Boletines de laboratorio

* * *

Introducción a la Ingeniería del Software y los Sistemas de Información II

Carlos Arévalo, Daniel Ayala, Margarita Cruz, Fernando Sola, Inma Hernández, Alfonso Márquez y David Ruiz

Curso 2024/25

Escuela Técnica Superior de
Ingeniería Informática

Índice general

Laboratorio 1

Configuración del proyecto e introducción a HTML

1.1. Objetivo

El objetivo de esta práctica es descargar y configurar un proyecto Silence que se usará a lo largo de las posteriores sesiones de laboratorio, así como a crear contenido básico en HTML5 usando las etiquetas aprendidas en teoría.

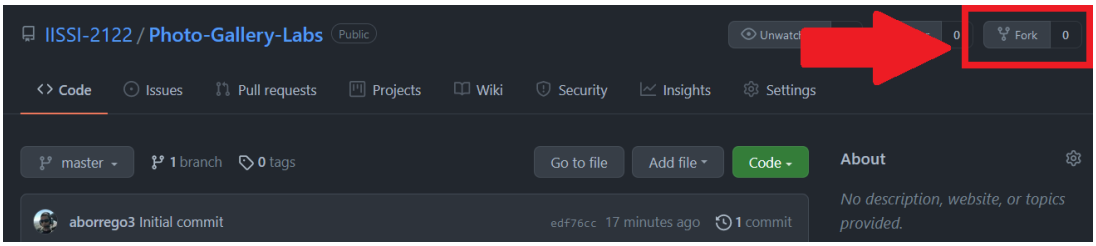
- Crear una copia de un proyecto Silence existente en GitHub.
- Descargar un proyecto en GitHub para modificarlo localmente.
- Conocer la estructura de archivos relativa a la aplicación web del proyecto.
- Usar las herramientas provistas por Silence para autogenerar archivos de configuración de endpoints y pruebas.
- Crear una nueva vista en el proyecto donde implementar contenido HTML básico.
- Subir cambios realizados localmente al repositorio GitHub.

1.2. Creación del proyecto

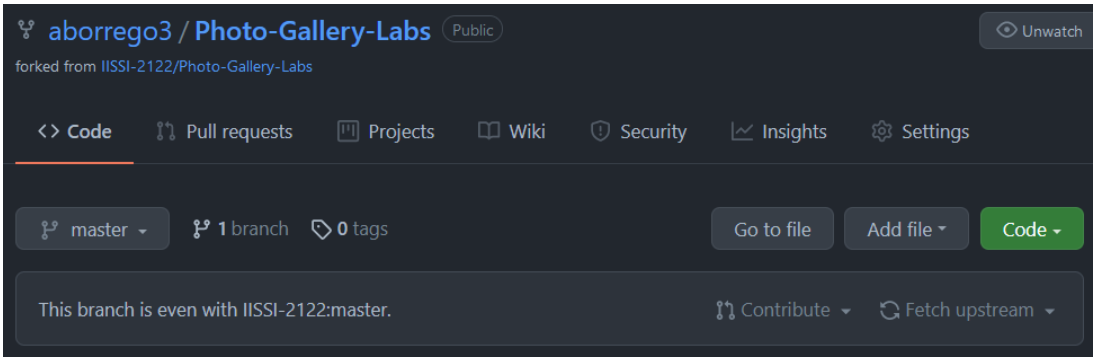
En las clases de laboratorio implementaremos un proyecto Silence que da soporte a una galería fotográfica. En la organización de la asignatura en GitHub se encuentra una versión inicial del mismo, que contiene los scripts SQL y la configuración necesaria para inicializar la base de datos. El proyecto puede encontrarse en este enlace.

Para trabajar en él en las siguientes clases de laboratorio, debemos hacer una copia de este repositorio en nuestro usuario de GitHub, para poder subir a nuestro repositorio los cambios realizados a lo largo de las sesiones de laboratorio.

Podemos hacer una copia del mismo accediendo a él, y pulsando en el botón "Fork":



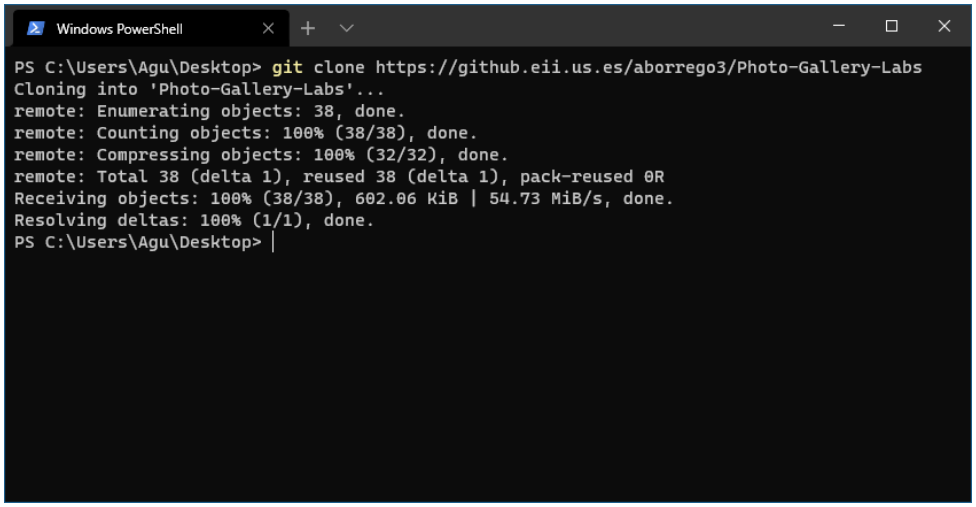
Si se nos pregunta dónde se debe realizar el fork, **seleccionaremos nuestro usuario**. Tras unos segundos de espera, la copia estará creada y podremos trabajar sobre ella:



Como se aprecia en la parte superior izquierda, se indica que este repositorio ha sido creado a partir del existente en la organización del curso. Además, se indica que en este momento, nuestra copia es idéntica al original ("*This branch is even...*"). Cuando subamos nuevos cambios, nuestra copia estará por delante de la original.

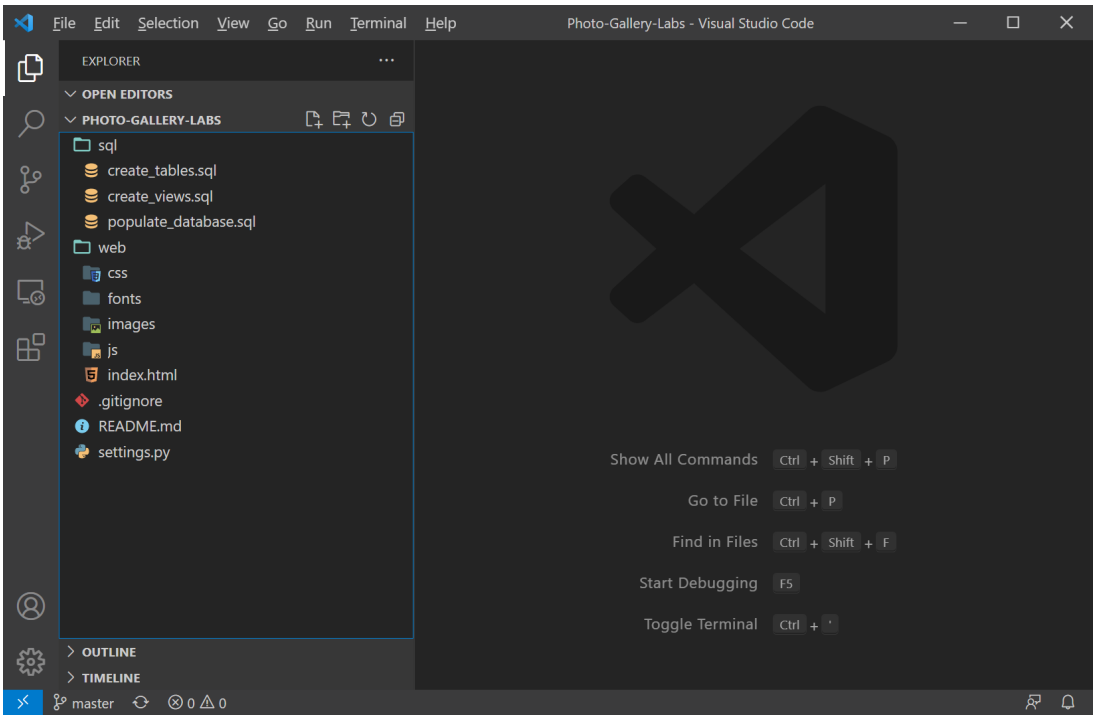
1.3. Descarga y configuración

Para trabajar en el proyecto, debemos descargar el "fork" que acabamos de crear en nuestro ordenador. Para ello, usaremos el comando "git clone":



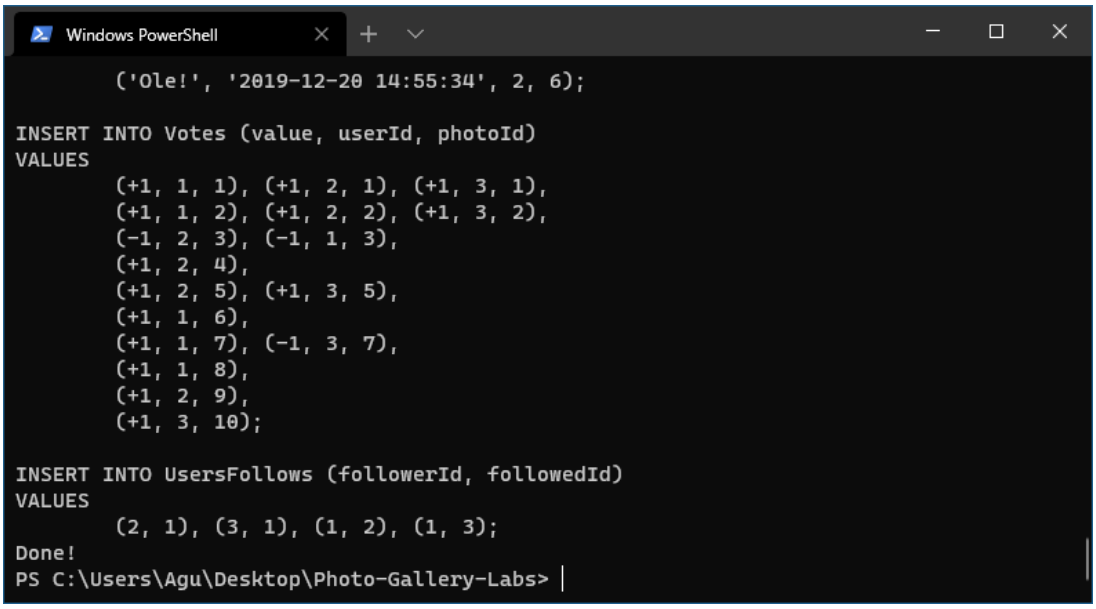
Asegúrese de que el proyecto que está clonando es su *fork*, y no el original de la organización del curso. Su UVUS debería aparecer en la URL que se está clonando, y no el de IISSI.

Esto creará una carpeta en el directorio actual de la consola con el mismo nombre del repositorio, donde se encuentran los archivos del proyecto. Si lo abrimos con VSCode, podremos comenzar a realizar cambios en el mismo:

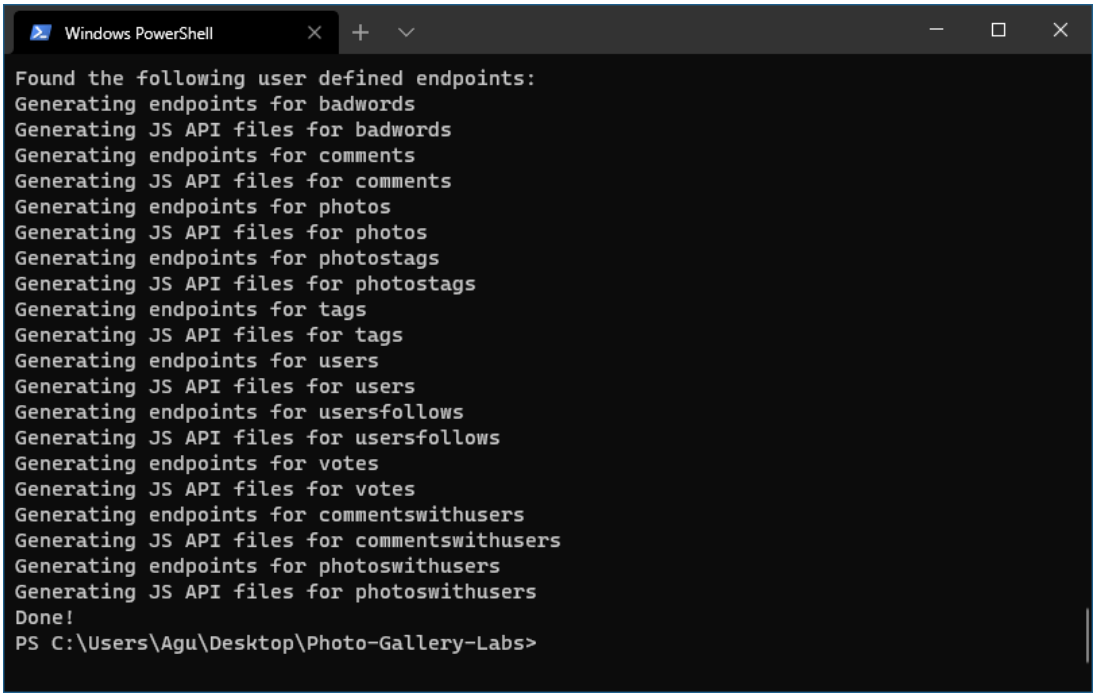


Por defecto, el proyecto está configurado para usar el usuario `iissi_user` de MariaDB, con contraseña `iissi$user`, y la base de datos `gallery`. Debemos asegurarnos de que la base de datos configurada existe, y que el usuario que se usa para la conexión tiene permisos en ella. Si alguno de estos parámetros debe modificarse, se deberá actualizar el archivo `settings.py` en la carpeta del proyecto.

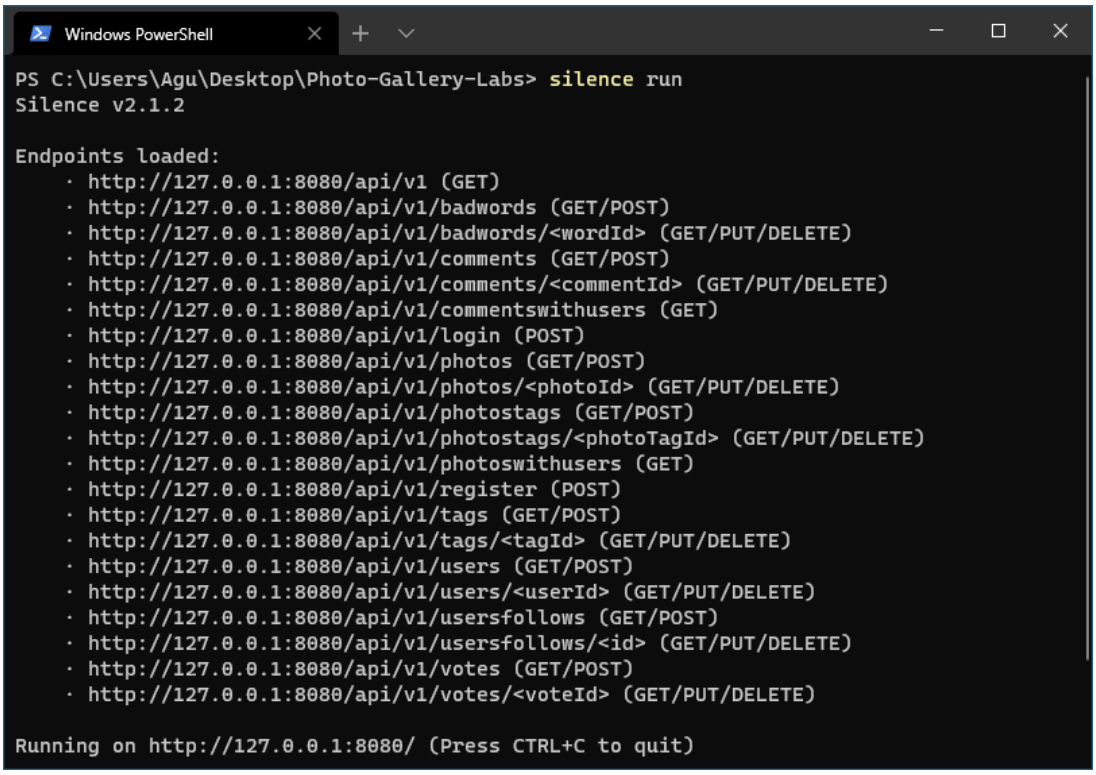
Para comprobar que la configuración es correcta, desplegaremos la base de datos de la galería fotográfica. El proyecto descargado ya incluye los ficheros SQL y la configuración adecuada, por lo que basta con ejecutar `silence createdb` en la consola. Si todo es correcto, se mostrarán las sentencias SQL ejecutadas:



Finalmente, Silence permite desplegar una API REST creada automáticamente a partir de la estructura de tablas existente en la base de datos. El comando `silence createapi` crea los archivos Python correspondientes en la carpeta `endpoints` del proyecto para definir dicha API, así como los archivos JavaScript correspondientes en la aplicación web necesarios para consumir la API creada:



Finalmente, podemos lanzar el servidor con `silence run`, donde deberían aparecer todos los endpoints disponibles en el proyecto:



1.4. Estructura de archivos

La estructura general de archivos de un proyecto Silence se introdujo en la asignatura IISSI-1. En ella, existe una carpeta `web/` que contiene los archivos asociados a la aplicación web del proyecto. La estructura de esta carpeta es la siguiente:

- `css/`: Carpeta que contiene las hojas de estilo.
- `fonts/`: Carpeta que contiene los archivos de fuentes tipográficas.
- `images/`: Carpeta que contiene las imágenes usadas en la aplicación.
- `js/`: Carpeta que contiene los ficheros JavaScript.
- `*.html`: Vistas HTML de la aplicación.

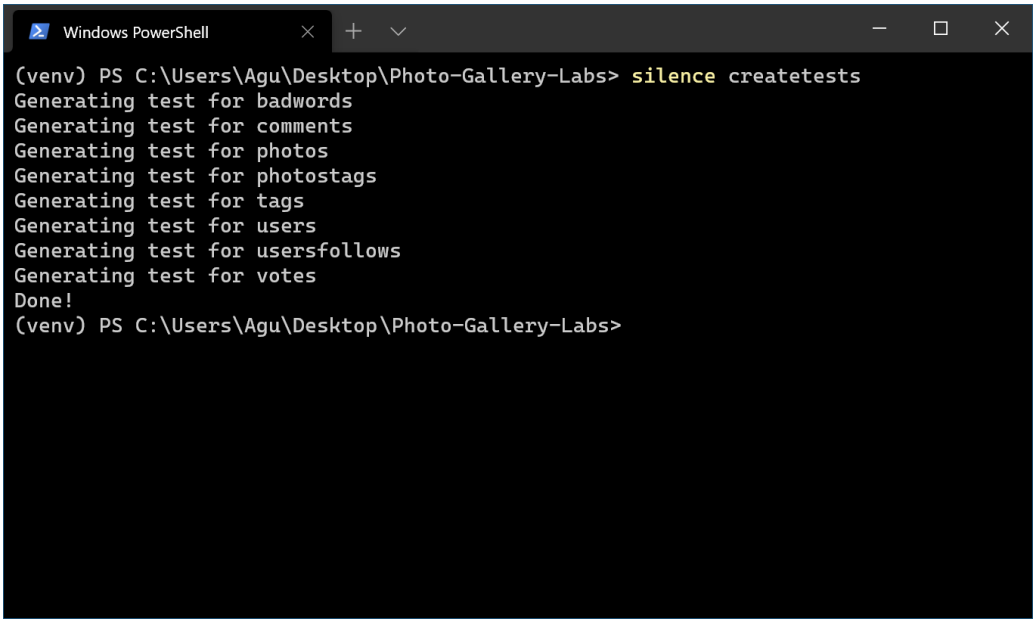
A su vez, la carpeta `js/` contiene:

- `api/`: Módulos de acceso y consumo de la API RESTful del proyecto
- `libs/`: Librerías JS externas usadas en la aplicación
- `renderers/`: Módulos de renderizado de entidades
- `utils/`: Módulos de utilidad general
- `validators/`: Módulos de validación de formularios
- `*.js`: Archivos JS asociados a cada una de las vistas de la aplicación

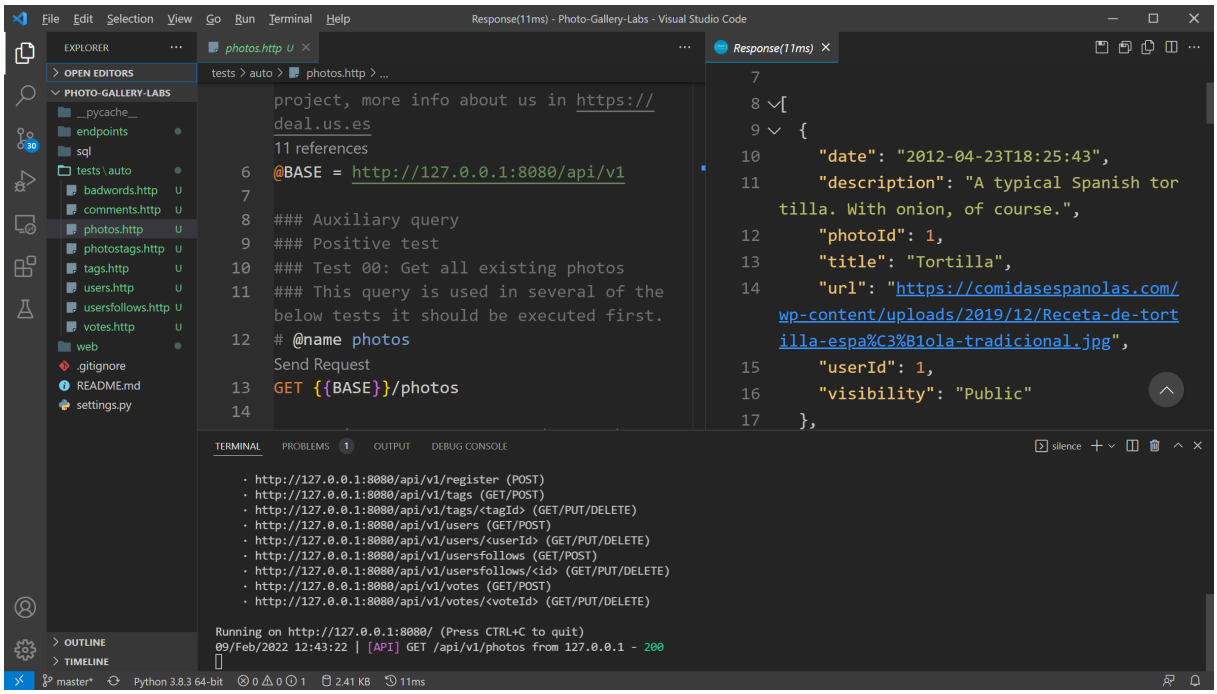
En esta sesión trabajaremos con archivos `.html` para crear contenido usando este lenguaje de marcado. En posteriores sesiones de laboratorio se cubrirán el contenido y uso del resto de componentes de la aplicación web.

1.5. Autogeneración de tests

Silence es capaz de generar archivos de tests automáticamente para los endpoints generados a partir de las tablas de la base de datos. Para ello, debemos ejecutar el comando `silence createtests` en la consola:



Los tests se generan en la carpeta `tests/auto/`, y se pueden ejecutar si se tiene instalada la extensión REST Client de Visual Studio Code y el servidor se encuentra corriendo mediante `silence run`:



1.6. Creación de contenido HTML

Para las siguientes secciones, deberemos haber iniciado el servidor mediante el comando `silence run`, tal y como se explicó en la anteriormente.

1.6.1. Elementos HTML básicos

Si accedemos en nuestro navegador a la dirección en la que se está ejecutando el servidor web de Silence, se nos presentará una página en blanco. Esto es debido a que la página por defecto es la definida en el fichero `index.html`, y éste está vacío.

Comenzaremos a darle contenido al `index.html` con la estructura común a todos los archivos HTML:

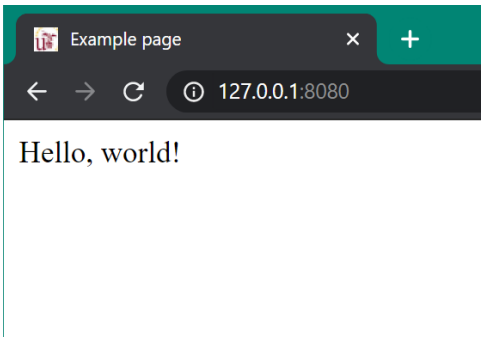
```
<!DOCTYPE html>
<html>
    <head>

    </head>

    <body>

    </body>
</html>
```

La primera línea sirve para declarar la versión HTML usada, en este caso `html` representa HTML5. Todo el documento está contenido en la etiqueta `<html>`, que a su vez contiene una cabecera `<head>` con información para el navegador, y un contenido `<body>` que es el que se muestra al usuario.

Dentro de la cabecera podemos incluir metadatos sobre la página, por ejemplo, su título o una imagen de miniatura o *favicon* que aparecerá en la pestaña del navegador. A su vez, todo lo que se encuentre dentro del cuerpo de la página será mostrado en el navegador:

```
<head>
    <title>Example page</title>
    <link rel="icon" href="/images/favicon.ico">
</head>

<body>
    Hello, world!
</body>
```

Si refrescamos la página, aparecen los cambios realizados en `index.html`:

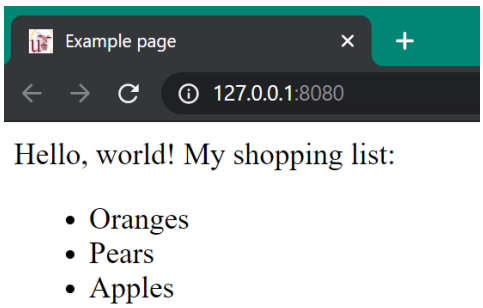


Note cómo el título y el icono de la pestaña se corresponden con lo definido dentro de la etiqueta `<head>`. A su vez, la ruta del icono hace referencia al archivo que puede encontrarse en la carpeta `images` de la aplicación web. La carpeta `docs/` del proyecto es la raíz de la aplicación web, por lo que todas las rutas relativas a archivos HTML, CSS o JS hacen referencia al contenido de esa carpeta.

Podemos seguir añadiendo contenido en el cuerpo del documento, por ejemplo, una lista no numerada mediante las etiquetas `<ul>` (***u**nordered **l**ist*) y `<li>` (***l**ist **i**tem*):

```
My shopping list:
<ul>
    <li>Oranges</li>
    <li>Pears</li>
    <li>Apples</li>
</ul>
```

Si incluimos la lista después del "hola mundo", el resultado es el siguiente:



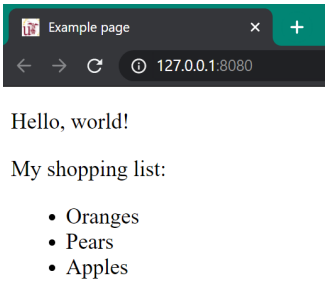
Pruebe a cambiar la etiqueta `<ul>` por `<ol>` (***o**rdered **l**ist*) y observe el resultado.

Note que, pese a separar el texto con saltos de línea, estos no se reflejan en el contenido mostrado por el navegador. Esto es debido a que los saltos de línea en un fichero HTML son generalmente ignorados por los navegadores y no influyen en la forma de organizar el contenido que se muestra.

Para forzar a que la lista de la compra aparezca bajo el "hola mundo" podemos envolver cada línea en una etiqueta de párrafo `<p>` (***p**aragraph*):

```
<p>Hello, world!</p>

<p>My shopping list:</p>
<ul>
    <li>Oranges</li>
    <li>Pears</li>
    <li>Apples</li>
</ul>
```

Un elemento de párrafo, que debe contener texto, ocupa todo el ancho posible, y se disponen visualmente unos sobre otros:

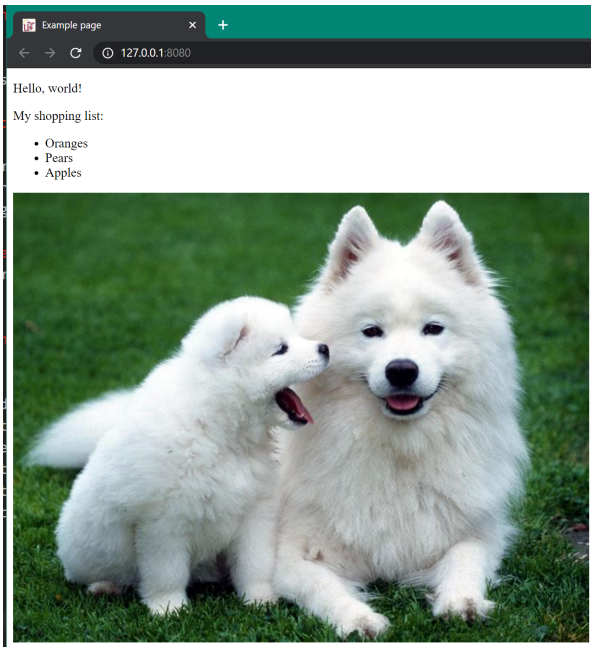


También se pueden incluir imágenes, para lo que se deben guardar en la carpeta `images` del proyecto. Suponiendo una imagen llamada `dogs.jpg`, se puede referenciar del siguiente modo:

```
<img src="/images/dogs.jpg"
     title="Two dogs"
     alt="Two dogs rest in a grass field">
```

La ruta de la imagen, relativa al directorio de nuestra aplicación web, se indica en el atributo `src`. El atributo `title` almacena el título de la foto, que se muestra al detener el ratón sobre ella, mientras que el atributo `alt` contiene una descripción de la imagen. Los dos últimos atributos no son obligatorios, pero sí altamente recomendados por motivos de accesibilidad. Note también que la etiqueta `<img>` no necesita etiqueta de cierre, ya que no es necesaria al no poder almacenar en su interior otras etiquetas.

El resultado es el siguiente:



Por defecto, las imágenes aparecen a su tamaño original. En el boletín correspondiente a estilos CSS se mostrará cómo modificar este comportamiento.

1.6.2. Enlaces a otros documentos

Una pantalla de la aplicación web, representada mediante un documento HTML, puede contener enlaces a otras vistas contenidas en otros documentos HTML. A modo de ejemplo, crearemos un archivo `register.html` en la carpeta `web`, junto al ya existente `index.html`. En este archivo introduciremos un formulario que podrá servir en el futuro para dar de alta a un usuario, por ejemplo.

El contenido inicial de `register.html` será el siguiente:

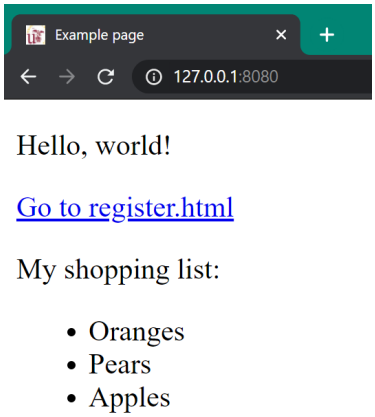
```
<!DOCTYPE html>
<html>
    <head>
        <title>Register form</title>
        <link rel="icon" href="/images/favicon.ico">
    </head>

    <body>
        TO-DO
    </body>
</html>
```

Note como las cabeceras en el `<head>` de los documentos HTML de una misma aplicación suelen ser similares, mientras que el contenido del `<body>` varía.

A continuación, modificaremos `index.html` para añadir un enlace al nuevo documento creado usando una etiqueta `<a>` (***a**nchor*):

```
(...)
<p>Hello, world!</p>

<a href="register.html">Go to register.html</a>

<p>My shopping list:</p>
(...)
```

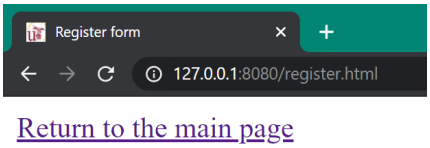
El documento al que se enlace se indica en el atributo `href`, mientras que el interior de la etiqueta contiene el texto que se mostrará como enlace:



Pulsar en este enlace nos llevará a `register.html`. Implementemos en este archivo un enlace para ir de vuelta a `index.html`:

```
<a href="index.html">Return to the main page</a>
```

Tendremos así navegación bidireccional entre ambas vistas:



1.6.3. Formularios

Los formularios son una parte fundamental de cualquier aplicación web, ya que permiten al usuario introducir datos para su procesamiento por parte del servidor. A modo de ejemplo, implementaremos un formulario de registro en `register.html`. Los formularios están contenidos dentro de etiquetas `<form>`:

```
<form>
    ...
</form>
```

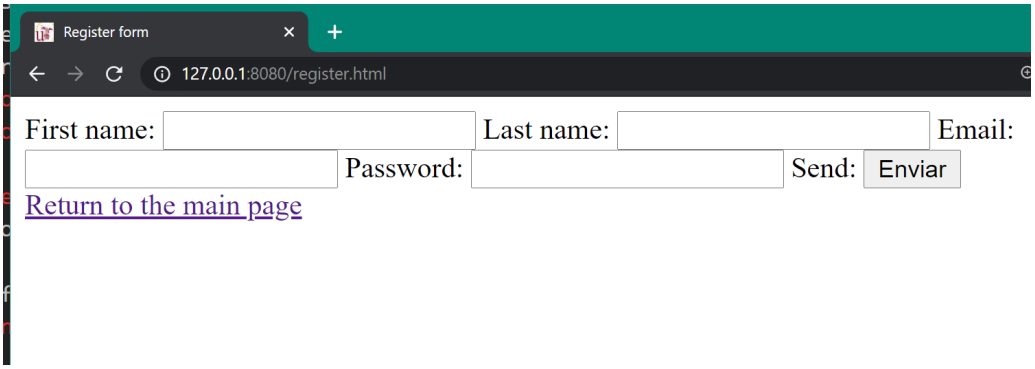
Opcionalmente, las etiquetas `<form>` pueden tener atributos como `action` para definir la ruta del servidor que ha de procesar los datos enviados, y `method` para establecer el método HTTP a usar.

Cada campo del formulario se define mediante etiquetas `<input>`:

```
<form>
    First name: <input type="text" name="firstName">
    Last name: <input type="text" name="lastName">
    Email: <input type="email" name="email">
    Password: <input type="password" name="password">
    Send: <input type="submit">
</form>
```

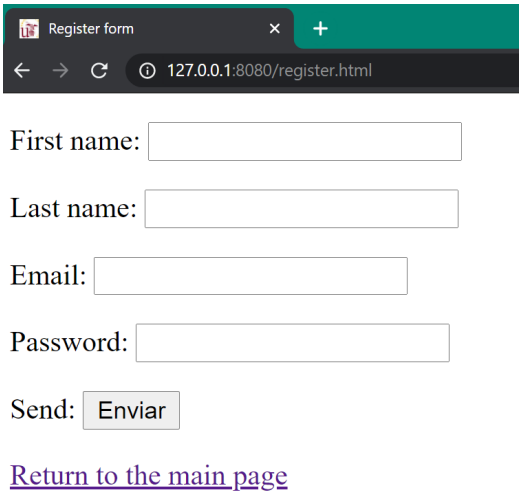
Observe que los elementos `<input>` tienen un atributo `type` que indican el tipo de elemento de entrada del que se trata (dispone aquí de un listado completo de tipos de inputs), y un atributo `name` que determina el nombre del campo que se envía al servidor con el valor proporcionado.

El resultado que muestra el navegador es:



Como ya ocurrió antes, los saltos de línea no tienen un efecto directo en la estructura visual del documento, y los campos se muestran uno tras otro. Podemos solucionar este problema incluyendo cada campo del formulario dentro de una etiqueta `<p>`, que se muestran una encima de otra:

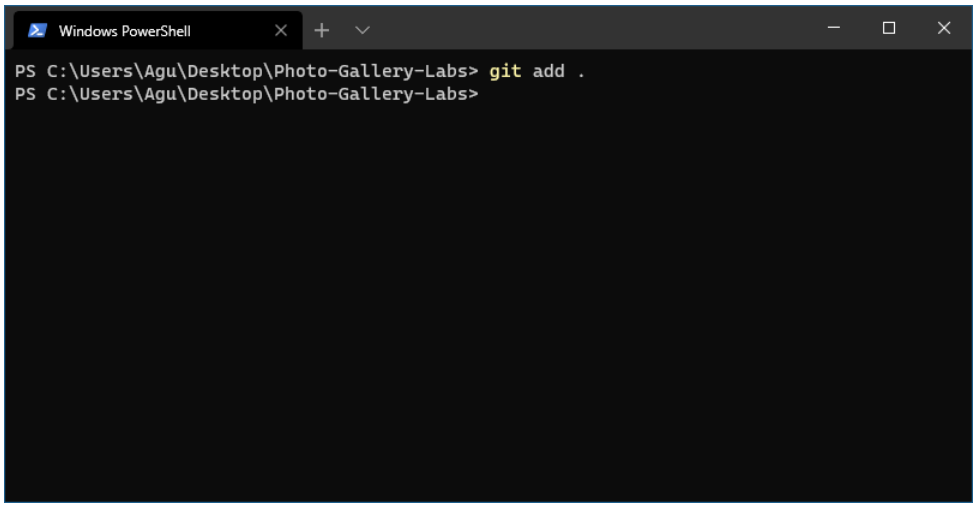
```
<form>
    <p>
        First name: <input type="text" name="firstName">
    </p>
    <p>
        Last name: <input type="text" name="lastName">
    </p>
    <p>
        Email: <input type="email" name="email">
    </p>
    <p>
        Password: <input type="password" name="password">
    </p>
    <p>
        Send: <input type="submit">
    </p>
</form>
```



1.7. Subida de cambios a GitHub

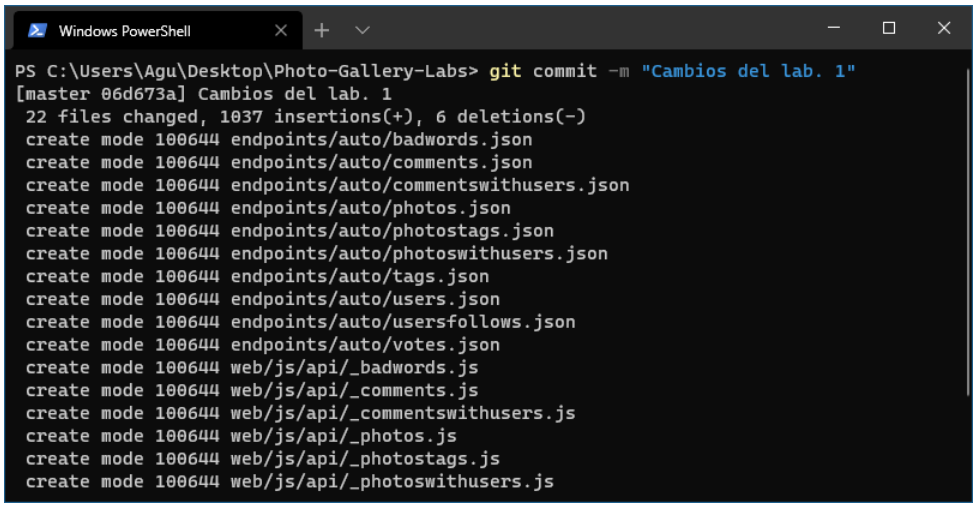
Durante el trascurso de las sesiones de laboratorio, iremos actualizando nuestro repositorio en GitHub con los cambios que hagamos en la galería fotográfica. A modo de ejemplo, realizaremos un nuevo *commit* con los cambios realizados en este laboratorio:

En primer lugar, añadiremos todos los archivos modificados al commit a realizar mediante el comando `git add .`:



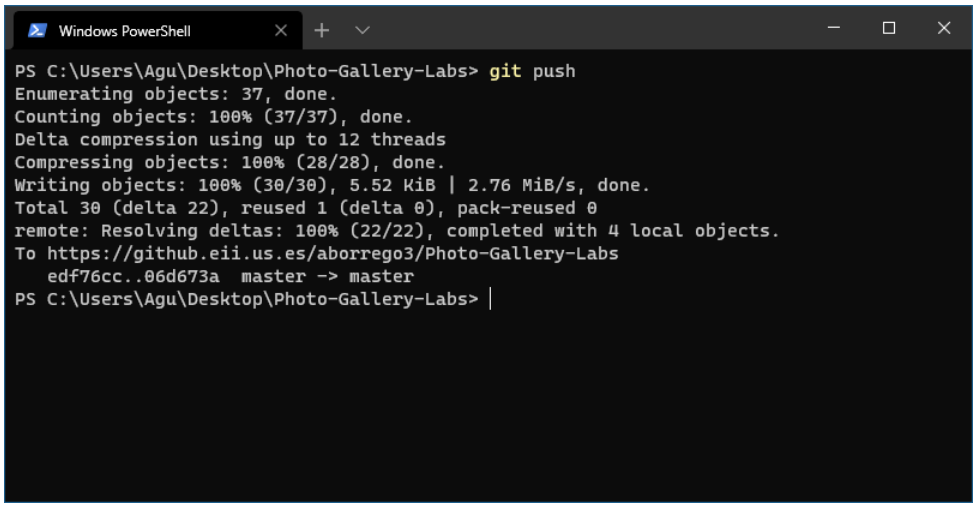
Por defecto, el comando `git add` no muestra ninguna salida si su ejecución es correcta. Si, en lugar de añadir todos los archivos modificados al commit, se desearan añadir sólo algunos en particular, se podrían especificar en el comando anterior, en lugar de usar el punto para incluirlos todos.

A continuación haremos un *commit*, con un mensaje descriptivo:

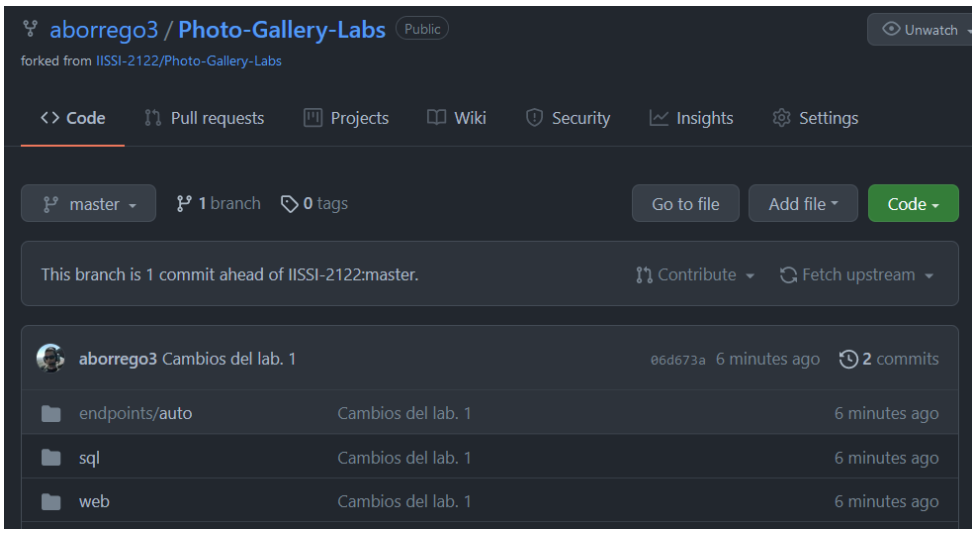


Note que la gran cantidad de archivos añadidos se deben a la ejecución del comando `silence createapi`, que creó una serie de archivos autogenerados.

Finalmente, subiremos los cambios a GitHub con `git push`:



Podremos ver que nuestro repositorio en GitHub ahora contiene los cambios más recientes:



Laboratorio 2

HTML y CSS básico

2.1. Objetivo

El objetivo de esta práctica es introducir al alumno a los conceptos básicos de HTML y CSS necesarios para construir una versión preliminar de la aplicación a desarrollar. El alumno aprenderá a:

- Utilizar las etiquetas básicas HTML para organizar los elementos en un sitio web.
- Usar y enlazar hojas de estilo CSS para dar formato lógico básico a los elementos mostrados.

2.2. Introducción

HTML5 y CSS3 son tecnologías básicas para la web que conocemos hoy en día. Si bien es cierto que el uso de frameworks que automatizan gran parte del trabajo de escritura de etiquetas HTML y maquetación CSS se ha vuelto común, el manejo básico de estos lenguajes es una competencia fundamental que todo desarrollador debe tener.

En esta práctica se mostrará una posible implementación, usando estos lenguajes, de algunos de los requisitos de la aplicación a desarrollar. Dado que el aspecto final de la interfaz queda a elección del alumno, no es necesario que ésta tenga el mismo aspecto que el mostrado en esta práctica, siempre que el acabado final cumpla con todos los requisitos funcionales solicitados en el documento de requisitos.

Es también importante recalcar que, en la mayoría de ocasiones, el maquetado

final de una aplicación web estará a cargo de una o varias librerías CSS, y no únicamente de código CSS desarrollado a mano. Cuando esto ocurre, es común que el código HTML sufra cambios sustanciales para adaptarlo a las demandas de cada librería CSS concreta. Por ello, el código desarrollado en esta práctica debe interpretarse como un segundo paso en el prototipo de la aplicación a desarrollar, que ahora será ejecutable en el navegador, pero que aún estará sujeto a cambios para mejorar sustancialmente su acabado. En la siguiente práctica se mostrará como integrar una de estas librerías CSS, Bootstrap, con nuestra aplicación, así como los beneficios y desventajas que ésto conlleva.

2.3. Vista global de la galería

Como se explicó en la práctica anterior, la vista contenida en el archivo `index.html` es la que se mostrará por defecto al acceder a la aplicación. Es por ello que, idealmente, es este archivo el que debería contener la vista a la que se accederá cuando el usuario ingrese a la aplicación. En nuestro caso, aprovecharemos el `index.html` para mostrar las fotos que contiene la galería.

Dado que nuestra aplicación no cuenta aún con funcionalidad JavaScript, no podremos comprobar si el usuario está logueado o no, a fin de mostrarle únicamente las opciones relevantes (por ejemplo, un usuario no logueado no puede crear fotos). En esta versión incluiremos todas las opciones disponibles, y más adelante aprenderemos a ocultarlas si es necesario.

Comenzaremos por añadir un título a la vista de listado de fotos, junto con un enlace para poder crear una nueva foto, que nos llevará a una página diferente:

```
<body>
 <h2>Recent pictures</h2>
 <a href="upload_picture.html">Upload a new picture</a>
...
```

A continuación, crearemos un bloque lógico para mostrar una imagen junto con alguna de su información relacionada: título, descripción y puntuación media. Además, añadiremos iconos para valorar la imagen, junto con botones para editarla o borrarla:

```
<div class="photo-block">
 <div class="photo-header-block">
  <h2 class="photo-title">Samoyed</h2>
  <h3 class="photo-score">Score: 4.52</h3>
 </div>

 <div class="photo-image-block">
    <img class="photo-image" src="https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg">
 </div>

 <div class="photo-metadata-block">
  <p class="photo-description">A very good boy.</p>
 </div>

 <div class="photo-edit-block">
  <button>Edit this photo</button>
  <button>Delete this photo</button>
 </div>
</div>
```

Es buena idea agrupar elementos similares dentro de bloques `<div>` y establecer clases para cada uno de ellos. Las etiquetas `<div>`, por defecto, no tienen un impacto visual directo, sino que se usan para agrupar elementos que, juntos, forman conjuntos lógicos en nuestras vistas. Así, podremos aplicarles estilos conjuntos a todos ellos simplemente referenciando el `<div>` correspondiente, en lugar de repetir el mismo estilo CSS para cada uno de ellos. El resultado visual de este código HTML es el siguiente:

Ahora, con una única foto en nuestra galería, es buen momento para afinar el estilo de cada bloque de imagen mediante CSS. Para ello, crearemos un archivo `style.css` en la carpeta `css/`, y lo enlazaremos a la vista incluyedo la siguiente etiqueta dentro de la cabecera `<head>`:

```
<link rel="stylesheet" href="/css/style.css">
```

Una vez enlazado, podemos escribir estilos en `style.css` para, por ejemplo, centrar horizontalmente todo el contenido, limitar la altura máxima de cada imagen, ajustar los tamaños y fuentes de cada texto, etcétera. Además, podemos añadir un borde alrededor de todo el bloque, para separarlos visualmente cuando haya varios. Este estilo será de aplicación en cualquier vista que enlace la hoja de estilos `style.css`. Se muestra a continuación un ejemplo de código CSS para lograr estos objetivos:

```
div.photo-block {
    text-align: center;
    border: 1px solid gray;
    border-radius: 5%;
}

div.photo-title {
    font-size: 140%;
    font-family: Arial;
    text-decoration: underline;
}

div.photo-score {
    font-family: 130%;
    font-family: monospace;
}

img.photo-image {
    max-height: 200px;
    width: auto;
}
```

Así, resulta:

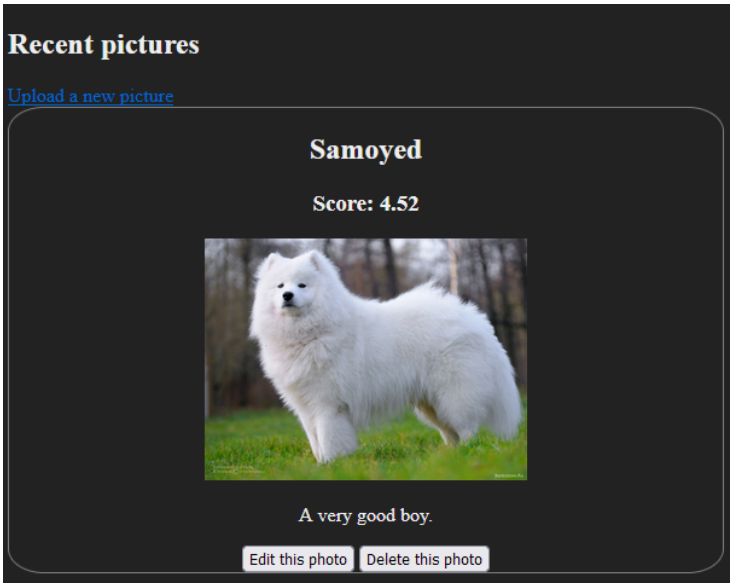
Adicionalmente, podemos hacer cambios en el estilo que afecten a toda la página. Como ejemplo, desarrollaremos nuestra galería con un tema oscuro (texto claro, fondo oscuro) para reducir la fatiga visual. Para ello, añadiremos una clase `dark` a la etiqueta `<body>` y aplicaremos el estilo CSS correspondiente:

```
<body class="dark">
...
```

Y el CSS resulta:

```
body.dark {
 background-color: #222222;
 color: #eeeeee;
}
```

Donde `background-color` y `color` son los atributos CSS que definen el color de fondo y el color del texto, respectivamente. El resultado visual es el siguiente:



Dado que la etiqueta `<body>` contiene todas las demás, los estilos aplicados a ella también a todo el contenido de la página. Sin embargo, cualquier etiqueta interior que defina su propio estilo tendrá prioridad sobre el estilo de la etiqueta `<body>`. Por ejemplo, los enlaces `<a>` siguen siendo azules, pese a que el color de texto de `<body>` ha sido cambiado a blanco.

Una vez hayamos adaptado el estilo de cada "bloque" fotográfico a nuestro gusto, podemos componer una galería reutilizando estos bloques. En el pasado, se estructuraban este tipo de páginas usando etiquetas `<table>` y organizando la vista mediante una tabla oculta, por ejemplo, mostrando tres imágenes por fila, usando tantas filas como sean necesarias. Esta práctica está considerada obsoleta y por lo tanto se desaconseja.

Actualmente CSS ofrece un sistema potente de tablas lógicas o `grid` para realizar una estructura similar. La principal ventaja de usar un grid CSS es que este formato no está determinado por el contenido HTML (formateado físico), y por lo tanto sus posteriores cambios serán mucho más sencillos al requerir únicamente modificaciones en la hoja de estilos, no en el contenido HTML. Además, se adapta automáticamente a pantallas de todo tipo (responsividad), mientras que el formateado físico no lo hace.

Para agrupar nuestras fotos en una grid CSS, comenzaremos por definir una etiqueta `<div>` que contendrá todas las imágenes. Esta etiqueta será el *container*. Indicaremos que los elementos dentro del *container* serán mostrados en un grid usando CSS. Además, declararemos que nuestro grid tendrá tres columnas, cada una de ellas ocupando el 33 % del ancho de la página:

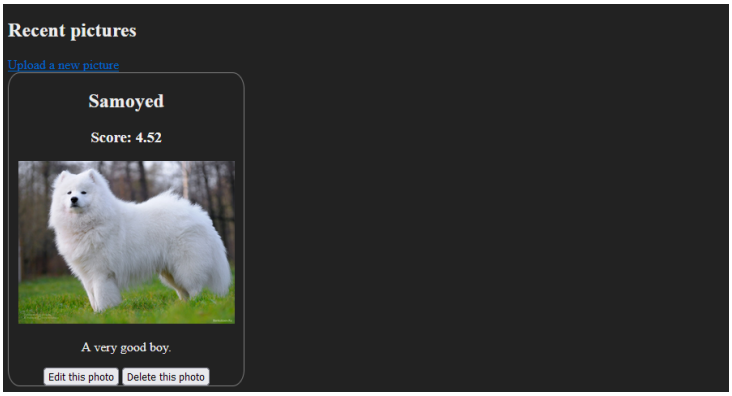
index.html:

```
<div class="gallery-container">
 <!-- Our photo-blocks will be here -->
</div>
```

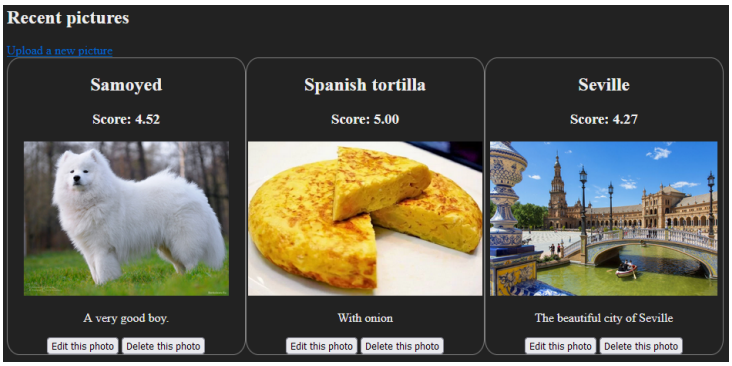
style.css:

```
.gallery-container {
    display: grid;
    grid-template-columns: 33% 33% 33%;
}
```

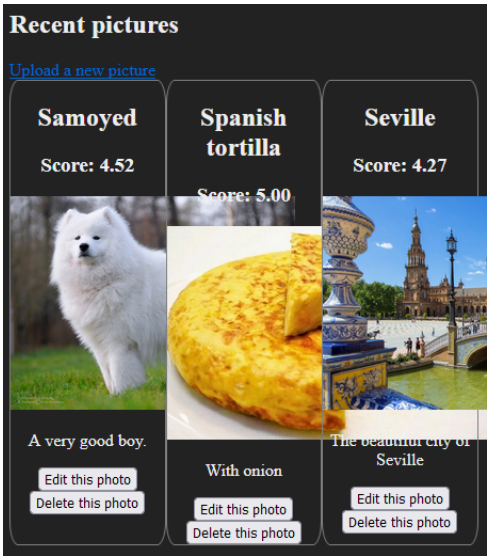
Podemos mover el `<div>` que contiene toda la información de nuestra imagen dentro de este contenedor, y el resultado será el siguiente:



Ahora, nuestro bloque de imagen ocupa la primera posición en esta tabla virtual con un ancho del 33 % del total. Podemos replicar el `<div class="photo-block">` y su interior para seguir poblando nuestra galería, con datos diferentes en cada uno de ellos. Observaremos que, gracias al grid CSS, cada bloque se coloca en su lugar sin necesidad de indicarlo en el código HTML:



Si siguiéramos añadiendo fotos, dado que ya se ha rellenado una fila de 3 fotos, la siguiente se colocaría debajo de la primera en una nueva fila. Además, si redimensionamos la página, la anchura de cada columna cambia para adaptarse. Sin embargo, las fotos no se redimensionan automáticamente:



Podemos solucionar este problema añadiendo un ancho dinámico a las imágenes en cada columna: si establecemos su ancho al 100 %, ésta ocupará todo el ancho de la columna, haciéndose más pequeña si es necesario. El alto se redimensionará automáticamente para mantener la proporción de aspecto:

```
img.photo-image {
    width: 100%;
    height: auto;
}
```

2.4. Formulario de registro

En la práctica anterior creamos una versión preliminar del formulario de registro de la aplicación. Ahora procederemos a refinarlo, añadiendo los campos adicionales que

sean necesarios y perfeccionando su estilo mediante CSS:

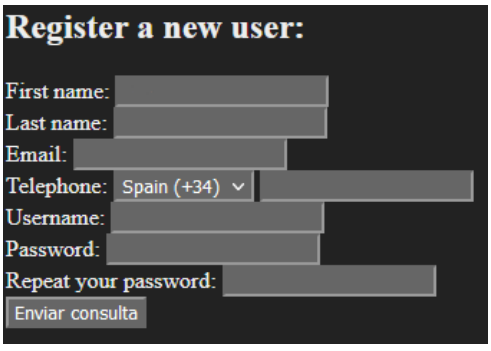
```
<h2>Register a new user:</h2>
<form>
 <div class="input-group">
  <label for="firstname-input">First name:</label>
  <input type="text" id="firstname-input" name="firstName" required>
 </div>

 <div class="input-group">
  <label for="lastname-input">Last name:</label>
  <input type="text" id="lastname-input" name="lastName" required>
 </div>

 <div class="input-group">
  <label for="email-input">Email:</label>
  <input type="email" id="email-input" name="email" required>
 </div>

 <div class="input-group">
  <label for="telephone-input">Telephone:</label>
  <select name="prefix" id="prefix-input">
   <option value="34">Spain (+34)</option>
   <option value="1">USA (+1)</option>
   <option value="other">Other</option>
  </select>
  <input type="text" id="telephone-input" name="telephone" required>
 </div>

 <div class="input-group">
  <label for="username-input">Username:</label>
  <input type="text" id="username-input" name="username" required>
 </div>

 <div class="input-group">
  <label for="password-input">Password:</label>
  <input type="password" id="password-input" name="password" required>
 </div>

 <div class="input-group">
  <label for="password2-input">Repeat your password:</label>
  <input type="password2" id="password2-input" name="password2" required>
 </div>

 <div class="input-group">
  <input type="submit">
 </div>
</form>
```

Nótese cómo se añaden a los elementos de entrada el atributo “required”, que añade una capa de validación por defecto haciendo que el formulario no pueda ser enviado si el campo en cuestión está vacío. Además, especificar el tipo del campo (text, email, password...) añade detalles como verificaciones adicionales sobre emails o no mostrar la entrada en el caso de contraseñas (aunque éstas viajan como texto plano al enviar el formulario).

Cada entrada se acompaña de una etiqueta `<label>`, que muestra al usuario lo que debe introducir en cada campo. Las label deben ser vinculadas al input correspondiente por razones de accesibilidad, lo cual se consigue haciendo que coincidan el atributo `for` del `<label>` con el `id` del `<input>`. Finalmente, se agrupan las parejas label-input en `<div>`s para su formateado posterior con CSS. El resultado visual es:

Claramente, el resultado puede mejorar. Por ejemplo, los campos `<input>` y `<select>` no concuerdan visualmente con el estilo oscuro. Podemos usar CSS para modificar su color de fondo y de texto, así como el color de su borde:

```
input, select {
    background-color: #666666;
    border-color: #999999;
    color: white;
}
```



Otra posible mejora es alinear todas las etiquetas de los campos del formulario a la derecha y las entradas de texto a la izquierda para darle un aspecto más uniforme. Para ello, haremos que los `<label>` tengan un ancho fijo. Además, gracias a los `<divs>` que hemos definido para separar los grupos del formulario, podemos aumentar la separación vertical entre ellos. Para efectuar estos cambios, enlace el archivo `style.css` en el documento `register.html` de la misma manera mostrada anteriormente. Podemos entonces añadir nuevos estilos destinados a los formularios:

```
.input-group > label {
    display: inline-block;
    width: 200px;
    text-align: right;
}

.input-group {
    margin-bottom: 0.5em;
}
```

El resultado es:

Los pasos a seguir para implementar la vista de login serían muy similares, empleando un formulario con los campos relevantes. Además, se pueden reutilizar los estilos CSS ya definidos para que todos los formularios presentados en la aplicación tengan un estilo uniforme.

2.5. Creación de una cabecera común

Al desarrollar una aplicación Web, es deseable contar con una cabecera y un pie común a todas las páginas, con objeto de unificar su estilo. Dado que se tratará de un elemento compartido por todas las vistas de nuestra aplicación, y no es una buena práctica duplicar el mismo código en varias vistas diferentes, la definiremos en un archivo HTML separado (por ejemplo, `header.html`) y la enlazaremos a nuestras vistas:

```
<div class="title-block">
    <h1 id="title">Photo Gallery</h1>
    <h3 id="subtitle">IISSI-2 - University of Seville</h3>
</div>

<hr>
```

En este ejemplo, nuestra cabecera tendrá un título, un subtítulo y una división horizontal al final. Además, incluiremos todos los elementos de la cabecera en un bloque lógico `<div>`. Una vez hecha, debemos hacer los siguientes cambios en las vistas en las que deseemos incluir la cabecera:

```
<html>
 <head>
  ...
  <script src="/js/utils/include.js"></script>
 </head>

 <body>
  <div id="page-header"></div>

  ...

  <script>
            include("header.html", "#page-header");
       </script>
 </body>
</html>
```

Incluiremos en la etiqueta `<head>` el archivo JavaScript `include.js`, que nos permite incluir un archivo HTML externo en una etiqueta de nuestra vista (este archivo se incluye por defecto en la plantilla del proyecto). En el cuerpo de la vista, definiremos una etiqueta vacía al principio, en cuyo interior se colocará automáticamente la cabecera. Finalmente, antes de terminar el cuerpo de la página, se incluye una línea de código JavaScript que indica que el contenido de `header.html` debe volcarse dentro de la etiqueta seleccionada mediante `#page-header`, es decir, aquella cuyo atributo `id` es el especificado.

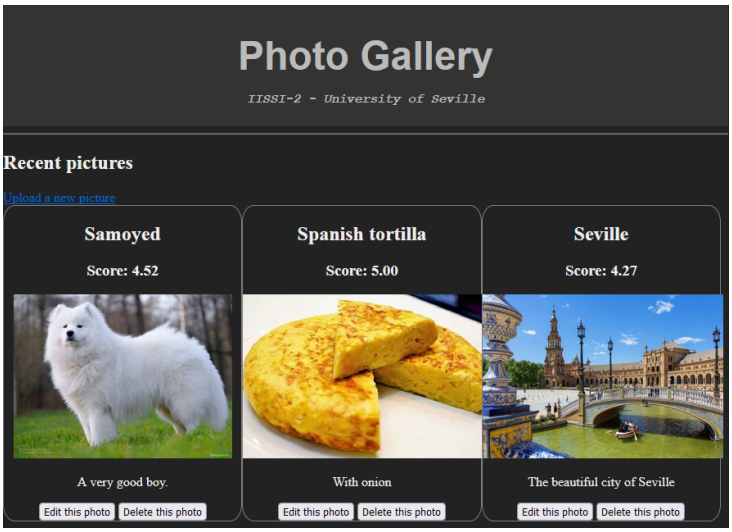
Puede repetir este proceso para incluir la cabecera que ha definido en `header.html` en tantas vistas como desee.

Sin embargo, como es de esperar, se muestra con el estilo por defecto. Mediante CSS podremos hacer cambios más finos, por ejemplo: centrar horizontalmente tanto título como subtítulo, cambiar la fuente y el estilo del texto, su color, y la distancia que cada elemento mantiene con respecto a los demás. Las modificaciones se pueden hacer sobre el archivo `style.css` provisto o crear uno nuevo, en cuyo caso se deberá enlazar con una etiqueta `<link rel="stylesheet"...>`

```
div.title-block {
 text-align: center;
 padding-top: 1%;
 padding-bottom: 1%;
 background-color: #333333;
}

#title {
 font-size: 300%;
 text-decoration: none;
 color: #BBBBBB;
 font-family: sans-serif;
 margin-bottom: 0;
}

#subtitle {
 font-size: 120%;
 font-style: italic;
 color: #BBBBBB;
 font-family: monospace;
}
```

Mediante la etiqueta `<div>` que hemos definido podemos aplicar estilos comunes a título y subtítulo: centrado horizontal, afinado del *padding* (distancia entre los bordes del elemento y su contenido interior) y color de fondo para el bloque. Para los estilos concretos de cada elemento interior (tamaño y estilo de fuente, color, márgenes...) los referenciamos por su ID, usando el selector `#id-elemento`.

El resultado visual es el siguiente:



Nótese que, por defecto, las etiquetas `<h1>` tienen subrayado (`text-decoration: underline`), pero este estilo no aplica al título. Dado que existe un selector más específico (por ID) aplicando un estilo que desactiva el subrayado (`text-decoration: none`), es este último el que prevalece.

2.5.1. Barra de navegación

La cabecera de la aplicación es común a todas las vistas, por que es un lugar muy apropiado para añadir una barra de navegación simple, que contendrá todas las vistas a las que el usuario puede acceder. Inicialmente, podemos usar etiquetas <a> para implementar la navegación:

```
<div class="title-block">
    <h1 id="title">Photo Gallery</h1>
    <h3 id="subtitle">IISSI-2 - University of Seville</h3>

    <div class="navigation">
        <a href="index.html">Recent photos</a>
        <a href="register.html">Register</a>
        <a href="login.html">Login</a>
    </div>
</div>
```

Ya que se encuentra en el bloque definido anteriormente, se beneficia de su centrado horizontal. Podemos ajustar el estilo mediante CSS, para que se asemeje más a una barra de navegación clásica:

```
div.navigation > a {
    color: white;
    font-size: 120%;
    background-color: #555555;
    font-family: monospace;
    text-decoration: none;
    display: inline-block;
    min-width: 150px;
}
```

En la siguiente práctica, aprenderemos a usar la librería CSS Bootstrap para dar un acabado visual más profesional a los elementos implementados en ésta.

2.6. Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo *commit* a efectuar.
- `git commit -m "mensaje"` crea un nuevo *commit* con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

Laboratorio 3

CSS avanzado: Bootstrap

3.1. Objetivo

El objetivo de esta práctica es introducir al alumno al framework CSS Bootstrap, para diseñar aplicaciones Web responsivas y con un nivel de acabado visual satisfactorio. El alumno aprenderá a:

- Integrar el framework Bootstrap en un proyecto ya existente.
- Comprender y usar la organización básica de Bootstrap en filas y columnas.
- Conocer y emplear los elementos que provee Bootstrap para los componentes más habituales de una aplicación web.
- Leer y asimilar documentación técnica para poder emplear nuevos elementos Bootstrap no cubiertos en esta práctica.

3.2. Introducción

Bootstrap fue creado en 2011 por Mark Otto y Jacob Thornton, desarrolladores de interfaz de usuario en Twitter como un medio para acelerar la creación de interfaces web de calidad. Es de código abierto y su licencia permite un uso libre, tanto privado como con ánimo de lucro. El repositorio en GitHub de Bootstrap es el tercero más popular en número de estrellas relativo a tecnologías Web, sólo por detrás de los frameworks JavaScript Vue y React.

Gran parte del motivo de su popularidad reside en que permite hacer interfaces web de gran calidad visual con mucho menor esfuerzo que escribiendo manualmente todo el código CSS requerido. Además, está pensado para ser *mobile-first*, esto es, que su

sistema de desarrollo de interfaces prima las pantallas pequeñas de los móviles, y los navegadores más grandes escalan a partir de éstas. Usando Bootstrap, se pueden hacer interfaces adaptables tanto a navegadores de escritorio como a pequeñas pantallas de móviles sin esfuerzo adicional.

3.3. Importando Bootstrap en nuestro proyecto

Para usar Bootstrap es necesario incorporar a nuestro proyecto una serie de archivos CSS y JavaScript. Estos pueden ser referenciados online, pero la plantilla de proyecto Silence descargada ya los incorpora. Añadiremos las siguientes líneas a nuestro `<head>`:

```
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
<link rel="stylesheet" href="/css/bootstrap.min.css">
<link rel="stylesheet" href="/css/font-awesome.min.css">
<script src="/js/libs/bootstrap.min.js"></script>
```

Incorporaremos estas dependencias primero al archivo `index.html`, en el que trabajaremos en las siguientes secciones. Es de esperar que el estilo de la página cambie, ya que se han incorporado las hojas de estilo propias de Bootstrap. A continuación aprenderemos a usar los estilos que definen éstas en nuestros elementos.

3.4. Sistema de filas y columnas

Cuando usamos Bootstrap, el contenido de una vista se organizará en filas. Cada fila puede ser a su vez subdividida en un máximo de 12 columnas, que por defecto tendrán el mismo ancho. Por ejemplo, si definimos una fila con una sola columna, obtendremos un elemento con un ancho del 100 % de la fila. Si definimos dos columnas en una fila, cada una tendrá el 50 % del ancho de la fila, y así sucesivamente.

Para empezar a definir contenido, se debe crear un `<div>` padre que contendrá todo el contenido de la vista. Esta etiqueta debe tener clase `container`:

```
<div class="container">
 <!-- Cuerpo de la vista aqui -->
</div>
```

A continuación, definiremos las siguientes filas en las que organizar los elementos. Las filas son tan anchas como su elemento padre (en este caso, tan anchas como la ventana en sí) y su altura está determinada por los elementos que contienen. Para definir una fila, crearemos un `<div>` con clase `row`:

```
<div class="container">
 <div class="row">
  Contenido de la fila aqui
 </div>
</div>
```

Finalmente, dentro de una fila podemos definir tantas columnas como deseemos, hasta 12. Por defecto, cada una de las columnas tendrá el mismo ancho, ocupando toda la fila. Las columnas se definen mediante `<div>`s con clase `col-md`. Podemos probar introduciendo el siguiente `<div>` en nuestra vista, después de la cabecera y antes del resto del contenido:

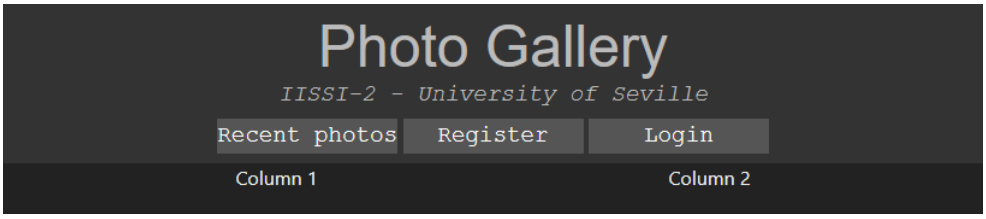
```
<div class="container">
 <div class="row">
  <div class="col-md">
   Column 1
  </div>
  <div class="col-md">
   Column 2
  </div>
 </div>
</div>
```

El resultado del código anterior es:

Como se aprecia, el contenido aparece en dos columnas. Además, Bootstrap añade márgenes automáticos entre el elemento raíz (`<div class="container">`) y su contenido. Estos márgenes varían automáticamente según el tamaño de la ventana, por lo que se recomienda no modificarlos.

Sin embargo, dentro de cada columna, el texto aparece alineado a la izquierda. Si deseamos centrar el texto del interior de un elemento, podemos añadir a los elementos en cuestión la clase `text-center`, también provista por Bootstrap. Otras clases posibles para alinear texto dentro de un elemento son `text-start` (izquierda) y `text-end` (derecha). Así, resultaría:

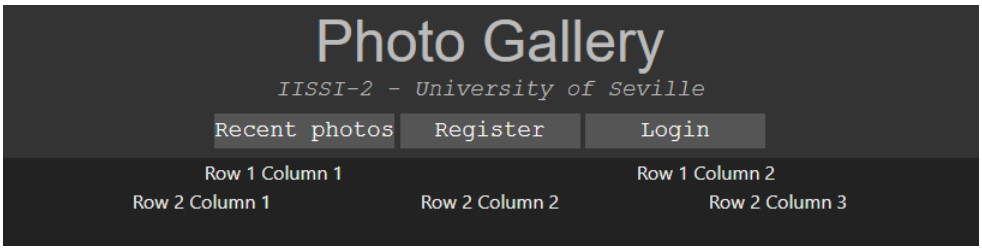
```
<div class="container">
 <div class="row text-center">
  <div class="col-md">
   Column 1
  </div>
  <div class="col-md">
   Column 2
  </div>
 </div>
</div>
```



Observe como la clase `text-center`, aplicada a toda la fila, centra los elementos de todas las columnas de dicha fila. Si se desea un ajuste más fino, se puede aplicar la clase de alineamiento de texto deseada a cada columna.

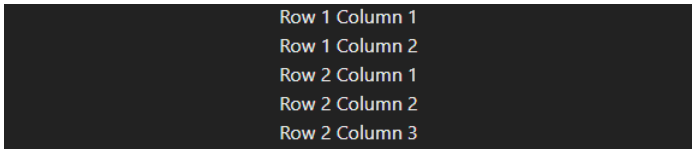
Podemos añadir más filas, cada una con sus propias columnas. No es necesario que todas las filas tengan el mismo número de columnas: si añadiésemos una segunda fila con tres columnas, el resultado sería:

```
<div class="container">
 <div class="row text-center">
  <div class="col-md">
   Row 1 Column 1
  </div>
  <div class="col-md">
   Row 1 Column 2
  </div>
 </div>

 <div class="row text-center">
  <div class="col-md">
   Row 2 Column 1
  </div>
  <div class="col-md">
   Row 2 Column 2
  </div>
  <div class="col-md">
   Row 2 Column 3
  </div>
 </div>
</div>
```



Como se aprecia, el número de columnas es diferente en las dos filas, aunque la separación entre filas puede ser insuficiente. Para ello, podemos aplicar código CSS personalizado para adaptar el margen inferior y superior de cada elemento `div.row` si así lo deseamos.

Como se explicó anteriormente, una de las principales ventajas de Bootstrap es que la adaptación a dispositivos móviles no requiere un esfuerzo adicional. Si se reduce el ancho de la ventana lo suficiente, las columnas pasan a apilarse verticalmente para adaptarse a pantallas más estrechas:



Las columas definidas mediante la clase `col-md` se apilan verticalmente cuando la anchura del elemento padre es menor de 768px. Existen otras clases para definir columnas, como `col-sm` y `col-lg`, que requieren una anchura mínima menor y mayor respectivamente para pasar a apilar verticalmente los elementos.

También se pueden hacer columnas de tamaño asimétrico, para lo que se debe indicar su anchura relativa de 1 a 12 mediante la clase `col-md-X`, donde X es el valor de anchura. Una columna de anchura 6 ocupará la mitad de la fila, una de anchura 9 ocupará el 75 %, etcétera. El resto de columnas de la misma fila se redimensionan automáticamente para completar la fila. Por ejemplo, si deseamos dedicar el 75 % del ancho de una fila a un elemento y el restante a otro:

```
<div class="container">
 <div class="row text-center">
  <div class="col-md-9">
   Wide column (75%)
  </div>
  <div class="col-md">
   This column takes up the remaining space
  </div>
 </div>
</div>
```

Existe una gran cantidad de opciones adicionales que permiten mucha más flexibilidad y que, por brevedad, no se incluyen en este boletín, pero que pueden consultarse en la página relevante de la documentación de Bootstrap.

3.5. Componentes Bootstrap relevantes

Bootstrap nos provee una gran cantidad de componentes de uso muy común en interfaces: desde botones, listas y desplegables hasta carruseles de imágenes, índices de paginación y mensajes de confirmación. La lista completa de estos componentes, junto con instrucciones para su uso y ejemplos, puede encontrarse en la documentación. En las siguientes subsecciones se mostrarán algunos de los más relevantes para nuestra aplicación.

3.5.1. Navbar: la barra de navegación, revisada

En la práctica anterior, creamos una barra de navegación simple mediante CSS personalizado, pero Bootstrap nos provee un componente llamado Navbar destinado a cubrir específicamente este propósito.

La barra de navegación más sencilla posible incluye sólamente los elementos relevantes de navegación, y se puede construir con el siguiente código HTML:

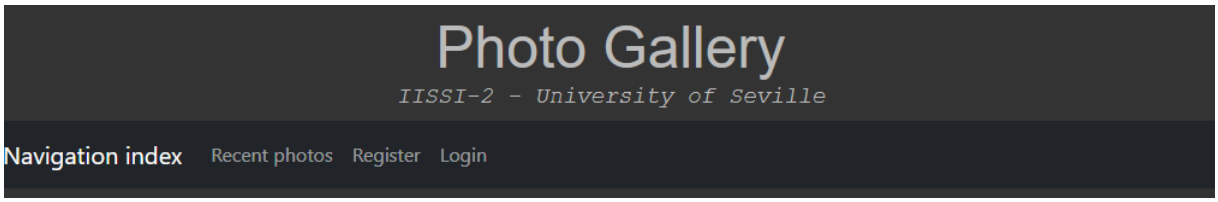
```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
<div class="container-fluid">
 <a class="navbar-brand" href="#">Navigation index</a>
 <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
     data-bs-target="#navbarSupportedContent"
  aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle
      navigation">
  <span class="navbar-toggler-icon"></span>
 </button>
 <div class="collapse navbar-collapse" id="navbarSupportedContent">
  <ul class="navbar-nav me-auto mb-2 mb-lg-0">
   <li class="nav-item">
    <a class="nav-link" href="index.html">Recent photos</a>
   </li>
   <li class="nav-item">
    <a class="nav-link" href="register.html">Register</a>
   </li>
   <li class="nav-item">
    <a class="nav-link" href="login.html">Login</a>
   </li>
  </ul>
 </div>
</div>
</nav>
```

Observe cómo se le ha dado un tono oscuro a la barra de navegación añadiéndole las

clases `navbar-dark` y `bg-dark`. Si se deseara tener la misma en tonos claros, se pueden usar las clases `navbar-light` y `bg-light`.

Como suele ser habitual al usar este tipo de frameworks, la estructura del código HTML viene impuesta y es más compleja que hacerlo a mano. Esta es una de las principales desventajas de las librerías CSS como Bootstrap. En este caso, los elementos de la barra de navegación se encuentran dentro de un `<ul class="navbar-nav">`.

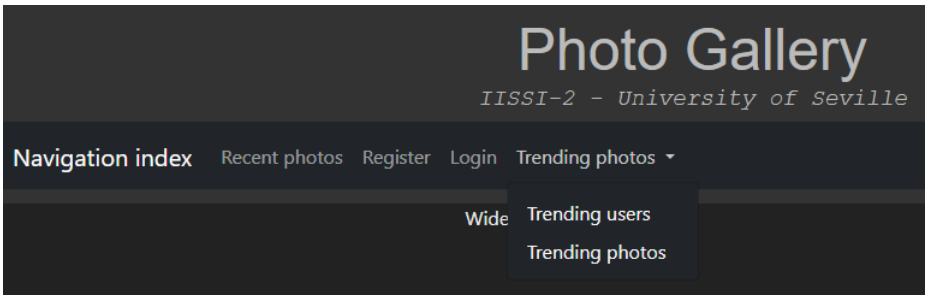
Sin embargo, este esfuerzo se ve recompensado con una barra de navegación de un estilo más uniforme y que, además, se colapsa automáticamente al hacer más estrecha la ventana, para adaptarse a móviles:



Además, se pueden añadir elementos colapsables, para agrupar varios elementos de la barra en uno sólo. Para ello, en lugar de añadir un `<li class="nav-item">` como los anteriores, añadiremos el siguiente elemento:

```
<li class="nav-item dropdown">
<a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button"
    data-bs-toggle="dropdown"
 aria-expanded="false">
 Trending photos
</a>
<ul class="dropdown-menu bg-dark" aria-labelledby="navbarDropdown">
 <li><a class="dropdown-item text-light" href="trending_users.html">Trending users</a>
     </li>
 <li><a class="dropdown-item text-light" href="trending_photos.html">Trending photos
     </a></li>
</ul>
</li>
```

En cada caso, deberemos adaptar los ID para que sean únicos, y modificar los enlaces de las etiquetas `<a>` para que referencien a los archivos HTML adecuados dentro de nuestra aplicación. Al igual que antes, la mayor complejidad en el código HTML se recompensa con un elemento de mayor calidad visual. Así, la nueva barra de navegación tras añadir este elemento resulta:



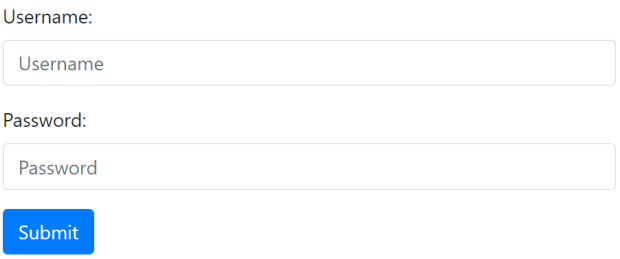
Existe una gran cantidad de opciones adicionales de personalización de la Navbar, que se pueden encontrar en la documentación correspondiente.

3.5.2. Formularios

Recuerde: Cualquier contenido que use elementos Bootstrap en el cuerpo de la página (filas, columnas, componentes...) debe estar dentro de un `<div class="container">` para que se visualize correctamente.

Bootstrap también da formato a formularios siempre que éstos tengan un formato concreto. Un ejemplo de formulario simple en Bootstrap es el siguiente:

```
<form>
 <div class="form-group">
  <label for="username-input">Username:</label>
  <input type="text" class="form-control" id="username-input" name="username"
      placeholder="Username">
 </div>
 <div class="form-group">
  <label for="password-input">Password:</label>
  <input type="password" class="form-control" id="password-input" name="password"
      placeholder="Password">
 </div>
 <button type="submit" class="btn btn-primary">Submit</button>
</form>
```



Los grupos de `<label>` e `<input>` deben ser agrupados mediante un `<div>` con clase `form-group`. Además, los `<input>` pueden contener un atributo `placeholder` para mostrar un texto por defecto a modo de ayuda cuando éstos esten vacíos. La

clase `form-control` les da a los elementos de entrada el estilo propio de Bootstrap, mientras que el resto de atributos son los propios de cualquier elemento `<input>` que ya conocemos.

Observe que el formato de colores de los `<input>` no es oscuro, ya que Bootstrap define sus propios estilos para ellos que toman preferencia sobre los que definimos nosotros previamente. Podemos hacer que pasen a un tema oscuro añadiéndole las clases `bg-dark text-light` a los `<input>`:

El color del botón puede ser modificado mediante su clase (ver la documentación de Bootstrap respecto a colores). Además, los campos del formulario pueden mostrarse en cualquier estructura deseada usando el sistema de filas y columnas de Bootstrap, siempre que todos los elementos se encuentren dentro de la etiqueta `<form>`.

Otros elementos de formulario relevantes se muestran a continuación:

Entradas de tipo `<select>` (por defecto son de opción única, si se añade a la etiqueta `<select>` el atributo `multiple` se convierte en un select de opción múltiple):

```
<div class="form-group">
 <label for="select-input">Select element from a list:</label>
 <select class="form-control" id="select-input" name="select">
  <option value="1">Option 1</option>
  <option value="2">Option 2</option>
  <option value="3">Option 3</option>
  <option value="4">Option 4</option>
  <option value="5">Option 5</option>
 </select>
</div>
```

Recuerde que los atributos `for` de los `<label>` se asocian con los atributos `id` de los `<input>` para mejorar la accesibilidad, y que un `id` debe ser único en todo el documento.

Checkboxes (en este caso, el `<div>` de agrupación pasa de tener clase `form-group` a `form-check`):

```
<div class="form-check">
 <input class="form-check-input" type="checkbox" id="checkbox-id" name="checkbox">
 <label class="form-check-label" for="checkbox-id">
  Option with a checkbox
 </label>
</div>
```

Opciones con radio button (nótese cómo se indica que las opciones son mutuamente excluyentes dándoles el mismo atributo `name`):

```
<div class="form-check">
 <input class="form-check-input" type="radio" name="radio-button" value="1" checked>
 <label class="form-check-label">
  Option 1
 </label>
</div>
<div class="form-check">
 <input class="form-check-input" type="radio" name="radio-button" value="2">
 <label class="form-check-label">
  Option 2
 </label>
</div>
```

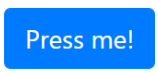
Por defecto, los checkboxes y radios se muestran apilados verticalmente. Para mostrar varios uno al lado del otro, se puede usar la clase `form-check-inline` en lugar de `form-check`. Además, se puede indicar qué opción está seleccionada por defecto añadiendo el atributo `checked`.

La documentación detallada sobre formularios, junto con el resto de opciones de personalización disponibles, se encuentra disponible en la página de Bootstrap sobre formularios y elementos input.

3.5.3. Botones

Con Bootstrap se le puede dar formato de manera muy sencilla a cualquier botón, añadiéndole las clases `btn` y `btn-primary` (esta última determina su estilo de color):

```
<button type="button" class="btn btn-primary">Press me!</button>
```



Al igual que la mayoría de elementos Bootstrap, se puede cambiar el estilo de color del botón mediante las clases apropiadas. Los estilos disponibles son los siguientes:

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
```

Se puede establecer un botón como desactivado añadiéndole el atributo `disabled`. Esto modificará su aspecto visual e impedirá que se pulse:

```
<button type="button" class="btn btn-primary">Enabled</button>
<button type="button" class="btn btn-primary" disabled>Disabled</button>
```

El resto de opciones relevantes se encuentran en la documentación del componente.

3.5.4. Etiquetas

Con el componente Badge de Bootstrap, se puede envolver un texto determinado en un elemento parecido a una etiqueta, para propósitos meramente estéticos:

```
This photo has the following categories:
<span class="badge bg-primary">Landscapes</span>
<span class="badge bg-primary">Nature</span>
```

Note que las etiquetas `<span>` sirven para contener texto, al igual que las etiquetas `<p>`. Sin embargo, a diferencia de las últimas, las primeras muestran el texto en la misma línea en lugar de separarlo en párrafos con saltos de línea.

Al igual que con los botones, se puede modificar su tema de color mediante las clases `primary`, `secondary`, `success`, etcétera. Además, se puede usar el mismo conjunto de clases `badge bg-*` con una etiqueta `<a href="...">` en lugar de `<span>`, para darle a un enlace aspecto de etiqueta. El resto de opciones relevantes se encuentran en la documentación del componente.

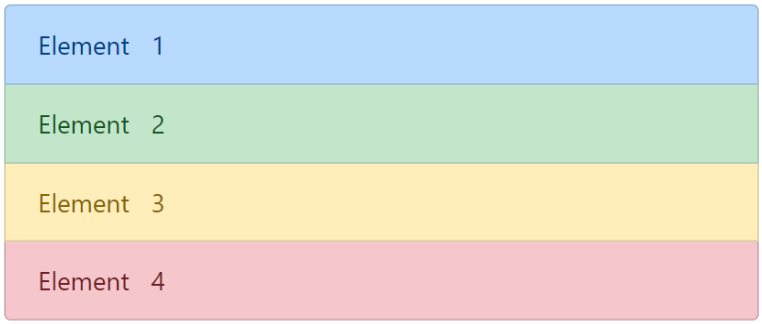
3.5.5. Listas

Las listas también se pueden formatear con Bootstrap añadiendo algunas clases definidas por el framework:

```
<ul class="list-group">
 <li class="list-group-item">Element 1</li>
 <li class="list-group-item">Element 2</li>
 <li class="list-group-item">Element 3</li>
 <li class="list-group-item">Element 4</li>
</ul>
```

Para darle un mayor grado de personalización, a los elementos de la lista se les puede añadir la clase `list-group-item-*` donde `*` es `primary`, `secondary`, `success`, etc.

```
<ul class="list-group">
 <li class="list-group-item list-group-item-primary">Element 1</li>
 <li class="list-group-item list-group-item-success">Element 2</li>
 <li class="list-group-item list-group-item-warning">Element 3</li>
 <li class="list-group-item list-group-item-danger">Element 4</li>
</ul>
```



El resto de opciones relevantes se encuentran en la documentación del componente.

3.5.6. Tarjetas

Las tarjetas o cards son una manera útil de agrupar varios elementos, como por ejemplo una imagen y texto variado. Una tarjeta sencilla se puede construir con Bootstrap mediante el siguiente código HTML:

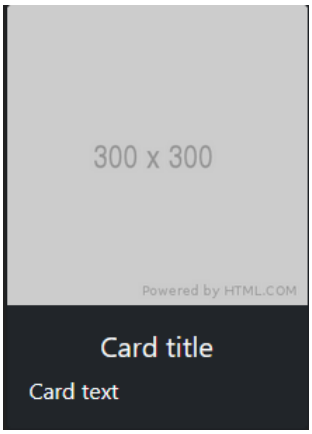
```
<div class="card text-light bg-dark">
 <img src="https://via.placeholder.com/300.png" class="card-img-top">

 <div class="card-body">
  <h5 class="card-title text-center">Card title</h5>
  <p class="card-text">Card text</p>
 </div>
</div>
```

Como se aprecia, todo el código HTML de la tarjeta está contenido en un `<div class="card">`. La imagen en el interior de la tarjeta tendrá clase `card-img-top` para que se redimensione automáticamente, y el contenido de la tarjeta se encuentra en un `<div class="card-body">`, cada uno con las clases apropiadas para darle estilo al texto.

Por defecto, como casi todos los elementos HTML, la tarjeta tenderá a ocupar todo el ancho disponible. Para limitar esto, se puede establecer una anchura máxima por CSS o incluirla en una columna Bootstrap para que su ancho se adapte dinámicamente. El siguiente ejemplo incluye la tarjeta en una columna cuyo ancho es el 33 % de la fila:

```
<div class="row">
 <div class="col-md-4">
  <!-- Aqui el codigo anterior de la tarjeta -->
 </div>
</div>
```

Así, resulta:



El estilo oscuro es, al igual que para los componentes anteriores, aportado mediante las clases `bg-dark text-light`. Si se quisiera usar un color específico diferente, se puede usar CSS personalizado en su lugar.

Los componentes pueden combinarse entre sí, por ejemplo, añadiendo botones, etiquetas o listas en el interior de una tarjeta. Las posibilidades de combinación sólo están limitadas por la imaginación y el buen gusto. El resto de opciones relevantes sobre las tarjetas Bootstrap están disponibles en la página correspondiente de la documentación.

3.6. Ejemplos de vistas de la aplicación

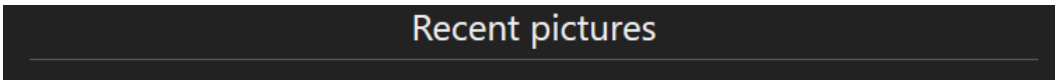
En las siguientes subsecciones se mostrarán sugerencias de implementación de algunas vistas del proyecto de curso. Conviene recordar que la implementación individual de cada vista es libre, y por tanto no tiene por qué seguir la implementación mostrada a continuación, siempre que en todo caso cumpla los requisitos del proyecto.

3.6.1. Listado de fotos más recientes

En todas las vistas, se debe partir de un elemento raíz `<div class="container">` que contendrá el resto del código HTML, ya que es necesario para que Bootstrap funcione correctamente. Comenzaremos, pues, por agregar una fila con una única columna con texto centrado, para mostrar el título de la vista:

```
<div class="container">
 <div class="row text-center">
  <div class="col-md">
   <h3>Recent pictures</h3>
  </div>
  <hr>
 </div>
</div>
```

Podremos comprobar que la combinación de una única columna y texto centrado hace que el título quede centrado en la ventana:

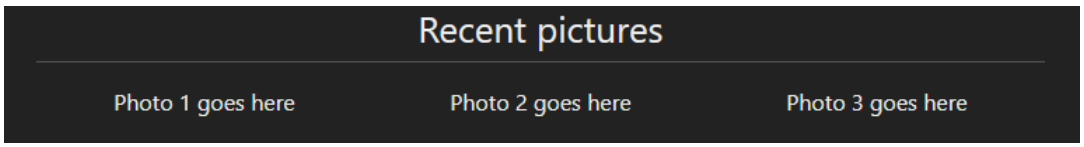


A continuación, podemos añadir las fotos por filas. Es conveniente decidir de antemano cuantas fotos deseamos mostrar en cada fila, a fin de establecer la clase necesaria para que cada columna ocupe el porcentaje de ancho correspondiente (esto es especialmente útil en el caso de que una fila no quede completamente llena, ya que por defecto las columnas que contenga intentarán ocupar todo el ancho disponible).

En este caso mostraremos tres fotos por fila, por lo que cada `row` tendrá tres columnas de clase `col-md-4` (12/3 = 4)

```
<div class="row text-center">
 <div class="col-md-4">
  Photo 1 goes here
 </div>
 <div class="col-md-4">
  Photo 2 goes here
 </div>
 <div class="col-md-4">
  Photo 3 goes here
 </div>
</div>
```

Añadir esta fila al interior del `<div class="container">`, tras el título, mostrará el contenido en tres columnas bajo el mismo:

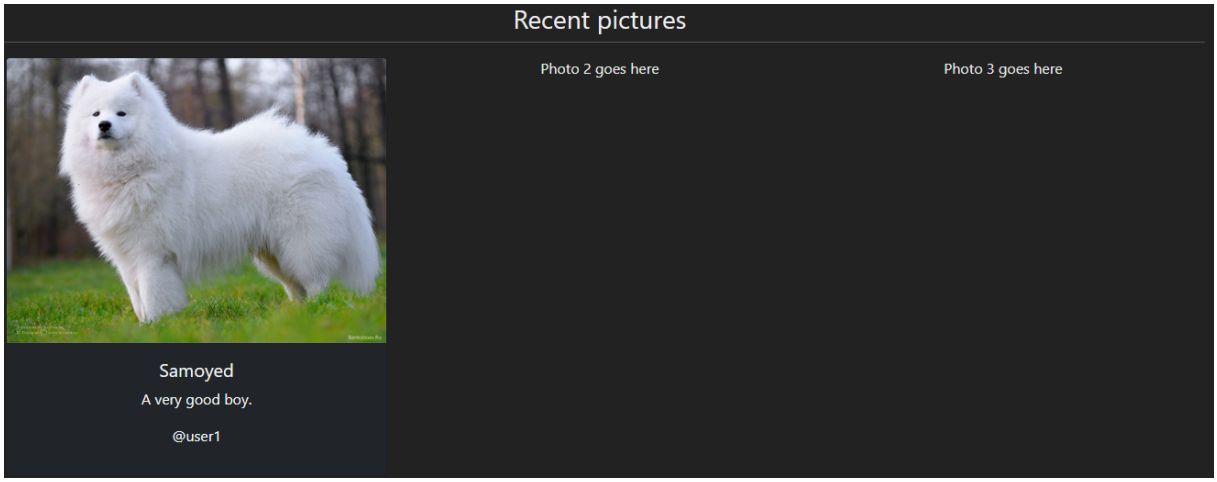


Una vez establecida la estructura, podemos trabajar en el código HTML que emplearemos para mostrar cada foto. Por ejemplo, para cada una de ellas podríamos mostrar su título, su descripción y su autor. Haciendo click en ella accederíamos a la vista de detalle de foto, donde se mostrarían el resto de atributos de la foto con más detalle.

Para mostrar cada una de las fotos de la galería podemos usar el componente tarjeta (ver Sección 3.5.6) de Bootstrap, usando los datos de las imágenes que introdujimos en la práctica anterior:

```
<div class="card bg-dark text-light">
 <img src="https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg" class="card-img-top">

 <div class="card-body">
  <h5 class="card-title text-center">Samoyed</h5>
  <p class="card-text">A very good boy.</p>
  <p class="text-end">@user1</p>
 </div>
</div>
```

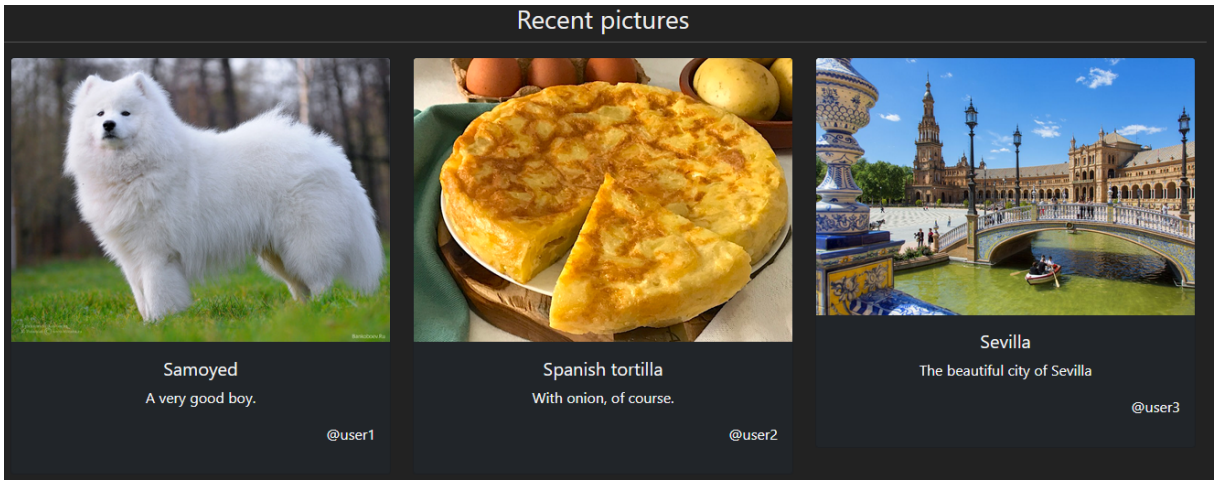
Si reemplazamos el texto que hemos puesto en las columnas como relleno con el código HTML anterior, podremos ver el resultado:



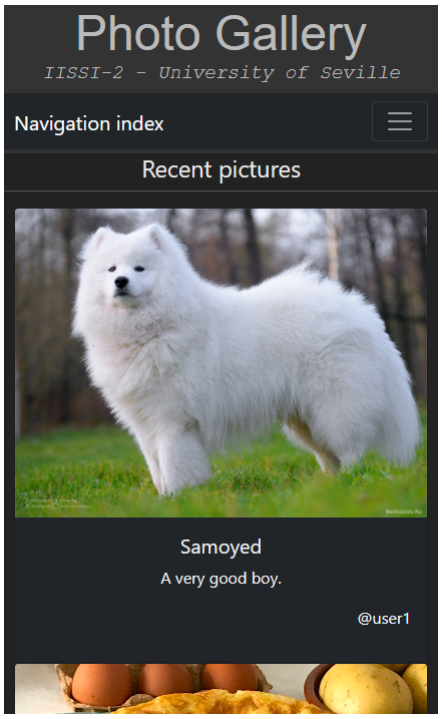
Para hacer que hacer click en la foto nos lleve a la página correspondiente de detalle de foto, podemos convertirla en un enlace envolviéndola en una etiqueta <a>:

```
<a href="photo_detail.html">
 <img src="..." class="card-img-top">
</a>
```

En clases posteriores, se mostrará cómo hacer una única página de detalle de fotos que sea capaz de mostrar la información de cualquier foto de la galería. Finalmente, podemos repetir el proceso hasta completar la fila, modificando los datos necesarios en cada foto:



Por defecto, las tarjetas ocupan el ancho de cada columna. Combinado con la capacidad que nos da Bootstap de organizar las columnas verticalmente cuando la ventana del navegador se estrecha, habremos conseguido que la galería se adapte automáticamente a móviles, mostrando en ese caso las fotos una encima de otra:



3.6.2. Detalle de foto

Una opción común es mostrar en la página principal de galería la información básica de cada foto, y luego tener una vista dedicada a mostrar todos los detalles de una foto concreta. La manera habitual de hacer esto es pasar el ID de la foto en cuestión a la vista de detalle usando un parámetro URL, y mediante JavaScript capturar este parámetro y cargar los datos de la foto en cuestión. Sin embargo, dado que JavaScript se introducirá en los próximos laboratorios, nos limitaremos a crear una vista con datos HTML estáticos. En el futuro, será fácil modificarla para que

cargue de manera dinámica los datos de cualquier foto aprovechando la estructura de la vista ya creada.

Trabajaremos ahora sobre un archivo HTML llamado `photo_detail.html`, que será al que accedamos al pulsar sobre una imagen en la página principal de la galería, implementada en la sección anterior. Para ello, usaremos el mismo esquema de archivo HTML mostrado hasta ahora. Recuerde importar todas las dependencias necesarias en el `<head>`, e incluir la cabecera común con la barra de navegación.

Los elementos que mostraremos en esta vista serán el título y la descripción de la fotografía, la propia imagen a un tamaño más grande, la puntuación actual de la imagen, el usuario que subió la imagen y la fecha en que lo hizo. Además, añadiremos botones para editar y borrar la foto, y un pequeño formulario para valorarla.

En este caso separaremos la vista en dos columnas: en una columna más ancha mostraremos la información de la foto, mientras que en otra más estrecha a su derecha colocaremos los botones y demás elementos para interactuar con ella:

```
<div class="row text-center">
 <div class="col-md">
  Photo details here
 </div>

 <div class="col-md-3">
  Buttons here
 </div>
</div>
```

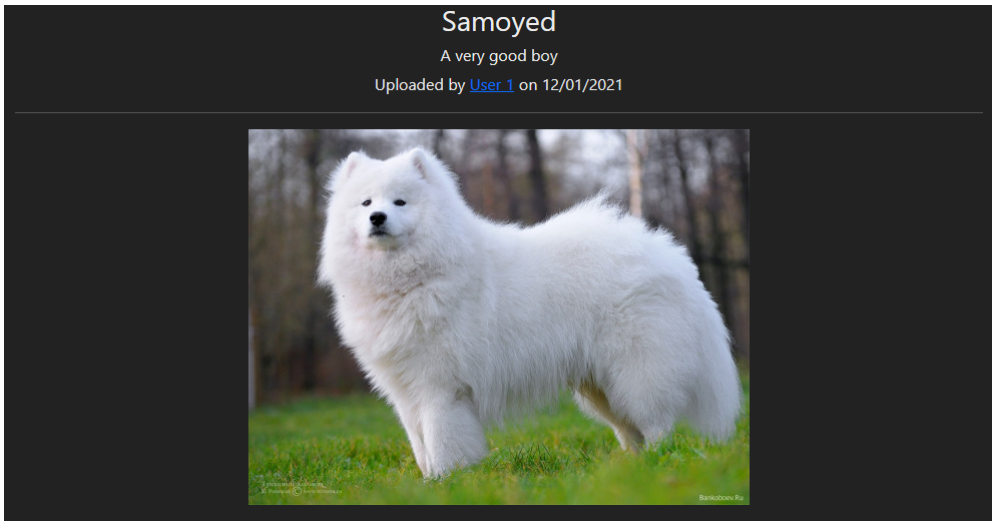
Como en todas las vistas que usen Bootstrap, recuerde que todo el contenido debe definirse dentro de un `<div class="container">`

En la primera columna, incluiremos los datos relevantes que queramos mostrar de la foto:

```
<div class="col-md">
 <h3>Samoyed</h3>
 <h6>A very good boy</h6>
 <p>Uploaded by <a href="user_profile.html">User 1</a> on 12/01/2021</p>

 <hr>

 <img src="https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg" class="img-fluid">
</div>
```

Recuerde que no debe incluirse contenido directamente en el `<div>` de la fila, en su lugar, debe introducirse al menos una columna y poner el contenido dentro de ella para su correcto posicionamiento.

Nótese cómo se ha añadido a la imagen la clase `img-fluid`. Esto hace que Bootstrap le aplique automáticamente los estilos necesarios para que escale automáticamente con el ancho y alto de la página, haciéndola responsiva. El resultado es el siguiente:



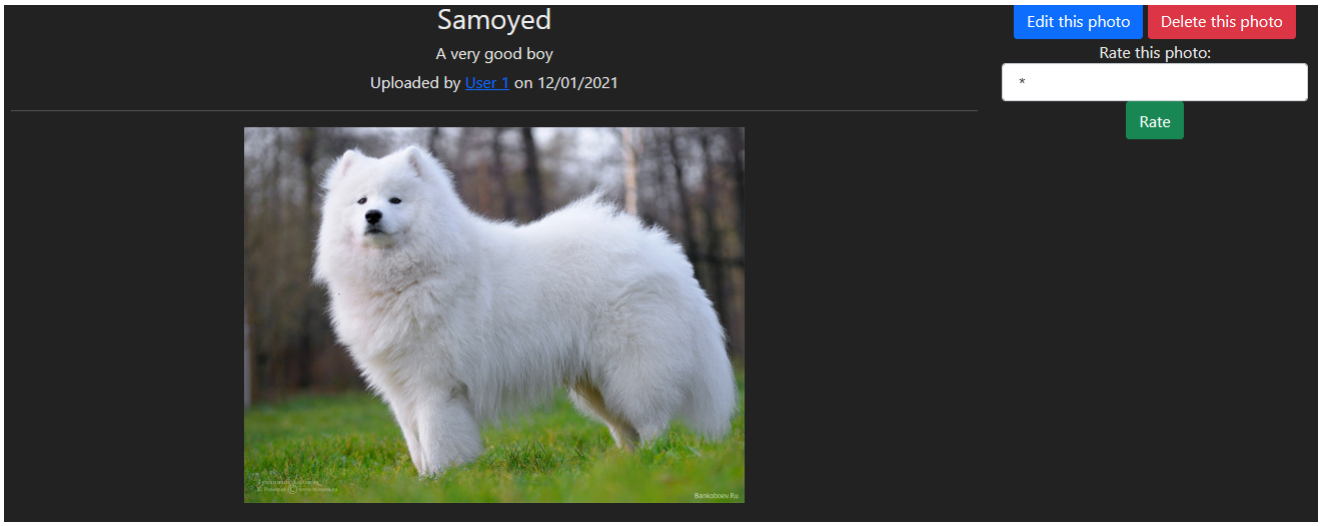
En la columna de la derecha, añadiremos dos botones para eliminar y editar la foto. En prácticas posteriores daremos funcionalidad a estos botones mediante JavaScript:

```
<div>
 <button class="btn btn-primary">Edit this photo</button>
 <button class="btn btn-danger">Delete this photo</button>
</div>
```

Además, añadiremos un formulario con un `<select>` para que el usuario pueda valorar la foto con entre 1 y 5 estrellas. De nuevo, en esta práctica nos centramos únicamente en proporcionar la interfaz visual, no la implementación de funcionalidad:

```
<form>
 <div class="form-group">
  <label for="rating-input">Rate this photo:</label>
  <select id="rating-input" name="rating" class="form-control">
   <option value="1">*</option>
   <option value="2">**</option>
   <option value="3">***</option>
   <option value="4">****</option>
   <option value="5">*****</option>
  </select>
  <button type="submit" class="btn btn-success">Rate</button>
 </div>
</form>
```

El resultado es el siguiente:



Por último, podemos opcionalmente añadir un enlace con estilo de botón en una fila nueva al final de la vista, para facilitar la navegación y poder volver a la página principal:

```
<div class="row text-center">
 <div class="col-md">
  <a href="index.html" class="btn btn-primary">Return to the gallery</a>
 </div>
</div>
```

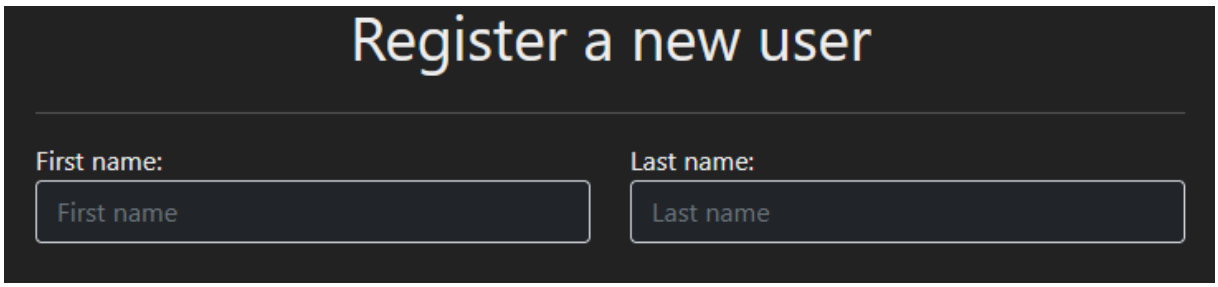
3.6.3. Registro de usuario

En la vista de registro de usuario podemos incluir un formulario Bootstrap para seguir unificando el estilo de la aplicación. Además, podemos aprovechar el sistema de filas y columnas para, por ejemplo, dividir el formulario en dos columnas diferentes y así aprovechar mejor el espacio en pantallas grandes. Para ello, comenzaremos por crear una etiqueta `<form>` y, dentro de ella, crear una fila con dos columnas:

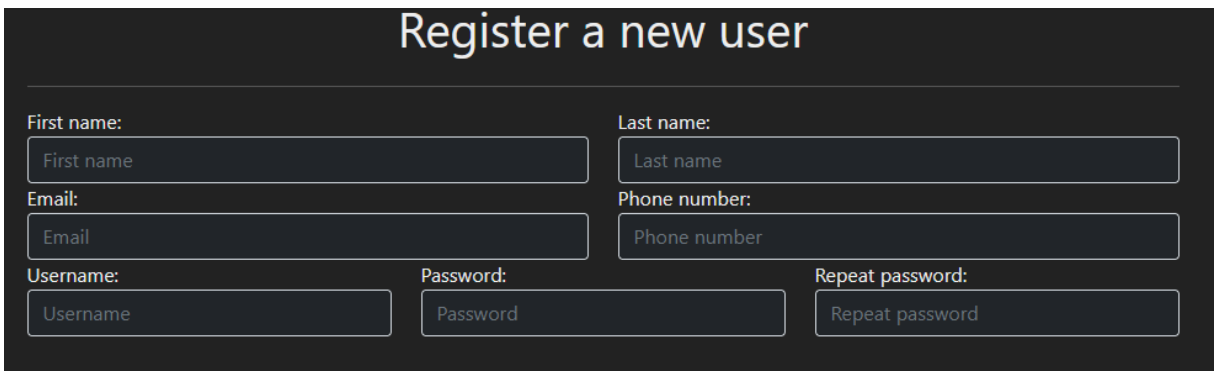
```
<form>
 <div class="row">
  <div class="col-md">
   Element 1
  </div>

  <div class="col-md">
   Element 2
  </div>
 </div>
</form>
```

Podemos entonces introducir en cada columna un elemento input, tal y como se explicó en la Sección 3.5.2:

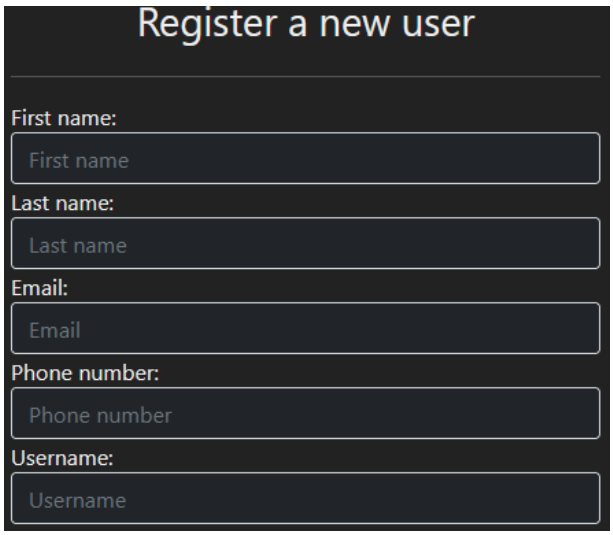
```
<div class="row">
 <div class="col-md">
  <div class="form-group">
   <label for="firstname-input">First name:</label>
   <input type="text" class="form-control text-light bg-dark" id="firstname-input"
       name="firstName" placeholder="First name">
  </div>
 </div>

 <div class="col-md">
  <div class="form-group">
   <label for="lastname-input">Last name:</label>
   <input type="text" class="form-control text-light bg-dark" id="lastname-input" name=
       "lastName" placeholder="Last name">
  </div>
 </div>
</div>
```



Podemos repetir tantas filas como sean necesarias para implementar todos los campos del formulario. Además, podemos crear filas con un número diferente de columnas, y Bootstrap las adapta automáticamente:

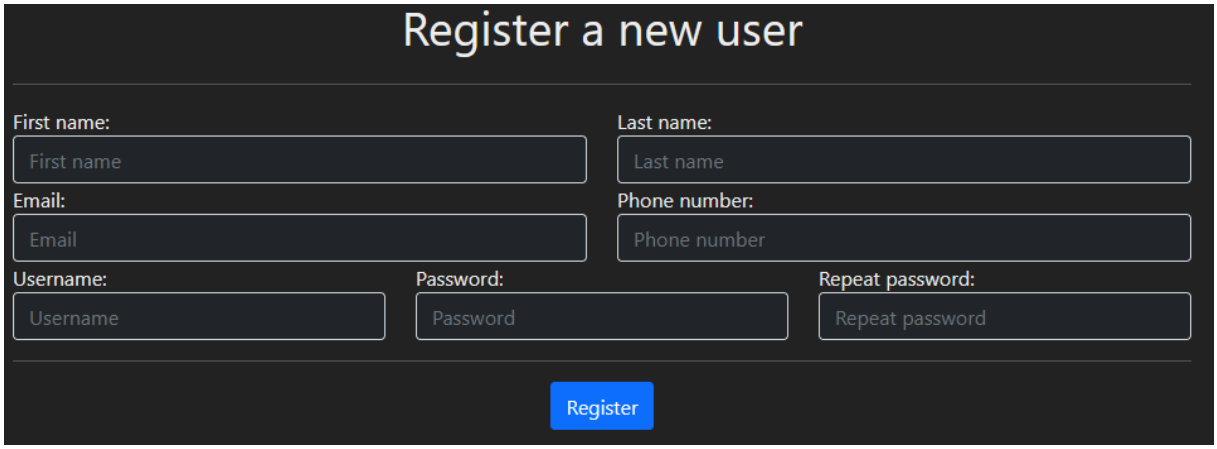


Por supuesto, Bootstrap se encarga de que la adaptación a dispositivos móviles sea inmediata y no requiera mayor esfuerzo por nuestra parte:



Finalmente, añadiremos una fila con una única columna, que contendrá el botón de enviar centrado horizontalmente:

```
<div class="row">
 <div class="col-md text-center">
  <button type="submit" class="btn btn-primary">Register</button>
 </div>
</div>
```



3.7. Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo *commit* a efectuar.
- `git commit -m "mensaje"` crea un nuevo *commit* con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

3.8. Anexo I: Iconos Font Awesome

Font Awesome, una librería diferente a Bootstrap, permite incorporar una gran cantidad de iconos a nuestra aplicación web de manera sencilla. **Pese a que la versión actual es la 5, se recomienda usar la 4** ya que tiene una mayor facilidad de instalación y uso. Font Awesome 4 se incluye en la plantilla de proyecto Silence, y si se han seguido los pasos mostrados en la Sección 3.3, se encontrará enlazado y listo para su uso. La línea relevante es la siguiente:

```
<link rel="stylesheet" href="/css/font-awesome.min.css">
```

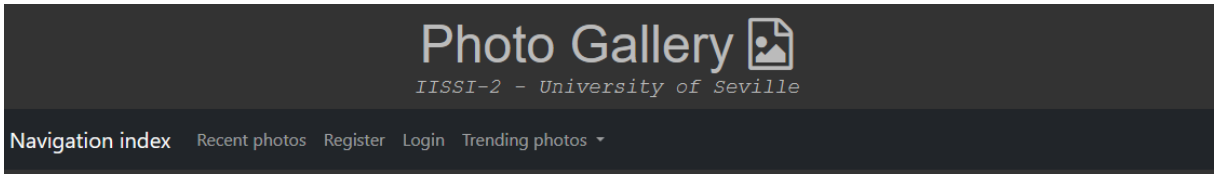
La lista de iconos disponibles está disponible en: https://fontawesome.com/v4.7.0/icons/ (hay que cerrar el mensaje emergente sobre la versión 5 de Font Awesome).

Al hacer click en uno de los iconos mostrados, se muestra el código HTML relevante para incluirlo. En general, es de la forma

```
<i class="fa fa-*" aria-hidden="true"></i>
```

Donde * es el código del icono a incluir. Por ejemplo, si incluimos un icono en el título de la aplicación presente en la cabecera:

```
<h1 id="title">Photo Gallery <i class="fa fa-file-image-o" aria-hidden="true"></i></h1
    >
```



Los iconos Font Awesome son vectoriales, por lo que mantienen su calidad al hacer zoom y se pueden mostrar a cualquier tamaño.

Laboratorio 4

Introducción a JS, DOM y renderizadores

4.1. Objetivo

El objetivo de esta práctica es introducir al alumno a JavaScript, el lenguaje de programación ejecutable por los navegadores que permite añadir comportamiento e interacciones complejas a la Web, así como algunos usos básicos de JS. El alumno aprenderá a:

- Escribir código con los elementos básicos de JavaScript.
- Manipular los elementos de una página Web con JavaScript.
- Programar módulos renderizadores para generar representaciones HTML de objetos del dominio.

4.2. Introducción

Los lenguajes estudiados hasta ahora (HTML, CSS) no son lenguajes de programación, ya que aunque definimos los elementos de una página web y su estilo, no permiten definir una secuencia de instrucciones ejecutable, ya sea al cargar la página, o como respuesta a alguna interacción por parte del usuario.

Esta funcionalidad está reservada a JavaScript, un lenguaje de programación conocido principalmente por su uso en navegadores, aunque no está limitado a éstos. JavaScript es un lenguaje interpretado, débilmente tipado[1] y multiparadigma. Los

[1] Existen alternativas con tipado fuerte como TypeScript.

navegadores modernos incluyen todo lo necesario para ejecutar código JavaScript durante la visualización de páginas. El código JavaScript puede interactuar con la página web, usando información relacionada con la navegación (como la posición del cursor), o alterando los elementos mostrados. Sin JavaScript, una página es completamente estática, ya que los elementos en ella son los que se definen en el código HTML mostrado, sin que estos cambien en algo más allá de su estilo. Como otros lenguajes de programación, JavaScript permite definir variables y usar tipos como objetos, arrays, funciones, incorporar librerías, etc. Pese a su nombre, es muy diferente a Java.

4.3. Introducción a JavaScript

El código JavaScript a ser ejecutado puede escribirse tanto en el mismo archivo en el que se encuentra el código HTML mediante la etiqueta `<script>`, o en archivos `.js` independientes. El primer caso está desaconsejado salvo en aquellos casos en los que el código a introducir no ocupe más de 4-5 líneas, y no sea reutilizable en otras páginas de la aplicación.

En nuestro proyecto, crearemos un archivo `.js` en la carpeta `web/js/` por cada vista `.html` de la aplicación, con el mismo nombre. Por ejemplo, la vista `index.html` tendrá asociado un archivo `index.js` en la carpeta anteriormente mencionada.

Para vincular el archivo JS a la vista HTML, incluiremos una etiqueta `<script>` al final del `<head>` de la vista en cuestión:

```
<script src="js/index.js" type="module"></script>
```

El atributo `type="module"` indica que usaremos las funcionalidades de módulos JS para importar elementos de otros archivos, como veremos más adelante en este boletín.

4.3.1. Hola, mundo

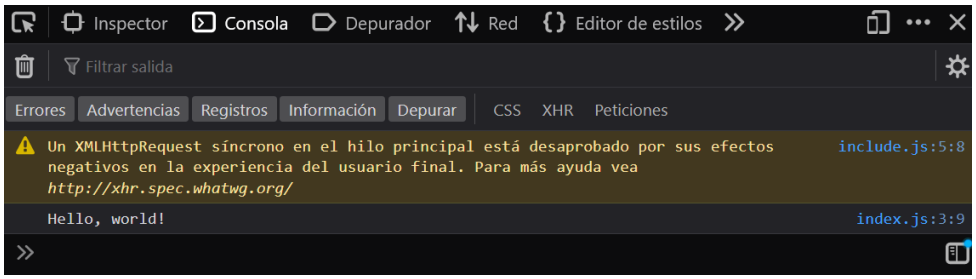
Para probar que el archivo se ha vinculado correctamente a la vista, podemos escribir un "Hola mundo" en `index.js`:

```
"use strict";

console.log("Hello, world!");
```

El `"use strict";` al principio del fichero activa el modo estricto de JavaScript, que convierte en errores algunos comportamientos de JS que de otro modo únicamente se considerarían advertencias. Usar siempre el modo estricto es una buena práctica, y

además garantiza la compatibilidad con futuras versiones de JS.

Más abajo, usamos la función `console.log` para imprimir una línea en la consola de depuración. Si actualizamos nuestra página, no veremos ningún cambio en ella. Para ver el mensaje impreso, debemos acceder a la consola del navegador pulsando F12 y seleccionando la pestaña "Consola":



Además de nuestro mensaje, aparecen advertencias producidas por otras librerías usadas, que pueden ser ignoradas o deshabilitadas en la configuración de la consola de depuración.

4.3.2. Punto de entrada

Pese a que se puede escribir directamente el código a ejecutar fuera de cualquier función, esta no es una buena práctica. Se recomienda definir una función `main`, que se ejecutará cuando la página esté completamente cargada:

```
"use strict";

function main() {
 // Your code here
}

document.addEventListener("DOMContentLoaded", main);
```

El código que deseemos ejecutar irá dentro de la función `main()`. La última línea indica al navegador que se debe ejecutar la función `main()` una vez todo el contenido de la página esté listo. Es posible definir funciones auxiliares fuera de la función principal. En una subsección posterior se cubre en más detalle el uso de funciones.

4.3.3. Variables

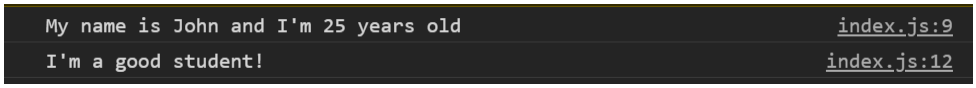
En JS se pueden definir variables mediante `let`. También es posible usar `var` para la definición de variables, pero se desaconseja su uso en código JS moderno. El estilo recomendado de nombrado es camelCase. Los tipos básicos son los comunes a la mayoría de lenguajes: cadenas, booleans, números enteros y decimales:

```
let age = 25;
let heightCm = 1.76;
let firstName = "John";
let goodStudent = true;
```

Las cadenas pueden escribirse usando comillas simples o dobles. También pueden usarse acentos graves ` para las cadenas e intercalar en ellas variables mediante la sintaxis ${}, por ejemplo:

```
let greeting = `My name is ${firstName} and I'm ${age} years old`;
console.log(greeting);

if(goodStudent) {
 console.log("I'm a good student!");
} else {
 console.log("I'm not a good student.");
}
```

Lo que imprimirá por consola:



Otros tipos básicos son los arrays y los objects. Un array es una lista ordenada de elementos, que no tienen por qué ser del mismo tipo (aunque se recomienda que lo sean, por consistencia). Un Object es similar a un diccionario en otros lenguajes, asociando valores de cualquier tipo a claves textuales.

```
let myList = ["one", "two", "three"];
let myObject = {
 one: 1,
 two: 2,
 three: 3
};
```

Los arrays pueden ser iterados mediante `for..of` o mediante un bucle `for` estándar (se recomienda usar el primero siempre que sea posible):

```
// These two loops are equivalent
for(let elem of myList) {
 console.log(elem);
}

for(let i = 0; i < myList.length; i++) {
 let elem = myList[i];
 console.log(elem);
}
```

Asimismo, los elementos de un Object pueden ser accedidos de forma manual mediante un punto, o usando una variable mediantes corchetes:

```
console.log(myObject.one); // Prints 1

for(let elem of myList) {
 console.log(myObject[elem]); // Prints 1, 2, 3
}
```

4.3.4. Funciones

Las funciones se definen mediante la palabra reservada `function`:

```
function addTwoNumbers(a, b) {
 let sum = a + b;
 return sum;
}

let result = addTwoNumbers(5, 3);
console.log(result);
```

También se pueden definir mediante la "notación flecha". Esta notación se suele usar, sobre todo, para crear una función anónima de una o dos líneas, por ser más compacta:

```
let addTwoNumbers = (a, b) => {
 return a + b
};

let result = addTwoNumbers(5, 3);
console.log(result);
```

Si la función tiene sólo una instrucción que devuelve un valor, no es necesario incluir las llaves ni la palabra `return`:

```
let addTwoNumbers = (a, b) => a + b;
```

Se recomienda, por consistencia, definir las funciones usando `function`, salvo en el caso mencionado anteriormente.

En JavaScript, las funciones también son variables, por lo que se pueden pasar como parámetros referenciándolas por su nombre. Por ejemplo, la función `filter` de un array recibe como parámetro una función que determina si un elemento debe pasar el filtro o no:

```
function isEven(x) {
 return x % 2 === 0;
}

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let evenNumbers = numbers.filter(isEven);
console.log(evenNumbers); // [2, 4, 6, 8]
```

En estos casos, la notación flecha para funciones puede resultar más compacta:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let evenNumbers = numbers.filter(x => x % 2 === 0);
console.log(evenNumbers); // [2, 4, 6, 8]
```

4.4. Manipulación del DOM

En un documento HTML, todos los elementos contenidos en él se estructuran jerárquicamente en forma de árbol, formando una construcción llamada *Document Object Model* (DOM). El DOM es, por tanto, una representación del contenido HTML que se muestra en una web.

4.4.1. Selección y modificación básica

Los navegadores proporcionan una API programática para interactuar con el DOM de una página web mediante JS, pudiendo añadir, modificar y eliminar contenido de manera dinámica. Para ello, en nuestro código debemos localizar el elemento HTML con el que queremos interaccionar mediante, por ejemplo, su atributo `id`.

Podemos realizar una prueba añadiendo un nuevo elemento `<div>` con un ID determinado en cualquier lugar de nuestro `index.html`, por ejemplo:

```
<div id="dom-test"></div>
```

Normalmente, para acceder a cualquier elemento del DOM realizaremos una consulta al elemento raíz, llamado `document`. La variable `document` hace referencia al propio documento HTML y es de ámbito global, por lo que puede ser usada en cualquier lugar del código:

Recuerde escribir su código en la función `main()` tal y como se definió anteriormente. Si no, es posible que su código se ejecute **antes** de que los elementos HTML estén cargados en la página, lo que hará imposible acceder a ellos a través del DOM.

```
let myDiv = document.getElementById("dom-test");
myDiv.textContent = "This was added using JavaScript";
```

Observe lo siguiente:

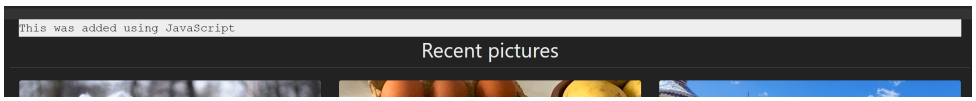
- La función `getElementById` permite, dado el ID de un elemento, buscarlo en el interior de otro elemento. En este caso, dado que `document` es el elemento raíz que contiene a todos los demás, lo estamos buscando en cualquier lugar del documento HTML.
- La función `getElementById` devuelve un objeto de tipo nodo HTML si se ha encontrado el elemento solicitado. Si no, devuelve `null`.
- El atributo `textContent` de un nodo HTML representa el texto contenido en él. Se puede sobreescribir para cambiar su contenido.

Pese a que el elemento `<div>` que hemos insertado en el HTML estaba vacío, se le ha dado un contenido de texto mediante JS:

This was added using JavaScript

Al elemento añadido dinámicamente se le añaden todos los estilos aplicables de las hojas de estilo de la página. Si deseamos modificar el estilo de un elemento en el árbol DOM, otro atributo de interés que tienen los nodos HTML en JS es `style`, que permite modificar de forma dinámica el estilo de un elemento alterando sus propiedades CSS:

```
myDiv.style.backgroundColor = "#EEEEEE";
myDiv.style.color = "black";
myDiv.style.fontFamily = "monospace";
```



Observará que los atributos accesibles mediante `style` tienen el mismo nombre que las propiedades CSS que se modifican mediante las hojas de estilo.

4.4.2. Creación de nodos HTML

Además de seleccionar elementos ya existentes, es posible crear elementos HTML nuevos e incluirlos en el documento mediante la función `document.createElement()`. Se debe indicar el tipo de elemento a crear como parámetro. Por ejemplo, podemos añadirle un párrafo <p> al que creamos anteriormente:

```
let newP = document.createElement("p");
newP.textContent = "This is a new paragraph";
myDiv.appendChild(newP);
```

En este caso, `"p"` hace referencia a que deseamos crear un elemento de tipo <p>. Al igual que antes, podemos establecer su contenido textual mediante `textContent`. Finalmente, lo añadimos al usando la función `appendChild()`, que añade un elemento como hijo de otro.

La representación HTML de los elementos creados será:

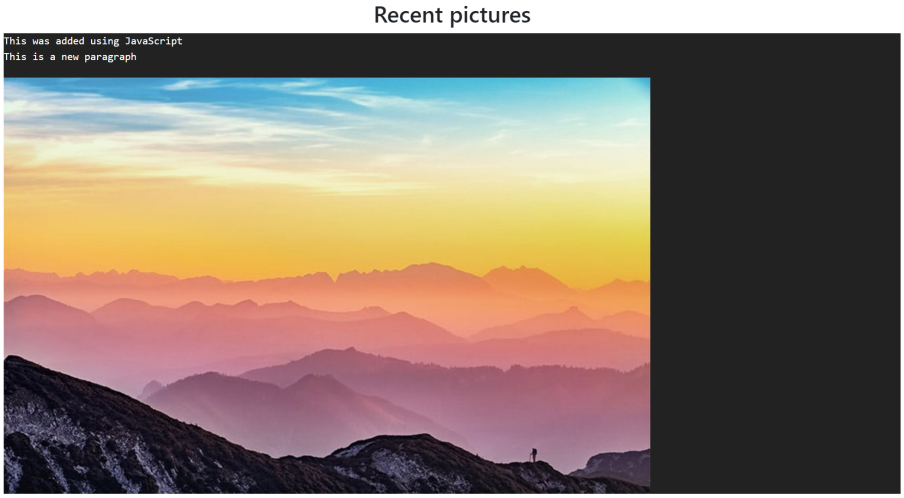
```
<div id="dom-test">
 This was added using JavaScript
 <p>
  This is a new paragraph
 </p>
</div>
```

Es importante destacar que crear un nuevo elemento HTML usando la función `document.createElement()` no hace que éste aparezca inmediatamente en el documento HTML. Hasta que no se introduzca dentro de algún otro elemento de la página, el elemento creado no será visible.

Un ejemplo más complejo es el siguiente, en el que se crea dinámicamente una imagen con unos atributos `src` y `title` determinados:

```
let newImage = document.createElement("img");
newImage.src = "https://loadedlandscapes.com/wp-content/uploads/2019/07/lighting.jpg";
newImage.title = "A beautiful landscape";

myDiv.appendChild(newImage);
```

Los atributos del elemento creado, como `src` o `title`, pueden establecerse modificando los atributos del nodo en JS. El resultado final es:



4.4.3. Selectores avanzados

En la sección anterior usamos la función `getElementById` para obtener un elemento contenido dentro de otro, según su atributo `id`. Sin embargo, en ocasiones puede ser útil seleccionar elementos según otros criterios. Una función relacionada es `querySelector`, que permite seleccionar nodos HTML usando los mismos selectores que se usan en CSS. Por ejemplo, podemos seleccionar el elemento `<div>` con atributo `class="container"` para cambiar su fondo:

```
let container = document.querySelector("div.container");
container.style.backgroundColor = "#BBBBBB";
```

Al igual que `getElementById`, `querySelector` devuelve el primer nodo encontrado que cumpla con el selector, o `null` si no se encuentra ninguno. Note que los siguientes dos selectores son equivalentes, ya que `querySelector` generaliza a `getElementById`:

```
document.getElementById("id");
document.querySelector("#id");
```

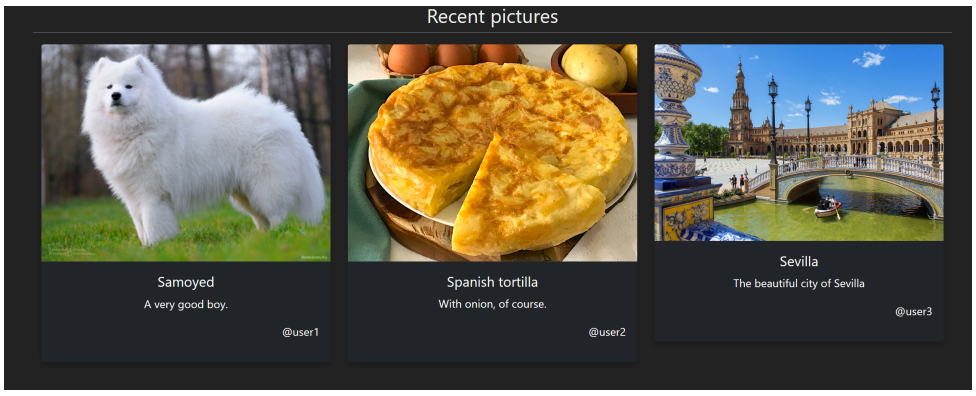
Existe una función relacionada, `querySelectorAll`, que devuelve un array de todos los elementos que cumplen un selector concreto. A modo de ejemplo, recordemos que todas las fotos están contenidas en una tarjeta, que es un `<div class="card">`. Podemos usar este selector para encontrar todas las tarjetas en el contenedor de la galería y cambiar su estilo, por ejemplo, añadiéndoles un ligero sombreado:

```
let container = document.querySelector("div.container");

let cards = container.querySelectorAll("div.card");
for (let card of cards) {
    card.style.boxShadow = "0 4px 8px 0 rgba(0, 0, 0, 0.2)";
}
```

Observe las siguientes consideraciones:

- `querySelectorAll` siempre devuelve un array de elementos, por lo que podemos iterar sobre ellos con `for..of`. Si no se encuentra ninguno, en lugar de `null` devuelve un array vacío.
- Todos los selectores estudiados (`getElementById`, `querySelector` y `querySelectorAll`) pueden usarse sobre `document`, para buscar los elementos en cualquier parte del DOM, o sobre un nodo concreto, para buscar los elementos en cuestión únicamente dentro de ese nodo.
- En el ejemplo mostrado, el punto anterior implica que un `<div class="card">` que no estuviera contenido en el `container` no sería seleccionado.

El código anterior modifica de forma dinámica el estilo de la galería, con este resultado:



4.4.4. Creación de nodos avanzada

En la Sección 4.4.2 aprendimos a crear e introducir manualmente nodos HTML en el documento a través del DOM. Sin embargo, este método es poco escalable para nodos complejos. Tomemos por ejemplo una tarjeta de la galería:

```
<div class="col-md-4">
 <div class="card bg-dark text-light">
  <img src="https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg" class="card-img-top">

  <div class="card-body">
   <h5 class="card-title text-center">Samoyed</h5>
   <p class="card-text">A very good boy.</p>
   <p class="text-end">@user1</p>
  </div>
 </div>
</div>
```

Cada tarjeta de este estilo está compuesta de un total de 7 nodos HTML, cada uno de ellos con una serie de atributos y texto. Sin duda, crearlos todos manualmente en el código sería tedioso. Por ello, en estos casos se suele almacenar una representación en forma de cadena del código HTML, y usar una función que interprete esa cadena, generando el nodo raíz y todos los que se encuentren en su interior automáticamente, para poder entonces colocarlos en el DOM.

La plantilla de proyecto Silence incluye esta función, llamada `parseHTML()`, en un módulo de utilidades. Para utilizarla, debemos importarla del módulo correspondiente al principio del fichero JS:

```
import { parseHTML } from "/js/utils/parseHTML.js";
```

La importación de elementos funciona de manera similar a otros lenguajes: se debe indicar entre llaves los elementos a importar, y el módulo (archivo) en el que se encuentran.

Podemos entonces usar la función para convertir una cadena, representando HTML, en un objeto correspondiente a un nodo HTML que podemos manipular mediante JS como hemos visto anteriormente:

```
let html = `<div class="col-md-4">
 <div class="card bg-dark text-light">
  <img src="https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg" class="card-img-top">

  <div class="card-body">
   <h5 class="card-title text-center">Samoyed</h5>
   <p class="card-text">A very good boy.</p>
   <p class="text-end">@user1</p>
  </div>
 </div>
</div>`;

let newCard = parseHTML(html);
let container = document.getElementById("gallery");
container.appendChild(newCard);
```

En este caso, hemos de asignarle un ID “gallery” al elemento que contiene las tarjetas correspondientes a las fotos, para poder acceder a él fácilmente e insrtar nuevos elementos.

Note que hemos usado los acentos graves ` ya que permiten que una cadena se extienda por varias líneas. Si accedemos a nuestra galería, podemos ver que contiene la nueva tarjeta.

4.5. Renderizadores

Como hemos visto en la sección anterior, es sencillo generar nodos a partir de una cadena que contiene HTML. Esto puede ser generalizado aún más: dado un objeto JS que represente a una entidad, se podrían acceder a los diferentes campos de la entidad y colocar la información relevante en los lugares pertinentes de la cadena HTML, generando programáticamente la representación adecuada.

4.5.1. Renderizador de fotos

Podemos definir una función que recibe una foto y devuelve un nodo HTML representándola como una tarjeta:

```
function photoAsCard(photo) {
    let html = `<div class="col-md-4">
  <div class="card bg-dark text-light">
   <img src="${photo.url}" class="card-img-top">

   <div class="card-body">
    <h5 class="card-title text-center">${photo.title}</h5>
    <p class="card-text">${photo.description}</p>
    <p class="text-end">User ${photo.userId}</p>
   </div>
  </div>
 </div>`;

    let card = parseHTML(html);
    return card;
}
```

Observe que se usa la sintaxis `${}` para intercalar variables en la cadena HTML, que se corresponden con los atributos del objeto `photo` recibido como parámetro de la función.

Esta función nos permite abstraer la representación de una foto como tarjeta, ya que recibe un objeto JS representando una foto y nos devuelve un nodo con todo el contenido necesario. Por tanto, es muy probable que queramos reutilizarla en varios lugares de nuestra aplicación, siempre que queramos mostrar una foto como tarjeta (por ejemplo, tanto en la página principal de la galería como en el perfil de un usuario).

La manera de lograr esto es mediante un módulo de renderizado de fotos, que contendrá un objeto con funciones para renderizar una foto de diferentes maneras. Para ello, crearemos un nuevo archivo `photos.js` en la carpeta `web/js/renderers/`:

```
"use strict";

const photoRenderer = {
 ...
};

export { photoRenderer };
```

Como en todo nuestro código JS, usaremos el modo estricto. A continuación definiremos un objeto `photoRenderer`, que contendrá funciones de renderizado de fotos. Finalmente, exportaremos el renderizador mediante `export { ... }` para que pueda ser importado en otros lugares del código con `import`. Observe que está definido mediante `const` para que las funciones del renderizador no puedan ser modificadas externamente.

Podemos añadir la función anterior al renderizador, llamándola `asCard`:

```
"use strict";
import { parseHTML } from "/js/utils/parseHTML.js";

const photoRenderer = {
    asCard: function(photo) {
        let html = `<div class="col-md-4">
   <div class="card bg-dark text-light">
    <img src="${photo.url}" class="card-img-top">

    <div class="card-body">
     <h5 class="card-title text-center">${photo.title}</h5>
     <p class="card-text">${photo.description}</p>
     <p class="text-end">User ${photo.userId}</p>
    </div>
   </div>
  </div>`;

  let card = parseHTML(html);
  return card;
    }
};

export { photoRenderer };
```

Observará que se trata de un objeto cuyos atributos son funciones, lo que permite invocarlas como si se trataran de métodos del objeto.

Esto nos permite añadir una foto a la galería dados sus atributos, sin tener que preocuparnos de generar su estructura HTML. Esto será especialmente útil en el futuro, cuando las fotos procedan de una consulta a la API del proyecto. Si

modificamos el código de `index.js` para usar el renderizador, resulta:

```
import { photoRenderer } from "/js/renderers/photos.js";

function main() {
    let container = document.getElementById("gallery");
    let photo = {
        title: "Samoyed",
        description: "A very good boy.",
        userId: 1,
        url: "https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg",
    };

    let card = photoRenderer.asCard(photo);
    container.appendChild(card);
}
```

El resultado es el mismo que en el apartado anterior, pero el código para generar el nodo HTML queda abstraído.

4.5.2. Renderizador de la galería

Una vez hemos abstraído la generación del código HTML para las fotos, podemos dar un paso más y crear un renderizador para toda la galería, que internamente usará el renderizador de fotos. Al igual que antes, crearemos un módulo (archivo) `web/js/renderers/gallery.js`:

```
"use strict";

import { parseHTML } from "/js/utils/parseHTML.js";

const galleryRenderer = {
 ...
};

export { galleryRenderer };
```

El renderizador de la galería usará el renderizador de fotos, por lo que debemos importarlo también:

```
import { photoRenderer } from "/js/renderers/photos.js";
```

En nuestro caso, tendremos una función para representar un array de fotos como una galería de tarjetas. Si se desea otro tipo de galería, se podría implementar también en el renderizador:

```
const galleryRenderer = {
    asCardGallery: function (photos) {
        ...
    }
};
```

En el código para renderizar nuestra galería, comenzaremos creando un elemento `<div>` que contendrá todas las filas de la galería, al que daremos clase `photo-gallery`. Crearemos también una fila inicial para insertar las fotos:

```
asCardGallery: function (photos) {
 let galleryContainer = parseHTML('<div class="photo-gallery"></div>');
 let row = parseHTML('<div class="row"></div>');
 galleryContainer.appendChild(row);
}
```

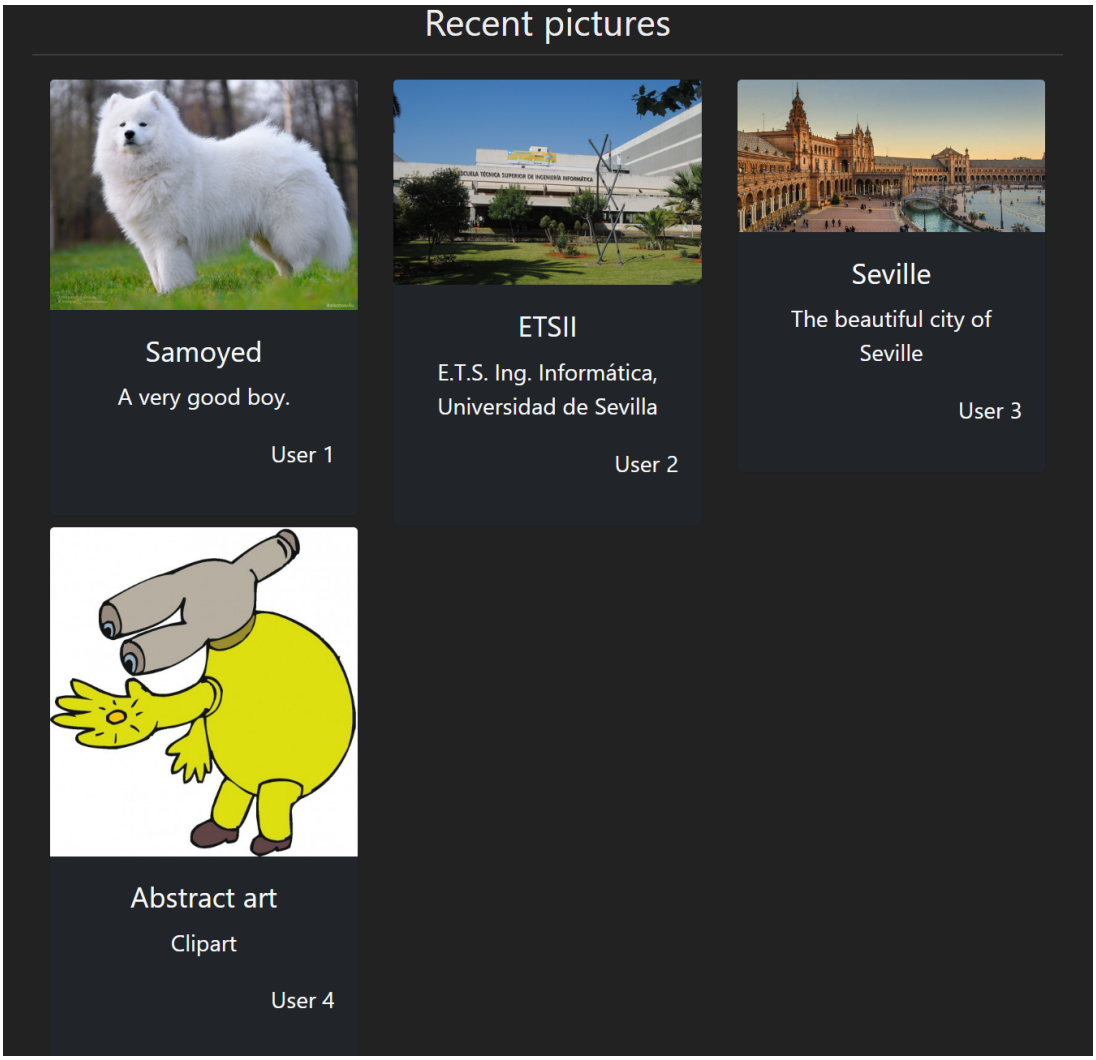
A continuación, iteraremos sobre todas las fotos recibidas como parámetro de la función, convirtiéndolas en su representación como tarjeta gracias al renderizador de fotos y añadiéndolas a la fila de la galería. Dado que nuestra galería contiene 3 fotos por fila, debemos llevar un conteo de las fotos insertadas. Cada 3 fotos, añadiremos una nueva fila a la galería y seguiremos insertando las nuevas fotos en esa fila creada. Finalmente, devolveremos el elemento que contiene toda la galería:

```
asCardGallery: function (photos) {
 let galleryContainer = parseHTML('<div class="photo-gallery"></div>');
 let row = parseHTML('<div class="row"></div>');
 galleryContainer.appendChild(row);

 let counter = 0;

 for (let photo of photos) {
  let card = photoRenderer.asCard(photo);
  row.appendChild(card);
  counter += 1;

  if (counter % 3 === 0) {
   row = parseHTML('<div class="row"></div>');
   galleryContainer.appendChild(row);
  }
 }

 return galleryContainer;
}
```

Para comprobar su correcto funcionamiento, crearemos un array de fotos en `index.js` y lo convertiremos dinámicamente en una galería, usando el renderizador recién creado:

```
import { galleryRenderer } from '/js/renderers/gallery.js';

function main() {
 let container = document.getElementById("gallery");
 let photos = [
  {
   title: "Samoyed",
   description: "A very good boy.",
   userId: 1,
   url: "https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg",
   date: "15/08/2020",
  },
  {
   title: "ETSII",
   description: "E.T.S. Ing. Informatica, Universidad de Sevilla",
   userId: 2,
   url: "https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/ETSI_Inform%C3%
       A1tica_Sevilla_y_DrupalCamp_Spain_2011.jpg/1920px-ETSI_Inform%C3%
       A1tica_Sevilla_y_DrupalCamp_Spain_2011.jpg",
   date: "01/01/2021",
  },
  {
   title: "Seville",
   description: "The beautiful city of Seville",
   userId: 3,
   url: "https://urbansevilla.es/wp-content/uploads/2019/03/
       urban-sevilla-foto-ciudad.jpg",
   date: "03/02/2019",
  },
  {
   title: "Abstract art",
   description: "Clipart",
   userId: 4,
   url: "https://clipartart.com/images/worst-clipart-ever-1.jpg",
   date: "14/08/2019",
  },
 ];

 let gallery = galleryRenderer.asCardGallery(photos);
 container.appendChild(gallery);
}
```

Observe que ya no es necesario crear la galería manualmente mediante HTML, sino que es generada automáticamente a partir de un array de fotos. Además, la abstracción en la creación de la galería permite insertarla en cualquier lugar de nuestra aplicación (por ejemplo, podemos mostrar una galería con las fotos subidas por un usuario en la página de su perfil). El resultado es el siguiente:



4.5.3. Renderizador de detalles de foto

El renderizador de fotos, por el momento, puede recibir una foto y mostrarla en formato tarjeta para la galería. Pero existe una vista en la que deseamos mostrar una imagen individual en un formato diferente: la de detalles de foto. En ese caso, querremos mostrar la misma foto no en formato miniatura como hasta ahora, sino a tamaño completo.

Dado que en la práctica anterior ya definimos la estructura HTML para mostrar una foto en detalle en `photo_detail.html`, podemos generalizarla añadiéndole un nuevo método al renderizador de fotos:

```
const photoRenderer = {
 asCard: function (photo) {
  ...
 },

 asDetails: function (photo) {
  let html = `<div class="photo-details">
    <h3>${photo.title}</h3>
    <h6>${photo.description}</h6>
    <p>Uploaded by <a href="user_profile.html" class="user-link">User ${photo.userId}
        </a> on ${photo.date}</p>

    <hr>

    <img src="${photo.url}" class="img-fluid">
   </div>`;

  let photoDetails = parseHTML(html);
  return photoDetails;
 },
};
```

Podemos, entonces, enlazar un archivo JavaScript `photo_detail.js` a la vista `photo_detail.html` y usar el renderizador para mostrar una foto concreta en detalle. En ese caso, dejaremos vacía la columna que contiene los detalles de la foto, y le daremos un ID para poder localizarla mediante JavaScript y añadirle el contenido dinámicamente:

```
<div class="col-md-9" id="photo-details-column">

</div>
```

Entonces, en `photo_detail.js` realizamos una operación similar a la de la página principal, definiendo una foto en JS y usando el renderizador para transformarla en una representación HTML adecuada, que colocamos en el DOM:

```
"use strict";

import { photoRenderer } from "/js/renderers/photos.js";

function main() {
    let photoContainer = document.querySelector("#photo-details-column");

    let photo = {
        title: "Samoyed",
        description: "A very good boy.",
        userId: 1,
        url: "https://i.ibb.co/tY1Jcnc/wlZCfCv.jpg",
        date: "12/01/1996",
    };

    let photoDetails = photoRenderer.asDetails(photo);
    photoContainer.appendChild(photoDetails);
}

document.addEventListener("DOMContentLoaded", main);
```

Pese a que ahora mismo parece una sobrecarga de código para lograr lo que ya estaba conseguido anteriormente usando sólo HTML, en posteriores laboratorios veremos que ésta estructura de código facilita enormemente poder cargar cualquier foto en esta vista mediante una consulta a la API del proyecto.

4.6. Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo *commit* a efectuar.
- `git commit -m "mensaje"` crea un nuevo *commit* con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

Laboratorio 5

Gestión de eventos y validación de formularios

5.1. Objetivo

El objetivo de esta práctica es introducir la gestión básica de eventos mediante JavaScript, desarrollando código que se ejecute cuando ocurre una determinada acción en la página Web. Además, se presentarán los conceptos fundamentales sobre la validación de los formularios de la página.

El alumno aprenderá a:

- Asignar funciones al lanzamiento de determinados eventos.
- Conocer los tipos de eventos más comunes.
- Realizar validación de formularios en una aplicación Web.

5.2. Introducción

Hasta ahora hemos realizado cambios en las vistas de nuestra aplicación Web de manera proactiva: mediante nuestro código, la página ha sufrido modificaciones en el momento que nosotros hemos considerado más oportuno, normalmente, tras la carga de ésta. Sin embargo, buena parte de la experiencia de usuario se centra en reaccionar a las interacciones de éste con la página: hacer algo cuando se pulsa un botón, se selecciona un campo, se pone el ratón en una zona o se envía un formulario.

Este código, a priori, es imposible conocer de antemano cuándo se ejecutará, ya que depende de la interacción del usuario. La gestión de eventos en JavaScript consiste en

desarrollar código asociado a cada una de estas situaciones, que se ejecutará cada vez que se cumpla una determinada condición.

Un aspecto muy relacionado es la validación de formularios en cliente: para evitar trabajo innecesario al servidor, se asocia un determinado código de validación al evento de envío de un formulario, cancelando el envío si el usuario ha introducido algún valor incorrecto.

En las siguientes secciones abordaremos estos aspectos y aprenderemos a gestionar los eventos más comunes de una aplicación Web.

5.3. Reaccionando a clicks

Uno de los eventos más comunes es `click`, que ocurre cuando el usuario hace click en un elemento determinado de la página, generalmente un botón.

Para hacer una prueba, añadiremos un botón a cualquier parte de nuestra vista principal. Es importante darle un `id`, ya que lo usaremos para acceder al botón en nuestro código JavaScript:

```
<button class="btn btn-primary" id="test-button">Press me!</button>
```

Es posible asociar directamente funciones que gestione los eventos relacionados con el botón en el código HTML, pero se desaconseja. En su lugar, accederemos al botón en nuestro código JS, y le asociaremos una función al evento `click`:

```
function main() {
    let button = document.getElementById("test-button");
    button.onclick = function (event) {
        alert("You've pressed the button!");
    };
}
```

Observe cómo la función asociada al evento `click` se asigna a un atributo llamado `onclick`. En general, las funciones asociadas a cualquier evento se asocian a un atributo que se llama igual que el evento, con el prefijo **on**. Pruebe a refrescar la página y comprobará que, al pulsar el botón, el navegador muestra un mensaje de alerta.

Además, las funciones que gestionan eventos reciben un parámetro de tipo Event[1], al que se le suele llamar `event`. Este objeto Event puede usarse, entre otras cosas, para saber qué elemento concreto provocó el evento o detener la propagación del evento.

En el ejemplo anterior, se le ha asignado al atributo `onclick` del botón una función anónima, dado que la operación a realizar era simple. Si un evento tiene asociada una

[1] `https://developer.mozilla.org/en-US/docs/Web/API/Event`

operación más compleja, que requiere de varias líneas de código, puede ser definida aparte y referenciada por su nombre.

Por ejemplo, podemos usar el parámetro `event` para acceder al nodo HTML que provocó el evento, y mostrar el texto que contiene:

```
function main() {
    let button = document.getElementById("test-button");
    button.onclick = clickHandler;
}

function clickHandler(event) {
 let target = event.target;
 let text = target.textContent;
 alert(text);
}
```

En este caso, cuando pulsemos el botón se nos mostrará el texto del propio botón, ya que es éste el elemento causante del evento:

5.4. Otros eventos relevantes

Pese a que `click` suele ser el evento más usado, hay muchos otros que merece la pena comentar. A continuación se detallan algunos de ellos:

5.4.1. mouseenter / mouseleave

Los eventos `mouseenter` y `mouseleave` se producen, respectivamente, cuando el cursor del mouse entra y sale de un determinado elemento. Usándolos en conjunto, es posible alterar un elemento cuando el usuario pone el cursor sobre él, y devolverlo a su estado original una vez lo mueve fuera.

Por ejemplo, podemos asignar a todas las tarjetas de la galería una función que, al poner el ratón sobre ellas, se ilumine su borde usando la propiedad CSS `border`:

```
function main() {
    let cards = document.querySelectorAll("div.card");
    for (let card of cards) {
        card.onmouseenter = handleMouseEnter;
        card.onmouseleave = handleMouseLeave;
    }
}

function handleMouseEnter(event) {
    let card = event.target;
    card.style.border = "2px solid blue"
}

function handleMouseLeave(event) {
    let card = event.target;
    card.style.border = "none";
}
```

5.4.2. focus / blur

Los eventos `focus` y `blur` están asociados a etiquetas de tipo `<input>`, y se lanzan cuando el elemento en cuestion recibe o pierde el foco de entrada, respectivamente.

5.4.3. change

El evento `change` también está asociado a etiquetas `<input>`, y se activa cuando el valor del campo ha cambiado. Generalmente, esto suele ocurrir cuando el elemento pierde el foco tras haber recibido cambios, o al pulsar Enter.

5.4.4. submit

El evento `submit` está asociado al envío de un formulario por parte de un usuario, usando el botón correspondiente. Este evento se suele usar para validar el formulario en cuestión cuando el usuario ha terminado de rellenar los campos, y se puede evitar el envío del formulario si algún campo introducido no es correcto. En la siguiente sección mostramos un ejemplo de uso.

5.5. Validación de formularios

Una aplicación Web suele contener uno o más formularios, que el usuario debe cumplimentar para enviar datos al servidor. Estos formularios deben ser validados en el navegador antes de su envío, para evitarle al servidor el procesamiento de formularios no válidos. Dado que los `<form>` envían un evento `submit` cuando el

usuario envía el formulario, se suele asociar a este evento código de validación, que impide el envío del formulario si se producen errores.

Como ejemplo, realizaremos una validación del formulario de registro de nuevo usuario, ubicando en `register.html`. Para ello, crearemos un archivo `register.js` y lo vincularemos a la vista de registro:

```
<script src="js/register.js" type="module"></script>
```

Además, le daremos un `id` al formulario de registro, para poder acceder a él mediante JavaScript:

```
<form id="register-form">
 ...
</form>
```

En el archivo `register.js`, estableceremos el punto de entrada habitual:

```
"use strict";

function main() {
 ...
}

document.addEventListener("DOMContentLoaded", main);
```

Recordemos que la función `main()` se ejecuta cuando ha cargado todo el contenido de la página. En ese momento, podemos asignar una función al evento de envío del formulario:

```
function main() {
 let registerForm = document.getElementById("register-form");
 registerForm.onsubmit = handleSubmitRegister;
}

function handleSubmitRegister(event) {
 alert("Form sent!");
}
```

Si enviamos el formulario, comprobaremos que se emite la alerta al capturarse el evento de envío.

A continuación, podemos proceder a acceder a cada uno de los campos que queramos validar. Para ello, nos ayudaremos de los objetos `FormData`, que permiten encapsular un formulario para acceder a su contenido programáticamente:

```
function handleSubmitRegister(event) {
 let form = event.target;
 let formData = new FormData(form);
}
```

En este caso, `event.target` hace referencia al `<form>` que está siendo enviado. Podemos, entonces, construir un `FormData` a partir del formulario en cuestión. Con este objeto, podemos usar los métodos `formData.get()` para acceder a los campos introducidos por el usuario. Este método recibe el atributo `name` del input cuyo contenido se desea consultar:

```
let firstName = formData.get("firstName");
let lastName = formData.get("lastName");
let password = formData.get("password");
let password2 = formData.get("password2");
```

Suele ser útil definir al principio un array de errores, que podemos ir rellenando en el caso de que no se cumplan determinadas condiciones:

```
function handleSubmitRegister(event) {
    let form = event.target;
    let formData = new FormData(form);

    let errors = [];

    let firstName = formData.get("firstName");
    let lastName = formData.get("lastName");
    let password = formData.get("password");
    let password2 = formData.get("password2");

    if (firstName.length < 3 || lastName.length < 3) {
        errors.push("The first and last name should have more than 3 characters");
    }

    if (password !== password2) {
        errors.push("The passwords must match");
    }
}
```

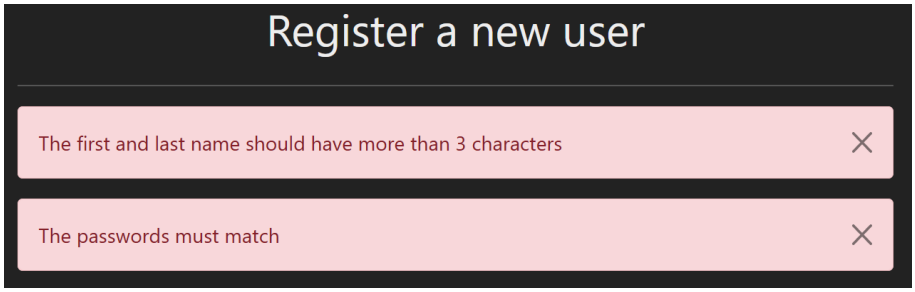
Lógicamente, debemos informar al usuario de todos los errores ocurridos. Una manera habitual es definir un `<div>` dedicado a mostrar mensajes al usuario. En nuestro caso, crearemos un `<div id="errors">` en la parte superior de la página, antes del formulario en sí, que usaremos para mostrar estos errores.

Para plasmar estos errores, existe un renderizador incluído por defecto en el proyecto, en el archivo `js/renderers/messages.js`. Este renderizador permite mostrar mensajes de información, error o éxito, que aparecerán automáticamente en cualquier `<div>` con `id=errors` que haya en la página:

```
import { messageRenderer } from "/js/renderers/messages.js";

function handleSubmitRegister(event) {
    let form = event.target;
    let formData = new FormData(form);

    let errors = [];

    // ...

    if (errors.length > 0) {
        event.preventDefault();
        let errorsDiv = document.getElementById("errors");
        errorsDiv.innerHTML = "";

        for (let error of errors) {
            messageRenderer.showErrorMessage(error);
        }
    }
}
```

Además, en el caso de que existan errores, deshabilitaremos el envío del formulario mediante `event.preventDefault()`:

El renderizador de mensajes, a diferencia de los demás, no devuelve un nodo HTML sino que coloca directamente el mensaje en el `<div id="errors">` que exista en la página. Es importante limpiar los mensajes de error anteriores, para evitar que se acumulen entre diferentes envíos:



Si no existen errores, no se mostrará ningún mensaje de error y se enviará el formulario. En posteriores prácticas, veremos cómo realizar este envío mediante una petición POST con AJAX a la API del proyecto.

5.6. Módulos validadores

En la sección anterior, vimos que el código para validar un formulario se suele encapsular en una función, que accede a los campos relevantes para realizar una serie de comprobaciones, y genera una serie de errores que deben ser mostrados al usuario para su subsanación.

Sin embargo, por su propia naturaleza, estas funciones suelen tener una cierta longitud, que aumentan la longitud del archivo JavaScript asociado a la vista. Además, es posible que el código de validación del formulario deba ser reutilizado en varias vistas, cosa que no es posible si se hace de forma específica para una vista concreta.

La solución es encapsular la función de validación en un módulo validador. Estos módulos se encuentran en la carpeta `js/validators/` y, de manera similar a los renderizadores, exportan un objeto que puede ser importado en otras vistas para su uso. En el caso del registro, dado que la entidad en cuestión son los usuarios, crearemos un archivo `users.js` en la carpeta mencionada, con el siguiente contenido:

```
"use strict";

const userValidator = {

 validateRegister: function (formData) {
  ...
 }
};

export { userValidator };
```

En la función de validación del registro, incorporaremos el código anterior:

```
validateRegister: function (formData) {
 let errors = [];

    let firstName = formData.get("firstName");
    let lastName = formData.get("lastName");
    let password = formData.get("password");
    let password2 = formData.get("password2");

    if (firstName.length < 3 || lastName.length < 3) {
        errors.push("The first and last name should have more than 3 characters");
    }

    if (password !== password2) {
        errors.push("The passwords must match");
    }

 return errors;
}
```

De este modo, el módulo de validación ofrece un método para, dado un objeto `FormData` concerniente al formulario de registro, devolver un array de errores relativos al formulario. Esto permite simplificar en gran medida el código de la vista en `register.js`:

```
import { messageRenderer } from "/js/renderers/messages.js";
import { userValidator } from "/js/validators/users.js";

function main() {
 let registerForm = document.getElementById("register-form");
 registerForm.onsubmit = handleSubmitRegister;
}

function handleSubmitRegister(event) {
 event.preventDefault();
 let form = event.target;
 let formData = new FormData(form);

 let errors = userValidator.validateRegister(formData);

 if (errors.length > 0) {
  let errorsDiv = document.getElementById("errors");
  errorsDiv.innerHTML = "";

  for (let error of errors) {
   messageRenderer.showErrorMessage(error);
  }
 }
}
```

5.7. Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo *commit* a efectuar.
- `git commit -m "mensaje"` crea un nuevo *commit* con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

Laboratorio 6

Peticiones AJAX: GET

6.1. Objetivo

El objetivo de esta práctica es introducir la comunicación del front-end implementado, usando HTML+CSS+JS, con el back-end del proyecto, consistente en una serie de endpoints REST. El alumno aprenderá a:

- Usar la librería JavaScript "Axios" para realizar peticiones AJAX de tipo GET a los endpoints del proyecto.
- Encapsular el código encargado de realizar este tipo de peticiones en módulos JavaScript de comunicación con la API del proyecto mediante programación asíncrona usando `async/await`.
- Usar y mostrar los datos obtenidos en las diferentes vistas, gestionando los posibles errores de manera adecuada.

6.2. Introducción

Hasta ahora, los datos mostrados en nuestra aplicación han sido introducidos a mano, ya sea directamente en el HTML de la página o definiéndolos mediante JavaScript para su posterior renderizado. Sin embargo, en aplicaciones reales estos datos provendrán de un servidor, normalmente a través de una API REST y en formato JSON. Por ello, es muy importante saber acceder a estos datos para interactuar con ellos en el front-end.

Tradicionalmente, este tipo de peticiones a un servidor por parte de un navegador mediante JavaScript reciben el nombre de peticiones AJAX (***A**synchronous **J**avaScript*

***A**nd **X**ML*), ya que se realizan de manera asíncrona para no bloquear la página durante la espera a que el servidor responda. Pese a que XML está en desuso como formato de intercambio de datos en favor de JSON, el acrónimo AJAX se sigue usando para denominar a estas peticiones.

En esta práctica aprenderemos a consultar algunos endpoints GET definidos en nuestro proyecto. Para ello, haremos uso de la librería Axios, ampliamente usada actualmente y que simplifica el proceso de hacer peticiones AJAX. Aprenderemos también a encapsular el código responsable de estas peticiones en módulos especializados, para simplificar el código de las vistas y poder reutilizar las peticiones en tantos sitios como sea necesario.

6.3. Configurando la conexión a la API

Todo lo relativo a las peticiones a la API del proyecto usando JavaScript se encuentra en la carpeta `web/js/api/`. Observe que en esta carpeta se incluye por defecto un archivo, `common.js`, que contiene definiciones comunes que son de utilidad para cualquier petición que hagamos:

```
const BASE_URL = "/api/v1";

const requestOptions = {
    headers: { Token: sessionManager.getToken() },
};
```

La constante `BASE_URL` hace referencia al prefijo de la URL que tenemos configurado en nuestro proyecto. **Es importante asegurarse de que es correcto**, ya que si no todas las peticiones fallarán al no incluir el prefijo correcto. El objeto `requestOptions` define las opciones y cabeceras que se enviarán en todas las peticiones. En este caso, está relacionado con el envío automático de tokens de sesión, que no son relevantes por el momento.

6.4. Módulos API

En el proyecto, los módulos JavaScript responsables de hacer peticiones a los diferentes endpoints de la aplicación se encuentran en la carpeta `/js/api/`. Dado que las operaciones que se suelen realizar contra estos endpoints son, en la mayoría de casos, peticiones CRUD, Silence permite generar automáticamente los módulos correspondientes a cada uno de ellos. Mediante el comando `silence createapi`, se crean en esta carpeta de manera automática los módulos correspondientes a los endpoints definidos en nuestro proyecto.

Para cada uno de los recursos almacenados en nuestra base de datos, tendremos un módulo que se encargará de realizar las peticiones correspondientes contra el

mismo. El nombre de los módulos autogenerados comienza con guión bajo (_) seguido del nombre del recurso. Esto permite poder crear manualmente módulos adicionales para peticiones más complejas que puedan ser necesarias. Por ejemplo, las operaciones CRUD básicas del endpoint de fotos se encuentran el el archivo autogenerado `/js/api/_photos.js`.

No modifique los módulos generados automáticamente, ya que los cambios se perderán si se vuelve a ejecutar el comando `silence createapi`. En su lugar, si necesita usar peticiones diferentes a las autogeneradas, cree un nuevo módulo y añada el código necesario para realizar las peticiones correspondientes.

Las peticiones GET se implementan como métodos dentro de un objeto definido en el módulo correspondiente. Por ejemplo, el método autogenerado para consultar todas las fotos es:

```
getAll: async function() {
   let response = await axios.get(`${BASE_URL}/photos`,requestOptions);
   return response.data;
},
```

Observe lo siguiente:

- Todos los métodos de petición deben ser declarados como `async` para que puedan ser usados en una función asíncrona.
- Se realiza la petición usando la librería `axios`, mediante el método `axios.get()`.
- La URL se construye a partir de la URL base común, definida en el archivo `/js/api/common.js`, que debemos importar.
- La llamada a `axios.get()` es asíncrona, por lo que se debe convertir en síncrona para esperar a que el servidor responda. Para ello, se utiliza `await` antes de la llamada, convirtiéndola en bloqueante.
- Una vez el servidor responde, se devuelven los datos de la respuesta, que se encuentran en el atributo `data` del objeto `response`. Axios se encarga de interpretar los datos JSON de la respuesta y convertirlos a objetos y arrays JavaScript.
- El parámetro `requestOptions`, importado también de `common.js`, incluye los datos que se deben enviar en todas las peticiones al servidor, por ejemplo, el token de sesión.

En las siguientes secciones, usaremos estos módulos autogenerados para realizar las peticiones a los endpoints de la aplicación.

6.5. Renderizando las fotos del servidor

En prácticas anteriores, definíamos manualmente el conjunto de fotos a mostrar en la página principal de la galería, en el archivo `index.js`. Ahora que contamos con módulos JavaScript con los que poder consultar las fotos existentes en la base de datos

a través de la API, podemos usarlos para obtener por esa vía el array de fotos a mostrar.

Para ello, debemos realizar dos modificaciones mínimas en el HTML de la vista principal: en primer lugar, incluiremos el `<script>` correspondiente para importar Axios en el `<head>`:

```
<script src="js/libs/axios.min.js"></script>
```

Además, añadiremos un div de errores dentro del contenedor principal:

```
<div id="errors"></div>
```

Con estos cambios realizados, podemos cambiar `index.js` para que utilice el módulo relativo a la API de fotos:

```
"use strict";

import { photosAPI_auto } from "/js/api/_photos.js";
import { galleryRenderer } from "/js/renderers/gallery.js";
import { messageRenderer } from "/js/renderers/messages.js";

async function main() {
    loadAllPhotos();
}

async function loadAllPhotos() {
    let galleryContainer = document.querySelector("div.container");

    try {
        let photos = await photosAPI_auto.getAll();
        let cardGallery = galleryRenderer.asCardGallery(photos);
        galleryContainer.appendChild(cardGallery);
    } catch (err) {
        messageRenderer.showErrorMessage("Error while loading photos", err);
    }
}

document.addEventListener("DOMContentLoaded", main);
```

Observe las siguientes consideraciones:

- La función `loadAllPhotos()` es una función asíncrona, dado que realiza una petición a la API. Esto permite que pueda ser ejecutada en segundo plano, sin que la página se bloquee.
- La llamada a `photosAPI_auto.getAll()` es asíncrona, pero debemos esperar a que el servidor responda para poder renderizar las fotos. Por ello, utilizamos `await` antes de la llamada.
- El formato de las fotos es el mismo que el utilizado anteriormente, por lo que podemos pasar la lista de fotos al renderizador correspondiente y mostrar la

galería en la página.
- Si ocurre cualquier problema durante la carga de las fotos, se ejecutará el `catch`, capturando la excepción y mostrando un mensaje en el div de errores.
- La función de entrada `main()` también debe ser `async`, ya que en su interior se usan funciones asíncronas.
- La llamada a `loadAllPhotos()` se ejecuta en segundo plano al ser esta función asíncrona, por lo que se podrían seguir ejecutando instrucciones en la función `main()` sin esperar a que el servidor responda la petición.

6.6. Vista de detalle de foto

En sesiones anteriores, implementamos una vista de detalle de foto que, gracias a un renderizador, muestra una foto concreta. Sin embargo, esta foto está establecida por nosotros, y sería deseable que se adaptara para mostrar cualquier foto posible: por ejemplo, que al pulsar en una foto de la galería principal se mostrara ésta en detalle en `photo_detail.html`.

Para ello, es necesario que la vista de detalle de foto reciba información sobre qué foto debe mostrar (generalmente se usa el ID de la foto). La forma más habitual de lograr esto es mediante parámetros de URL: por ejemplo, se puede acceder a esta vista mediante la URL `(...)/photo_detail.html?photoId=X`. En este caso, el navegador carga la página `photo_detail.html` como es habitual, pero el parámetro `photoId` proporcionado en la URL puede ser leído mediante JavaScript.

Recordemos que el renderizador de fotos permite mostrar una foto como tarjeta, que incluye un enlace a `photo_detail.html`. Podemos editar el **renderizador de fotos** para que el enlace de cada foto en formato tarjeta incluya su `photoId` como parámetro de URL:

```
asCard: function(photo) {
 (...)
 <a href="photo_detail.html?photoId=${photo.photoId}">
     <img src="${photo.url}" class="card-img-top">
 </a>
 (...)
}
```

Esto hace que, al pulsar en una foto de la galería, se transmita la información relativa al `photoId` de la foto seleccionada mediante un parámetro de URL. Podemos capturar este parámetro usando JavaScript, en `photo_detail.js`:

```
let urlParams = new URLSearchParams(window.location.search);
let photoId = urlParams.get("photoId");
console.log("The photo ID to load is: " + photoId);
```

El objeto `URLSearchParams` sirve para acceder más fácilmente a los parámetros de

URL, que se encuentran en `window.location.search`. Con este objeto, podemos acceder a un parámetro determinado usando `urlParams.get()`. Esto hará que se muestre por consola el ID de la foto que debemos mostrar:

```
The photo ID to load is: 6
```

Teniendo el ID de la foto, podemos hacer una consulta a la API para obtener los datos de la misma, y proporcionárselos al renderizador para que muestre la foto en cuestión, de manera muy similar a la usada para la galería:

```
"use strict";

import { photoRenderer } from "/js/renderers/photos.js";
import { photosAPI_auto } from "/js/api/_photos.js";
import { messageRenderer } from "/js/renderers/messages.js";

// Get the ID of the photo to load from the URL params
let urlParams = new URLSearchParams(window.location.search);
let photoId = urlParams.get("photoId");

async function main() {
    // Check that we have an ID before doing anything else
    if (photoId === null) {
        messageRenderer.showErrorMessage("Please, provide a photoId");
        return;
    }

    loadPhotoDetails();
}

async function loadPhotoDetails() {
    let photoContainer = document.querySelector("#photo-details-column");
    try {
        let photo = await photosAPI_auto.getById(photoId)
        let photoDetails = photoRenderer.asDetails(photo);
        photoContainer.appendChild(photoDetails);
    } catch (err) {
        messageRenderer.showErrorMessage("Error loading photo", err);
    }
}

document.addEventListener("DOMContentLoaded", main);
```

En este caso, podemos definir `urlParams` y `photoId` fuera de la función `main()` ya que los parámetros de URL están disponibles en todo momento, no sólo cuando la página está completamente cargada, así, podemos usar `photoId` en cualquier lugar del código desarrollado en el archivo.

Finalmente, para que todos los elementos funcionen correctamente, es importante modificar la vista `photo_detail.html` para añadir:

- La etiqueta `<script src="js/libs/axios.min.js"></script>` para importar Axios.

- El contenedor `<div id="errors">` para mostrar los posibles errores.

6.7. Peticiones AJAX a vistas

En ocasiones, para proporcionar una buena experiencia de usuario, es necesario realizar peticiones a varios recursos a la vez. Por ejemplo, a la hora de mostrar las tarjetas de la galería, puede ser buena idea mostrar el nombre de usuario que subió la foto junto con su imagen de perfil. Sin embargo, la tabla `Photos` sólo contiene el `userId` de la foto. En estos casos, es conveniente crear una vista que contenga los campos adicionales que deseamos mostrar, y realiar la petición GET a la vista en cuestión.

En la plantilla de proyecto utilizada, la base de datos contiene una vista `PhotosWithUsers`, que devuelve todas las filas de la tabla `Photos` y adicionalmente, para cada una de ellas, el nombre del usuario que subió la foto y la URL de su avatar.

Los módulos JS autogenerados para consultar la API incluyen también peticiones de tipo GET para las vistas de la BD. Concretamente, las peticiones para la vista anterior se encuetran en el archivo autogenerado `/js/api/_photoswithusers.js`.

Para implementar esta modificación, debemos cambiar el módulo usado para hacer la petición en `index.js`, usando el de esta vista en lugar del de las fotos:

```
import { photoswithusersAPI_auto } from "/js/api/_photoswithusers.js";

(...)

async function loadAllPhotos() {
    let galleryContainer = document.querySelector("div.container");

    try {
        let photos = await photoswithusersAPI_auto.getAll();
    (...)
```

A continuación, debemos acomodar esta nueva información en el renderizador de fotos, para que se muestre en la vista:

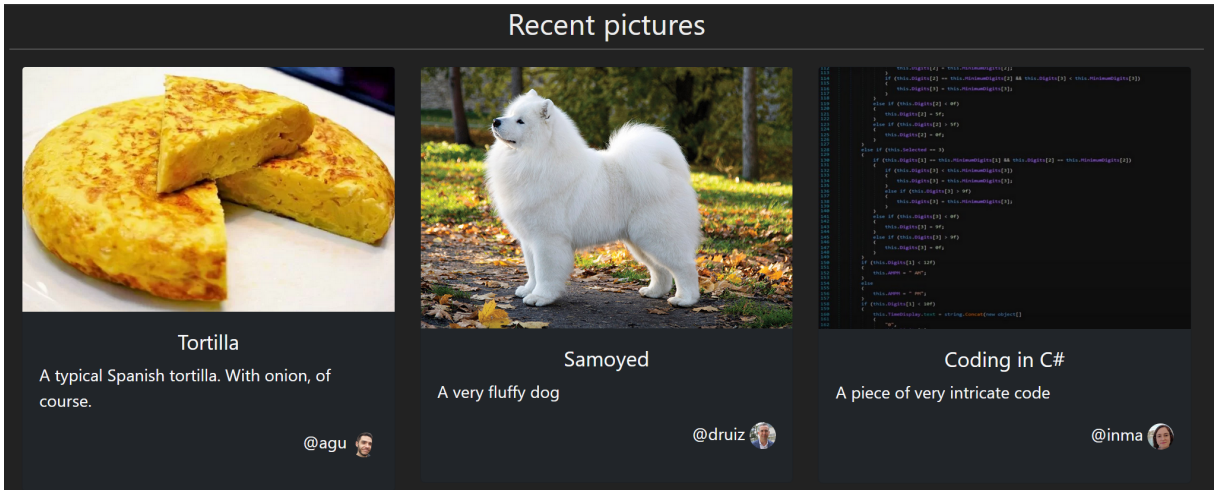
Dado que el renderizador de la galería hace uso, a su vez, del que acabamos de modificar, este cambio hace que automáticamente todas las fotos de la galería muestren el nombre del usuario que las subió:

```
asCard: function (photo) {
    let html = `<div class="col-md-4">
        <div class="card bg-dark text-light">
            <a href="photo_detail.html?photoId=${photo.photoId}">
                <img src="${photo.url}" class="card-img-top">
            </a>

            <div class="card-body">
                <h5 class="card-title text-center">${photo.title}</h5>
                <p class="card-text">${photo.description}</p>
                <p class="text-end">
                    @${photo.username}
                    <img src="${photo.avatarUrl}" class="photo-user-avatar">
                </p>
            </div>
        </div>
    </div>`;

    let card = parseHTML(html);
    return card;
},
```

Por último, debemos limitar el tamaño de las imágenes de perfil que aparecen junto a las fotos mediante CSS, usando la clase que les hemos asignado:

```
img.photo-user-avatar {
    max-width: 25px;
    height: auto;
    border-radius: 50%;
}
```

El resultado es el siguiente:



6.8. Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo *commit* a efectuar.
- `git commit -m "mensaje"` crea un nuevo *commit* con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

Laboratorio 7

Peticiones AJAX: POST, PUT y DELETE

7.1. Objetivo

El objetivo de esta práctica es profundizar en la comunicación del back-end con el front-end mediante peticiones AJAX que modifiquen el estado de la base de datos. El alumno aprenderá a:

- Usar la librería JavaScript "Axios" para realizar peticiones AJAX de tipo POST, PUT y DELETE a los endpoints del proyecto, mediante el enfoque orientado a módulos JavaScript aprendido en la práctica anterior.
- Realizar envíos de formulario por la vía indicada en el punto anterior.

7.2. Introducción

En la práctica anterior, realizamos una comunicación básica con el back-end de nuestra aplicación, realizando consultas GET para obtener datos almacenados en la base de datos y poder mostrarlos en las diferentes pantallas. En esta sesión profundizaremos en este aspecto realizando otro tipo de operaciones, tales como la creación, modificado y borrado de recursos usando métodos POST, PUT y DELETE respectivamente.

Para ello, implementaremos una nueva vista con un formulario que servirá tanto para subir nuevas fotos como para modificar una foto existente. Además, dotaremos de funcionalidad a los botones de "Editar foto" y "Borrar foto" ubicados en la vista de detalle de foto.

7.3. Formulario de edición de foto

Para poder crear y modificar fotos, necesitaremos una vista que contenga un formulario para poder introducir los datos propios de cada foto, a saber:

- URL de la imagen
- Título
- Descripción
- Visibilidad (pública / privada)

Para ello, crearemos un archivo `edit_photo.html` en el que, dentro de la plantilla usual para documentos HTML5, crearemos el formulario correspondiente:

```
<form id="form-photo-upload">
    <div class="form-group">
        <label for="input-url">URL:</label>
        <input required type="text" class="form-control" id="input-url" name="url"
            placeholder="Photo URL">
    </div>

    <div class="form-group">
        <label for="input-title">Title:</label>
        <input required type="text" class="form-control" id="input-title" name="title"
            placeholder="Title">
    </div>

    <div class="form-group">
        <label for="input-description">Description:</label>
        <textarea class="form-control" id="input-description" name="description"
            placeholder="Describe your image..." rows="4"></textarea>
    </div>

    <div class="form-group">
        <label for="input-visibility">Visibility:</label>
        <select required class="form-control" id="input-visibility" name="visibility">
            <option value="Public">Public</option>
            <option value="Private">Private</option>
        </select>
    </div>

    <div class="row">
        <div class="col-md text-center">
            <button type="submit" class="btn btn-primary">Send</button>
        </div>
    </div>
</form>
```

Nota: Se recomienda añadir un enlace a `edit_photo.html` en la barra de navegación para su fácil acceso.

7.4. Creación de fotos con POST

Tras estos cambios en el proyecto, estamos listos para implementar la creación de nuevos fotos en el formulario creado. Como es habitual, crearemos un archivo JavaScript con el mismo nombre de la vista en cuestión, en este caso, `edit_photo.js`:

```
"use strict";

import { photosAPI } from "/js/api/photos.js";
import { messageRenderer } from "/js/renderers/messages.js";

async function main() {
    ...
}

document.addEventListener("DOMContentLoaded", main);
```

Crearemos una función destinada a gestionar el envío del formulario de creación de foto, que tiene ID `form-photo-upload`:

```
async function main() {
    let registerForm = document.getElementById("form-photo-upload");
    registerForm.onsubmit = handleSubmitPhoto;
}

function handleSubmitPhoto(event) {
    ...
}
```

En esta función, crearemos un objeto FormData a partir del formulario que está siendo enviado, y usaremos el módulo de API para enviarlo mediante una petición POST. Si la petición tiene éxito, redirigiremos al usuario de nuevo a la página principal para que pueda ver la nueva foto creada. Si hay algún fallo, mostraremos el mensaje:

```
async function handleSubmitPhoto(event) {
    event.preventDefault();

    let form = event.target;
    let formData = new FormData(form);

    // Add the current user ID
    formData.append("userId", 1);

    try {
        let resp = await photosAPI_auto.create(formData);
        let newId = resp.lastId;
        window.location.href = `photo_detail.html?photoId=${newId}`;
    } catch (err) {
        messageRenderer.showErrorMessage(err.response.data.message);
    }
}
```

Son importantes las siguientes consideraciones adicionales:

- La primera línea, `event.preventDefault()`, evita que el navegador envíe por su cuenta el formulario, ya que lo estamos haciendo nosotros mediante Axios.
- Un atributo de la foto es el ID del usuario que la sube, información que no se puede obtener del formulario. En el siguiente laboratorio aprenderemos a gestionar todo lo relacionado con sesiones, por ahora proporcionaremos un valor por defecto.
- El método `.append()` de un objeto FormData permite añadir un par clave/valor a los datos a enviar.
- La página actual se guarda en `window.location.href`. Si cambiamos su valor, hacemos que el navegador vaya a otra página.
- En la implementación proporcionada no se realiza ninguna validación del formulario. Deberá implementar la validación que considere necesaria, y únicamente realizar la petición POST si no se ha producido ningun error.
- Podemos acceder a los datos devueltos por el servidor tras crear la foto, y usar la ID devuelta para dirigir al usuario a la vista de detalle de la foto que acaba de crear.

Si ha establecido `auth_required` a `true` en el endpoint de creación de foto se mostrará un error al enviar el formulario, ya que no se ha iniciado sesión. En la siguiente sesión de laboratorio resolveremos este problema, por el momento, puede establecerlo a `false`.

7.5. Edición y borrado de fotos

En la vista de detalle de fotos, `photo_detail.html`, existen dos botones para editar y modificar la foto en cuestión, pero hasta el momento no tenían ninguna funcionalidad.

Para asignarles funciones manejadoras de eventos al pulsar en ellos, primero debemos darles un ID a cada uno:

```
<button id="button-edit" class="btn btn-primary">Edit this photo</button>
<button id="button-delete" class="btn btn-danger">Delete this photo</button>
```

Entonces, editaremos el archivo `photo_detail.js` para añadirle funciones manejadoras a los eventos `click` de ambos botones. En el caso del botón de borrado, crearemos una función que pregunte al usuario si realmente desea eliminar la foto mediante la función `confirm()`[1]. En el caso de una respuesta afirmativa, usaremos el módulo de API para emitir una petición DELETE y devolveremos al usuario a la página principal:

```
async function main() {
    // Assign the handler function to the delete button
    let deleteBtn = document.querySelector("#button-delete");
    deleteBtn.onclick = handleDelete;
}

async function handleDelete(event) {
    let answer = confirm("Do you really want to delete this photo?");

    if (answer) {
        try {
            await photosAPI_auto.delete(photoId);
            window.location = "/index.html";
        } catch (err) {
            messageRenderer.showErrorMessage(err.response.data.message);
        }
    }
}
```

Como observará, dado que en la práctica anterior definimos `photoId` en el ámbito global, podemos usarlo en la función `handleDelete` para saber el ID de la foto que estamos mostrando.

Para editar la foto, redirigiremos al usuario a un formulario de edición de foto. En principio, podríamos estar tentados a crear una nueva vista para editar una foto. Sin embargo, ya contamos con un formulario que contiene todo lo necesario para modificar los campos de una foto: el formulario de creación de foto. Podemos aprovechar esta vista para darle dos propósitos y evitar tener vistas duplicadas:

- Si `edit_photo.html` no recibe ningún parámetro de URL, lo usaremos para crear una nueva foto.
- Si recibe un `photoId` como parámetro de URL (p.ej. `edit_photo.html?photoId=6`), lo usaremos para modificar la foto en cuestión.

En ese caso, el botón para editar foto de `photo_detail.js` se limitará a redireccionar a esta vista, proporcionando el ID de foto correspondiente a la foto actual:

[1] La función `confirm()` está disponible de manera nativa en los navegadores, y muestra al usuario un mensaje proporcionado como parámetro junto con botones para confirmar o cancelar. Devuelve un boolean indicando si el usuario ha aceptado o no.

```
async function main() {
    ...
    let editBtn = document.querySelector("#button-edit");
    editBtn.onclick = handleEdit;
}

function handleEdit(event) {
    window.location.href = "edit_photo.html?photoId=" + photoId;
}
```

En la siguiente sección realizaremos las modificaciones necesarias en el formulario de creación de foto para permitir editar una foto.

7.5.1. Actualizando la vista de creación de fotos

Nos centraremos ahora en `edit_photo.html` y su archivo JS asociado, `edit_photo.js`. Como explicamos anteriormente, debemos determinar si hemos recibido un `photoId` (en ese caso, editaremos una foto existente) o si no lo hemos recibido (en ese caso, crearemos una foto nueva). Podemos usar el mismo código usado en `photo_details.js` para capturar los parámetros de URL recibidos y obtener uno de ellos:

```
let urlParams = new URLSearchParams(window.location.search);
let photoId = urlParams.get("photoId");
let currentPhoto = null;
```

Además, usaremos una variable `currentPhoto` para almacenar los atributos de la foto que estamos editando, si es el caso. Si no, esta variable permanecerá como `null`.

En la función `main()` comprobaremos si hemos recibido algún parámetro de URL. Si es el caso, realizaremos las siguientes operaciones:

- Modificar el título de la página para que sea “Editando una foto” en lugar de “Creando una nueva foto”.
- Consultar la foto con el ID recibido a la API.
- Almacenar la foto en la variable `currentPhoto`.
- Editar los campos del formulario para establecer sus valores iniciales a aquellos que tenga la foto que estamos editando.

```
async function main() {
    if (photoId !== null) {
        loadCurrentPhoto();
    }

    ...
}

async function loadCurrentPhoto() {
    let pageTitle = document.getElementById("page-title");
    let urlInput = document.getElementById("input-url");
    let titleInput = document.getElementById("input-title");
    let descriptionInput = document.getElementById("input-description");
    let visibilityInput = document.getElementById("input-visibility");

    pageTitle.textContent = "Editing a photo";

    try {
        currentPhoto = await photosAPI_auto.getById(photoId);
        urlInput.value = currentPhoto.url;
        titleInput.value = currentPhoto.title;
        descriptionInput.value = currentPhoto.description;
        visibilityInput.value = currentPhoto.visibility;
    } catch (err) {
        messageRenderer.showErrorMessage(err.response.data.message);
    }
}
```

Esto provocará que, si estamos editando una foto, ya aparezcan cargados todos los atributos de la foto en los inputs correspondientes. Finalmente, debemos modificar la función encargada del envío del formulario, `handleSubmitPhoto`, para realizar una operación u otra dependiendo de si estamos editando una foto o creando una nueva:

```
async function handleSubmitPhoto(event) {
    event.preventDefault();

    let form = event.target;
    let formData = new FormData(form);

    if (currentPhoto === null) { // Creating a new photo
        // Add the current user ID
        formData.append("userId", 1);

        try {
            let resp = await photosAPI_auto.create(formData);
            let newId = resp.lastId;
            window.location.href = `photo_detail.html?photoId=${newId}`;
        } catch (err) {
            messageRenderer.showErrorAsAlert(err.response.data.message);
        }
    } else { // Updating an existing photo
        formData.append("userId", currentPhoto.userId);
        formData.append("date", currentPhoto.date);

        try {
            await photosAPI_auto.update(formData, photoId);
            window.location.href = `photo_detail.html?photoId=${photoId}`;
        } catch (err) {
            messageRenderer.showErrorAsAlert(err.response.data.message);
        }
    }
}
```

Si estamos editando una foto, `currentPhoto` no será `null`, y podemos acceder a atributos como el usuario que subió la foto y la fecha de subida para añadirlos al formulario.

7.6. Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo *commit* a efectuar.
- `git commit -m "mensaje"` crea un nuevo *commit* con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

Laboratorio 8

Gestión de sesiones

8.1. Objetivo

El propósito de esta práctica es aprender a realizar operaciones de registro, login y cierre de sesión en el cliente, haciendo uso de los endpoints provistos por el servidor a tal efecto. Además, se mostrarán conceptos básicos sobre gestión de la sesión en el cliente usando JavaScript. El alumno aprenderá a:

- Realizar el registro de un nuevo usuario en la aplicación.
- Almacenar en el cliente, mediante JavaScript, los datos del usuario con la sesión iniciada.
- Implementar un elemento para cerrar la sesión.
- Mantener y gestionar tokens de sesión para poder usar endpoints protegidos.
- Mostrar y usar datos del usuario actual en las diferentes vistas.
- Adaptar las vistas en función de si el usuario está autenticado o no.

8.2. Introducción

La gestión de sesiones de usuarios es, sin duda, un aspecto fundamental en cualquier aplicación Web. Este aspecto varía en gran medida según el framework Web usado, aunque algunos conceptos básicos son de aplicación general.

En esta práctica dotaremos de funcionalidad al formulario de registro implementado anteriormente, y aprenderemos a realizar una gestión básica de las sesiones de usuario en cliente.

8.3. Configuración de Silence

El framework Silence provee por defecto dos endpoints para realizar estas operaciones:

- `/login` recibe los datos de inicio de sesión de un usuario (identificador y contraseña).
- `/register` recibe todos los datos de un usuario y lo da de alta en la apliación, almacenando su contraseña de manera segura en la BD.

Estos dos endpoints devuelven **la misma respuesta** JSON: un token de sesión en un atributo llamado "sessionToken" y los datos completos del usuario logueado o registrado en un atributo llamado "user". **En esta práctica implementaremos únicamente el registro, dado que la gestión del inicio de sesión es idéntica**. El cierre de sesión, por otro lado, se efectúa únicamente por parte del cliente (navegador), eliminando de la memoria local el token de sesión almacenado.

Para que funcionen correctamente los dos endpoints anteriormente descritos es fundamental configurar adecuadamente el parámetro `USER_AUTH_DATA` del proyecto, especificando el nombre de la tabla que contiene los usuarios, así como las columnas que se usan como identificador y contraseña respectivamente.

8.4. Módulo API de autenticación

Al igual que para todas las interacciones que hemos hecho hasta el momento con la API del proyecto, las operaciones de login y registro las efectuaremos haciendo uso de un módulo JavaScript destinado a interactuar con estos endpoints. Por ejemplo, podemos implementar las operaciones POST de login y registro en un módulo `js/api/auth.js`:

```
"use_strict";

import { BASE_URL, requestOptions } from "./common.js";

const authAPI = {

    login: async function(formData) {
       let response = await axios.post(`${BASE_URL}/login`, formData, requestOptions);
       return response.data;
    },

    register: async function(formData) {
       let response = await axios.post(`${BASE_URL}/register`, formData, requestOptions
           );
       return response.data;
    },

};

export { authAPI };
```

8.5. Módulo de gestión de sesiones

El proyecto descargado incluye un módulo encargado de ofrecer métodos para facilitar la gestión de las sesiones en el navegador, que se puede encontrar en `js/utils/session.js`. Los métodos que ofrece el objeto `sessionManager` exportado por el módulo son:

- `login(token, user)` inicia sesión con el token de sesión y los datos de usuario proporcionados.
- `logout()` cierra la sesión activa.
- `getToken()` devuelve el token de sesión actual, o `null` si la sesión no está activa o ha caducado.
- `isLogged()` devuelve un boolean indicando si hay una sesión activa o no.
- `getLoggedUser()` devuelve los datos del usuario actual, o `null` si la sesión no está activa.
- `getLoggedId()` devuelve el `userId` del usuario actual, o `null` si la sesión no está activa.

Internamente, este módulo hace uso del `localStorage` del navegador para almacenar localmente datos recibidos del servidor, tales como el token de sesión que identifica a una sesión activa, y que es necesario para acceder a endpoints protegidos. En las siguientes secciones, veremos cómo usar este módulo de manera prática.

8.6. Registro de usuario

En prácticas anteriores, implementamos un formulario de registro en HTML en la vista `register.html` y lo dotamos de validación en `register.js`. Ahora, es hora de modificar este último fichero para realizar una petición a la API de registro y usar el resultado para iniciar una sesión.

Comenzaremos por importar en `register.js` tanto el módulo de API creado anteriormente como el módulo de gestión de sesiones:

```
import { sessionManager } from "/js/utils/session.js";
import { authAPI } from "/js/api/auth.js";
```

La función que realiza la validación del formulario es `handleSubmitRegister`, que se ejecuta cada vez que el usuario envía el formulario. Modificaremos esta función para que, si la validación es exitosa (no hay errores), se envíe el formulario haciendo uso del módulo de API correspondiente, usando una función aparte. Si no, se le mostrarán al usuario los errores:

```
function handleSubmitRegister(event) {
    event.preventDefault();
    let form = event.target;
    let formData = new FormData(form);

    let errors = userValidator.validateRegister(formData);

    if (errors.length > 0) {
        let errorsDiv = document.getElementById("errors");
        errorsDiv.innerHTML = "";

        for (let error of errors) {
            messageRenderer.showErrorMessage(error);
        }
    } else {
        sendRegister(formData);
    }
}
```

En la función `sendRegister`, que hemos de definir, recibiremos los datos del formulario ya validados y los enviaremos al endpoint de registro. Esta función será **asíncrona**, ya que haremos en ella una petición a la API. Inicialmente, mostraremos por consola los datos recibidos del servidor, para poder inspeccionar su contenido:

```
async function sendRegister(formData) {
    try {
        let loginData = await authAPI.register(formData);
        console.log(loginData);
    } catch (err) {
        messageRenderer.showErrorMessage("Error registering a new user", err);
    }
}
```

Si creamos un nuevo usuario mediante el formulario, aparecerá la respuesta del servidor en la consola:

Como se explicó en la Sección 8.3, esta respuesta tiene dos atributos: "sessionToken", que contiene el token de sesión, y "user", que contiene los datos del usuario. Estos dos parámetros son los requeridos por el método `login()` del módulo de gestión de sesiones, explicado en la Sección 8.5. Así, podemos guardarlos como variables y usarlos para iniciar sesión:

```
async function sendRegister(formData) {
    try {
        let loginData = await authAPI.register(formData);
        let sessionToken = loginData.sessionToken;
        let loggedUser = loginData.user;

        sessionManager.login(sessionToken, loggedUser);
        window.location.href = "index.html";
    } catch (err) {
        messageRenderer.showErrorMessage("Error registering a new user", err);
    }
}
```

Si el registro es exitoso, iniciaremos la sesión del usuario creado y redirigiremos al usuario a la página principal. Si no, mostraremos el error correspondiente.

Recuerde que los nombres de los atributos de los usuarios se deben corresponder con los valores de los atributos "name" de los `<input>` del formulario.

8.7. Actualizando la barra de navegación

Tras implementar el inicio de sesión, existen varias modificaciones que podemos hacer en la barra de navegación:

8.7.1. Mostrar el usuario actual

Las modificaciones anteriores son suficientes para registrar un usuario e iniciar sesión con él, pero no hay ningún tipo de feedback visual que indique al usuario si éste tiene una sesión iniciada o no. Para lograr este propósito, dotaremos de dinamismo al texto que aparece a la izquierda en la barra de navegación:

- Si hay una sesión iniciada, mostraremos un saludo y el nombre de usuario.
- Si no, mostraremos "Anónimo".

El elemento en cuestión es el enlace `<a>` con `class="navbar-brand"` que se encuentra en el archivo HTML de la cabecera (`header.html`). Para poder acceder a él mediante JS, le daremos un ID:

```
<a class="navbar-brand" id="navbar-title" href="#"></a>
```

A continuación, crearemos un archivo `header.js` para realizar operaciones en la cabecera, con la estructura habitual:

```
"use strict";

function main() {
    ...
}

document.addEventListener("DOMContentLoaded", main);
```

Deberá enlazar `header.js` en todas las vistas de su aplicación, junto con el resto de archivos JavaScript:

```
<script src="js/header.js" type="module"></script>
```

En la función `main()`, comprobaremos si hay una sesión iniciada y, en ese caso, accederemos al nombre de usuario actual, usando el módulo `session.js`:

```
import { sessionManager } from "/js/utils/session.js";

function main() {
    showUser();
}

function showUser() {
    let title = document.getElementById("navbar-title");
    let text;

    if (sessionManager.isLogged()) {
        let username = sessionManager.getLoggedUser().username;
        text = "Hi, @" + username;
    } else {
        text = "Guest";
    }

    title.textContent = text;
}
```

8.7.2. Cierre de sesión

Es habitual incluir en la barra de navegación un botón para cerrar la sesión rápidamente. En nuestro caso, podemos hacerlo añadiendo un nuevo elemento a la barra, con un ID determinado para poder darle funcionalidad mediante JavaScript:

```
<li class="nav-item"><a class="nav-link" href="#" id="navbar-logout">Logout</a></li>
```

Podemos implementar una función que se lance cuando el usuario pulse en él, y cierre su sesión enviándolo a la página de inicio:

```
function main() {
    showUser();
    addLogoutHandler();
}

function addLogoutHandler() {
    let logoutButton = document.getElementById("navbar-logout");

    logoutButton.onclick = function () {
        sessionManager.logout();
        window.location.href = "index.html";
    };
}
```

8.7.3. Ocultar opciones no disponibles

Actualmente, nuestra aplicación muestra al usuario todas las opciones disponibles con independencia de si está logueado o no. Mediante JavaScript podemos ocultar aquellos elementos de la barra de navegación que no deben mostrarse a un usuario invitado, como por ejemplo, el cierre de sesión o el enlace al formulario de crear foto. Asimismo, le ocultaremos a un usuario logueado opciones como el login y el registro.

Como es habitual, debemos darle IDs a los elementos de la barra de navegación para poder acceder a ellos mediante JS y ocultarlos si es necesario:

```
<ul class="navbar-nav me-auto mb-2 mb-lg-0">
    <li class="nav-item">
        <a class="nav-link" href="index.html" id="navbar-recent">Recent photos</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="edit_photo.html" id="navbar-create">Upload photo</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="register.html" id="navbar-register">Register</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="login.html" id="navbar-login">Login</a>
    </li>
    <li class="nav-item dropdown" id="navbar-trending">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button"
            data-bs-toggle="dropdown"
            aria-expanded="false">
            Trending photos
        </a>
        <ul class="dropdown-menu bg-dark" aria-labelledby="navbarDropdown">
            <li><a class="dropdown-item text-light" href="trending_users.html">Trending
                 users</a></li>
            <li><a class="dropdown-item text-light" href="trending_photos.html">
                Trending photos</a></li>
        </ul>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="#" id="navbar-logout">Logout</a>
    </li>
</ul>
```

En la función `main()`, llamaremos por último a una función auxiliar que ocultará los elementos adecuados según el estado de la sesión del usuario:

```
function main() {
    showUser();
    addLogoutHandler();
    hideHeaderOptions();
}

function hideHeaderOptions() {
    let headerRegister = document.getElementById("navbar-register");
    let headerLogin = document.getElementById("navbar-login");
    let headerLogout = document.getElementById("navbar-logout");
    let headerRecent = document.getElementById("navbar-recent");
    let headerCreate = document.getElementById("navbar-create");
    let headerTrending = document.getElementById("navbar-trending");

    if (sessionManager.isLogged()) {
        headerRegister.style.display = "none";
        headerLogin.style.display = "none";
    } else {
        headerRecent.style.display = "none";
        headerCreate.style.display = "none";
        headerLogout.style.display = "none";
        headerTrending.style.display = "none";
    }
}
```

Nota: Ocultar enlaces y otros elementos no es, en sí mismo, una medida de seguridad, ya que el usuario avanzado puede acceder de todos modos introduciendo la URL a mano o usando herramientas como REST Client. La seguridad radica en configurar adecuadamente el back-end para que el servidor no permita a un usuario realizar operaciones que no le corresponden, por ejemplo, limitándolas a usuarios autenticados o que tengan un determinado rol.

8.8. Ocultando elementos en otras vistas

Al igual que hemos hecho con las opciones de la barra de navegación, existen elementos en otras vistas que debemos ocultar a usuarios no autenticados, ya que si intentaran interactuar con ellos se les mostraría un error por falta de permisos. Uno de esos casos son los botones de edición y modificación de fotos en la vista de detalle de foto, así como el formulario para emitir una valoración, dado que esas operaciones están reservadas a usuarios autenticados.

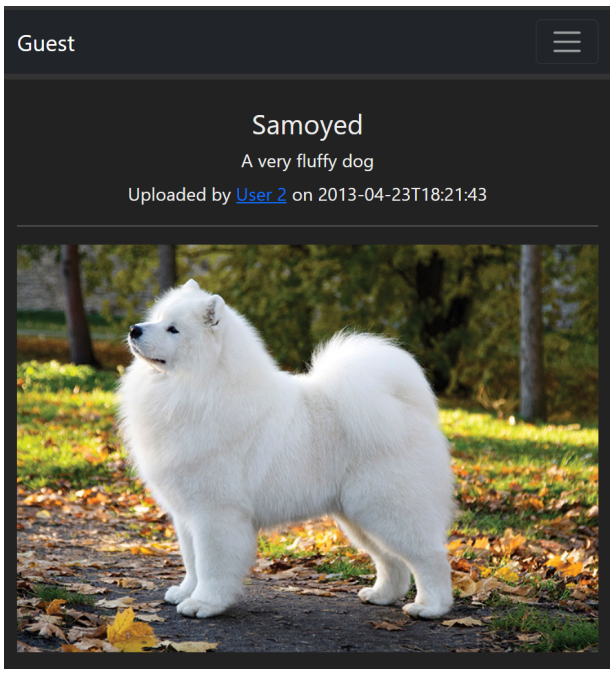
La forma de proceder es similar a la anterior: se le debe dar a esos elementos una ID, y posteriormente, en el código JS asociado a la vista, ocultarlos si el usuario no tiene la sesión iniciada. Por ejemplo, en `photo_detail.html` le daremos un ID a la columna que contiene estos elementos:

```
<div class="col-md-3" id="actions-col">
    <p>
        <button id="button-edit" class="btn btn-primary">Edit this photo</button>
        <button id="button-delete" class="btn btn-danger">Delete this photo</button>
    </p>
    ...
</div>
```

En el archivo JS asociado, `photo_details.js`, eliminaremos este elemento si no hay una sesión iniciada:

```
import { sessionManager } from "/js/utils/session.js";

async function main() {
    ...
    hideActionsColumn();
}

function hideActionsColumn() {
    let actions_col = document.getElementById("actions-col");
    if (!sessionManager.isLogged()) {
        actions_col.style.display = "none";
    }
}
```

Observe que, si se elimina esta columna, la anterior pasa a tener todo el ancho de la página, optimizando así el uso de la vista:



8.9. Subida de fotos

En la práctica anterior, dejamos un elemento pendiente en la gestión del formulario de subida de fotos: las nuevas fotos se asignaban al ID de un usuario prestablecido, en lugar de aquel que tenía la sesión iniciada. Ahora que tenemos control sobre las sesiones de los usuarios, podemos usar el módulo de sesiones para acceder al ID del usuario actual.

Realizaremos entonces la siguiente modificación en `edit_photo.js`, reemplazando el ID del usuario que introdujimos manualmente por la función que obtiene el ID del usuario logueado:

```
import { sessionManager } from "/js/utils/session.js";

async function handleSubmitPhoto(event) {
    ...

    if (currentPhoto === null) { // Creating a new photo
        // Add the current user's ID
        formData.append("userId", sessionManager.getLoggedId());
    ...
}
```

Recuerde que la creación de fotos es una operación reservada a usuarios logueados, por lo que debe configurar su endpoint adecuadamente.

Como sabe, los endpoints configurados con `auth_required: true` requieren el envío del token de sesión para su uso. Sin embargo, los módulos API que hemos creado en las sesiones anteriores envían siempre el token de sesión almacenado en caso de que esté disponible, gracias al objeto común `requestOptions`. Por ello, comprobará que no es necesario que realice ningún cambio adicional para que la creación de fotos se comporte como se espera: si está autenticado podrá crear, editar y borrar fotos; mientras que si no tiene la sesión iniciada se mostrará un mensaje de error.

8.10. Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo *commit* a efectuar.
- `git commit -m "mensaje"` crea un nuevo *commit* con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

Laboratorio A

Entorno de trabajo

A.1. Objetivo

El objetivo de esta práctica es configurar el entorno de trabajo inicial para el desarrollo de la asignatura. El alumno aprenderá a:

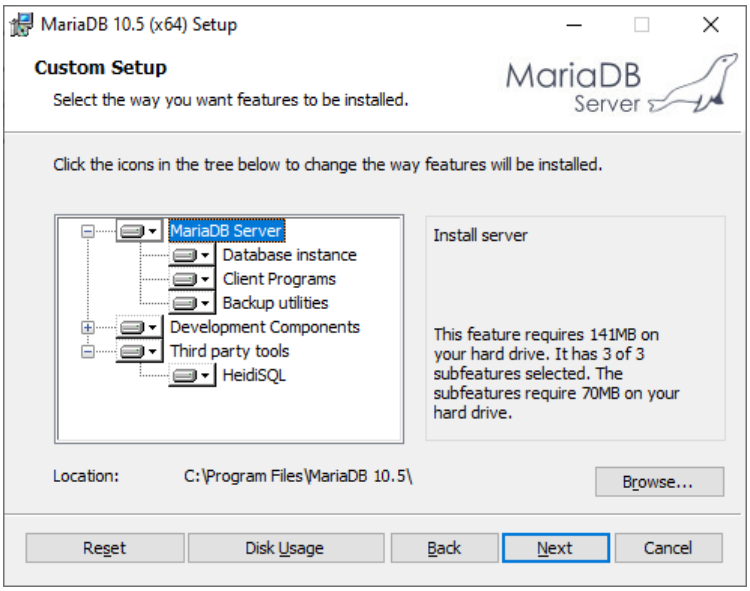
- Instalar MariaDB y HeidiSQL.
- Crear conexiones con BBDD locales.
- Crear usuarios para las BBDD.
- Cargar y ejecutar un script SQL.
- Ejecutar una consulta simple sobre una BD.
- Instalar y configurar Python.
- Instalar Visual Studio Code y extensiones relacionadas.
- Instalar y configurar Git.

A.2. Instalación de MariaDB y HeidiSQL

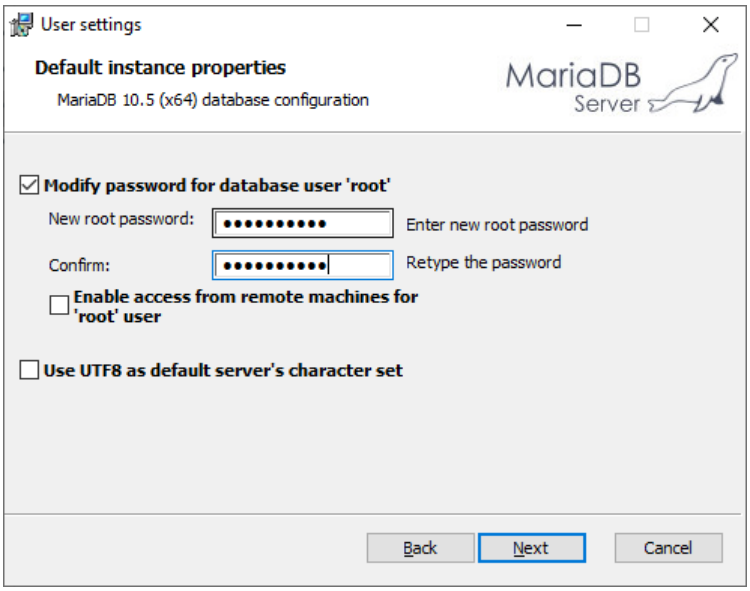
Durante el curso usaremos MariaDB, un *fork* de MySQL que comparte sus mismas funciones pero tiene una licencia más permisiva. En esta sección se explica su instalación para Windows, cuyo instalador puede encontrarse aquí.

En Linux, puede instalarse el paquete `mariadb-server` usando `apt` o el gestor de paquetes correspondiente. En macOS, se puede instalar MariaDB usando Homebrew, como se explica aquí. El cliente HeidiSQL sólo está disponible para Windows, en otros SO pueden usarse alternativas como MySQL Workbench o DBeaver.

Accederemos a la página de descargas de MariaDB y descargaremos e iniciaremos el instalador para Windows. Cuando se nos pregunte, dejaremos marcadas todas las características a instalar:



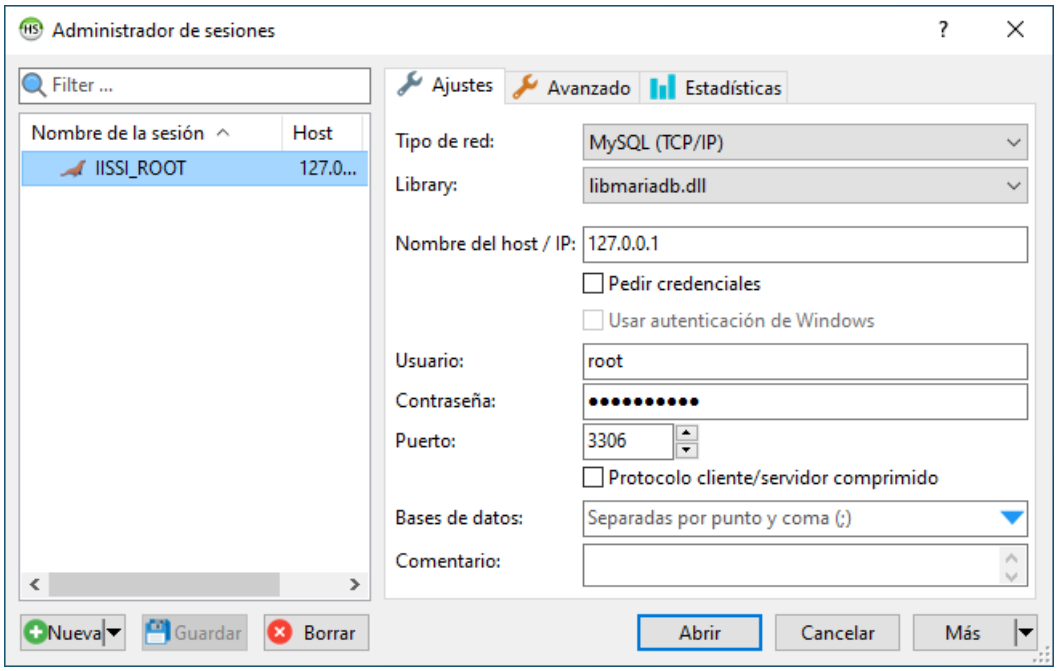
A continuación se nos preguntará por la contraseña del usuario *root* (administrador). Para nuestra BD usaremos la contraseña `iissi$root`:



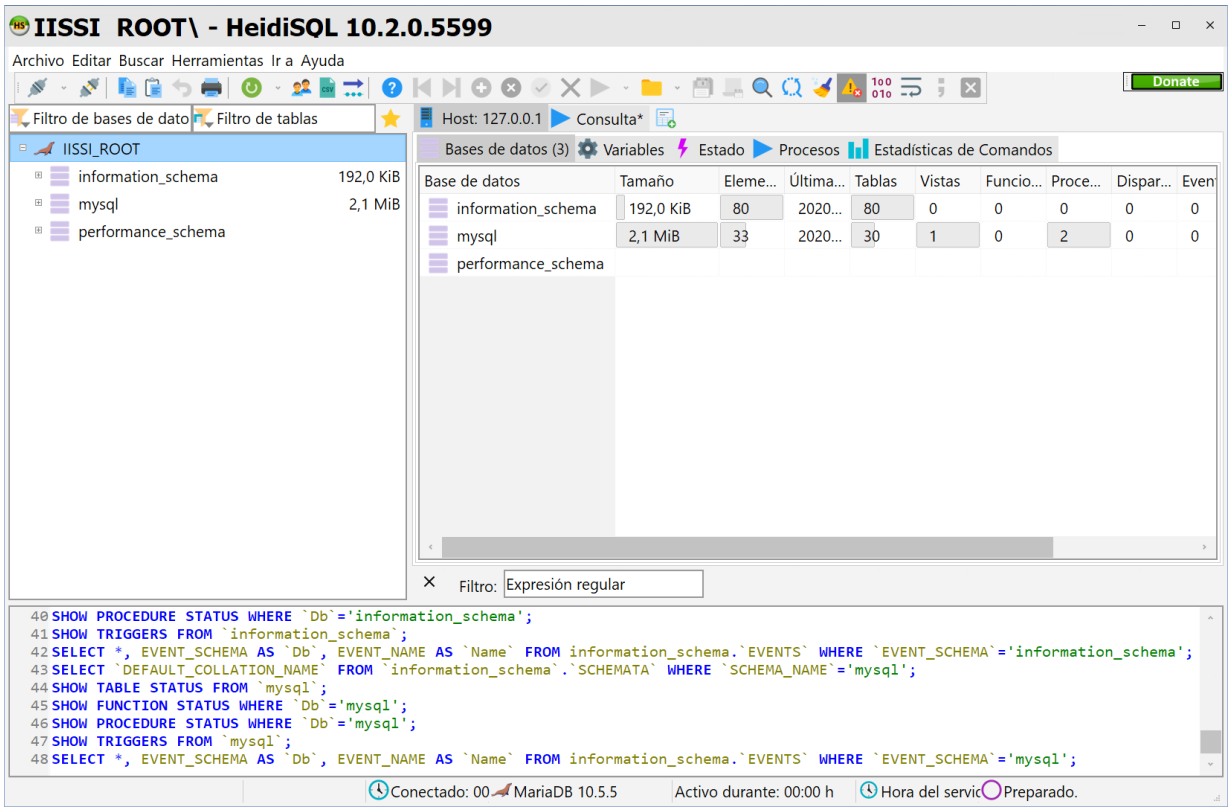
Aceptaremos las opciones por defecto a partir de este diálogo y esperaremos a que finalice la instalación, tras lo cual habrán quedado instalados tanto MariaDB como el cliente HeidiSQL.

A.3. Creación de una conexión con HeidiSQL

Para trabajar con MariaDB usaremos el cliente HeidiSQL, que debería estar instalado si se han seguido adecuadamente las pautas de la sección anterior. Ejecutamos HeidiSQL y creamos una nueva sesión, a la que llamaremos `IISSI_ROOT`, en la que indicaremos los datos de acceso del usuario root que hemos configurado anteriormente:



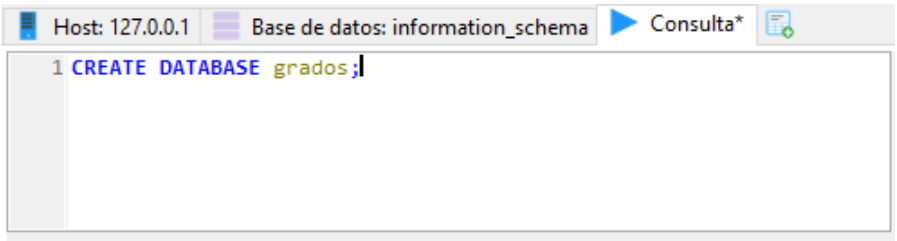
Tendremos acceso a todas las BD que tiene el SGBD. Las que aparecen por defecto (information_schema, mysql y performance_schema) son propias del SGBD y no deben ser modificadas manualmente.



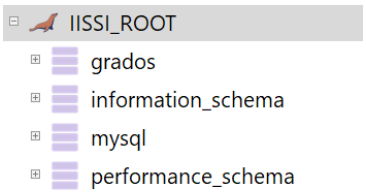
A.4. Creación de una base de datos

Una “base de datos” o “database” en MariaDB (también llamada “schema”) es un conjunto de tablas, que normalmente se corresponde con un proyecto o aplicación concreto. Mediante el uso de databases, se pueden tener varias aplicaciones con conjuntos diferentes de tablas funcionando sobre un mismo SGBD.

Como ejemplo, crearemos la base de datos “grados”. Para ello, usando el usuario *root*, accederemos a la pestaña “Consulta” y ejecutaremos `CREATE DATABASE grados;`, tras ello, pulsaremos F9 para ejecutar nuestra consulta:



Si pulsamos sobre la conexión y la actualizamos mediante F5, podremos ver que se ha creado la base de datos “grados”:

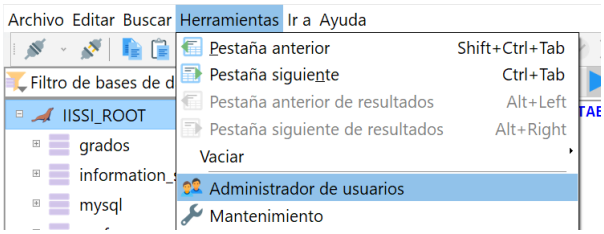


Sin embargo, realizar todas las operaciones con el usuario *root* no es aconsejable. En la siguiente sección, crearemos un nuevo usuario para operar con la base de datos recién creada.

A.5. Creación de usuarios

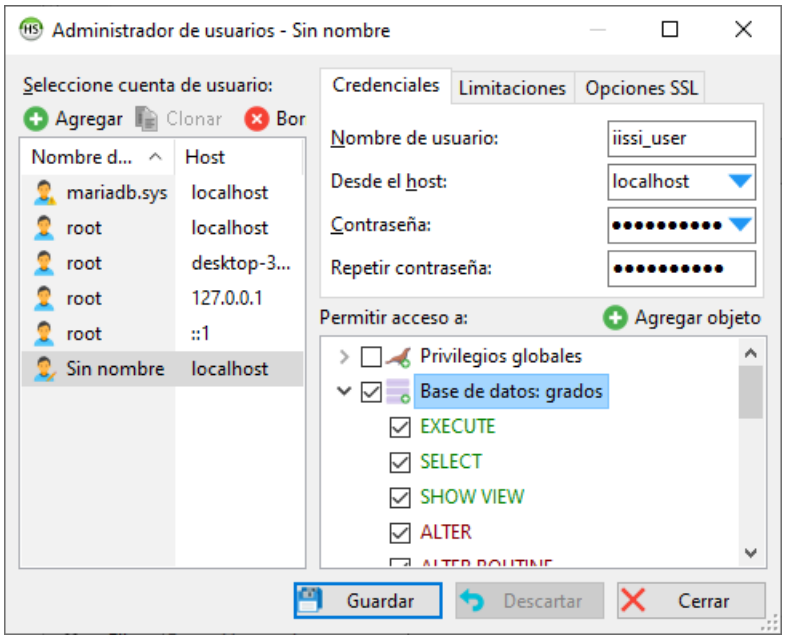
Operar directamente con el usuario *root* está altamente desaconsejado, ya que éste tiene permisos ilimitados sobre el SGBD, y cualquier error cometido puede resultar potencialmente grave. En su lugar, crearemos un usuario y le otorgaremos los permisos adecuados para poder crear y manipular bases de datos.

Crearemos un usuario usando Herramientas → Administrador de usuarios:



Pulsamos en Agregar y creamos un nuevo usuario con nombre `iissi_user` y clave `iissi$user`. En "Desde el host" se deja marcado "localhost", lo cual indica que el usuario a crear sólo podrá ser accedido desde nuestra máquina, no por conexiones remotas.

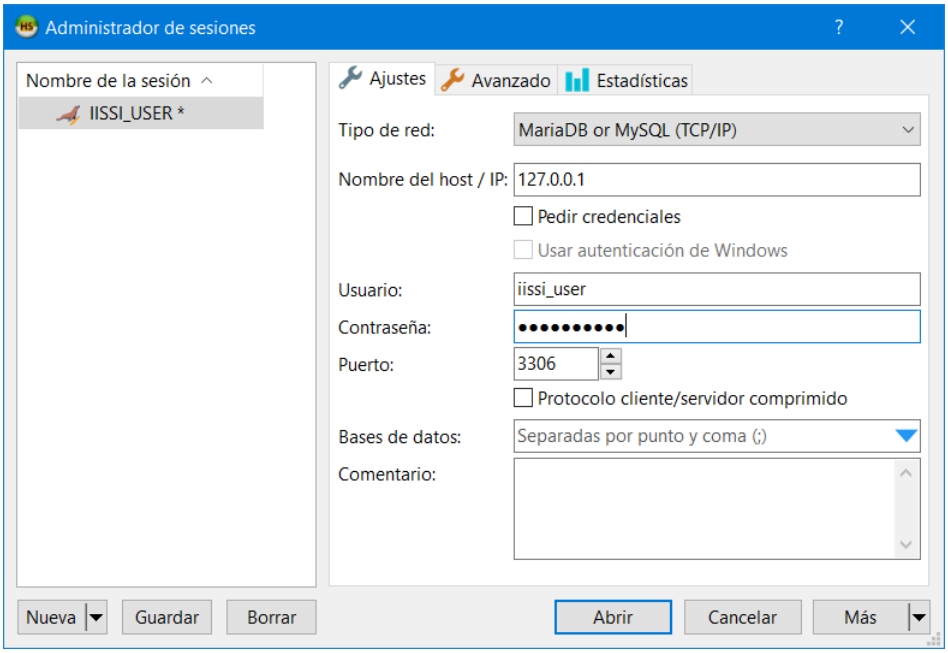
En la parte inferior podremos asignarle permisos al usuario que vamos a crear. Es aconsejable otorgar los mínimos permisos imprescindibles, por lo que el nuevo usuario sólo tendrá permisos para modificar la base de datos "grados". Para otorgarle permisos en la BD que acabamos de crear, la seleccionamos en Agregar objeto y marcamos todos los permisos.



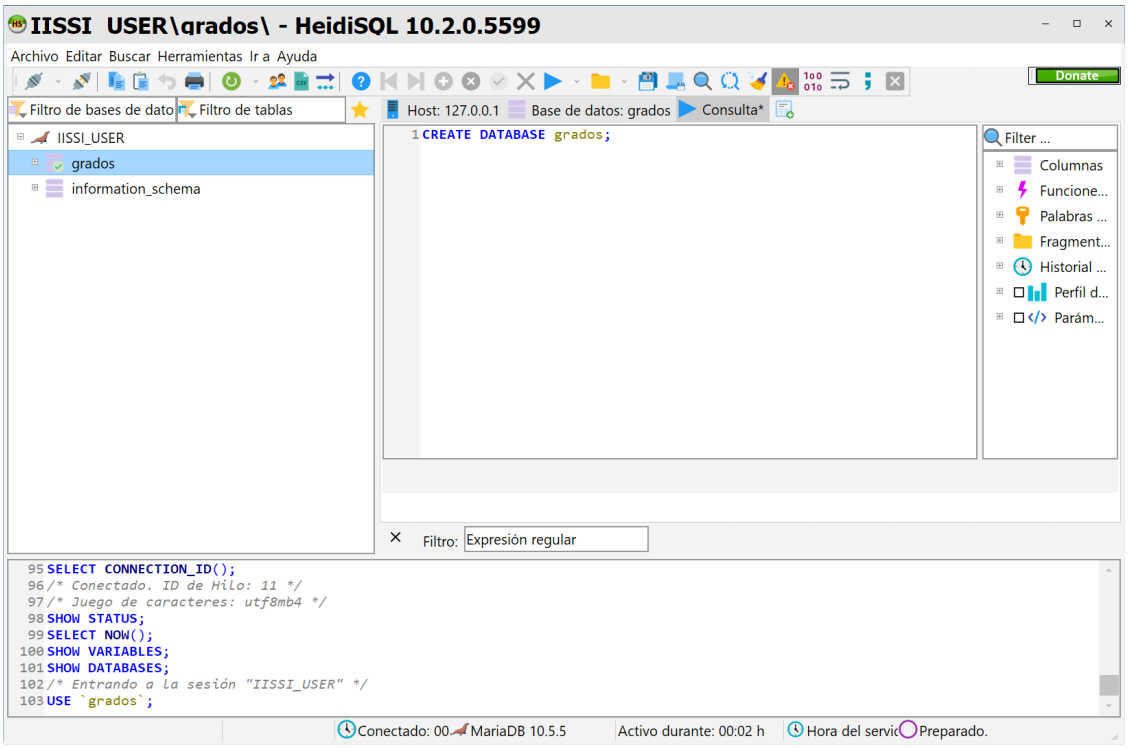
Finalmente, pulsaremos en “Guardar” para registrar el nuevo usuario.

A.6. Conexión con el nuevo usuario

Crearemos una nueva conexión “IISSI_USER” para el usuario que acabamos de crear, que será la que usaremos para trabajar con nuestras bases de datos:



En esta sesión sólo se tiene acceso a las BD del usuario, no a las del sistema:



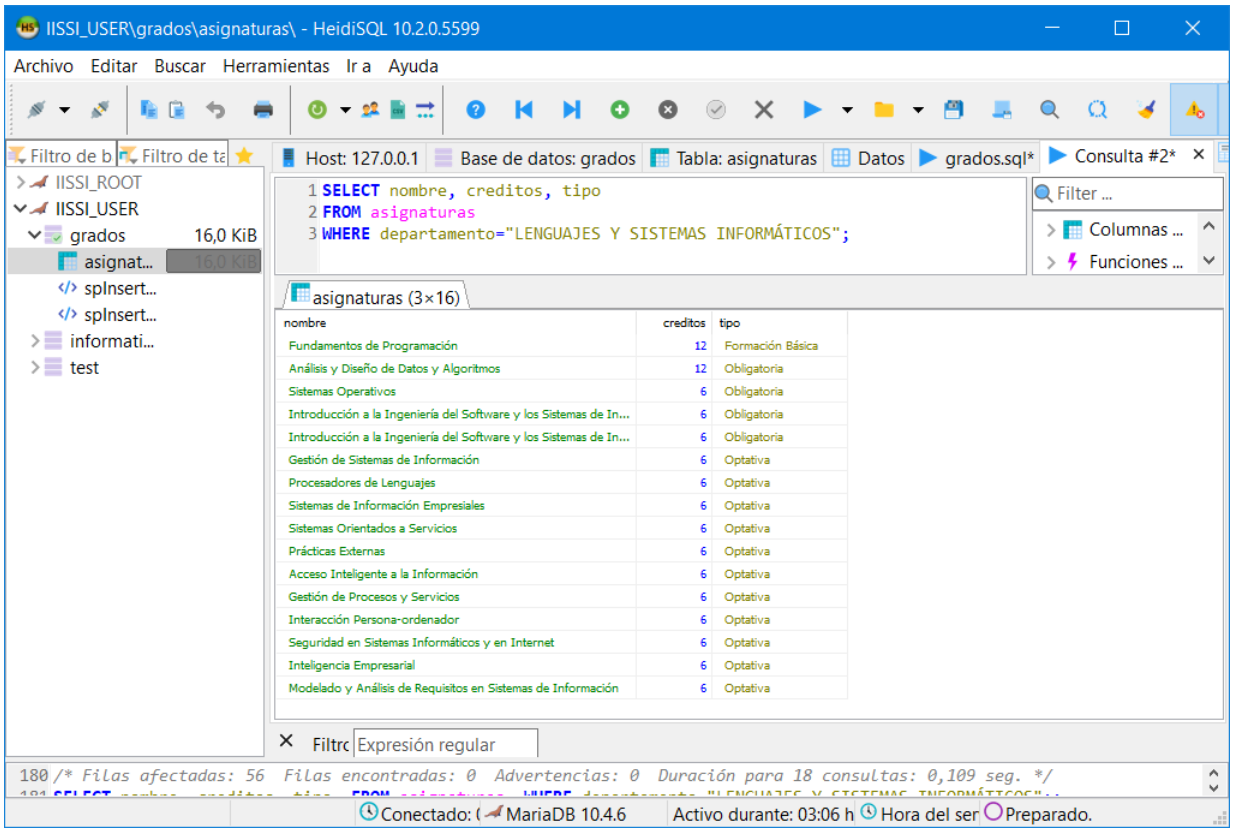
A.7. Ejecutar script de prueba

Para importar datos en la BD que acabamos de crear, utilizaremos el archivo grados.sql. Seleccionamos. Archivo → "Cargar archivo SQL" → grados.sql (Ejecutar F9 , "Enviar lote de una sola vez"). Durante su ejecución se pueden producir advertencias, pero no es algo inusual.

Al actualizar usando F5 , podrá comprobar que se ha creado una nueva tabla "Asignaturas" en la BD "Grados".

Podemos ejecutar una consulta SQL para obtener el nombre, el número de créditos y el tipo de las asignaturas impartidas por el departamento "LENGUAJES Y SISTEMAS INFORMÁTICOS" usando la pestaña "Consulta":

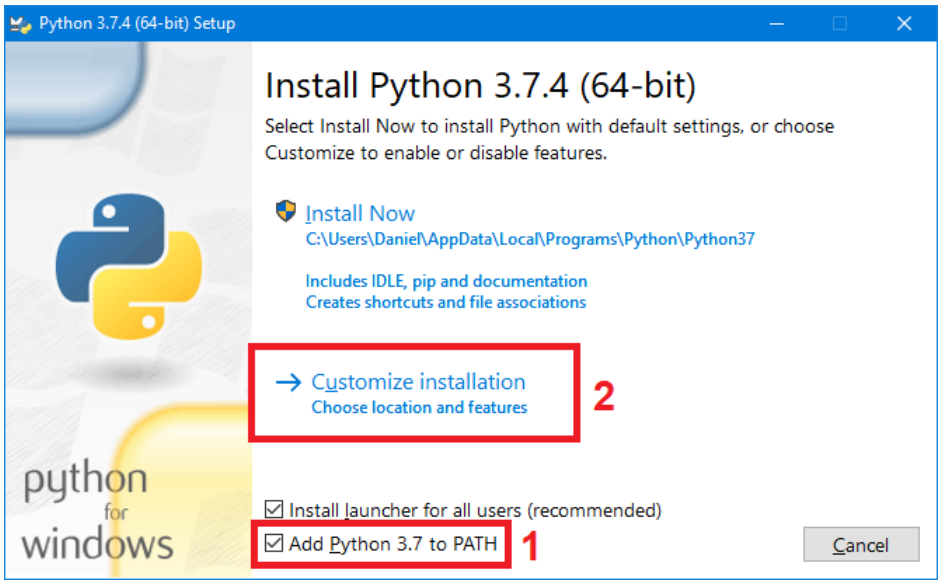
```
SELECT nombre, creditos, tipo
FROM Asignaturas
WHERE departamento = "LENGUAJES Y SISTEMAS INFORMÁTICOS";
```



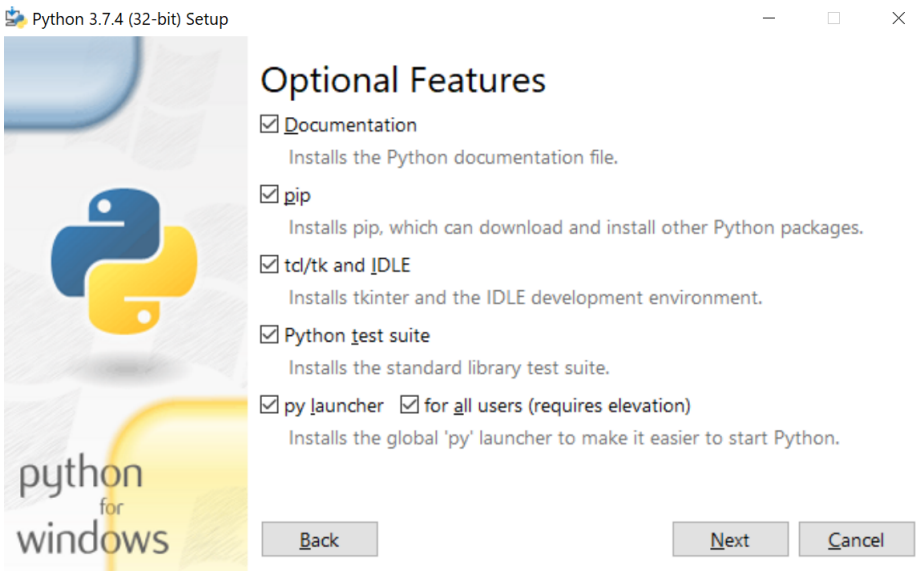
A.8. Instalación y configuración de Python

Nota: si se tiene Python instalado mediante Anaconda, es posible que surjan problemas de compatibilidad. En ese caso, se recomienda desinstalar Anaconda antes de proceder.

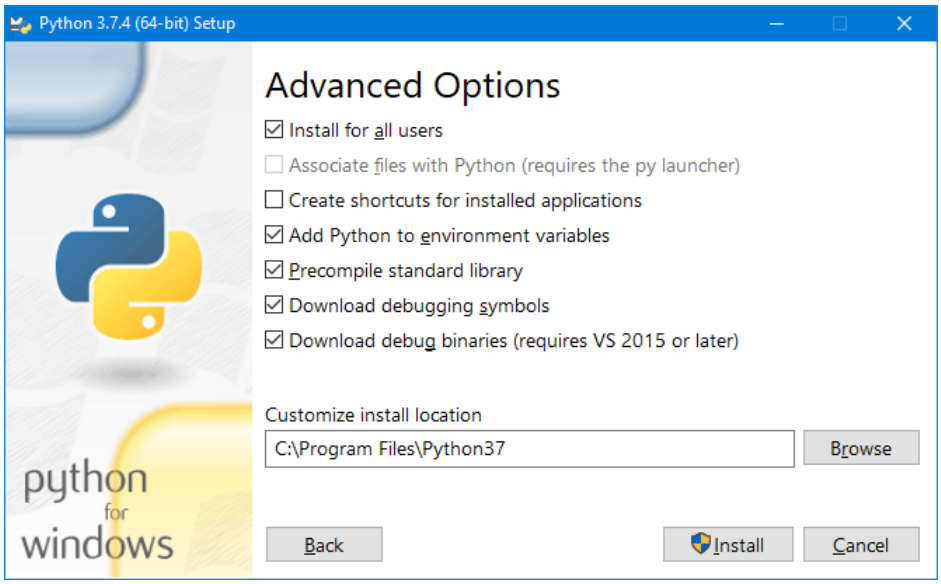
Python es necesario para usar el framework Silence, que se empleará al final de IISSI-1 y durante IISSI-2. Descargamos e instalamos la versión 3.X más reciente de Python. Seleccionamos personalizar instalación ("customize installation"). **Es importante marcar antes la opción "Add Python 3.X to PATH":**



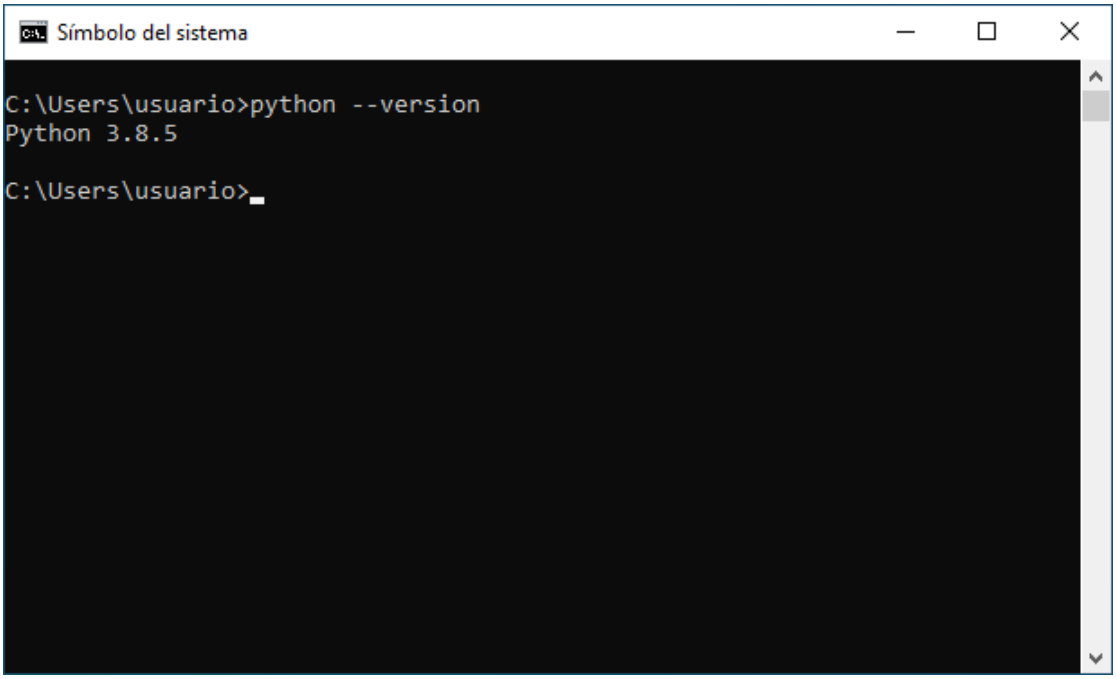
Entre las opciones de instalación opcionales, **dejamos marcadas todas las opciones**:



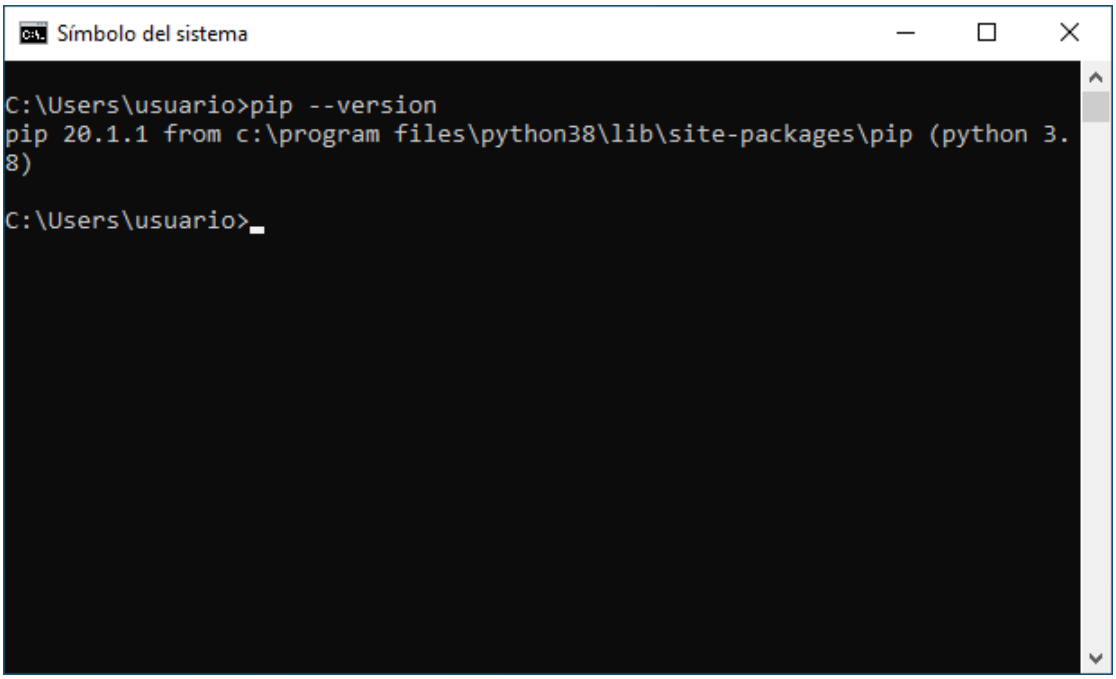
Marcamos las siguientes opciones avanzadas:



Una vez concluya la instalación, podemos comprobar que ésta ha sido correcta abriendo una consola y consultando la versión de Python instalada mediante `python --version`:

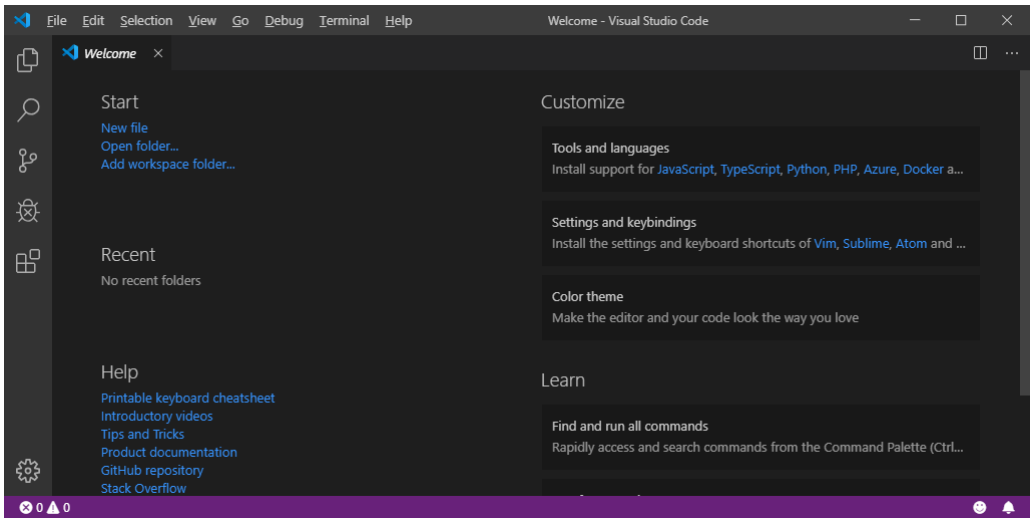


Igualmente, comprobaremos que pip, el gestor de paquetes de Python, está correctamente instalado, ya que lo necesitaremos más adelante. Para ello podemos ejecutar `pip --version`:



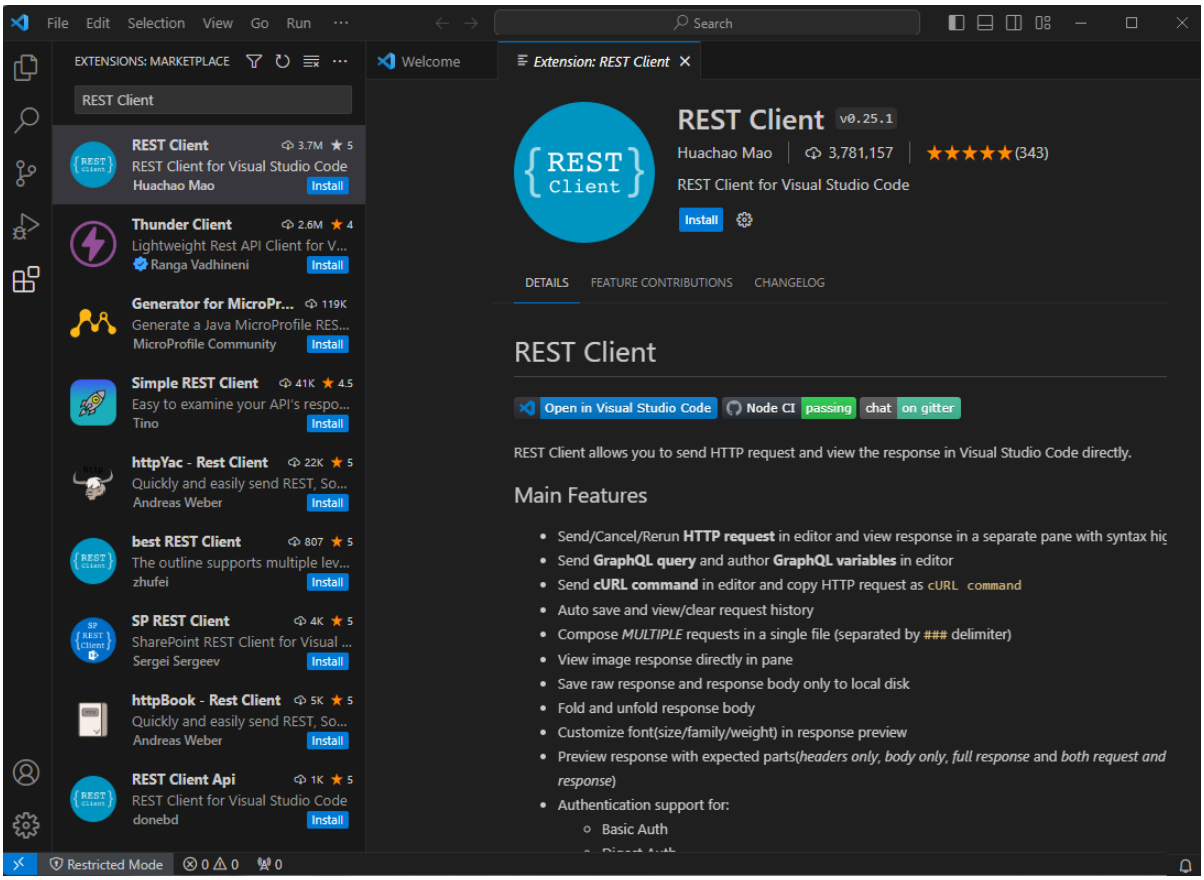
A.9. Instalación de Visual Studio Code

Como editor de código se usará Visual Studio Code. Descargamos el instalador y lo ejecutamos. Mantenemos las opciones en sus valores por defecto e iniciamos Visual Studio Code una vez finalice la instalación:



Accedemos a File → Preferences → Extensions, seleccionamos la extensión de Python y la instalamos:

Asimismo, instalaremos también la [extensión REST Client]:



A.10. Instalación de Git

Git es un sistema de control de versiones que usaremos a lo largo de IISSI-1 y 2 para descargar material relacionado con la asignatura, registrar nuestros cambios y mantener una copia de ellos en GitHub.

Para instalar Git, por favor, consulte el boletín auxiliar de Git y GitHub que está publicado.

Si desea consultar las nociones básicas sobre el funcionamiento de repositorios GitHub, en el boletín Flujo de trabajo con GitHub se describe cómo crear una cuenta y gestionar un repositorio. En este boletín se utiliza una versión de GitHub de la escuela, que puede encontrarse en https://github.eii.us.es