mapa en este primera iteración. Además, también dentro de este bucle, rellenamos otros 2 arrays: “name” e “identifier”. Estos arrays contienen los nombres e identificadores de todos los monumentos que aparecen en el mapa y los valores de estos arrays se utilizan en las llamadas a la función *.bindPopup* de Leaflet para que en cada marcador aparezca el nombre del monumento relacionado y además este nombre actúe como enlace a la página de ese monumento en concreto generada dinámicamente sobre la plantilla monumento.html.

Al igual que en el mapa de monumento.html, se utiliza la API *Geolocation* para mostrar en el mapa un marcador verde que representa la posición actual del usuario en el momento de cargar la página. El permiso de ubicación se solicita tras un mensaje de la página cuando el usuario interactúa con el mapa haciendo click o arrastrando.

Finalmente, con la llamada a la función L.Routing.control de Leaflet, se genera la ruta, que consiste en una o varias líneas rojas dibujadas entre los marcadores de los monumentos. Para esto, es necesario la utilización de un motor de rutas externo a Leaflet y nosotros hemos utilizado el que ofrecen las dos siguientes APIs:

- *Mapbox.*
- *graphHopper.*

El motor de rutas se pasa como parámetro a la funcion *L.Routing.control*, asociado al parámetro “router:”. Utilizamos 2 motores por si acaso se nos acabaran las peticiones diarias o mensuales de uno de los 2, tendríamos disponible el otro simplemente descomentando el bloque de código que le corresponde.

Al utilizar *Mapbox*, la función que se establece como valor del parámetro “router:” es *L.Routing.mapbox* y esta a su vez recibe como parámetro la API_KEY de nuestra cuenta de *Mapbox*.

De manera equivalente, al utilizar *graphHopper*, la función que se establece como valor del parámetro “router:” es *L.Routing.graphHopper* y esta a su vez recibe como parámetro la API_KEY de nuestra cuenta de graphHopper.

Para acabar se añade una leyenda en la parte inferior derecha para orientar al usuario sobre la naturaleza de los marcadores y un panel en la parte inferior izquierda que indica la comunidad autónoma de la cual se está visualizando el mapa en este momento.

- **getFromDB():**

  Es la misma función que aparece en otros ficheros .js y permite obtener los datos necesarios de *localStorage* para la implementación de la página rutas.html. El código cambia ligeramente para adaptarla a esta página en concreto, por ejemplo, pueden cambiar los nombres de las variables y/o la *key*, pero la lógica de la función es exactamente la misma.

- **saveIntoDB(comment):**

  Es la misma función que aparece en otros ficheros .js y permite guardar los datos necesarios en *localStorage* para la implementación de la página rutas.html. El código cambia ligeramente para adaptarla a esta página en concreto, por ejemplo, pueden cambiar los nombres de las variables y/o la *key*, pero la lógica de la función es exactamente la misma.

- **writeComment(e):**

```
function writeComment(e) {
  // e.preventDefault();
  console.log("submit was clicked");
  const name = document.getElementById("nameInput");
  const text = document.getElementById("commentInput");
  console.log(uuidv4());
  const comment = {
    id: uuidv4(),
    name: name.value,
    text: text.value,
    date: Date.now(),
    liked: false
  };
  saveIntoDB(comment);
  generateCommentFeed();
  name.value = "";
  text.value = "";
}
```

Ilustración 25: Función writeComment(e)

Función que permite que el usuario escriba comentarios en la página rutas.html. Cada comunidad tiene un conjunto específico de comentarios en la página rutas.html. Los comentarios están implementados con *localStorage*.

Al principio de la función, se guardan en 2 variables JavaScript los elementos html donde el usuario puede escribir su nombre y el texto del comentario. A continuación, se forma un struct llamado “comment” que contiene lo siguiente:

- El identificador del comentario, generado aleatoriamente y de manera única mediante la librería de identificadores únicos (uuid), importada en rutas.html de la siguiente manera:

  <script src="https://cdnjs.cloudflare.com/ajax/libs/uuid/8.1.0/uuidv4.min.js" ></script>

  Una vez importada la librería, se puede llamar a la función *uuidv4()* que se asegura de generar un identificador único mediante la utilización de varios parámetros como por ejemplo la fecha y el tiempo del momento exacto en que se llama a la función.

- El nombre del usuario que escribe el comentario, guardado en la variable name.
- El texto del comentario, guardado en la variable text.
- La fecha en que se escribe el comentario, generada a partir de la función nativa *Date.now(),* guardada en la variable date.
- Un booleano para establecer si el comentario está en estado de “me gusta” o no (likes de comentario).

  Una vez generado este struct, se guarda en el conjunto de comentarios almacenado en *localStorage* mediante la función *saveIntoDB(comment)*, y se llama a la función

generateCommentFeed() para actualizar la lista de comentarios en el contenedor correspondiente de la página rutas.html.

Finalmente se reestablecen los campos de nombre y texto al carácter vacío “” para dejarlos preparados para escribir otro comentario.

- **generateCommentFeed():**

Imágenes de la función disponibles en Drive.

Esta función escribe en el contenedor html correspondiente el conjunto de comentarios almacenados en *localStorage*.

Primero, obtiene este conjunto de comentarios de *localStorage* mediante la función *getFromDB()*. Con la notación *getFromDB()[community]* se obtiene solo el conjunto de comentarios de la comunidad que está seleccionada por el usuario en la página rutas.html, la cual está almacenada en la variable global “community”. En caso de que no haya ningún comentario todavía se muestra en la pantalla un texto que invita al usuario a poner uno. En caso contrario se obtiene el conjunto de comentarios.

Una vez obtenido este conjunto de comentarios, se itera sobre él y por cada comentario se genera una estructura HTML que se compone de todos los elementos que contiene el struct que forma el comentario. Este struct es el que se explica en detalle en la función writeComment(e). Para obtener el formato de fecha que a nosotros nos interesa, se han ejecutado las funciones *.getDay()*, *.getMonth()* y *.getFullYear()* sobre la variable “date” que originalmente contenía el valor devuelvo por la función *Date(comment.date).*

Cuando se ha generado todo el código HTML dinámicamente para todos los comentarios del conjunto, se accede al elemento de la página

rutas.html que debe contener estos comentarios y se establece mediante la función nativa *elemento.innerHTML* el contenido de este elemento con la variable “commentsHTML”, que es donde se ha ido guardando todo el código HTML generado con JavaScript.

- **toggleLiked(id):**

```
function toggleLiked(id) {
  let comments = getFromDB();
  comments[community] = comments[community].map(comment => {
    if(comment.id === id){
      comment.liked = !comment.liked;
    }
    return comment;
  });
  console.log(comments);
  localStorage.setItem("comments", JSON.stringify(comments));
  generateCommentFeed();
}
```

Ilustración 26: Función toggleLiked(id)

Función cuyo propósito es actualizar la propiedad “liked” del comentario que tiene el identificador que la función recibe como parámetro.

Obtiene el conjunto de comentarios de *localStorage* mediante la función *getFromDB()* y lo guarda en la variable “comments”. Entonces, itera sobre este conjunto hasta encontrar el comentario cuyo identificador coincide con el que se ha pasado por parámetro a la propia función *toggleLiked(id).* Cuando ha encontrado el comentario, accede a la propiedad ‘liked’ mediante *comment.liked* e invierte el valor, ya que al ser un boolean solo puede ser true o false.

Finalmente devuelve el comentario con la propiedad ‘liked’ cambiada y actualiza el conjunto de comentarios en *localStorage* con este comentario actualizado a través de la función *setItem* indicando como clave la key correspondiente a los comentarios, que en nuestro caso es “comments”. Como siempre, formateamos con *JSON.stringify* para que se almacenen con el formato adecuado para *localStorage*.

Para que el cambio se vea reflejado en la página rutas.html del usuario, vuelve a generar la lista de comentarios realizando una llamada a la función *generateCommentFeed()*.

- **submitSearch(e):**

```
function submitSearch(e) {
  e.preventDefault();
  const term = document
    .getElementById("searchBarRutas")
    .value.trim()
    .toLowerCase()
    .normalize("NFD")
    .replace(/[\u0300-\u036f]/g, "");
  window.location.href = `/?term=${term}#filaTituloCatalogo`;
}
```

Ilustración 27: Función submitSearch(e)

La función recibe por parámetro el evento lanzado al ejecutar el botón de la searchBar de la página rutas.html y utilizando el mismo código que el ya explicado en la función *setSearch(id)* del fichero .js relacionado con la página principal, permite al usuario utilizar la barra de búsqueda pero en este caso redirige al usuario a la página principal, ya que esta search bar está en la página rutas.html (como se puede ver, el identificador pasado a la función *.getElementByID* es "searchBar**Rutas**") mientras que el catálogo de cartas se encuentra en la página principal.

- **autocomplete(inp):**

Esta función aparece en otros archivos .js y se encarga del autocompletado de texto de la barra de búsqueda. Se explica con más exactitud en el apartado 6.1.2.

- **completeMonumentsArray():**

  Esta función aparece en otros archivos .js y se encarga de rellenar un array con todos los monumentos contenidos en el JSON. Se explica con más exactitud en el apartado 6.1.2.

- **window.onload = async function():**

```
window.onload = async function () {
  await getJSONFile();
  try {
    await getExternJSONFile();
  }
  catch (error) {
    console.error(error);
  }
  finally {
    const select = document.getElementById("selectComunidad");
    select.value = getQueryParams();
    getRouteMap();
    autocomplete(document.getElementById("searchBarRutas"));
    generateCommentFeed();
  }
};
```

Ilustración 28: window.onload = async function():

La idea de esta función es la misma que las equivalentes en los otros dos ficheros .js. Es la función que se ejecuta cuando la ventana del navegador termina de cargar y en ese momento llama al resto de funciones contenidas en su mismo fichero para que combinadas generen la página para el usuario.

En este script, se caracteriza por obtener de la URL el nombre de la comunidad correcta mediante la llamada a la función *getQueryParams()* y establecer mediante *select.value* este nombre como valor seleccionado de la botonera que permite elegir la comunidad cuya ruta se quiere mostrar. Esto sirve para cargar la página de manera correcta cuando el usuario accede a ella desde el botón “ruta relacionada” que se encuentra en la página monumento.html.

En esta función se utiliza el bloque try-catch-finally para que, en caso de que se produzca un error con la obtención del JSON externo, no afecte el funcionamiento normal de la página y se muestre todo correctamente.

7. Web-responsive. Evaluación de la web en varios dispositivos.

Para comprobar que la web es responsive y se ajusta de manera adecuada a los dispositivos y tamaños más importantes, hemos utilizado los siguientes métodos:

- Toggle Device Toolbar, de la herramienta para desarrolladores de Google Chrome. Esta herramienta ofrece un menú desplegable donde se pueden seleccionar varios tamaños para ver cómo se visualiza la web en estos.

- Redimensión manual. Para comprobar aspectos muy concretos, hemos hecho uso de la redimensión manual de la ventana que ofrecen todos los navegadores.

Dividiremos las pruebas en las 3 páginas. Para el tamaño de dispositivo móvil, las pruebas se han realizado con el formato iPhone SE, que representa un tamaño general de dispositivo móvil.

Por otra parte, se mostrarán capturas realizadas para tamaños intermedios como portátiles y tablets.

Página Principal

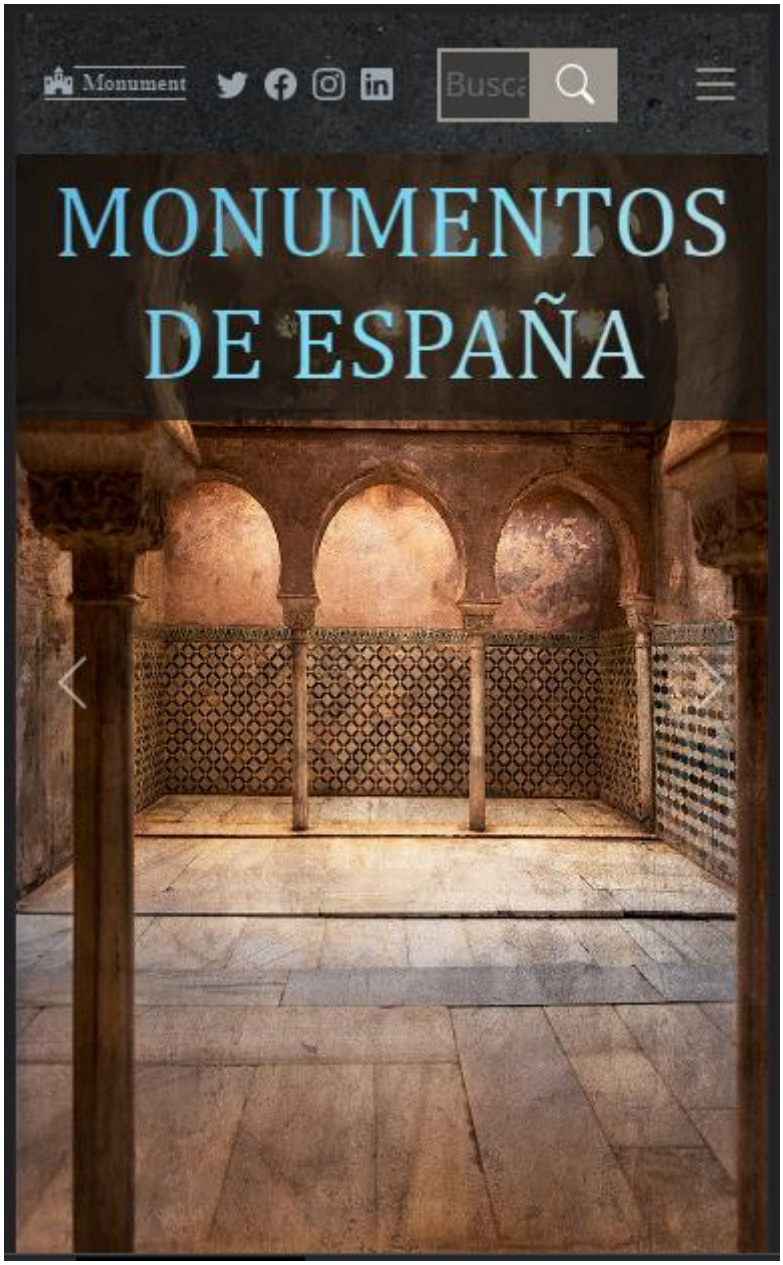


Ilustración 29: vista móvil página principal

En esta imagen se puede ver como la barra de navegación se reduce horizontalmente y los enlaces a las páginas se agrupan en un menú desplegable situado en la esquina superior derecha. Por otra parte, las imágenes del slider se centran y algunas de ellas se cambian por otra, como la que se ve en la imagen de arriba, para que quede mejor estéticamente. Para detectar cuándo hay que sustituir las imágenes se ha utilizado *media query*.

Ilustración 30: vista móvil sidenav catálogo monumentos

En esta imagen se puede ver como el sidenav cambia de forma, de manera que ahora en lugar de estar el sidenav a la izquierda y el catálogo de cartas a la derecha, están uno encima del otro y el sidenav ocupa toda la pantalla horizontalmente.

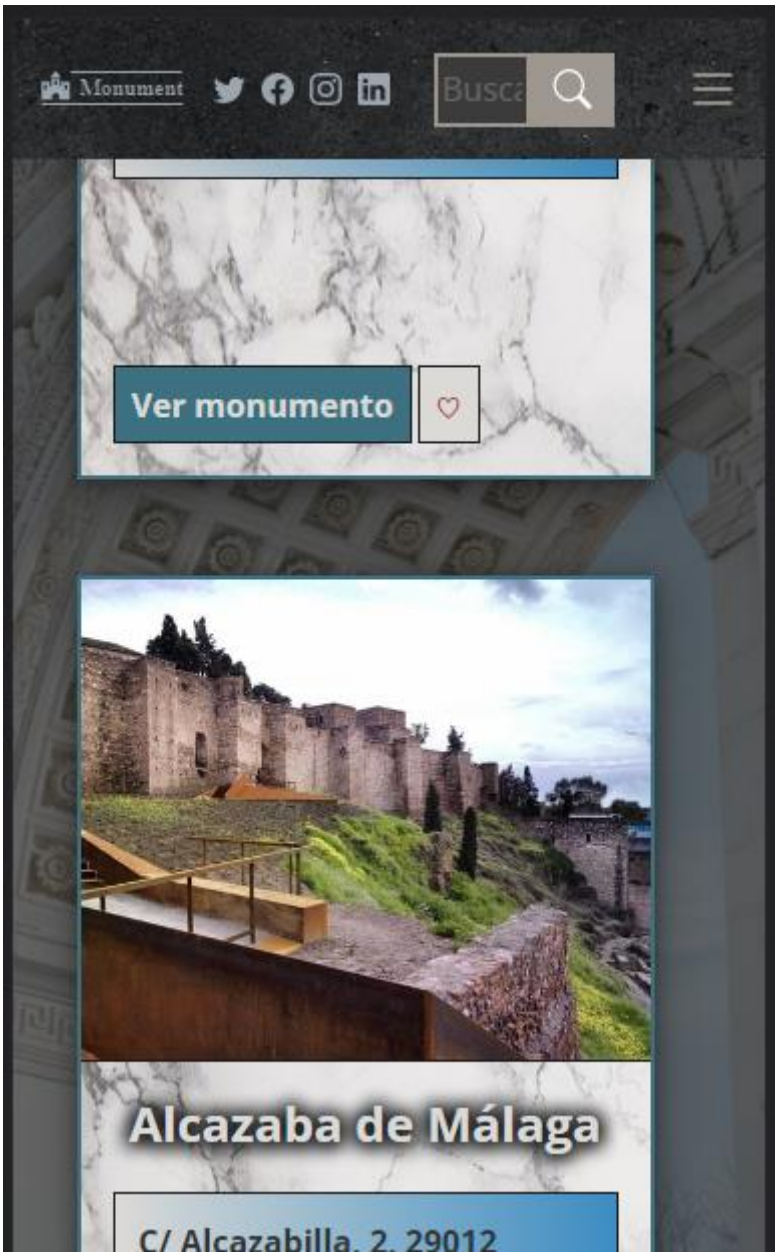


Ilustración 31: vista móvil catálogo monumentos

En cuanto al catálogo de cartas en concreto, todas las imágenes aparecen una encima de la otra, ya que en ningún movil existirá el espacio suficiente horizontalmente para colocar 2 cartas una al lado de la otra y que sean bien legibles.

Ilustración 32: vista móvil footer

Finalmente, en cuanto al *footer*, el formato pasa a ser vertical y los 3 componentes de este se agrupan uno encima del otro.

Página monumento.html

Ilustración 33: vista móvil página rutas

En esta página ocurre lo mismo que en las demás en cuanto a la barra de navegación ya que es compartida por las 3 páginas.

En cuanto al resto del contenido, en esta página no hay mucho que decir. Simplemente se distribuyen los elementos verticalmente uno encima del otro.

Ilustración 34: vista móvil datos monumento

Por otra parte, dentro de la carta informativa, los iconos svg y la información asociada a estos se agrupan lo mejor posible para ocupar menos espacio, pero manteniendo la legibilidad y estética. La descripción se estira verticalmente.

Página rutas.html

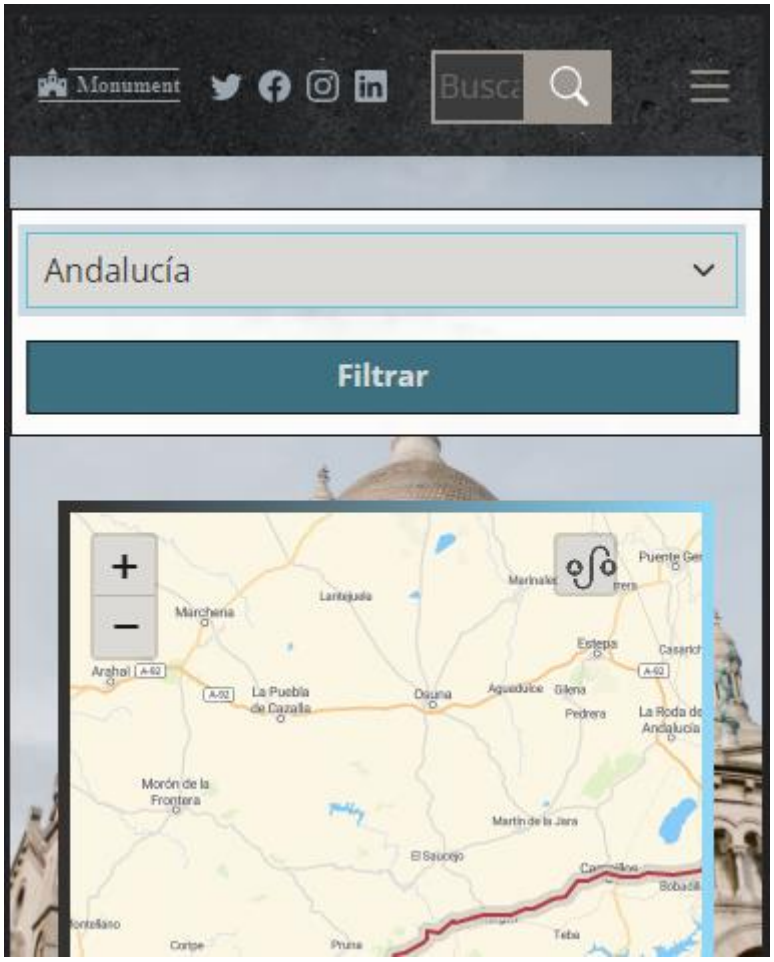


Ilustración 35: vista móvil página rutas

Para tamaños menores, en rutas.html el sidenav no es un conjunto de botones situados uno encima del otro, sino que se transforma en un menú desplegable que ocupa poco espacio, ya que consideramos que así es más estético. Al desplegar el menú para ver las posibles comunidades se muestra de la siguiente manera:

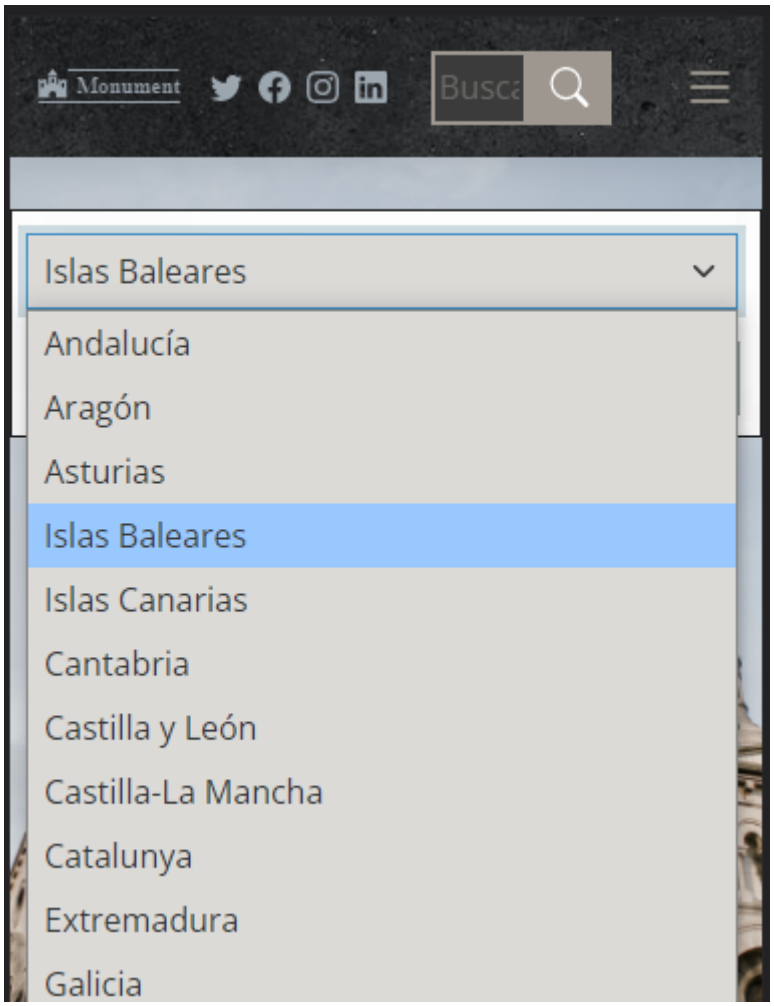


Ilustración 36: vista del dropdown de selección de comunidad desde móvil

En cuanto al mapa y la caja de comentarios, siguen apariendo uno encima del otro, pero se reduce su superficie horizontal y se estiran verticalmente:

Ilustración 37: vista móvil del mapa de la página de rutas

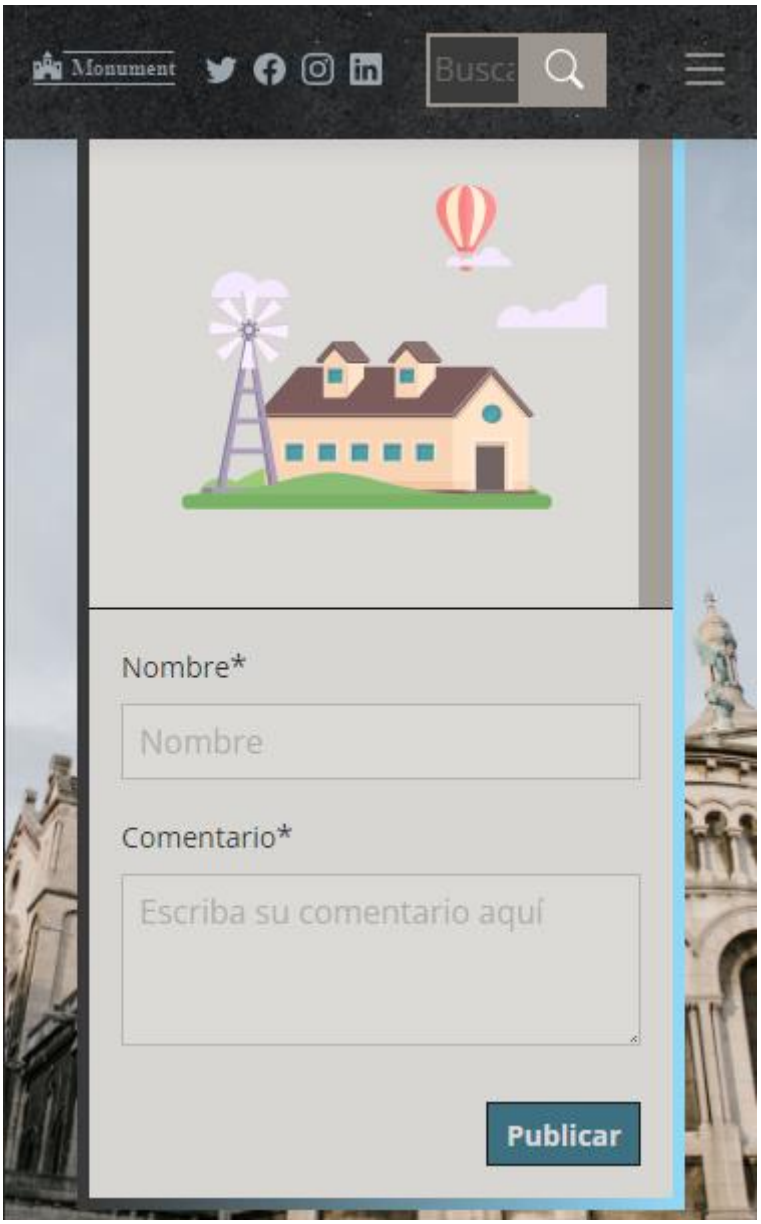


Ilustración 38: vista móvil caja comentarios página rutas

El aspecto responsive de la web no solo funciona para dispositivos móviles, sino que se adapta automáticamente también a cualquier tamaño intermedio. Algunos ejemplos son los siguientes:

Por ejemplo, en una pantalla de tamaño estándar 1920x1080 el catálogo de cartas se muestra de la siguiente manera:

Ilustración 39: vista 1920x1080 catálogo monumentos

Como se puede ver, aparece el sidenav a la izquierda y en la parte de la derecha cada fila contiene 4 monumentos.

Si reducimos el tamaño de la pantalla un poco, se adaptará al nuevo tamaño.

Por ejemplo, para la resolución 1366 x 524, que es una resolución común para portátiles o algunas tablets, obtenemos el siguiente resultado:

Ilustración 40: vista 1366x524 catálogo monumentos

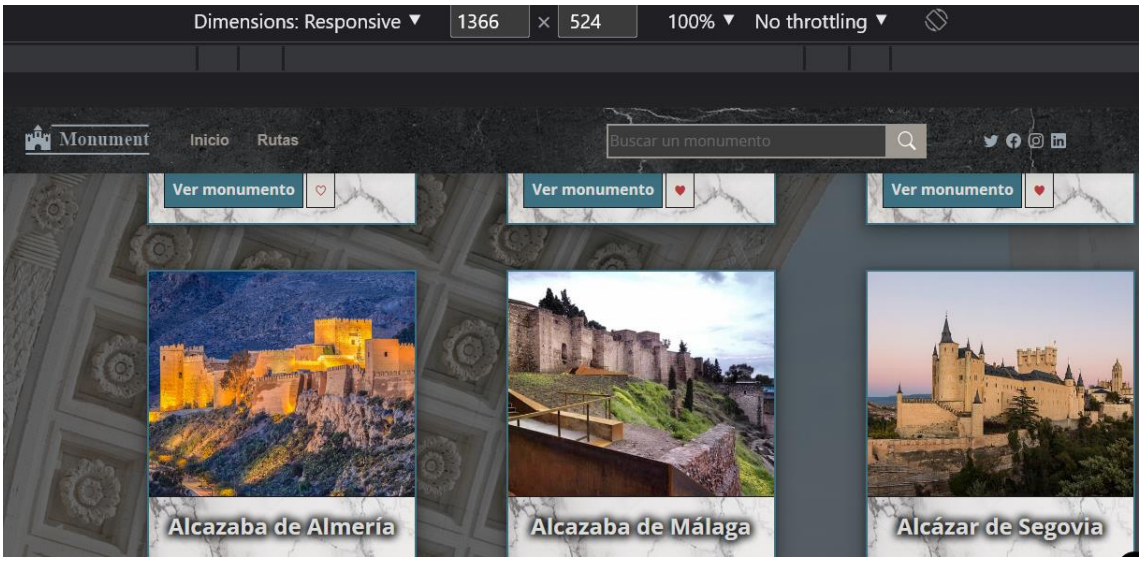


Ilustración 41: vista 1366x524 catálogo monumentos

Ahora en lugar de aparecer 4 cartas en cada fila, aparecen solo 3. La Alcazaba de Almería, que era el cuarto monumento de la primera fila en la primera imagen, pasa a la fila de abajo, donde es ahora el primer monumento de esta.

También se puede ver que el espacio entre el sidenav y la primera carta se ha reducido.

Si lo reducimos un poco más, volverá a pasar lo mismo, 2 cartas en lugar de 3:

Ilustración 42: vista 1200x524 catálogo de monumentos

O 1 sola carta en lugar de 2:

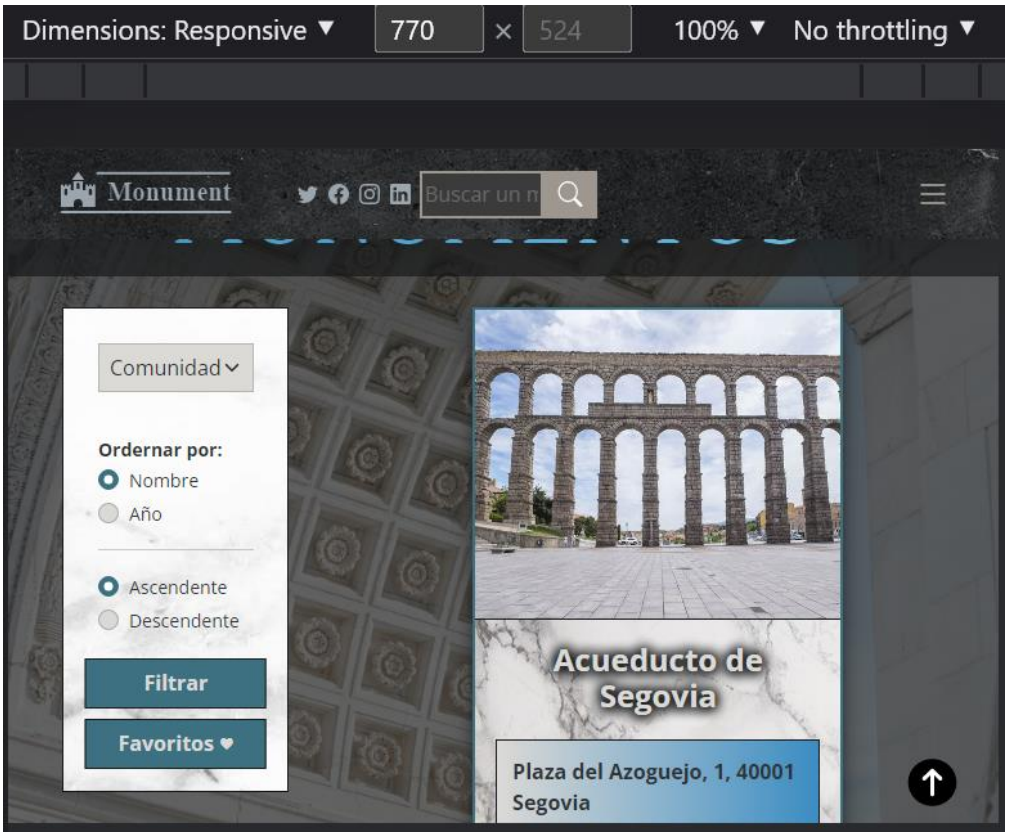


Ilustración 43: visión 770x524 catálogo de monumentos

Llegados a este punto, si se reduce aún más el tamaño, ya se entraría en el formato de dispositivo móvil, el cual ya se ha mostrado más arriba dentro de este mismo apartado de Web Responsive.

Esta misma dinámica se refleja en el resto de los contenidos y páginas, por lo que no mostramos todos los ejemplos de toda la web para no sobrecargar la documentación.

Todos los aspectos responsive suceden de manera automática gracias al uso del sistema Grid de Bootstrap, y algunos ajustes manuales realizados por nosotros mediante *media query*.

8. Bootstrap

Utilizamos la herramienta Bootstrap mediante un *import* de los siguientes enlaces en la cabecera y el *body* de los ficheros HTML, respectivamente:

- https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css, importado con la etiqueta <link>.
- https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js, importado con la etiqueta <script>.

Esta herramienta nos permite utilizar módulos HTML y CSS base para ahorrar tiempo y nosotros los modificamos a nuestro gusto.

El aspecto más importante de utiliza Boostrap es conseguir que la página sea *responsive* a través de su *Grid system* que consiste en crear filas y columnas que hacen que la web sea *responsive* automáticamente al redimensionarla.

Intentamos utilizar únicamente Bootstrap 5, que es la última versión, pero en algunos casos cogemos cosas de la versión 4 si lo consideramos oportuno.

9. Plantillas/temas utilizados

Además de Bootstrap y el fichero .css que trae con él, estamos utilizando un tema CSS que consiste en un fichero .css que hemos descargado de una página que se dedica a crear temas y presentarlos en una galería.

No utilizamos ninguna plantilla completa precodificada ya que hemos considerado que es más cómodo ir creando nuestra página web de manera modular, hecho que nos da una mayor libertad y control sobre el diseño, además de simplificar significativamente el código al poner solo cosas que sabemos que son explícitamente necesarias.

Algunas de las páginas que hemos visitado para buscar temas o plantillas son las siguientes:

- https://bootswatch.com/
- https://bootstrap.themes.guide/#themes

10. Librerías utilizadas

Las librerías que utilizamos en nuestra práctica son las siguientes:

- jQuery: Esta importada con el siguiente enlace dentro de la etiqueta <script>:
  https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js%22%3E
  jQuery es una librería de JavaScript que permite realizar las mismas funcionalidades, pero con un código más simple y legible.
- Lodash: Esta importada con el siguiente enlace dentro de la etiqueta <script>:
  https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.js%22%3E
  Esta es una librería de JavaScript que no está enfocada a webs concretamente, sino que se construyó para facilitar el tratamiento de estructuras de datos en general y nosotros lo utilizamos para que la gestión de los datos de fichero JSON sea mucho más cómoda.

11. APIs utilizadas

El listado de todas las APIs utilizadas con sus enlaces correspondientes es el siguiente:

- *Leaflet*
- *Mapbox*
- *GraphHopper*
- *HTML geolocation API*
- *OpenWeather*
- *WebStorage (localStoragey sessionStorage)*
- *HTML Video and Audio*
- *Esri Leaflet*

12. Información de otro grupo utilizada

Para cumplir con el requisito de utilizar un JSON externo de otro grupo hemos decidido añadir al mapa de la página de rutas restaurantes con un marcador rojo. Tanto la localización de estos restaurantes como su nombre los obtenemos iterando sobre el JSON del grupo de gastronomía de España.

Estos restaurantes se muestran según la comunidad autónoma que el usuario tenga seleccionada. Los compañeros no tienen en su JSON la comunidad autónoma a la que pertenecen dichos restaurantes por lo que se ha llevado a cabo el procedimiento que se describe en la función *getRouteMap()* de la página de rutas. Al hacer click sobre un restaurante se abre un *popup* que muestra el nombre de ese restaurante en concreto como se muestra a continuación.

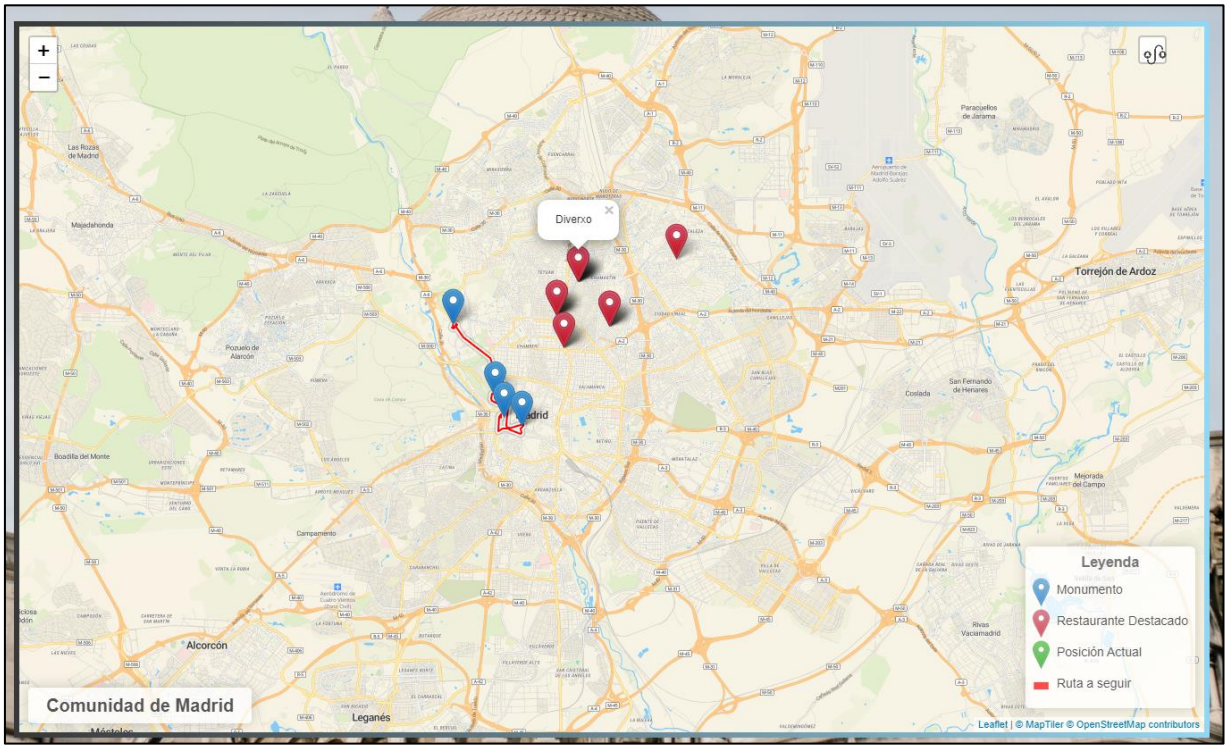


Ilustración 44: ejemplo de la visualización de los restaurantes de Madrid obtenidos del JSON externo

13. Web Semántica

El concepto de Web Semántica consiste en mejorar sustancialmente las tareas de buscar y gestionar información y recursos web específicos, haciendo los datos legibles por las aplicaciones conocidas como “Agentes Web” que gestionan la información de manera semiautomática.

La característica web semántica se implementó inicialmente con microdata. Sin embargo, después nos dimos cuenta de que es más cómodo mediante JSON-LD. Por lo tanto, la versión final de la web tiene la semántica realizada completamente con JSON-LD.

Para esto hemos creado una función “*generateJSONld*” en cada uno de los ficheros .js que controlan las 3 páginas de la web (pág. principal, monumento.html y rutas.html).

Las funciones JavaScript son las siguientes:

Página principal:

```
function generateJSONld(monumento){
  const script = document.createElement("script");
  script.setAttribute("type","application/ld+json");

        let s = {
          "@context":"https://www.schema.org",
          "@type": "LandmarksOrHistoricalBuildings",
          "image": monumento.image[0],
          "name": monumento.name,
          "address": {
            "@type": "PostalAddress",
            "streetAddress": monumento.address
          },
          "additionalProperty": {
            "@type": "PropertyValue",
            "name": "yearBuilt",
            "value": monumento.yearBuilt
          }
        };
        script.textContent+=JSON.stringify(s);
  document.head.appendChild(script);
}
```

Ilustración 45: función para generar JSONld de forma dinámica de los monumentos de la página principal

Como se puede ver, esta función permite generar de manera automática un script de tipo application/ld+json y añadirlo al fichero .html de la página principal. El fichero se crea al comienzo de la función con la función nativa *document.createElement("script")*. A continuación, se le asigna el tipo de script mediante la función nativa *setAttribute*.

Una vez hecho esto, se genera el contenido del script de manera dinámica. Se utilizan las propiedades estandarizadas de Schema.org relacionadas con monumentos como, por ejemplo:

- type: "LandmarksOrHistoricalBuildings".
- name.
- address.
- yearBuilt.
- Etc.

La asignación de valores a estas propiedades se realiza mediante la notación *monumento.propiedad* siendo monumento el parámetro recibido por la función. Este parámetro es enviado desde la función que realiza los filtros y genera una carta por cada monumento encontrado en el fichero .json:

```
_.orderBy(filtered, [filter], [order]).forEach((monumento) => {
  // Libreria lodash

  cards += generateCard(monumento);
  generateJSONld(monumento);
});
```

Ilustración 46: generación de la información semántica de la página principal

De esta manera, se genera una instancia para cada monumento, por lo que los motores de búsqueda sabrán que en nuestra página principal hay 82 monumentos y qué características tienen estos monumentos. Esto se puede comprobar con el validador de Schema.org. Para esto, primero introducimos la URL a validar, que en este caso es la de la página principal:

Ilustración 47: validación página principal

Cuando el validador inspecciona la semántica de la web, muestra la siguiente información:

Ilustración 48: resultados validación página principal

Como se puede ver, detecta 83 elementos, de los cuales uno de ellos es la información de carácter *LocalBusiness* contenida en el *footer* de la página y los otros 82 son los monumentos contenidos en cartas.

Si desplegamos el apartado *LocalBusiness*, aparece la siguiente información:

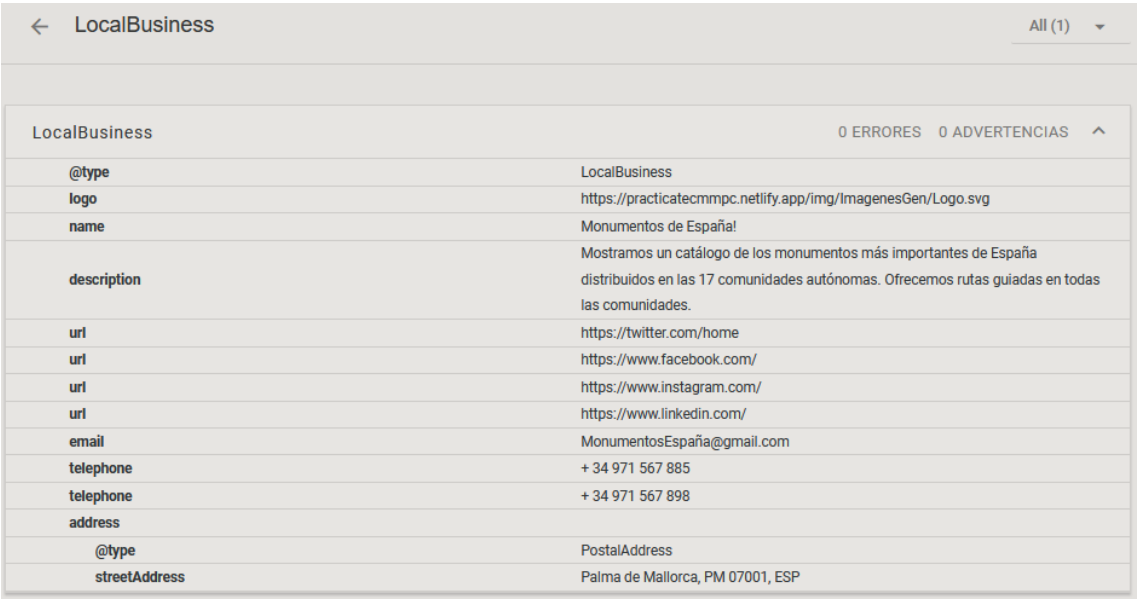


Ilustración 49: despliegue resultados obtenidos LocalBusiness

En esta imagen se puede ver que el validador detecta correctamente que la información contenida en el *footer* es de tipo *LocalBusiness* y al mismo tiempo sabe en concreto qué concepto es cada uno de los valores contenidos en este. Por ejemplo, sabe que los dígitos +34 971 567 885 consisten en un número de teléfono.

Esta información se define de forma estática en el <head> de cada uno de los 3 .html de la siguiente forma:

```
<script type="application/ld+json">
    {
        "@context":"https://www.schema.org",
        "@type": "LocalBusiness",
        "logo":"/img/ImagenesGen/Logo.svg",
        "name": "Monumentos de España!",
        "description": "Mostramos un catálogo de los monumentos más importantes de España distribuidos en
        las 17 comunidades autónomas. Ofrecemos rutas guiadas en todas las comunidades.",
        "url":[
            "https://twitter.com/home",
            "https://www.facebook.com/",
            "https://www.instagram.com/",
            "https://www.linkedin.com/"
        ],
        "address": {
            "@type": "PostalAddress",
            "streetAddress": "Palma de Mallorca, PM 07001, ESP"
        },
        "email": "MonumentosEspaña@gmail.com",
        "telephone":[
            "+ 34 971 567 885",
            "+ 34 971 567 898"
        ]
    }
</script>
```

Ilustración 50: generación estática de la información semántica de LocalBusiness

Si desplegamos el apartado *LandarmarksOrHistoricalBuildings* y hacemos click sobre un monumento en concreto, aparece la siguiente información:

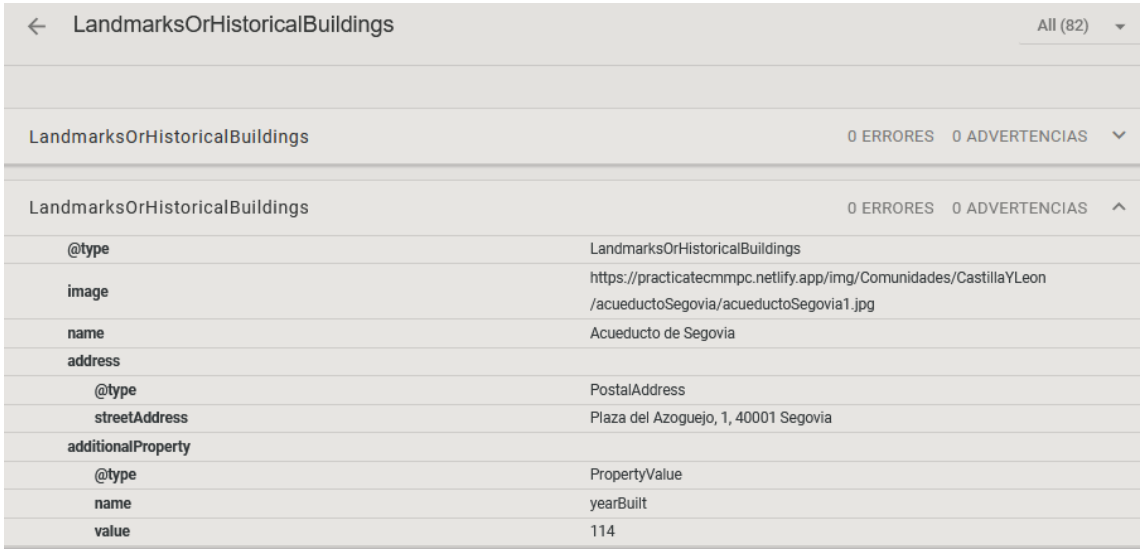


Ilustración 51: despliegue resultados obtenidos LandmarksOrHistoricalBuildings

Al igual que con *LocalBusiness*, podemos comprobar que el validador sabe que la información que aparece en la página es de un *LandmarkOrHistoricalBuildings*, es decir, un monumento en nuestro caso. También sabe en qué consiste cada valor de las cartas. Por ejemplo, sabe que Acueducto de Segovia es el nombre del monumento.

De esta manera se facilita a los motores de búsqueda encontrar información en nuestra web, lo que supone una ventaja para nuestra web a la hora de los resultados que aparecen en los navegadores cuando los usuarios navegan.

Página monumento.html

De manera similar a la página principal, en el script details.js, que es el script que controla monumento.html, tenemos la siguiente función:

```
function generateJSONld(monumento){
  const script = document.createElement("script");
  script.setAttribute("type","application/ld+json");

        let s = {
          "@context":"https://www.schema.org",
          "@type": "LandmarksOrHistoricalBuildings",
          "name":monumento.name,
          "identifier": monumento.identifier,
          "image":[
            monumento.image[0],
            monumento.image[1],
            monumento.image[2]
          ],
          "description": monumento.description,
          "geo": {
            "@type": "GeoCoordinates",
            "latitude": monumento.latitude,
            "longitude": monumento.longitude
          },
          "address": {
            "@type": "PostalAddress",
            "streetAddress": monumento.address
          },
          "additionalProperty": {
            "@type": "PropertyValue",
            "name": "yearBuilt",
            "value": monumento.yearBuilt
          },
          "subjectOf":{
            "@type": "VideoObject",
            "name": [
              monumento.video[0],
              monumento.video[1]
            ],
            "thumbnail": "Monument image",
            "description": "Short video about the monument",
            "uploadDate": "27/05/2022",
            "thumbnailURL": monumento.image[0]
          }
        };
        script.textContent+=JSON.stringify(s);
  document.head.appendChild(script);
}
```

Ilustración 52: función para generar la información semántica de la página de monumentos de forma dinámica

A diferencia de la función para la página principal, esta función contiene más información porque en la página monumento.html se muestra toda la información de un monumento en concreto y por lo tanto esto incluye información extra como las coordenadas y el vídeo del monumento que no se muestran en la página principal. Por otra parte, se llama a esta función una única vez, pasando por

parámetro el único monumento contenido en la página. El código sabe cuál es este monumento porque lo obtiene del valor del parámetro *?identifier* de la URL, que se establece al hacer click en el botón de alguna de las cartas de la página principal.

A parte de estas diferencias, la idea es exactamente la misma. Para la comprobación se utiliza de nuevo el validador de Schema.org. La URL introducida esta vez es la siguiente:

Ilustración 53: validación página monumento

Tras el análisis de la página, la información devuelta es la siguiente:

Ilustración 54: resultados página monumento

El apartado *LocalBusiness* es exactamente el mismo que el de la página principal, ya que el *footer* es compartido. Por lo tanto, no se volverá a explicar. Sin embargo, el apartado *LandmarksOrHistoricalBuildings* sí que se diferencia del de la página principal, ya que podemos ver que solo contiene un único elemento. Esto es porque esta página muestra información de un solo monumento, que es el monumento asociado a la carta donde se encuentra el botón que permite acceder a esta página. Entonces, la información de web semántica detectada por el validador en este caso es la siguiente:

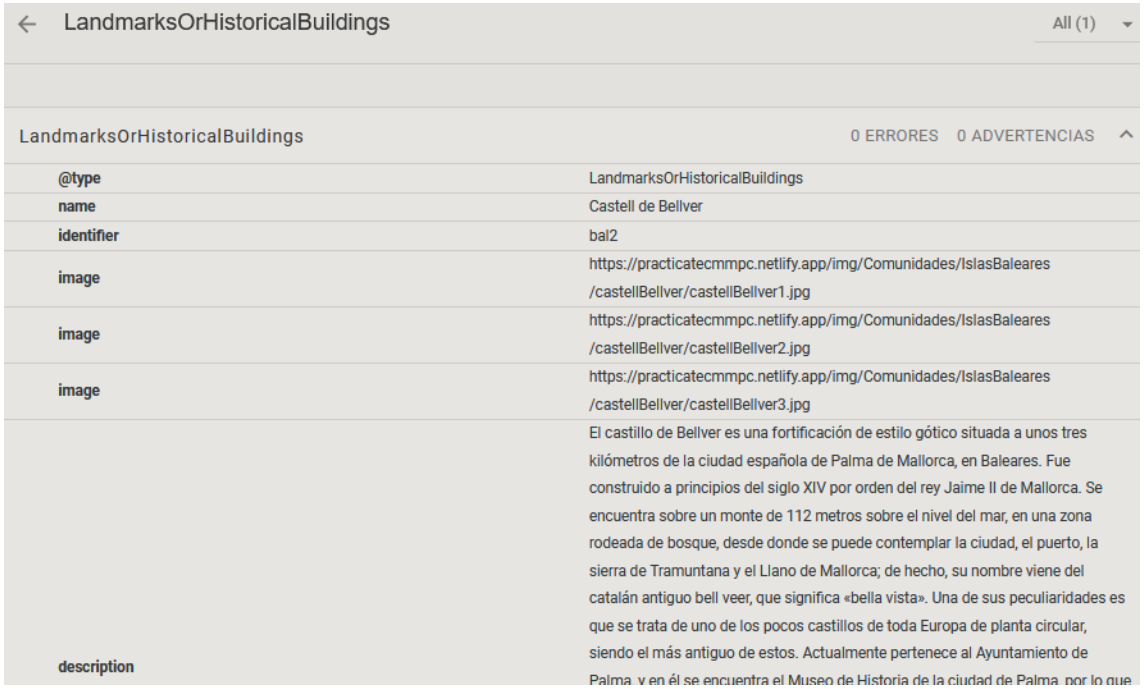


Ilustración 55: despliegue resultado LandmarksOrHistoricalBuildings

Como se puede ver, esta información corresponde al monumento conocido como Castell de Bellver, que es el que tiene asociado el identificador “bal2”. Este identificador es el que aparece como valor de la propiedad *?identifier* en la URL introducida en el validador. Por lo tanto, está detectando la información solo de ese monumento en concreto, como debe ser.

Al igual que en la página principal, con esta semántica los motores de búsqueda sabrán que el texto “El castillo de Bellver es una fortificación…” es la descripción de este monumento, sabrán que el enlace https://practicatecmmpc.netlify.app/img/Comuni... es una imagen de este monumento, etc.

Página rutas.html

De nuevo, se utiliza la misma idea. Se ha programado la siguiente función en rutas.js, que es el código JavaScript utilizado para controlar el fichero rutas.html:

```
function generateJSONld(monumento) {
  const script = document.createElement("script");
  script.setAttribute("type", "application/ld+json");

  let s = {
    "@context": "https://www.schema.org",
    "@type": "LandmarksOrHistoricalBuildings",
    "name": monumento.name,
    "geo": {
      "@type": "GeoCoordinates",
      "latitude": monumento.latitude,
      "longitude": monumento.longitude
    },
  };
  script.textContent += JSON.stringify(s);
  document.head.appendChild(script);
}
```

Ilustración 56: función para generar la información semántica de la página de rutas de forma dinámica

En este caso solo se configura el nombre y las coordenadas, ya que es la única información de los monumentos que aparece en esta página.

De nuevo la idea es exactamente la misma. Se pasa la siguiente URL al validador:

Ilustración 57: validación página rutas

Y devuelve la siguiente información:

Ilustración 58: resultados obtenidos página rutas

LocalBusiness vuelve a ser lo mismo que en la página principal y la página monumento.html.

La diferencia de nuevo se encuentra en el apartado *LandmarksOrHistoricalBuildings*, donde esta vez detecta exactamente 8 instancias. Este número coincide con el número de marcadores que aparecen en la ruta de la comunidad Andalucía.

Esto es porque por defecto, al entrar en la página de rutas desde el enlace de la barra de navegación, la ruta mostrada es la de Andalucía.

Si vamos a la página de otra ruta que tenga un número diferente de marcadores, el número de instancias cambiará. Por ejemplo, la comunidad Islas Baleares solo tiene 4 marcadores, para 4 monumentos. Por lo tanto, la información devuelva si cambiamos la URL es la siguiente:

Ilustración 59: resultados obtenidos de otra comunidad

Como se puede ver, detecta solo 4 elementos en lugar de los 8 de antes. Si desplegamos este apartado, al igual que en las otras páginas, nos mostrará que detecta correctamente los valores del monumento y los asocia a propiedades estandarizadas de Schema.org:

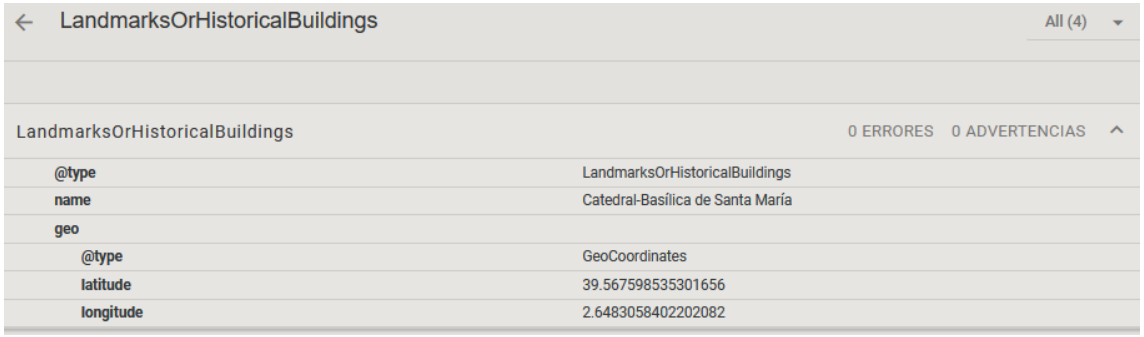


Ilustración 60: despliegue resultado obtenido LandmarksOrHisotricalBuildings

Por lo tanto, por ejemplo, el validador sabe que el valor numérico “39.5675…” es la latitud de las coordenadas del monumento con nombre “Catedral-Basílica de Santa María”.

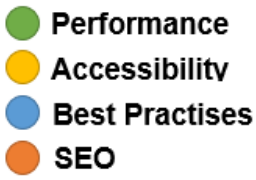
14. Auditoría de la página web

Para asegurarnos de que nuestra página web cumple con los diferentes requisitos de integridad, accesibilidad, rendimiento y SEO la hemos evaluado utilizando diferentes herramientas externas:

Lighthouse y PageSpeed Insights

Lighthouse es una herramienta de auditoría externa que permite comprobar los resultados de forma muy visual puntuándola de 0 a 100 según 4 aspectos: Rendimiento, Accesibilidad, Mejores prácticas y SEO y otorga una descripción de los diferentes problemas que pueden afectar a la puntuación de estos aspectos. Por otra parte, la página PageSpeed Insights se centra en medir el rendimiento de la web.

Siguiendo las recomendaciones de Lighthouse hemos modificado todos los aspectos posibles para aumentar la puntuación con las siguientes acciones:



- Se han redimensionado las imágenes para que tengan un tamaño lo más adaptado posible al marco de la página y se han comprimido para reducir el tiempo de carga de estas.

- Se ha añadido el atributo [alt] a todas las imágenes (descripción alternativa para mejorar la accesibilidad de la página).

    - En la página de rutas no se detecta el atributo [alt] de los iconos de los marcadores, pero están bien puestos según la documentación oficial del Leaflet.

- Nos hemos asegurado de que los botones tengan un nombre accesible de cara a los lectores de pantalla por voz.

- Se ha asegurado que todos los [id] son únicos en los elementos activos y “focuseables” como botones.

- Se han arreglado los títulos (h1, h2, etc) para que sigan un orden secuencial descendente.

- Se solicita el permiso de ubicación después de que el usuario realice una acción (interactuar con los mapas) y no al entrar a la página. (Monumento en concreto y Rutas).

- Se han intentado eliminar todos los errores por consola que dependen de nuestros documentos.

- Se han añadido los autores y una meta descripción en el head de los documentos.

A continuación, se muestran una serie de capturas de pantalla donde se reflejan las puntuaciones obtenidas en cada una de las páginas de nuestra web.

Página principal

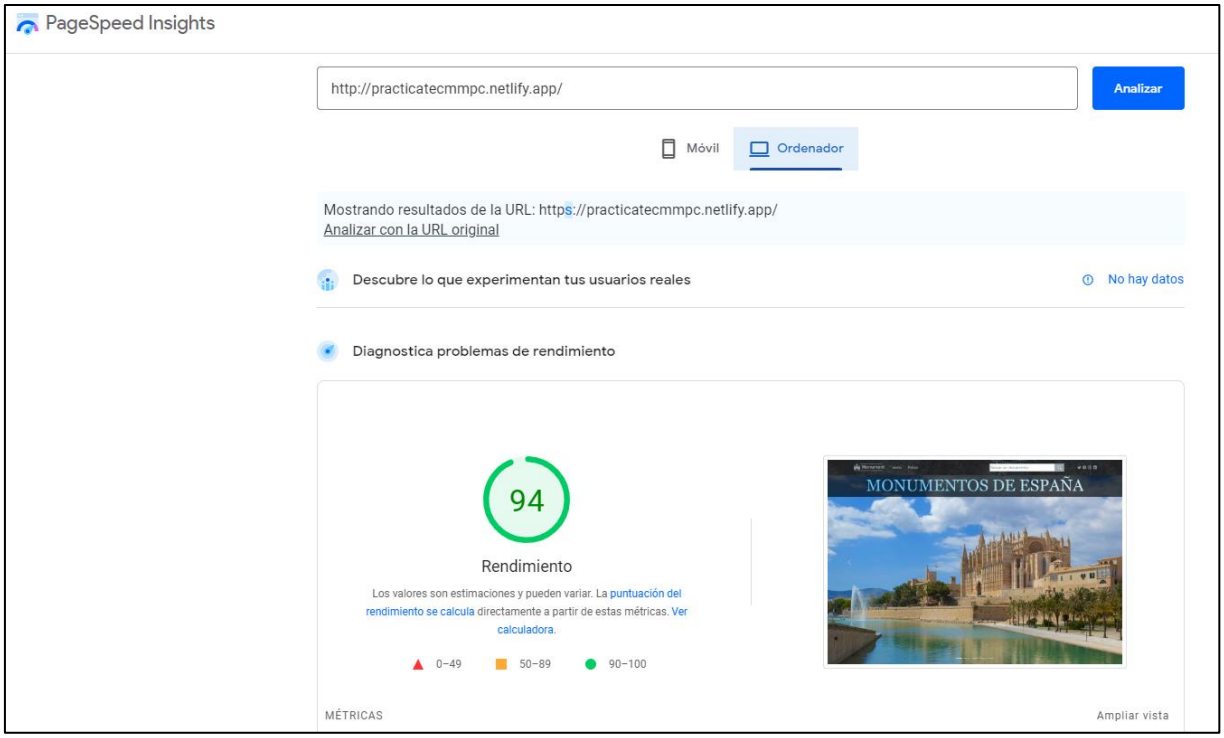


Ilustración 61: resultado obtenido con la página PageSpeed Insights de la página principal

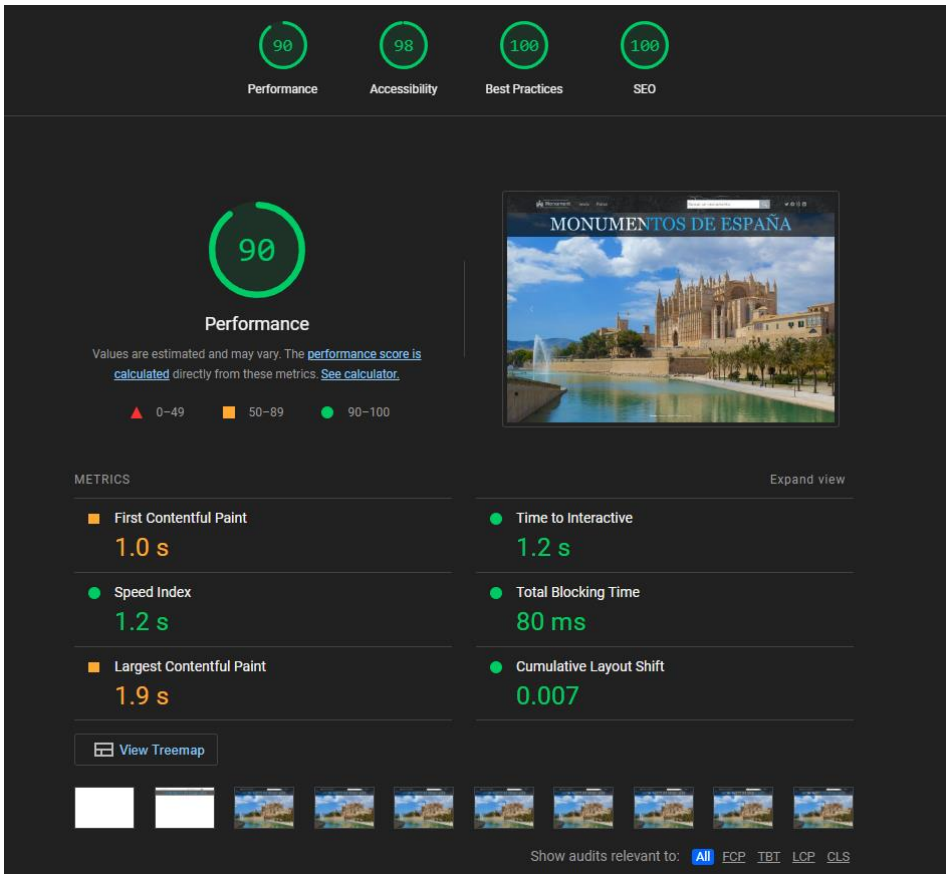


Ilustración 62: resultados obtenidos con Lighthouse de la página principal

En la página principal se han obtenido puntuaciones muy cercanas al 100. La accesibilidad no llega a tal valor debido a que considera que el contraste entre el fondo y las letras de los botones de las cartas de los monumentos no tienen suficiente contraste. En cuanto al rendimiento, se ve afectado debido al formato en que están guardadas las imágenes (jpg) y al propio host.

Página monumento en concreto

En la página de monumento en concreto la puntuación de rendimiento baja significativamente a un 75-76. Esto es debido a que hay un valor elevado de *Cumulative Layout Shift*. Hemos intentado reducir este valor tomando medidas como reservar el espacio de los elementos antes de cargar la página, pero no ha servido de mucho. Pensamos que una de las causas por las que se produce esto es por la API del tiempo, que tarda unos segundos en mostrar los datos obtenidos de la misma. El resto de los aspectos tienen una puntuación de 90 o superior.

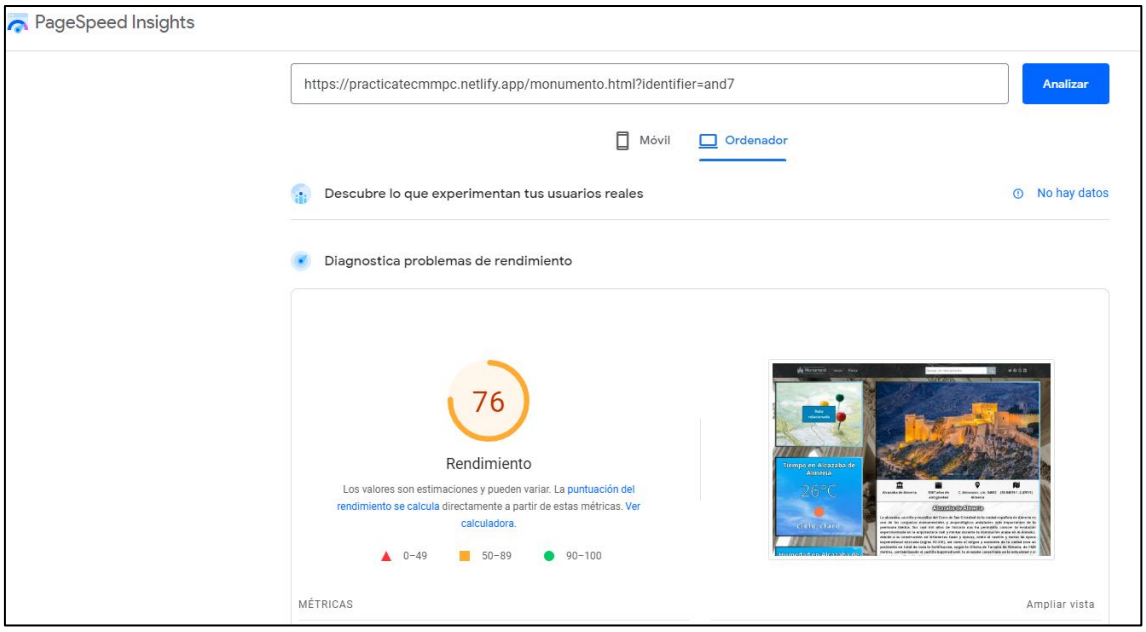


Ilustración 63: resultado obtenido con la página PageSpeed Insights de la página de monumento concreto

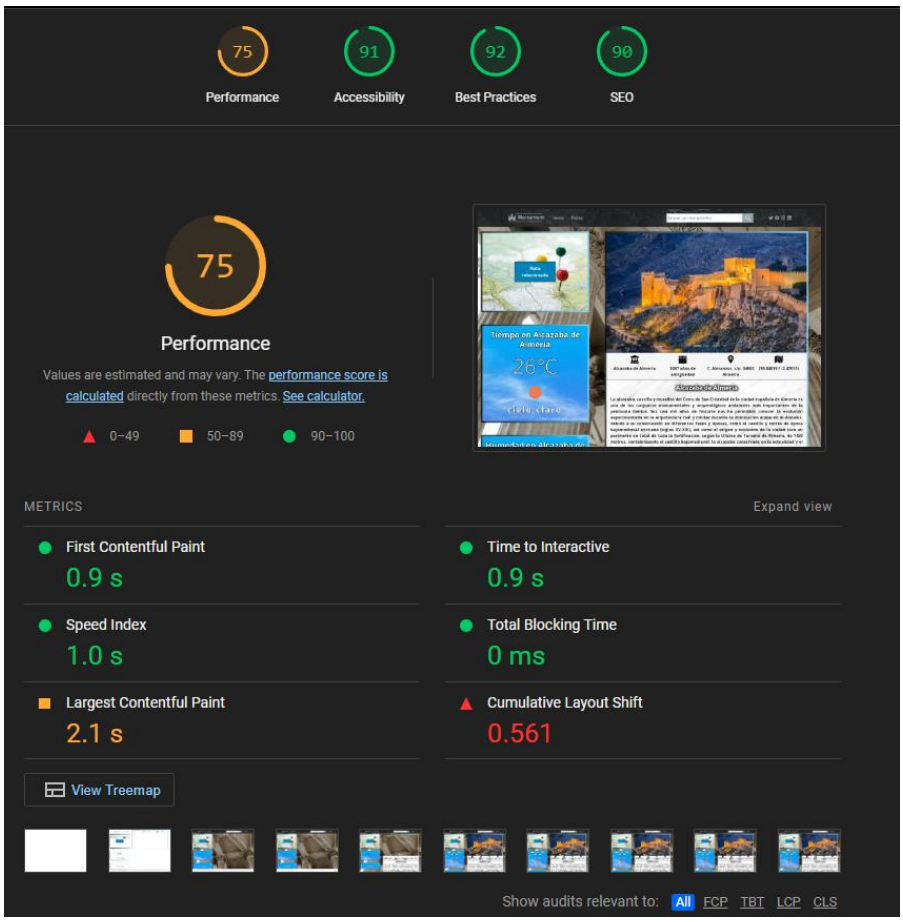


Ilustración 64: resultados Lighthouse página monumento concreto

Página de rutas

En la página de rutas todos los valores vuelven a ser bastante buenos con una puntuación superior a 90. Cabe mencionar que las puntuaciones de Acesibilidad y SEO se reducen debido a que la herramienta no detecta el texto alternativo [alt] de los iconos del mapa. Este atributo se ha añadido siguiendo la documentación oficial de la API de Leaflet pero es ignorado por la herramienta de auditoría por lo que la puntuación se ve afectada.

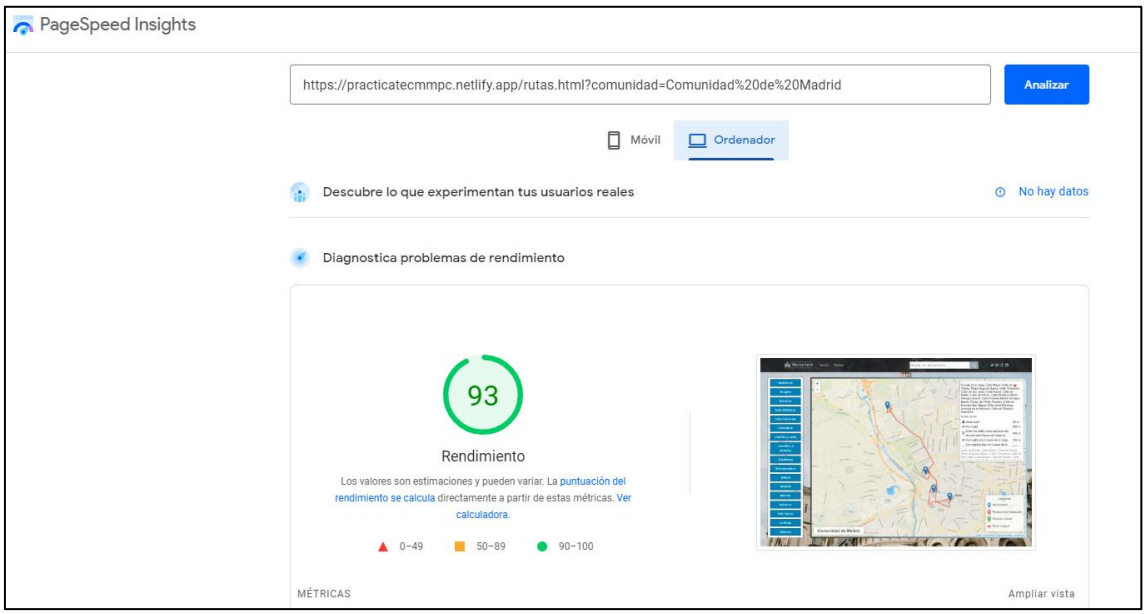


Ilustración 65: resultado obtenido con la página PageSpeed Insights de la página de rutas

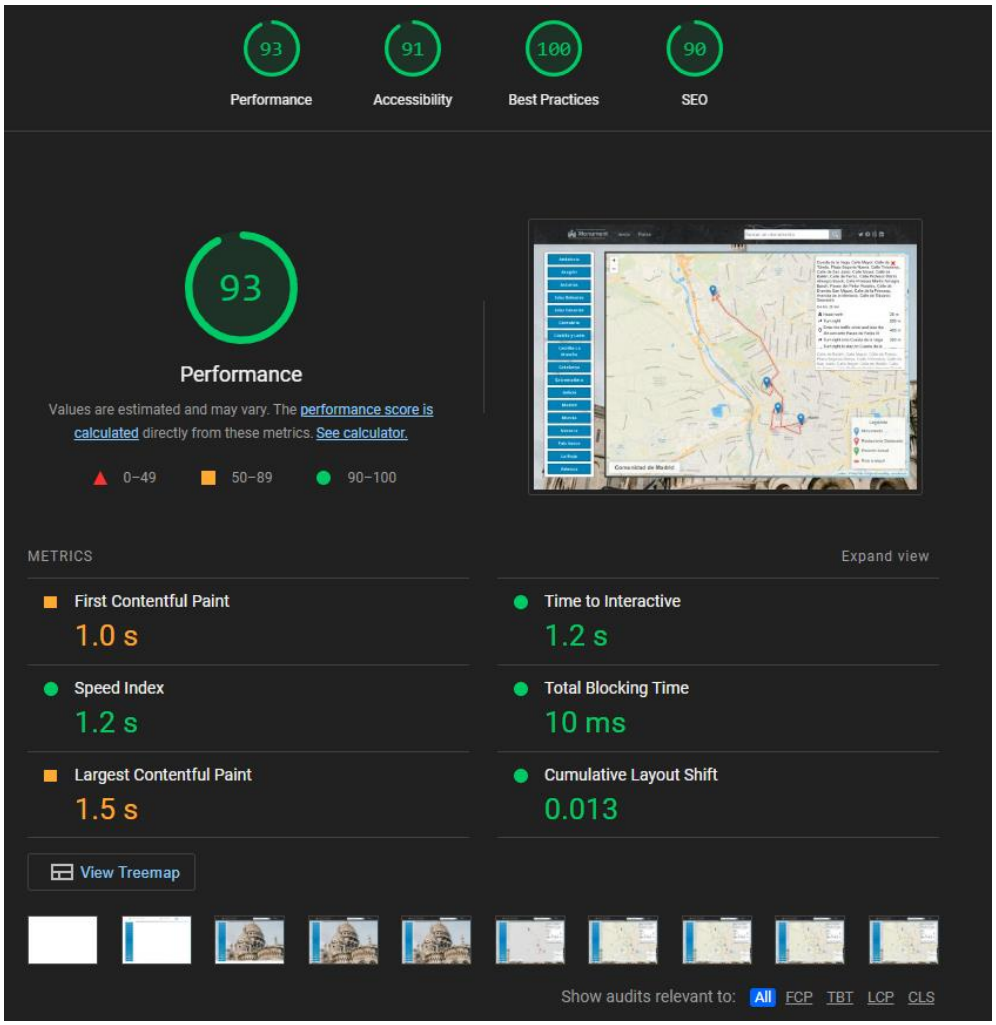


Ilustración 66: resultados Lighthouse página rutas

Nibbler

Además de las dos páginas anteriores también hemos utilizado la página Nibbler para llevar a cabo una auditoría de la web en general teniendo en cuenta otros criterios además de los anteriores.

Los apartados que nos interesan en Nibbler son los de *Accessibility* y *Technology* donde tenemos un 10 de ambos. El apartado de *Experience* también es interesante pero su nota es más baja porque se ve afectado por parámetros como tener una cuenta de Twitter vinculada, la popularidad definida por Alexa o el contenido de la página, que es supuestamente insuficiente ya que solo tiene en cuenta el texto estático.

Además de las acciones anteriormente descritas para mejorar la puntuación de las auditorías de Lighthouse (que también han servido para mejorar la de Nibbler) se ha eliminado la extensión ".html" del link de la página de rutas. Esto es para mejorar el SEO de la página. Para hacerlo se ha definido un documento .htaccess que permite reescribir la URL y eliminar dicha extensión.

De esta forma, el resultado obtenido es el siguiente:

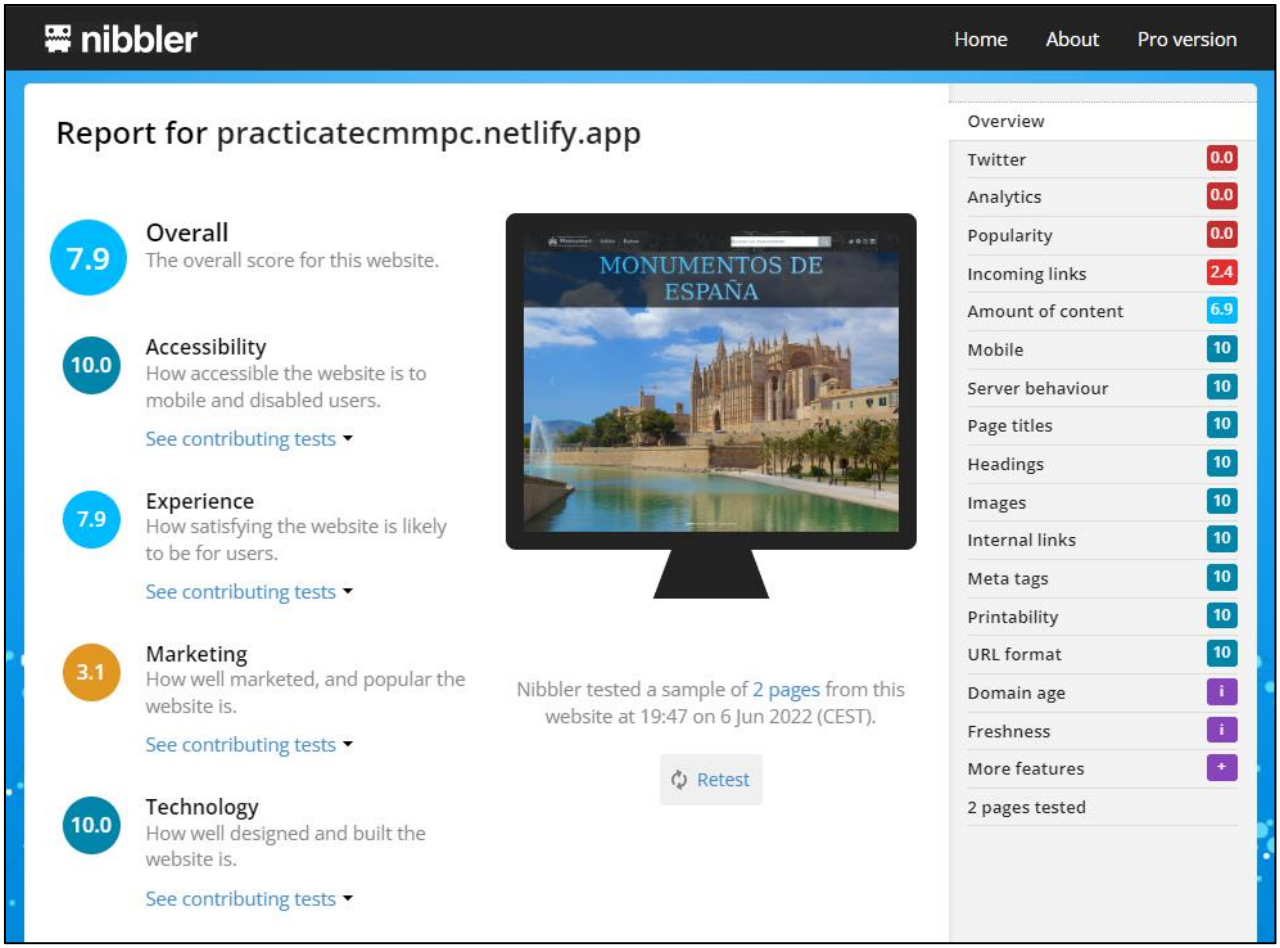


Ilustración 67: resultado obtenido auditoría con Nibbler de la web entera

15. Opinión sobre la práctica

Creemos que la práctica es imprescindible en la carrera para aprender los conceptos básicos de desarrollo web. Es importante estar realizándola sin la ayuda de un *framework* ya que de esta manera estamos obteniendo los conocimientos imprescindibles y si algún día sí utilizamos *frameworks* ya iremos con confianza porque habremos construido al menos una web con HTML, CSS y JavaScript puro.

Esta práctica nos ha permitido experimentar de primera mano en qué consiste el desarrollo web de *frontend* y, aunque nos hemos encontrado algunas dificultades, las hemos podido superar con éxito logrando crear una página web bastante vistosa y completamente funcional.

16. Autoevaluación

La nota final que nos asignamos es 8 sobre 10. La justificación es la siguiente:

Nos hemos asegurado de que nuestra página web cuente con todos los elementos necesarios para cumplir con los requisitos de la práctica. También nos hemos preocupado de obtener la mejor calificación posible en las diferentes páginas de auditoría y hemos intentado utilizar todas las tecnologías posibles dentro del tiempo disponible.

Aun así, si hubiéramos tenido algo más de tiempo nos habría gustado implementar algunas características adicionales que le podrían haber aportado más valor a nuestra página y que aumentarían nuestra nota de autoevaluación de un 8 a un 9,5 o 10:

- Comentarios implementados con PHP
- Progressive Web-App
- API de Web-Speech para aumentar la accesibilidad

17. Conclusiones

Aunque en alguna ocasión esta práctica nos ha supuesto un reto en algunos aspectos ya que no dominamos la programación web, nos ha permitido mejorar y aumentar nuestros conocimientos sobre este tipo de programación.

Echando la vista atrás, donde esta práctica nos parecía increíblemente complicada y enorme, nos damos cuenta de que, tras muchas horas de trabajo e investigación, hemos logrado crear una página web bastante decente para no tener conocimientos de desarrollo web previos. Además, el hecho de no utilizar un *framework* para desarrollarla ha permitido que adquiramos unos conocimientos más profundos sobre este tipo de programación que seguro serán útiles de cara al futuro.