RASTREADOR DE VIVIENDAS ADAPTADAS

ANEXO III
Especificación de diseño

VNiVERSiDAD D SALAMANCA

Trabajo de Fin de Grado
INGENIERÍA INFORMÁTICA 2023

Autor
Adrián Torre Salinero

Tutores
Sara Rodríguez González
Guillermo Hernández González

Tabla de contenido

Tabla de imágenes

1. Introducción

En el siguiente anexo se describe el proceso de diseño realizado para el desarrollo del proyecto. En él se describe la estructura del proyecto. Además, se detallan los algoritmos utilizados para implementar la funcionalidad deseada. Se explican de una forma teórica el diseño de interfaces, indicando los elementos que deberán estar presentes en todas y cada una de las interfaces del sistema.

Se va a realizar una descripción concreta de los datos del sistema, tanto del formato de almacenamiento que es a través de un servidor de base de datos de MariaDB, como los detalles de cada una de las tablas que aparecen en la base de datos.

Se va a realizar también una descripción del proceso de diseño de las interfaces, con bocetos que intenten reflejar las interfaces que finalmente se implementarían en el sistema.

El diseño de datos y el diseño procedimental, explicando la estructura interna de los datos y explicando los algoritmos implementados, se detallan a lo largo de los apartados 0 y 0

Finalmente, y reuniendo toda la información presentada a lo largo del anexo, se presenta el diagrama de clases de diseño final.

2. Estructura del proyecto

En el proyecto se ha decidido utilizar la estructura MVC. La estructura Modelo-Vista-Controlador (Model-View-Controller, MVC) es un patrón de arquitectura de software que permite separar los datos y lógica del negocio de su representación y el módulo de gestión de eventos y comunicaciones. En el caso de este proyecto en particular, la vista se corresponde con las diferentes interfaces que se utilizan, representadas por los diferentes archivos html, el controlador está representado por el archivo flask, que proporciona los *endpoints* de acceso a la información, y el modelo es la base de datos del sistema y el fichero para el acceso a la misma, AccesoBaseDatos. Además, se pueden incluir en el modelo ciertas tareas que se realizan de forma independiente y que no tienen una representación ni necesitan de una lógica especial. Estos scripts se comunicarán también con el script de acceso a la base de datos, pero de una forma directa.

2.1. View - Vista

Como se ha dicho antes, la vista se compone de todas las interfaces de las que dispone la aplicación para comunicarse con el usuario. Servirán como representación de los datos que hay en el modelo y se comunicarán con el controlador a través de los *endpoints* o rutas que éste ponga a disposición. Las diferentes interfaces sólo se comunican con el controlador, el cual le suministra los datos que hayan pedido.

Una vez explicado esto, cabe destacar que para simplificar los diagramas de la estructura total del proyecto, se va desarrollar el paquete por funcionalidades. voy a dividir las diferentes interfaces en 3 grupos en función de quién tiene acceso a las mismas.

2.1.1. Interfaces generales

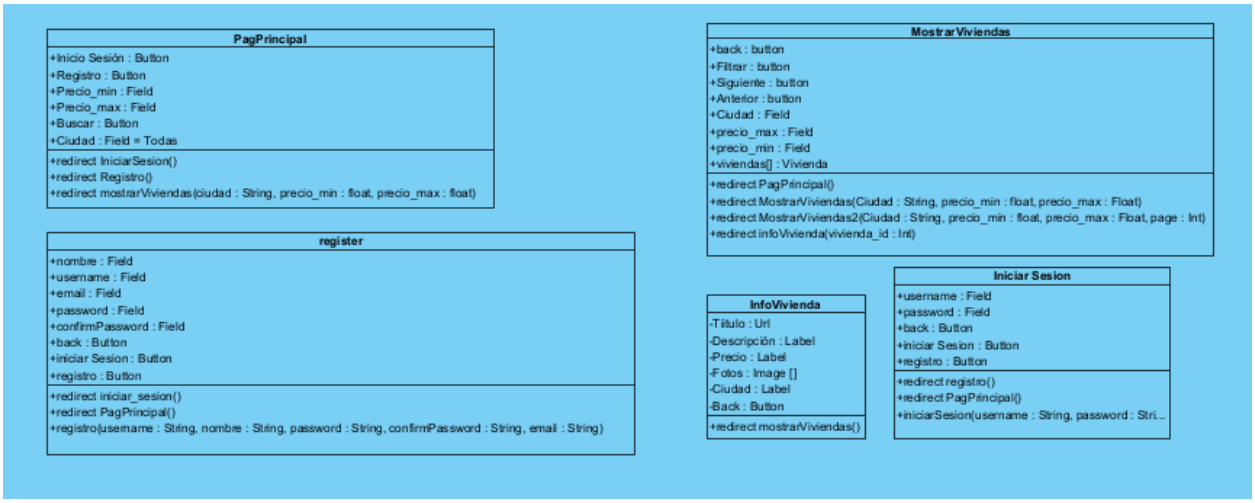

Imagen 1 - Interfaces generales

A estas interfaces tienen acceso todos los usuarios, ya estén registrados o no. En estas interfaces no hay ninguna función propia, por lo que las funciones que aparece que tienen definidas son simplemente conexiones con los respectivos endpoints del controlador. Se realizan mediante requests, en las que se especifican si es necesario los atributos que sean necesarios, como por ejemplo en el inicio de sesión la contraseña y el usuario. Se ha querido especificar en los atributos de las interfaces los campos a rellenar, botones, bloques de texto o colecciones de imágenes que son fundamentales para el funcionamiento de la aplicación, ya sea porque su contenido es una representación del modelo, por ser campos de un formulario para comunicarse con el controlador o porque son botones con funciones asociadas a los botones.

Todos los botones back, el cual es recurrente en la aplicación, tiene una función relacionada para volver a la página anterior, dependiendo de la interfaz en la que estemos se realizará la comunicación con un endpoint diferente.

2.1.2. Interfaces de usuario registrado

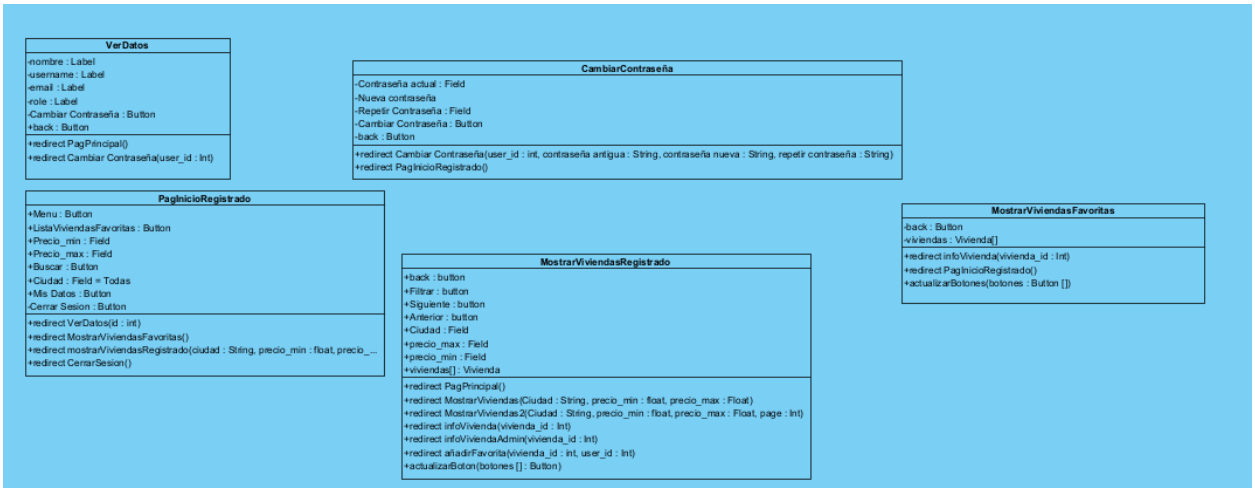

Imagen 2 - Interfaces usuario registrado

Estas interfaces son propias de un usuario registrado. En este caso la página principal sufre ligeras variaciones, eliminando los botones para el acceso al registro y al inicio de sesión, pero añadiendo un menú en el que aparecen dos opciones, ver datos y lista de viviendas favoritas. Mostrar viviendas también sufre ligeras modificaciones, apareciendo botones para añadirlas a favoritas. En esta interfaz se implementa una función utilizando js la cual actualiza en tiempo real el estado de los botones, aunque la actualización real de los datos se realiza en segundo plano. De igual forma pasa en mostrar viviendas favoritas, donde se actualizan los botones de favorito en tiempo real, mostrando en todo momento si la vivienda la tienes en la lista de favoritos o no en ambos casos. En cuanto a las interfaces de ver datos y cambio de contraseña no hay mucho que destacar, ya que simplemente tienen funciones de comunicación con el controlador.

2.1.3. Interfaces de administrador

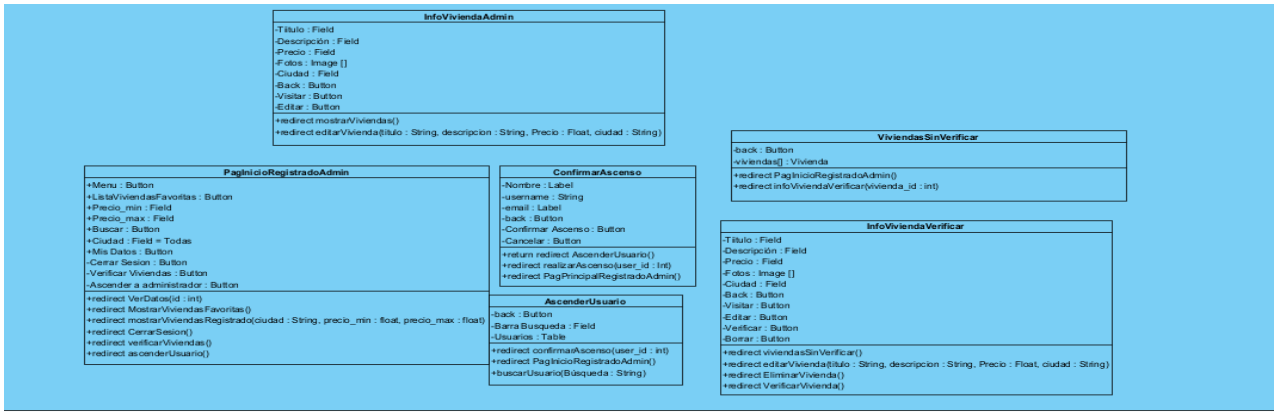

Imagen 3 - Interfaces de administrador

Estas últimas interfaces son exclusivas de los usuarios administradores. El control de acceso tanto de estas interfaces como las correspondientes a un usuario registrado (que como se explicó en el modelo del dominio, el actor administrador hereda las funcionalidades del actor usuario registrado), es realizado por el controlador.

La página principal de los administradores es igual que la de los usuarios registrados añadiendo dos opciones al menú de opciones: Verificar viviendas y ascender usuario a administrador. Además de esas dos opciones, la gran diferencia con los usuarios registrados es la capacidad de editar la información de las viviendas, siendo capaces de hacerlo tanto para las viviendas sin verificar como para las ya verificadas y mostradas normalmente. Las dos interfaces que permiten realizar esta funcionalidad son infoVivivendaAdmin, para las viviendas «normales» o ya verificadas, e infoViviendaVerificar para las que no han sido aún verificadas. Ambas interfaces comparten una estructura similar, teniendo un botón para redirigir al usuario a la lista de viviendas correspondiente, mostrar viviendas y viviendas sin verificar, unos campos editables correspondientes a la información de la vivienda y dos botones de editar, para cambiar los datos de la base de datos a los que tienen los campos en ese momento, y otro para visitar la página de la vivienda original. La interfaz infoViviendaVerificar además dispone de dos botones, para verificar la vivienda y añadirla así a la base de datos, y borrar para eliminarla del sistema, cuando la vivienda no es accesible, pero ha pasado el filtro.

La interfaz ViviendasSinVerificar es accesible a través del botón del menú correspondiente. En ella se muestra una lista de las viviendas que aún no han sido verificadas, funcionamiento idéntico a mostrarViviendas. AscenderUsuario es accesible a través del otro botón exclusivo de administrador. En esta interfaz, se muestra una tabla con todos los usuarios del sistema y una barra de búsqueda. La tabla mostrará en todo momento los usuarios que tengan un username que coincida total o parcialmente con la búsqueda introducida. Para ello, se define una función en js, buscarUsuario(busqueda). Al pulsar sobre cualquiera de los nombres de usuario se redirigirá al usuario a la interfaz ConfirmarAscenso. Esta interfaz muestra los datos del usuario seleccionado y tiene 3 botones, back, que redirigirá a la interfaz principal de administrador, otro para confirmar el ascenso, que redirigirá a ejecutarAscenso, y un tercer botón cancelar, que redirigirá a AscenderUsuario de nuevo. Cabe decir que ejecutarAscenso no es un endpoint como tal, la intenión es que utili-

ce el mismo endpoint que el utilizado para mostrar la interfaz, ConfirmarAscenso, pero cambiando los parámetros en la petición. Para mejorar la comprensión del diagrama se ha decidido utilizar ejecutarAscenso.

En resumen, la vista se compone de muchas interfaces para distintas funcionalidades, implementadas utilizando html, css y javaScript (js). Todas ellas se comunican con el controlador a través de los endpoints que este proporciona. Esto permite separar la representación, implementada por la vista y todas sus interfaces, de la lógica interna y las comunicaciones, implementadas en controlador y modelo. Existen algunas funciones de js que permiten una mejor experiencia de usuario, pero en ningún caso conocen la estructura interna de los datos y se limitan únicamente a mostrarlos.

2.1.4. Diseño de interfaces

Para realizar el diseño de las interfaces descritas, se han realizado varios bocetos de las mismas, para que el desarrollador pudiera hacerse una idea de como quedaría la interfaz finalmente y la línea de trabajo que debía seguir.

La primera de las interfaces bocetadas se corresponde con PagPrincipal, en la que aparecen en la parte central los campos para introducir los filtros y el botón de la lupa para realizar la búsqueda con los filtros introducidos. Además en la parte superior derecha se encuentran los botones para iniciar sesión y registro.

Imagen 4 - Boceto PagPrincipal

Otras de las interfaces bocetadas son las correspondientes a iniciarSesion y register. En ellas aparece en la parte central un formulario para realizar ambas acciones, con sus respectivos campos y un atajo para acceder a la opuesta. En iniciar sesión se piden dos datos, nombre de usuario y contraseña. En el registro se pide nombre, correo, nombre de usuario, contraseña y repetir contraseña. Una vez introducidos los datos correctamente, se debe pulsar el botón correspondiente a iniciar sesión o registro.

Imagen 5 - Boceto iniciarSesion y register

Se ha bocetado también la interfaz principal de usuario registrado, muy similar a PagPrincipal, pero sin los botones de registro e inicio de sesión y con un menú desplegable en la parte izquierda de la pantalla. Para desplegar el menú hay que pulsar el botón con 3 líneas.

Imagen 6 - Boceto PagIncioRegistrado

La interfaz mostrarViviendas también ha sido bocetada. En ella, deberá aparecer en la parte superior una barra de búsqueda que se mantenga fija donde se encuentren los filtros introducidos en la búsqueda, los cuales pueden ser editados, un botón para rehacer la búsqueda, de forma similar a la página principal, y un botón para volver a la página principal correspondiente, en función del usuario que realice la acción. Además, se muestran una colección de viviendas separadas por líneas, en las que aparece la información de las mismas. Por último, en la parte inferior de la interfaz se deben incluir botones para navegar entre las páginas de viviendas que casen con los filtros.

Imagen 7 - Boceto mostrarViviendas

Por último, se ha considerado importante bocetar infoViviendaAdmin. En esta interfaz aparecen los campos de información de la vivienda, los cuales pueden ser editados. Aparecen dos botones, uno para editar, que realiza los cambios que se le hayan aplicado al texto y otro para visitar la página original de la vivienda. Aparece un carrusel de fotos el cual es navegable a través de los botones correspondientes. Se puede volver a la página de búsqueda mediante la flecha que se encuentra en la parte superior izquierda.

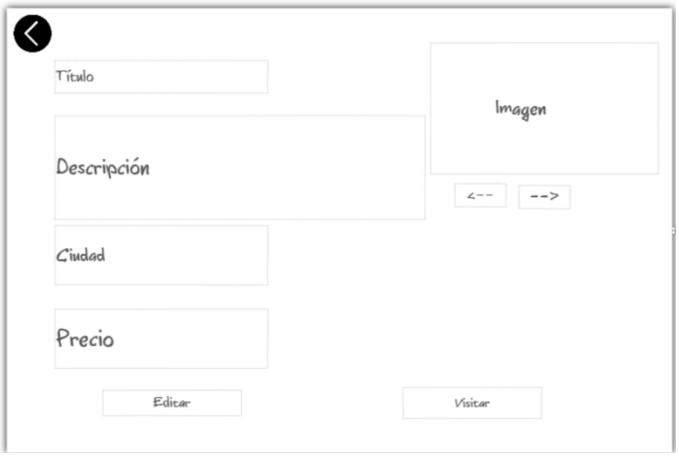

Imagen 8 - Boceto InfoVIviendaAdmin

2.2. Controller- Controlador

El controlador es el encargado de realizar las comunicaciones entre la vista y el modelo. Se encarga de recibir las peticiones realizadas por el usuario en la vista, adaptarlas al formato deseado por el modelo (o en este caso particular al sistema de acceso al modelo), recibir la respuesta del mismo, y adaptar la respuesta a las necesidades de la interfaz. Para poder realizar esta tarea, se definen unos endpoints o puntos de acceso, en los que través de la definición de url's es capaz de compartir funciones propias. Es lo que se denomina una API.

Una API (ApplicationPrograming Interface) es una interfaz de programación que permite la comunicación e interacción entre diferentes componentes de software. En el proyecto, la API maneja solicitudes y respuestas HTTP y se encarga de dirigir esas solicitudes a diferentes tareas de controlador para procesar la lógica correspondiente.

Para ello se crea un script accesibles.py, que va a contener la definición de la API y todas las funcionalidades asociadas. El script tiene el siguiente aspecto:

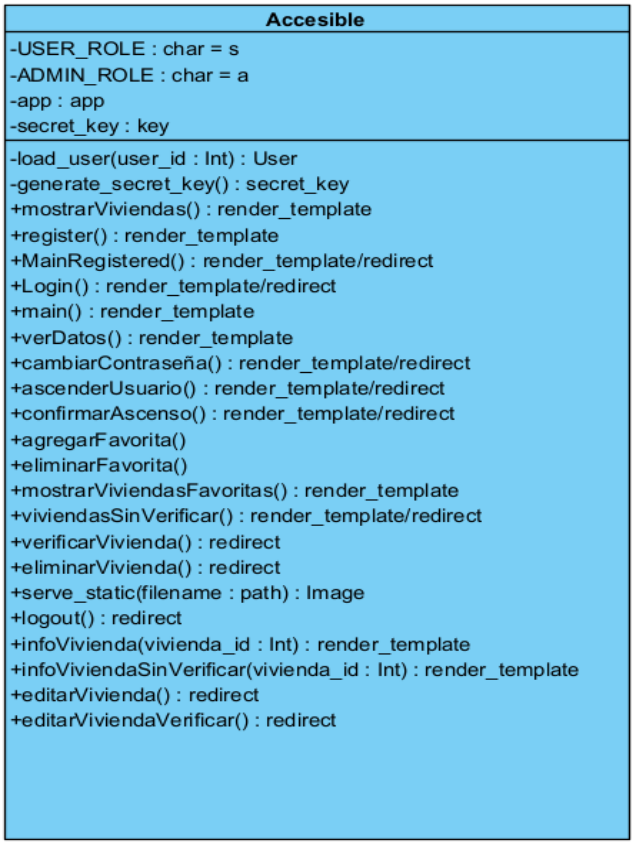

Imagen 9- Estructura Accesible

En él se definen unos atributos necesarios para el correcto funcionamiento de la API. Los atributos ADMIN_ROLE y USER_ROLE son dos constantes que marcan los valores de referencia para cada uno de los roles. Le sirve para llevar internamente la lógica de privilegios en el sistema, es decir, de limitar funcionalidades en función de tu role. En cuanto a las variables app y secret_key:

- App es una instancia de la clase flask, que representa a la aplicación flask. La clase Flask es proporcionada por el frameworkFlask. Se debe pasar el argumento __name__ , que es una variable especial de python que representa el nombre del módulo actual. Esta variable se utiliza para configurar y definir las rutas, vistas y comportamientos de la aplicación.
- Secret_key: Es una clave creada para mantener la seguridad en la sesión del usuario y proteger contra ataques de falsificación de solicitudes entre sitios. La clave secreta se configura en la aplicación flask.

Estos dos elementos son fundamentales para la configuración y el funcionamiento básico de la aplicación.

Se implementa de la siguiente manera:

```python
app = Flask(__name__)
login_manager = LoginManager(app)
login_manager.init_app(app)



def generate_secret_key():
    secret_key = Fernet.generate_key()
    return secret_key
```

Imagen 10 - Instanciación variable app y secret_key

```python
if __name__ == '__main__':
    secret_key = generate_secret_key()


    # Asignar la clave secreta a la aplicación Flask
    app.secret_key = secret_key

    app.run()
```

Imagen 11 - Uso de las variables app y secret_key

En cuanto a las funciones que tiene definidas el script, la mayoría de ellas corresponden al comportamiento de los end-points definidos por app en nuestro script. Las únicas que no lo son sonload_user y generate_secret_key. generate_secret_key se muestra en la Imagen 6, donde se utiliza la biblioteca Fernet para generar una clave secreta. Fernet proporciona un cifrado simétrico y autenticación de datos.

Load_user por otro lado tiene una función algo más compleja. Para empezar se debe explicar la definición de una clase User en el script. Esta clase es una implementación personalizada de la clase UserMixin provista por Flask-login. Esta clase tiene como objetivo representar a un usuario en la aplicación. Tiene dos atributos, id, para identificar al usuario, y role, para identificar sus privilegios. En Load_user se pasa un id de un usuario para cargar los datos del mismo en una variable de esta clase, la cual se retorna.

```python
@login_manager.user_loader
def load_user(user_id):

    """
    Obtiene los datos de usuario a través del módulo de acceso a la base de datos
    :user_id id del usuario
    """
    # Obtener los datos del usuario desde la base de datos
    existing_user=accesoBaseDatos.load_user(user_id)
    if existing_user:
        user = User(existing_user[0], existing_user[1])# Crear una instancia de la clase User con los datos del usuario
        return user


    return None
```

Imagen 12 - Función load_user

Esta función es pasada como callback al decorador «@login_manager.user_loader», que define como cargar los datos de usuario para flask_login.

Una vez explicadas estas dos funciones especiales dentro del script, el resto son todas muy similares y tienen las siguientes características:

- Se crea un endpoint o una ruta para la aplicación de flask, mediante «@app.route('/urlDeseada', methods=[metodos])», donde urlDeseada es el nombre que se le quiere asignar a la ruta de acceso y métodos son los métodos REST permitidos en ese acceso. El valor por omisión es GET.
- Retornan la renderización de una de las interfaces que existentes en la vista o redireccionan a otra ruta. Hay algunas funciones que pueden realizar ambas cosas en función del método utilizado, los privilegios del usuario o la información del request

Existen algunas de las funciones que tienen parámetros de entrada, los cuales están incluidos en la ruta de acceso. Estas funciones son las correspondientes a mostrar información de viviendas o para obtener imágenes estáticas y siguen todas la misma estructura. Se deben realizar pequeños cambios en la ruta proporcionada. Un ejemplo:

```
@app.route("/infoVivienda/<int:vivienda_id>")
def infoVivienda(vivienda_id):

    """
    Define el endpoint para mostrar la información de la vivienda seleccionada. Su comportamiento es diferente si el usuario es administrador,
    mostrando una interfaz más completa
    :vivienda_id id de la vivienda de la que se quiere obtener la información
    """
    vivienda=accesoBaseDatos.getViviendaInfo(vivienda_id)
    imagenes=accesoBaseDatos.getFotosVivienda(vivienda_id)
    if current_user.is_authenticated:
        if current_user.role==ADMIN_ROLE:
            return render_template('infoViviendaAdmin.html', vivienda=vivienda, imagenes=imagenes)
        else:
            return render_template('infoVivienda.html', vivienda=vivienda,imagenes=imagenes)
    else:
        return render_template('infoVivienda.html', vivienda=vivienda,imagenes=imagenes)
```

Imagen 13 - Función infoVivienda

En la función infoVivienda, se requiere del id de la vivienda de la cual se quiere mostrar la información. Para que los url de la información de cada vivienda sean únicos, se debe definir la ruta como se realiza en Imagen 8, incluyendo la variable entera vivienda_id en la misma. De esta forma, el link de cada vivienda será diferente.

En cuanto a la función para recuperar elementos estáticos, en el proyecto se recuperan imágenes, que corresponden a elementos de la interfaz. Las interfaces recuperan estas imágenes en tiempo de ejecución y sin saber realmente donde están ubicadas las imágenes. La función luce de la siguiente manera:

```
@app.route('/static/<path:filename>')
def serve_static(filename):
    root_dir = os.path.dirname(os.getcwd())
    return send_from_directory(os.path.join(root_dir, 'static'), filename)
```

Imagen 14 - Función serve_static

Esta función es la única vez en la que el controlador va a acceder directamente a algún tipo de almacenamiento de datos. El resto de veces lo hará a través del módulo AccesoBaseDatos. Este módulo pertenece al modelo de la aplicación y será explicado más adelante.

Existen algunas funciones que utilizan el decorador @login_required proporcionado por flask_login para indicar que el acceso a esos endpoints es exclusivo para usuarios registrados. También se suele usar la variable current_user para acceder a los datos del usuario actual. También es proporcionado por flask_login. Son dos funcionalidades muy importantes ya que permite realizar fácilmente restricciones por grupos e individuales en la aplicación

Cabe destacar que el fichero accesible no recibe directamente las peticiones de las interfaces. Las peticiones son procesadas por un servidor. En este caso, el servidor que procesa las peticiones html es el servidor de desarrollo WSGI proporcionado por flask, debido al modelo de implementación elegido. Este servidor podría y debería ser cambiado por otro servidor con estructura WSGI si se traslada la implementación a un entorno comercial. En este caso,se ha dockerizado el proyecto y el servidor de pruebas cumple las funciones que se necesitan.

En resumen, el script accesible.py es la aplicación Flask. En ella se definen varios endpoints para el acceso a determinadas funciones. Se encarga, junto con el servidor, de comunicar las peticiones de los usuarios, realizadas en las interfaces de la vista , al modelo. Se utiliza el moduloAccesoBaseDatos para acceder al modelo, abstrayendo el funcionamiento interno de la base de datos. Una vez obtenidos los datos del modelo, se adaptan y se responde de la forma adecuada según la lógica interna. El script define y utiliza la lógica interna de la aplicación para el correcto funcionamiento de la misma, con las restricciones de seguridad necesarias.

2.3. Model - Modelo

En el proyecto el modelo esta conformado principalmente por una base de datos implementada en MariaDB, en la que se han credo varias tablas para guardar y manejar la información de nuestro sistema. Las tablas que se han creado son:

- users
- infoViviendas
- imagenesViviendas
- viviendasFavoritas
- viviendasSinVerificar

2.3.1. Diseño de datos en el modelo

En este apartado se pretende explicar la representación que se ha realizado para los datos en el modelo. Para ello, se van a describir las tablas mencionadas anteriormente.

Users: Es la tabla que almacena la información relativa a los usuarios. Los campos de esta tabla son:

1. id : Es un entero no nulo y representa la clave primaria de la tabla, por lo que sus valores deben ser únicos. Es autoincrementativo, por lo que no es necesario incluirlo en la sentencia de creación de una tupla.

2. username: varchar(50), identifica el nombre de usuario.
3. name: varchar(50), identifica al nombre real del usuario.
4. enail: varchar(100), identifica el correo electrónico asociado al usuario.
5. password: varchar (255), es la contraseña asociada al usuario. Este campo se introduce en la tabla hasheado, no el valor real, para mejorar la seguridad.
6. role: char(1), identifica al rol del usuario. Debe tener uno de los valores de ADMIN_ROLE o USER_ROLE.

InfoViviendas: Es la tabla que contiene la información de las viviendas ya verificadas que se muestran en la aplicación. La información de la tabla se consigue a través del sistema scraping. Los campos de esta tabla son:

1. id: Idemusers.
2. titulo: varchar(255). Frase que resume la vivienda.
3. link: varchar(255). Url correspondiente a la vivienda
4. precio: decimal(10,2). Precio de la vivienda
5. ciudad: varchar(255). Ciudad donde se encuentra la vivienda
6. descripcion: varchar(10000). Descripción de la vivienda.
7. dispo: varchar(3). Parámetro de control para saber si la vivienda sigue estando disponible en la página original.

ImagenesViviendas: Tabla donde se almacenan las url de las imágenes que corresponden a cada vivienda. Sus campos son:

1. id: Idemusers.
2. link: varchar(255). Url de la imagen.
3. vivienda_id:Int. Clave externa que referencia al id de una vivienda en la tabla infoViviendas.

De esta forma se consiguen dos cosas. La primera es evitar la descarga de todas las imágenes para mostrarlas en la web, lo que consumiría muchos recursos de espacio. Además, la estructura de la tabla, permite el rápido acceso a las imágenes cuando se necesiten mediante el uso de la clave externa sobre la tabla infoViviendas.

ViviendasFavoritas: Tabla con estructura similar a imagenesViviendas que relaciona al usuario con las viviendas que haya agregado como favoritos. La tabla tiene los siguientes campos:

1. id: idemusers
2. vivienda_id: int. Clave externa referenciando al atributo id de infoviviendas.
3. user_id: int. Clave externa referenciando al atributo id de users.

De esta forma, tanto añadir como eliminar viviendas favoritas son operaciones poco costosas. Además, permite mantener la información de las viviendas favoritas separado tanto de la información de las viviendas como de la información de los usuarios.

InfoViviendasVerificar: Es la tabla donde se almacenan los datos de las viviendas encontradas por el programa scraper. Las viviendas que se encuentran en esta tabla lo hacen de manera temporal, suponiéndose que un futuro un usuario administrador la verificará o la borrará permanentemente. Los campos de esta tabla son los mismos que la de la tabla infoviviendas, salvo dispo.

2.3.2. Módulo de Acceso a la Base de Datos

En el proyecto, como se ha comentado a lo largo del anexo, se utiliza un módulo para el acceso a la base de datos. Es el único programa que accede y modifica directamente la base de datos. Para que pueda acceder a la base de datos se ha decidido utilizar la biblioteca mysql.connector, que permite realizar consultas en la base de datos. Para que el módulo pueda realizar consultas en la base de datos del sistema, es necesario proporcionarle los parámetros de configuración, como el puerto, usuario, contraseña... A continuación, se muestra una imagen de los parámetros utilizados:

```
db_config = {
    'host': '192.168.1.31',
    'user': 'root',
    'password': '16112000',
    'database': 'viviendasaccesibles',
    'port' : '3300'
}
```

Imagen 15 - Parámetros de acceso a la base de datos

Utilizando estos parámetros se realiza la conexión con la base de datos del sistema. El script tiene definidas funciones que satisfacen las necesidades de todos los elementos del sistema. Tiene el siguiente aspecto:

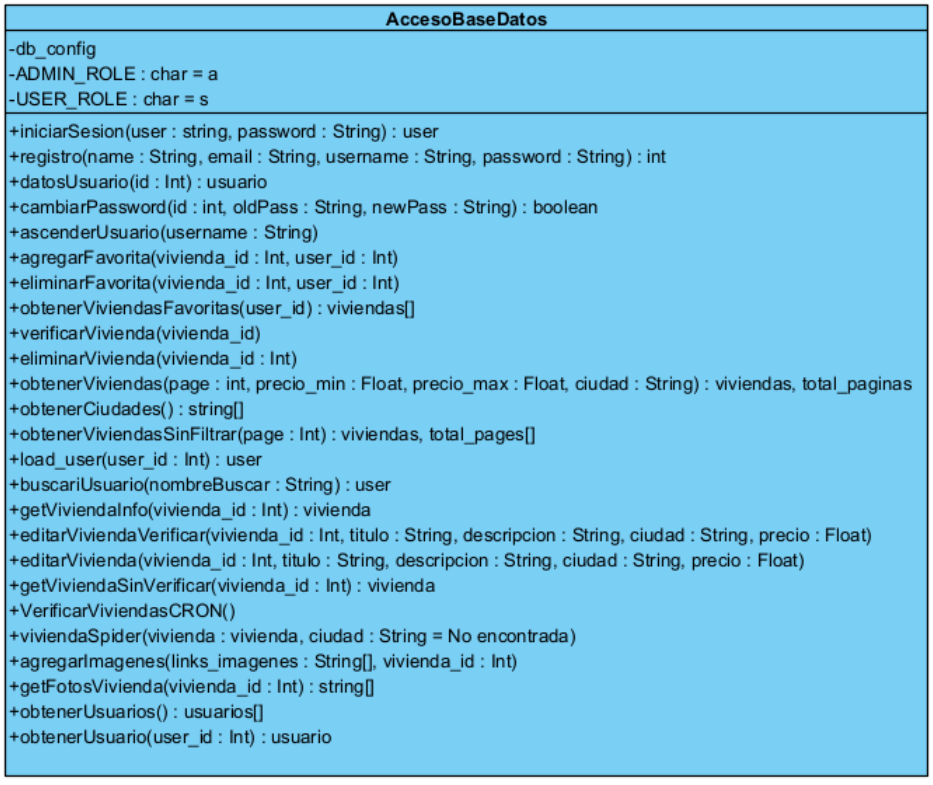

Imagen 16 - Estructura modulo AccesoBaseDatos

Cabe destacar que, aunque la mayoría de las funciones que implementa este fichero son utilizadas directamente por el controlador para el acceso a la base de datos, existen otros componentes que utilizan el módulo para acceder a la base de datos. Existen funciones especiales definidas para estos componentes, las cuales son:

- VerificarViviendasCRON: Es utilizada por un script que se encarga de verificar la disponibilidad de las viviendas en la página web original. La función se encarga de visitar todos los links que tenemos almacenados, tanto de viviendas verificadas como no verificados. Si la vivienda no está disponible puede realizar dos acciones: borrar la vivienda si no ha sido aún verificada o cambiarle el valor al campo dispo de la vivienda si sí está verificada. El script verificarViviendas se encarga de utilizar recursivamente esta función.
- ObtenerCiudades: Es utilizada cuando se realiza una búsqueda y se está en la interfaz mostrarViviendas o mostrarViviendasRegistrado. Recoge y ordena alfabéticamente las ciudades de las que existen viviendas en la base de datos para facilitar el filtrado.
- viviendaSpider: Es una función utilizada por el sistema scraper para incluir viviendas en la base de datos de viviendasSinFiltrar.

2.3.3. Buscar Imágenes

El módulo de acceso a la base de datos hace uso de un script para encontrar las imágenes correspondientes a un link de una vivienda. Esta acción se realiza cuando se verifica la vivienda, justo después de añadirla a la tabla de infoviviendas. Se realiza en ese momento porque para poder incluir las fotos en la tabla de la base de datos se necesita saber el id de la vivienda. Es por ello que no se agregan imágenes de las viviendas que no están verificadas. El script únicamente tiene la función de recoger las imágenes de manera similar a como se hace en el scraper.

2.3.4. Scraper

Una de las partes más importantes dentro del proyecto es el sistema scraper elegido. Inicialmente, se planteó un sistema más rudimentario para la obtención de información de viviendas en la web, utilizando selenium. Rápidamente se descartó esa idea y se pasó a utilizar Scrapy como herramienta para realizar esta función. Scrapy es un framework de scraping y crawling implementado en python. Permite la creación de una araña a la que puedes definir unas url que visitar y unos atributos que buscar. De esta forma el sistema va a encontrar información de las viviendas.

Para poder utilizar scrapy debemos definir las url's que debe visitar. Al principio del proyecto se intentaron utilizar los portales de viviendas más conocidos. Usaban una ruta compleja, pero eran la mejor opción en cuanto a escalabilidad del proyecto, ya que disponían de la base de viviendas más grande. Concretamente, se intentó utilizar fotocasa.es e idealista.es. Ambas páginas no han podido ser utilizadas finalmente, ya que disponían de programas anti-scraping y lo prohibían expresamente en las condiciones de uso. Es por eso que finalmente se decidió realizar el scraping en una página más pequeña: todopisos.es . Este portal tiene menos vivienda en la base de datos, pero no prohíbe de ninguna forma el scraping. Además, las rutas de las viviendas del portal son muy sencillas, siguiendo todas la siguiente estructura:

https://www.todopisos.es/inmueble/id

Donde id es un identificador numérico que identifica a cada vivienda subida en la página. De esta forma el sistema puede guardar el número de la última vivienda visitada con éxito y poder crear en tiempo de ejecución las url que debe visitar la araña. El ultimo número visitado con éxito se guarda en un fichero cuya ruta se almacena en el atributo CONF de la araña.

Una vez establecidas las rutas, se le debe indicar que información debe de recuperar de cada una de las páginas que visite. Para ello, se utiliza el campo css de la response recibida por parte del servidor. En nuestro caso particular se buscará por clases de estilo el título, el precio y la descripción.

```
title = response.css("h1::text").get()
price = response.css(".price::text").get()
description = response.css(".large::text").get()
# Crear el objeto de datos
data = {
    "title": title,
    "price": price,
    "description": description,
    "link": response.url,
}
```

Imagen 17 - Búsqueda de datos de la vivienda

Tras encontrar estos datos, coteja si la descripción encaja con la de una descripción de una vivienda accesible. Este proceso se explica más adelante.

Una vez decidido si la vivienda es accesible o no, se recogen otro tipo de datos, como la ciudad, que son más costosos, ya que la página web todopisos no expresa la ciudad en un campo exclusivo, por lo que se extrae de la descripción utilizando geonamescache. Esta biblioteca de Python permite obtener todas las ciudades de España y Portugal, países en los que esta página dispone de viviendas.

Una vez encontrados todos los datos relativos a la vivienda, se utiliza el módulo AccesoBaseDatos para añadir la vivienda a la base de datos de viviendas sin verificar.

2.3.5. Proceso de filtrado

Podría considerarse la funcionalidad clave del proyecto. Es una parte muy importante de la funcionalidad ofrecida por la aplicación, ya que el filtro aplicado debe ser capaz de diferenciar las descripciones de las viviendas que son accesibles de las que no lo son. Para implementar esta funcionalidad se ha decidido utilizar las bibliotecas Gensim y nltk.

Estas dos bibliotecas de Python han sido las utilizadas para poder filtrar las viviendas accesibles del resto de viviendas que se obtienen con la araña. Para ello se hace uso de los modelos LDA propocionados por gensim. Un modelo de Asignación Latente de Dirichlet (LDA) es un modelo generativo que permite generar una serie de temas o tópicos desde un conjunto de datos dados (corpus). Para poder trabajar correctamente con este modelo, no se puede usar directamente el conjunto de descripciones de las viviendas, ya que existe mucha información que no es útil a la hora de clasificar los tópicos. Para ello se usan las herramientas proporcionadas por NLTK. En concreto se hace uso de varias de sus funciones que permiten eliminar signos de puntuación, dígitos y reducir cada palabra restante a su raíz etimológica. La raíz resultante de este proceso se llama token. El proceso de entrenamiento del modelo LDA ha llevado su tiempo, realizando muchas pruebas hasta que se ha conseguido un modelo que satisfaga mínimamente las necesidades de elproyecto (filtrar viviendas según sean adaptadas o no). El proceso de entrenamiento ha sido siempre el mismo y es el siguiente:

```
diccionario = Dictionary(df.Tokens)
print(f'Número de tokens: {len(diccionario)}')

corpus = [diccionario.doc2bow(Descripcion) for Descripcion in df.Tokens]

lda = LdaModel(corpus=corpus, id2word=diccionario,
               num_topics=100, random_state=42,
               chunksize=509, passes=20, alpha='auto')
```

Imagen 18 - Creación diccionario y modelo LDA

Como se puede observar en la Imagen 18, lo primero es generar un diccionario con los tokens que se han obtenido de las descripciones analizadas. En el caso que se muestra se está utilizando un dataframe de la biblioteca pandas. Después se debe crear un corpus con todos los tokens que aparezcan en el dataframe. En el corpus, cada descripción se transformará en una bolsa de palabras, con las frecuencias de aparición de las mismas. Una vez creados estos dos objetos se procede a entrenar al modelo LDA, el cual tiene muchos parámetros de configuración. Los que han resultado más útiles a la hora de configurar el modelo LDA definitivo han sido:

- num_topics: Indica el número de temas o tópicos que debe de ser capaz de encontrar el modelo LDA. En el caso de este proyecto, finalmente se utilizó un número considerablemente alto de temas (100) ya que las viviendas accesibles es un tema bastante específico dentro de todos los temas que puede encontrar en una vivienda, ya que puede crear temas por ciudades, lujos de la vivienda, número de habitaciones y baños, tipo de vivienda...
- chunksize: Indica el tamaño de muestra que coge el modelo cada vez en cada interación del entrenamiento. Este valor fue variando en función del tamaño del corpus, pero en general fue variando entre 500 y 1000.
- passes: Indica el número de veces que se quiere que recorra el corpus completo. En este proyecto se le indicó un número de pasadas alto, de alrededor de 100, ya que al aumentar este número, los temas obtenidos suelen ser más diferentes entre sí.
- eta y alpha= son dos variables que permiten indicar la similitud entre los tópicos encontrados. A mayor sea el número que se introduzca, más parecidos serán los temas que proporcione el modelo LDA. En el caso concreto de este proyecto se hicieron varias pruebas y los valores que mejor funcionaron fueron siempre valores inferiores a 1, ya que si se utilizaban valores mayores que 1, en la mayoría de los tópicos aparecían palabras como baño, dormitorio, cocina... que no permiten para nada diferenciar unas viviendas de otras.

Tras varias pruebas con distinto tamaño en el corpus (inicialmente se probó con una base de datos de 200.000 viviendas, pero no se encontraban temas relevantes para viviendas adaptadas. Después se hizo un filtrado manual para encontrar unas 500 viviendas que utilizar como corpus, lo que finalmente dió resultados) y con los valores de configuración de LDA, se ha encontrado un modelo LDA que contiene un tema relativo a viviendas adaptadas. En concreto, encuentra aquellas viviendas que no tienen barreras arquitectónicas y que lo especifican en la descripción.

El filtrado de viviendas se hace en gran parte gracias a este tópico, que reduce la gran cantidad de viviendas rastreadas inicialmente (unas 200.000 como se ha comentado anteriormente) a una reducida lista de 1500. En estas 1500 se buscan los tokens clave que son barrer y arquitecton. Aunque se podría hacer esta búsqueda desde un primer momento, el modelo LDA ahorra gran parte del trabajo de filtrado y esta pequeña parte (la cual ha reducido el número de viviendas de 1500 a unas 700), la cual es más costosa, se reduce a un número ínfimo de viviendas.

Una vez realizado el proceso de creación del modelo LDA y el diccionario, se almacenan y este proceso no se repite nunca más. A partir de ahora, cuando se extraiga la información de una vivienda, se aplicarán los ajustes necesarios al texto para obtener los tokens y se utilizará el diccionario y modelo ya creados. Se define una función topicoAccesible dentro del script de la araña scrapy. En ella se le pasa un conjunto de tokens y determina si se corresponden con el tópico accesible.

```
lda= models.LdaModel.load("/home/usuario/PruebaServer/DEF/LDA/ldamodel")
diccionario=gensim.corpora.dictionary.Dictionary.load("/home/usuario/PruebaServer/DEF/LDA/casas.dictionary")

def topicoAccesible (vivienda):
    bow_articulo_nuevo = diccionario.doc2bow(vivienda)
    i=0

    for topico in lda[bow_articulo_nuevo]:
        i+=1
        if topico[0] == 67:


            if topico[1] > 0.015:

                return True

            else:

                return False

    return False
```

Imagen 19 - Función topicoAccesible

De esta forma determina si las viviendas encontradas son accesibles o no. Además, se realiza la búsqueda antes mencionada de tokens clave para evitar errores. Esta función es ejecutada con cada una de las descripciones obtenidas por la araña. Teniendo todo esto en cuenta, el script de la araña acaba teniendo la siguiente estructura:

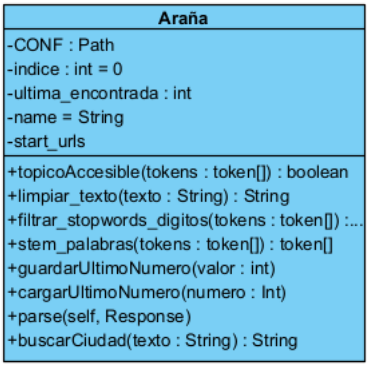

Imagen 20 - Estructura Araña

3. Diagrama de clases de diseño

Un diagrama de clases de diseño es una representación visual de las clases y sus interacciones en el sistema.

El diagrama final de clases del diseño es el siguiente. No se han especificado los diferentes componentes de la vista para facilitar la comprensión del diagrama

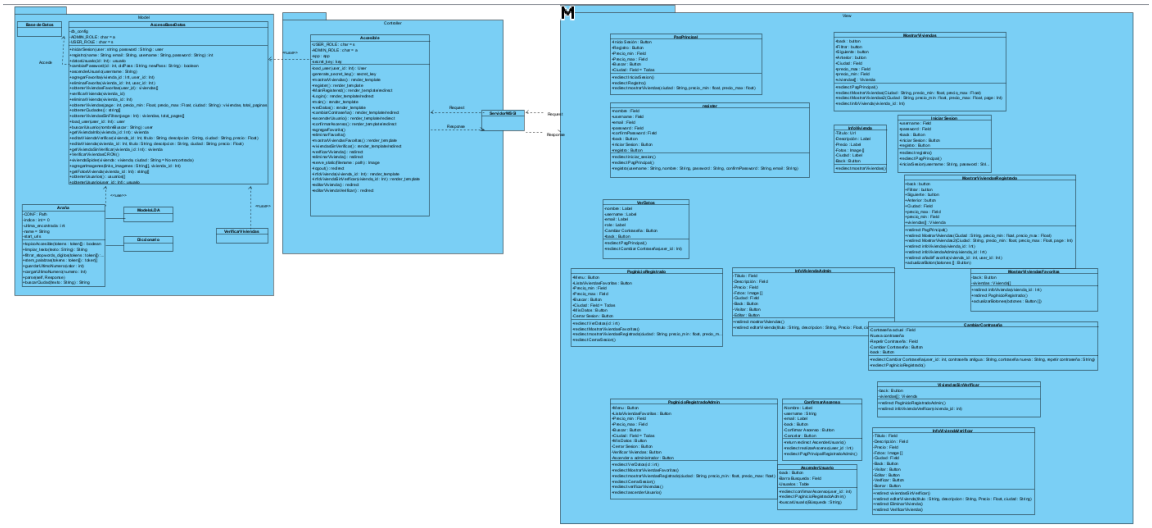

Imagen 21 - Diagrama de clases de diseño

En el diagrama mostrado en Imagen 21, se muestran los módulos de la aplicación. La comunicación entre la vista y el controlador se realiza mediante las peticiones realizadas por la interfaz, las cuales son contestadas por el controlador. El controlador se comunica con el modelo mediante el uso de funciones definidas en el módulo de acceso a la base de datos. Estas funciones han sido detalladas en este documento y permiten al controlador responder las peticiones de la vista. Los elementos del modelo que interactúan con la base de datos del sistema también utilizan el módulo de acceso a la base de datos, mediante las funciones definidas para ello.

La estructura interna de cada uno de los elementos se ha detallado a lo largo de este documento, al igual que la estructura de la vista.