ANTEPROYECTO DAW CURSO 2022

OnlinePlusLocal

Autor:

Cristo Manuel Rodríguez Rodríguez

TABLA DE CONTENIDO

1. Datos del alumno y título del proyecto

Ciclo: Desarrollo de Aplicaciones Web

Curso: Segundo

Nombre: Cristo Manuel

Apellidos: Rodríguez Rodríguez

Tutor académico: Jose Antonio Lara Sánchez

Título: OnlinePlusLocal (Artículos online de tiendas a tu alcance).

2. Introducción

Actualmente estamos en una era digital, en la que podemos ver como internet ya llega a casi todos sitios, estamos totalmente conectados, incluso para algunos de una forma intensiva, llegando a hacer todo por el teléfono u ordenador (compras, trabajo, conocer amistades, pago de deudas, etc..).

Esto puede ser un problema para algunas empresas pequeñas que empiezan, ya que las grandes de internet o las grandes superficies terminan cogiendo la atención del consumidor y no por el precio o productos atractivos, sino por la publicidad y el pensamiento del consumidor de 'allí hay de todo', por lo que ser visible es un problema grave, ya que sin visibilidad no hay clientes que conozcan lo que ofrecemos y como consecuencia no habrá ventas, lo que llevará al cierre del negocio.

Otros de los puntos malos a tener en cuenta es que muchos productos que antes conseguíamos en algunas superficies, como por ejemplo en el supermercado la marca Ariel u otras marcas de alimentos o ropa, han sido cambiadas en las grandes cadenas por las suyas propias, podemos ver los ejemplos de Hacendado (Mercadona) o Deluxe (Lidl), lo que afecta a los consumidores leales a alguna marca, algunos sin saber dónde comprar la marca que quieren y a buen precio.

3. Finalidad

Saber dónde comprar algo específico cerca de casa y no tener que realizar pedidos por internet que ocasionarían gastos extras por el envío y la larga espera o ir a muchas tiendas para localizarlo. Una plataforma donde los comercios cercanos publiquen sus artículos juntos y poder ir a verlos o contactar con el comercio para adquirirlo, sin hacer un gasto sin haberlo visto físicamente como sucede con la compra online.

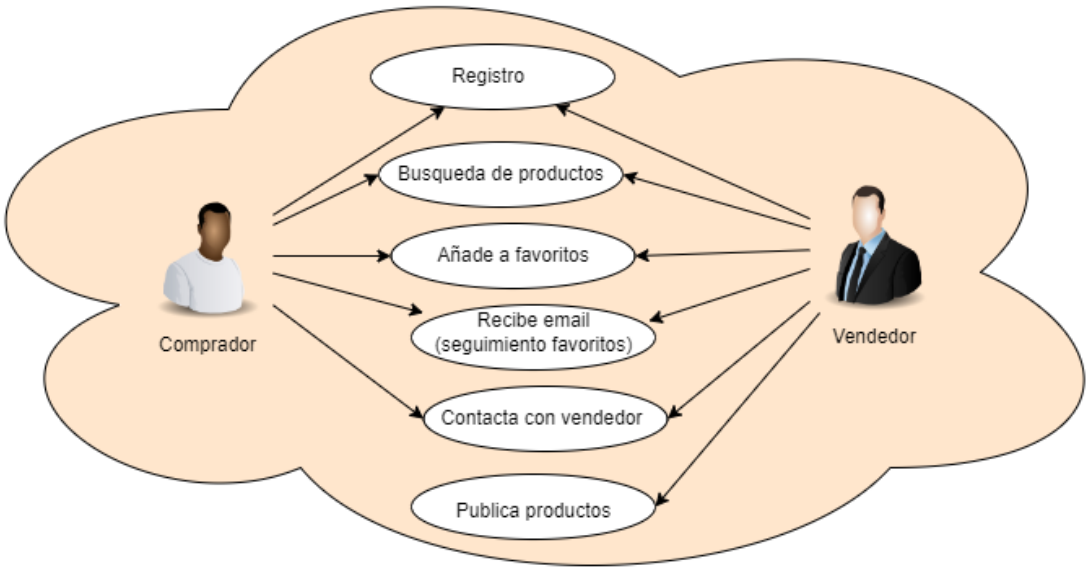
Como consumidor me gustaría tener una plataforma web que nos brinde la posibilidad de encontrar mayor cantidad de productos y cerca de casa, los cuales podemos ir a probar, como por ejemplo calzado o un vestido antes de adquirirlos, puede ser de gran ayuda, ya que nos evita el realizar grandes paseos a tiendas sin éxito o el comprar a 'ciegas' guiándonos por una simple imagen en internet, por lo que ha parecido una buena idea, siendo también algo poco común, ya que existen ya muchas plataformas de venta online, pero muchas veces compramos sin saber si exactamente se ajusta a nuestras necesidades lo que estamos comprando.

5. Objetivos

Nuestro objetivo será la realización de una plataforma de exposición de productos de los comercios registrados, en ella habrá dos tipos de usuarios:

- **Comprador**: usuarios que podrán realizar búsquedas de productos ya inscritos por los usuarios vendedores, teniendo el contacto y dirección para realizar el pedido o para ir a verlo en la tienda y pagarlo allí. Dispondrán también de un apartado de favoritos de los cuales se podrá recibir avisos cuando cambie el precio.

- **Vendedor**: Dispondrá de todas las funciones de los compradores (pudiendo ser ellos mismos compradores de otros vendedores) y además de las funciones de subir productos a la plataforma. Por temas de privacidad no contactarán con los usuarios compradores si ellos no contactan con ellos antes, sobre todo para evitar el acoso de publicidad que ya tenemos en la web y con el teléfono hoy en día.



Se ha realizado un Mockup en la plataforma de Figma, para visualizar de forma más detallada como será el proyecto (recibirá cambios según vaya avanzando el proyecto). Hay que aclarar que es para dar una idea general, por lo que está incompleto y no se han realizado los diseños necesarios del modo responsive (diseños de los diferentes dispositivos como Desktop, Tablet y mobile) para cumplir los plazos, pero ayudará a ver cómo va a quedar y las necesidades faltantes que se deseen agregar antes del comienzo de la fase de codificación:

Mockup

Añado también el prototipo de éste, que será una imitación de como será la navegación del usuario en la plataforma, viendo los pasos a seguir desde que se registra hasta que accede a la plataforma de búsqueda y perfil:

Prototipo

Estos enlaces serán accesibles en todo momento desde la bibliografía del proyecto para su fácil localización y acceso. Se irán modificando según se vea necesario.

Con todo esto, el resto del proyecto lo dividiremos en tres fases:

- **Fase 1**: Creación de la parte del *Frontend*. (Páginas web con navegación, pero sin funcionalidad real, obteniendo los datos de ejemplo de un JSON local)
- **Fase 2**: Creación de la parte del servidor, *Backend*, (Base de datos donde se almacenarán los datos y código del servidor, se realizarán pruebas con Postman de la API)
- **Fase 3**: Se conectará *Frontend* y *Backend*, se realizarán arreglos de *bugs*, se añadirán cosas necesarias que no hayan sido vistas antes del comienzo del proyecto, actualización, realización de pruebas y documentación.

6. Medios utilizados

Tecnologías a utilizar:	
Recurso	**Uso**
HTML	Creación de la estructura de las páginas web
CSS	Estilos y diseño de los documentos
Javascript	Funcionalidad del Frontend
PHP	Funcionalidad del Backend y conexión con la base de datos
SQL	Consultas a la base de datos
Git	Sistema de control de versiones
GitHub	Servicio de almacenamiento utilizado para el proyecto
Draw.io	Creación de diagramas
Figma	Creación de Mockup
Ubuntu	Sistema donde se desplegará la aplicación web
Visual Studio Code	Editor donde se creará el proyecto
Apache	Servidor utilizado para las pruebas y despliegue
Photoshop	Edición de imágenes
MariaDB	Base de datos de la aplicación
Postman	Pruebas de conexión con el servidor
Composer	Manejador de paquetes de PHP
WSL	Subsistema de linux en Windows (se desarrollará en Windows)
phpMyAdmin	Administración de la base de datos
Trello	Administración de proyectos
Plesk	Automatización de centro datos y alojamiento web

Además de las comentadas arriba, si se utilizara alguna más, se comentará a lo largo del proyecto. Estas son las que de momento tenemos pensadas utilizar.

También se usarán las siguientes páginas, las cuales nos ofrecerán ayuda, recursos y herramientas útiles para el desarrollo:

- **Tinypng**: Compresor de imágenes PNG.
- **Caniuse**: Ver compatibilidad de los navegadores.
- **Freepik**: Descarga de imágenes.
- **Google Font**: Iconos de la aplicación.

Añadir por último el enlace al repositorio de Github donde se alojará el código:

Repositorio OnlinePlusLocal

Y el enlace al servidor VPS propio donde se alojará la aplicación web para su visualización y uso:

onlinepluslocal.cmrr.es/

Ya creados para poder empezar el proyecto una vez lo den por válido. Se añadirá un apartado con todos los enlaces del proyecto en la próxima documentación, para un acceso rápido.

7. Planificación

Investigación

Tarea	Tiempo
Páginas similares, competencia o con aporte de ideas	1 hora
Obtención de recursos (imágenes, iconos, …)	1,5 horas
Documentación inicial del proyecto	1,5 horas
Total, investigación:	**4 horas**

Diseño

Tarea	Tiempo
Creación del diseño de la aplicación	5 horas
Edición de imágenes	1 hora
Total, diseño:	**6 horas**

Implementación

Tarea	Tiempo
Desarrollo de la parte del Frontend (Documentos web)	10 horas
Documentación del desarrollo del Frontend	1,5 horas
Desarrollo de la parte del Backend (código PHP)	9 horas
Creación de la base de datos	1 hora
Documentación del desarrollo del Backend y base de datos	1,5 horas
Total, Implementación:	**23 horas**

Pruebas

Tarea	Tiempo
Unitarias	1 hora
Integración	1 horas
Funcionales	1 horas
Corrección de errores	1 hora
Documentación final	1,5 horas
Total, pruebas:	**5,5 horas**

Tiempo total

Total, finalización del proyecto:	**38,5 horas**

8. Preparación del entorno de trabajo

Una vez visto los requisitos de nuestra aplicación, el primer paso es la preparación de nuestro sitio de trabajo. Lo primero será la instalación de las herramientas necesarias para el proyecto.

Instalaremos WAMP para disponer de una base de datos (*MariaDB*) y el servidor *Apache* con la que realizar pruebas en Windows:

https://www.wampserver.com/en/

A continuación, el editor de código que utilizaremos, *Visual Studio Code*, por el cual hemos optado por sus prestaciones para el desarrollo web. Lo obtendremos del siguiente enlace:

https://code.visualstudio.com/

La instalación es muy sencilla, al finalizar dedicaremos un poco de tiempo para su configuración e instalación de extensiones, de las cuales destacamos:

5. **PHP Intelephense**: Para el manejo del lenguaje PHP
6. **PHP Debug**: Utilizada para la depuración del lenguaje PHP

Para poder depurar con PHP no solo nos hará falta la extensión *PHP Debug*, además deberemos realizar unos pasos de configuración, los pasos a seguir los obtendremos de la siguiente página y de la extensión de *Visual Studio Code, PHP Debug*:

https://xdebug.org/wizard y PHP Debug

Para la edición e imágenes utilizaremos *Photoshop*, al ser una de las mejores herramientas que disponemos para la edición de imágenes, la cual utilizaremos para dar fondos transparentes y modificarlas dándoles un aspecto más atractivo:

https://www.adobe.com/es/products/photoshop.html

El control de versiones lo gestionaremos mediante *Git*, el cual obtendremos del siguiente enlace para Windows, disponiendo en el sitio de una buenísima documentación para su manejo:

https://git-scm.com/download/win

GitHub será el que utilizaremos como repositorio para el almacenamiento de nuestro proyecto en la nube:

https://github.com/

Los pasos para la creación del repositorio son muy sencillos, lo crearemos y ya solo nos faltará la preparación de nuestro servidor para el despliegue.

Para el despliegue de nuestra aplicación web utilizaremos un servidor propio, el cual gestionaremos mediante el panel Plesk para agilizar. En él crearemos un subdominio para nuestra aplicación e instalaremos un certificado SSL/TLS *Lets Encrypt*, para dar más confianza y seguridad a los usuarios cifrando la conexión. Aprovecharemos para la creación de la cuenta de correo con la que enviaremos mensajes a los usuarios de nuestro sitio web, como activación de cuenta o artículos favoritos y haremos las modificaciones de los registros DNS del subdominio para su correcta configuración.

Por último, añadiremos los requisitos del *Product Backlog* a la herramienta *Trello* con la que iremos haciendo el seguimiento y fechas plazo del proyecto:

En el siguiente enlace se podrá ir viendo el seguimiento de éste: Trello: OnlinePlusLocal

9. Product Backlog

ID	Requisito	Prioridad	Sprint	Estado
1	Diseño de página familiar para el usuario que facilité la navegación	Obligatorio	1	Terminado
2	Estilo *responsive* para que se pueda visualizar correctamente en diferentes dispositivos	Deseable	1	Terminado
3	Compatibilidad con los navegadores más utilizados: Chrome, Edge, Firefox y Opera	Obligatorio	1	Terminado
4	Estructura de la página ligera, con una carga rápida en el dispositivo, para compensar la carga de artículos	Deseable	1	Terminado
5	Confirmación de registro mediante correo electrónico	Obligatorio	2	En progreso
6	Las contraseñas deben estar cifradas en la base de datos	Obligatorio	2	En progreso
7	Se debe limitar el tamaño de las imágenes de los artículos, dando al usuario una opción para comprimir y editar dicha imagen	Deseable	2	En progreso
8	Si el usuario ingresa una ruta inexistente debería ser redireccionado a la página de inicio	Obligatorio	2	En progreso
7	La consulta a la base de datos debe ser rápida, dividiendo la cantidad de información que el usuario solicita para cargarla mediante un botón de *cargar más*	Deseable	3	Por empezar
8	Al abrir la información de un artículo el usuario debería poder acceder a diferentes formas de contacto con la tienda	Obligatorio	3	Por empezar
9	El usuario puede seguir una tienda para ser avisado por correo electrónico de nuevos artículos de ésta	Deseable	3	Por empezar
10	Debería no tener errores de carga ni de navegación, siendo revisado mediante testeo	Deseable	3	Por empezar

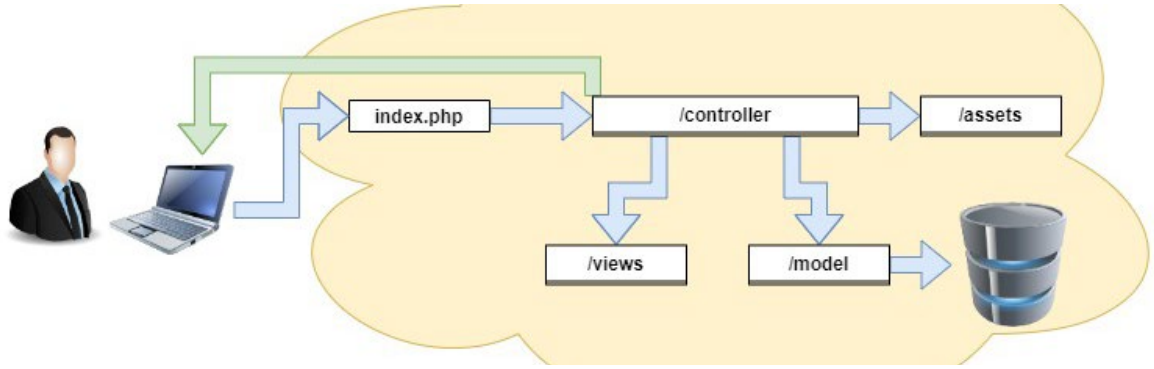
Product Backlog. Indicará el estado del proyecto, irá actualizándose con éste

9.1. Sprint 1 - Diseño

En el primer *Sprint* marcaremos las tareas a realizar y comenzaremos con el proyecto:

ID	Requisito	Prioridad	Sprint	Estado
1	Diseño de página familiar para el usuario, para facilitarle la navegación	Obligatorio	1	En progreso
2	Estilo *responsive* para que se pueda visualizar correctamente en diferentes dispositivos	Deseable	1	En progreso
3	Compatibilidad con los navegadores más utilizados: Chrome, Edge, Firefox y Opera	Obligatorio	1	En progreso
4	Estructura de la página ligera, con una carga rápida en el dispositivo, para compensar la carga de artículos	Deseable	1	En progreso

Lo primero que haremos será crear la estructura de carpetas y archivos que necesitaremos para la realización del proyecto, para saber que necesitaremos, realizaremos un esquema del funcionamiento que queremos lograr:



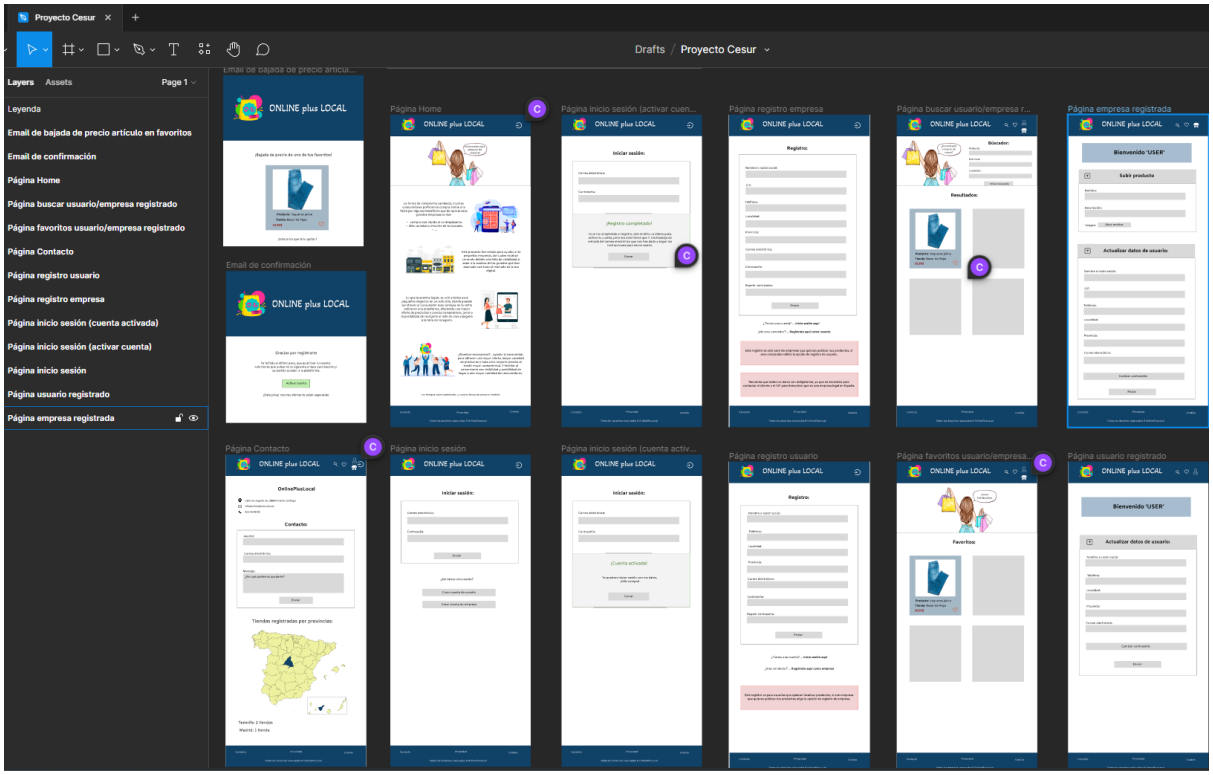
En la imagen podemos ver los siguientes pasos:

El usuario solicita un recurso a *index.php*, éste se encargará de la seguridad, como la inyección de código malicioso y otras funciones que en un futuro viésemos necesarias. A continuación, pasará al controlador la petición, el cual se encargará de:

- Solicitar al modelo los datos requeridos de la base de datos
- Obtener los recursos de la carpeta *assets* que sean necesarios
- Conseguir la vista requerida para completar la solicitud, agregando el contenido y devolviendo todo listo al usuario que generó la solicitud.

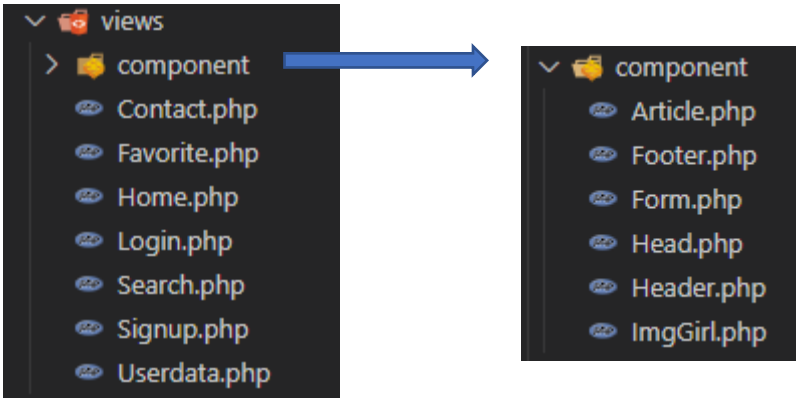
Una vez entendido el funcionamiento, crearemos la estructura de archivos y carpetas:

A Continuación, crearemos el Mockup del proyecto para poder tener una idea clara de lo que se va a crear y como va a estar repartido el contenido cumpliendo con los requisitos del sprint, por lo que acudiremos a la herramienta *Figma* y procederemos a realizarlo:



Se puede acceder al Mockup desde el siguiente enlace: [OnlinePlusLocal mockup]

Listo todo lo anterior, ya podremos empezar a crear las vistas de la aplicación web. En la carpeta *views* crearemos las vistas de las páginas y en la carpeta *component* las vistas de los componentes que se reutilizen en estas vistas, para la reutilización de código



En la entrada principal de nuestra aplicación, *index.php*, obtendremos el parámetro de la vista solicitada mediante la superglobal GET y en caso de el usuario de no haber escrito la variable el usuario, que redirija a la página home, para facilitar al usuario la escritura de la ruta en el navegador. Después imprimiremos la vista mediante la clase *ViewController*:

```php
index.php > ...
<?php
require_once './controller/ViewController.php';

$page = (isset($_GET['page']))
    ? strip_tags(trim($_GET['page']))
    : 'home';

$vc = new ViewController();
$vc->printView($page);
```

Puede ser un poco incómodo el tener que escribir parámetros en la ruta a la página, esto lo solucionaremos en la última parte del proyecto, creando rutas amigables mediante un archivo *.htaccess*.

Durante la creación del objeto *ViewController* no hemos de hacer ninguna acción extra, por lo que el constructor lo hemos dejado vacío:

```php
8 references | 0 overrides
public function __construct() {}
```

Posteriormente se hace uso del método *printView* pasándole por parámetro la vista solicitada por el usuario mediante la ruta del navegador:

```php
public function printView(string $page)
{
    $user = 'seller';

    match ($page) {
        'perfil-vendedor' => $this->viewUserdataSeller(),
        'perfil-cliente' => $this->viewUserdataBuyer(),
        'favoritos' => $this->viewFavorite($user),
        'buscador' => $this->viewSearch($user),
        'registro-cliente' => $this->viewSignupBuyer(),
        'registro-vendedor' => $this->viewSignupSeller(),
        'contacto' => $this->viewContact($user),
        'login', 'out-session' => $this->viewLogin($user),
        default =>
        $this->viewHome(),
    };
}
```

En este método vemos que se obtiene el parámetro y según el resultado de éste, nos imprimirá la página requerida. Es importante destacar que se ha usado *match* y no *switch* de PHP, por lo que se requerirá tener en el servidor como mínimo la versión 8 de PHP:

https://www.php.net/manual/es/control-structures.match.php

Cada uno de los métodos del *match*, están encargados de cargar la vista correspondiente y de cargar los recursos necesarios según cada una de ellas, veamos una de ellas, *viewSearch*:

```php
/**
 * Vista de la página de búsqueda
 *
 * @param string $user El tipo de cliente que inició sesión. Disponibles: buyer o seller
 */
1 reference
private function viewSearch($user)
{
    // TODO: datos de ejemplo para ver el resultado visual
    $data_tmp = file_get_contents('./test/data-product/articles_testing.json');
    $products = json_decode($data_tmp, true);

    $header =   new Header(['favorite', $user, 'logout']);
    $viewSearch = new Search(self::URL_SERVER, $products);
    $this->setPage($header->getCode(), $viewSearch->getCode());
}
```

Vemos que el método recibe de parámetro el tipo de usuario que inició sesión, lo que hacemos es que el icono de acceso al perfil en el *header* del sitio cambie según el tipo de usuario, aunque en ciertas páginas no será visible, dejaremos que se acceda al tocar los botones de acceso, como si ya estuviera registrado un vendedor, menos en la página de home, lo dejaremos con acceso al *login*.

Home → 

Resto de páginas → 

También vemos en el código que cargamos unos datos de ejemplo desde un archivo JSON, para ver el resultado de los artículos y si la estética cumple los requisitos.

Después de la carga de los archivos de ejemplo se carga el componente *Header* del sitio. Al ser construido el objeto, se crea el código del componente:

```php
public function __construct(array $linksHeader)
{

    $this->code = '<header>
                    <div class="title-box">
                        <a href="/">
                            <img class="logo" src=' . self::LOGO . ' alt="Logo y enlace a la página principal" />
                        </a>
                        <h1>ONLINE plus LOCAL</h1>
                    </div>
                    ' . $this->getIconsHeader($linksHeader) . '
                    </header>';
}
```

Aunque hemos separado en un método el código de los enlaces del *Header*, para dejar el código más fácil de leer y mantener:

```php
private function getIconsHeader(array $iconsSelected)
{
    $codeLinks = '';
    foreach ($iconsSelected as $selected) {

        $link = self::ICON_HEADER[$selected];

        $codeLinks .= '<a class="box-button" href=' . $link['href'] . ' target=_self>
                        <span class="material-symbols-outlined ' . $link['color'] . '" title=' . $link['title'] . '>
                            ' . $link['text'] . '
                        </span>
                        <p class="white hidden">
                            ' . $link['title'] . '
                        </p>
                    </a>';
    }

    return '<div class="icon-box">' . $codeLinks . '</div>';
}
```

Por lo que, si en un futuro se quisiera modificar los enlaces del sitio, solo sería necesario acudir a ese método.

Los iconos de enlace junto a su configuración hemos decidido sacarlos del código en un array, del cual se obtendrá solo los que hayan sido solicitados como parámetro:

```
private const ICON_HEADER =
[
'search' => ['href' => './?page=buscador', 'color' => 'white', 'title' => 'Buscador', 'text' => 'search'],
'favorite' => ['href' => './?page=favoritos', 'color' => 'red', 'title' => 'Favoritos', 'text' => 'favorite'],
'login' => ['href' => './?page=login', 'color' => 'white', 'title' => 'Login', 'text' => 'login'],
'logout' => ['href' => './', 'color' => 'white', 'title' => 'Logout', 'text' => 'logout'],
'seller' => ['href' => './?page=perfil-vendedor', 'color' => 'white', 'title' => 'Perfil', 'text' => 'store'],
'buyer' => ['href' => './?page=perfil-cliente', 'color' => 'white', 'title' => 'Perfil', 'text' => 'person'],
];
```

La ventaja de eso es que cuando queremos agregar un nuevo enlace solo será necesario agregarlo en el array y pasando por parámetro el nombre lo agregaremos, sin tocar código que pueda estropear el sitio.

Y para finalizar la clase *Header*, comentar el método *getCode*, que devuelve el código del componente ya creado y configurado:

```
/**
 * Código HTML del componente Header
 *
 * @return string Devuelve el código HTML con el Header del documento
 */
1 reference | 0 overrides
public function getCode(): string
{
    return $this->code;
}
}
```

Este método tiene la misma función en los diferentes componentes, por lo que no será necesario comentarlo más adelante.

En el método de *ViewController*, donde creamos la vista de la página búsqueda, vimos que después del *Header* se creaba la vista del contenido de esa vista:

```
$viewSearch = new Search(self::URL_SERVER, $products);
```

El mecanismo es parecido al *Header*, con algunas pequeñas diferencias.

Le enviamos dos parámetros, *URL_SERVER*, que es una constante de la clase que indica la ruta al archivo principal de entrada al servidor, hecho así por si en un futuro debemos modificarla, solo hacerla de un sitio:

```
private const URL_SERVER = './server.php';
```

Y el parámetro *products* que en este caso son los artículos de ejemplo cargados del JSON, pero en el siguiente Sprint cargaremos los datos de una base de datos.

La creación de la vista *viewSearch*, contenido principal de la página, es similar a lo visto con alguna pequeña diferencia:

```php
public function __construct(string $urlServer, array $articles = [])
{
    $imgGirl = new ImgGirl('¡Nos vamos de compras!');

    $this->code = '<section class="search">
                    <div class="girl">
                        '. $imgGirl->getCode() .'
                    </div>
                    <div class="form-search">
                        '. self::getForm($urlServer) .'
                    </div>
                </section>';

    $this->code .= '<section class="result-search">
                    <h4 class="global-title-plane">Resultados:</h4>
                    <div class="articles">';

    foreach ($articles as $k => $v) {
        $this->code .= Article::getPreview(
            $v['id'],
            $v['img'],
            $v['name'],
            $v['shop'],
            $v['description'],
            $v['price'],
            false
        );
    }

    $this->code .= '</div>
                </section>';
}
```

Vemos que se genera el código de la misma forma y en una variable de clase con el mismo nombre, para ayudar al programador que lea el código a entender con más facilidad la funcionalidad.

La diferencia es que aquí se hace uso de vistas componentes, la imagen de una chica con mensaje personalizado por parámetro (*ImgGirl*), componente *Article*, que serán los resultados de artículos en la búsqueda y por último el formulario de búsqueda, el cual lo hemos puesto en un método aparte.

Si viajamos a ese método de la misma clase veremos que se hace uso de la clase *Form*:

```php
/**
 * Crea el formulario para búsqueda de artículos
 *
 * @param string $urlServer Ruta al archivo del servidor al que se le enviará los formularios
 * @return string Código HTML del formulario para ser insertado al documento
 */
1 reference
private function getForm(string $urlServer): string
{

    $form = new Form(
        [
            'name' => 'search',
            'legend' => 'Buscador',
            'url' => $urlServer,
            'method' => 'GET',
            'fieldset' => false
        ],
        self::INPUTS_SEARCH,
        [
            'text' => 'Iniciar búsqueda',
            'icon' => ''
        ],
        ''
    );

    return $form->getCode();
}
```

El motivo de separarlos en clases es por la reutilización, los formularios se usan en varios sitios como en página de búsqueda, login, perfil de usuario, contacto…; facilitándonos el mantenimiento al tener un formulario con el mismo estilo y si necesitáramos hacer un cambio de imagen, solo sería necesario hacerlo en un sitio.

Lo único a tener en cuenta, las diferencias como la ruta de envío del formulario y los campos de éste, que serán enviados como parámetros al instanciar la clase en forma de array:

```php
private const INPUTS_SEARCH = [
    ["id" => "name", "label" => " Producto:", "type" => "text", "component" => "input"],
    ["id" => "province", "label" => " Provincia:", "type" => "text", "component" => "input"],
    ["id" => "city", "label" => " Localidad:", "type" => "text", "component" => "input"]
];
```

Y los datos estéticos y nombre que identifique al formulario al llegar al servidor que irán en el primer parámetro también como array:

```php
'name' => 'search',
'legend' => 'Buscador',
'url' => $urlServer,
'method' => 'GET',
'fieldset' => false
```

La última línea en el método *viewSearch* envía todas las vistas para crear la página e imprimirla en el navegador, vemos que envía el código del *Header* y del *main*:

```php
/**
 * Vista de la página de búsqueda
 *
 * @param string $user El tipo de cliente que inició sesión. Disponibles: buyer o seller
 */
1 reference
private function viewSearch($user)
{
    // TODO: datos de ejemplo para ver el resultado visual
    $data_tmp = file_get_contents('./test/data-product/articles_testing.json');
    $products = json_decode($data_tmp, true);

    $header = new Header(['favorite', $user, 'logout']);
    $viewSearch = new Search(self::URL_SERVER, $products);
    $this->setPage($header->getCode(), $viewSearch->getCode());
}
```

Si vamos a ese método vemos que monta las vistas en la estructura de un documento HTML y lo envía con el comando *echo*:

```php
/**
 * Crea e imprime en el navegador la página agregando las vistas pasadas por parámetro al documento
 *
 * @param string $header Cabecera de la página (Header)
 * @param string $content Contenido de la página (Main)
 */
9 references
private function setPage(string $header, string $content)
{
    $head = new Head();
    $footer = new Footer();

    $doc = '<!DOCTYPE html>
            <html lang="es">
                ' . $head->getCode() . '
                <body>
                    ' . $header . '
                    <main class="content-body">
                        ' . $content . '
                    </main>
                    ' . $footer->getCode() . '
                </body>
            </html>';

    echo $doc;
}
```

La cabecera del documento (*Head*) y el pie de página (*Footer*) hemos decidido ponerlo dentro de este último método al ser igual en todas las páginas, evitando así líneas de código inecesaria en todas los métodos de creación de vistas.

Para los estilos, hemos agregado a la cabecera del documento una hoja de estilos principal para todo el sitio, pero apuntando a cada elemento con la sección padre donde está, para evitar modificar etiquetas de otras secciones, un ejemplo para que se entienda:

```css
header .logo {
  cursor: pointer;
}

header h1 {
  color: #fff;
  font-family: "Carter One", cursive;
  font-size: 4vw;
  text-shadow: 1px 1px 2px #104167;
}
```

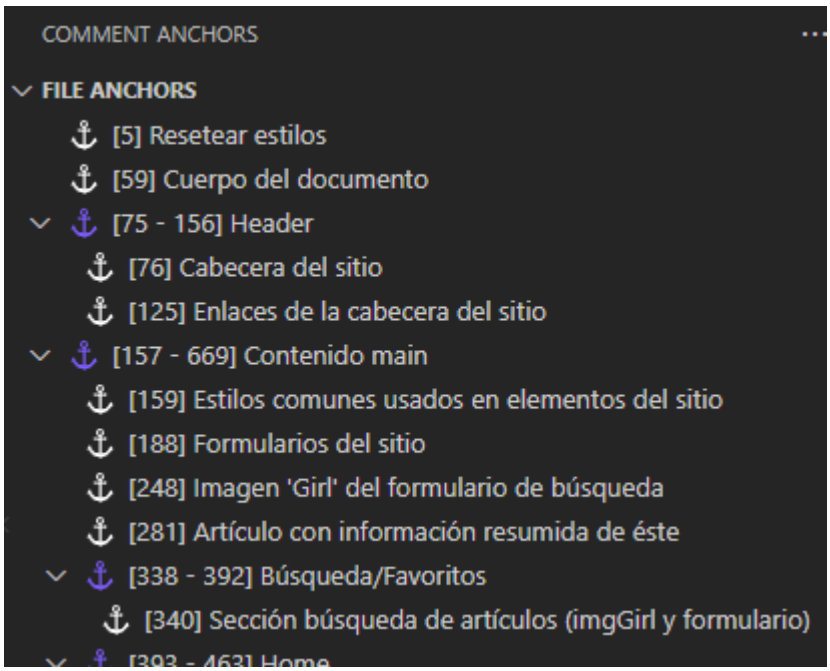
```css
.form legend {
  margin: 0 auto;
}

.form label {
  display: flex;
  flex-flow: column nowrap;
  max-width: 100%;
}
```

Como se puede observar, se agrega el selector de etiqueta *header* antes del selector a modificar el estilo, como con el selector de clase *form*, para evitar así modificar estilos de otras secciones por herencia y tenerlo más ordenado y fácil de entender que está modificando.

Se ha agregado también comentarios para separar las secciones de estilos, mediante una extensión, para localizar de forma rápida los estilos que necesitemos cambiar:

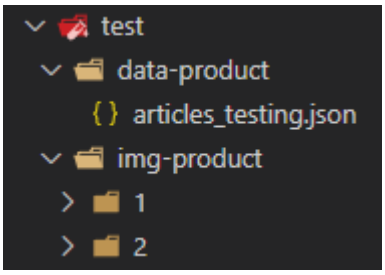
```css
/* !SECTION: Header */
/* SECTION: Contenido main */

/* ANCHOR: Estilos comunes usados en elementos del sitio */
```

Gracias a esos comentarios podemos mediante la extensión *Comment Anchors* localizar rápidamente esa sección de código:

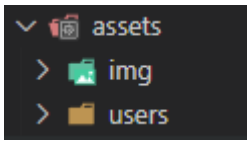


Los estilos como vemos en la imagen anterior, reseteamos los estilos del navegador y posteriormente vamos modificando los estilos de arriba a abajo en el documento, modificando primero el general del *body*.

Por último, en este Sprint, hemos creado las carpetas de test, para poder visualizar contenido de artículos en la página de búsqueda y favoritos, almacenando en cada carpeta por numeró de id las imágenes de cada tienda:



Algo que haremos en el Sprint 2 en la carpeta *assets*, donde almacenaremos las imágenes en la carpeta *users* por su número de id y las imagens del sitio web en la carpeta *img*:



Por lo que el primer Sprint lo damos por finalizado, habiendo creado los documentos y estilos de las páginas web, ahora en el Sprint 2 pasaremos a la parte del servidor, donde crearemos base de datos, envío de email y funcionamiento del sitio mediante scripts.

5. Bibliografía

MDN Web Docs. developer.mozilla.org (1998-2022). Recuperado de: Enlace

Can I use. caniuse.com (2022). Recuperado de: Enlace

Tipos de registros DNS. ionos.es (2022). Recuperado de: Enlace

Crear dirección de correo. ionos.es (2022). Recuperado de: Enlace

WAMP. wampserver.com (2022). Recuperado de: Enlace

Visual Studio Code. code.visualstudio.com (2022). Recuperado de: Enlace

XDebug PHP. xdebug.org (2002-2022). Recuperado de: Enlace

PHP Debug extensión VSCODE.es (2022). marketplace.visualstudio.com. Recuperado de: Enlace

Photoshop. adobe.com (2022). Recuperado de: Enlace

Git. git-scm.com (2022). Recuperado de: Enlace

GitHub. github.com (2022). Recuperado de: Enlace

Plesk. plesk.com (2022). Recuperado de: Enlace

Let's Encrypt. letsencrypt.org (2022). Recuperado de: Enlace

Figma. figma.com (2022). Recuperado de: Enlace

MariaDB. mariadb.com (2022). Recuperado de: Enlace

Draw.io. drawio-app.com (2022). Recuperado de: Enlace

Apache. httpd.apache.org (1997-2022). Recuperado de: Enlace

Composer. getcomposer.org (2022). Recuperado de: Enlace

Postman. postman.com (2022). Recuperado de: Enlace

WSL. microsoft.com (2022). Recuperado de: Enlace

phpMyAdmin. phpmyadmin.net.com (2003-2022). Recuperado de: Enlace

Ubuntu Docs. ubuntu.com (2022). Recuperado de: Enlace

Trello guide. trello.com (2021). Recuperado de Enlace