RASTREADOR DE VIVIENDAS ADAPTADAS

ANEXO IV
Documentación técnica de programación

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 contenido ii
Tabla de imágenes iii
1. Introducción 1
2. Estructura general del proyecto 2
3. Configuración del sistema 3
4. Módulo de vista 4
    4.1. CSS y JavaScript 6
5. Módulo de gestión 10
    5.1. Configuración de la API 10
    5.2. Puntos de acceso 12
6. Módulo de persistencia 21
    6.1. Base de datos del sistema 21
    6.2. Acceso base de datos 21
        6.2.1. Funciones de accesoBaseDatos.py 22
    6.3. Sistema scraper 32
    6.4. Modelo LDA y diccionario 34
    6.5. VerificarViviendas.py 35
    6.6. BuscarImagenes.py 36
    6.7. Cron 36
7. Modelo de despliegue 37
8. Vista general del proyecto 39

Tabla de imágenes

Imagen 1 - Requirements.txt 3
Imagen 2 -Construcción Request /mostrar_viviendas 5
Imagen 3 - Construcción Requests para el acceso de las diferentes opciones del menú 5
Imagen 4 - Formulario de registro 6
Imagen 5 - Ejemplo Style en Html 7
Imagen 6 - Actualizar botones JS I 8
Imagen 7 - Actualizar botones JS II 8
Imagen 8 - buscarUsuarios () JS 9
Imagen 9 - Bibliotecas utilizadas en la API 10
Imagen 10 - Clase User 11
Imagen 11 - Instanciación variable app y secret_key 12
Imagen 12 - Uso de las variables app y secret_key 12
Imagen 13 - Función load_user 12
Imagen 14 - Endpoint /mostrar_viviendas 13
Imagen 15 - endpoint /register 13
Imagen 16 - endpoint /MainRegistered 14
Imagen 17 - endpoint /Login 14
Imagen 18 - endpoint / 14
Imagen 19 - endpoint /VerDatos 15
Imagen 20 - endpoint /cambiarContraseña 15
Imagen 21 - endpoint /ascenderUsuario 15
Imagen 22 - endpoint /confirmarAscenso 16
Imagen 23 - endpoint /agregarFavorita 16
Imagen 24 - /endpoint eliminarFavorita 16
Imagen 25 - endpoint /mostrarViviendasFavoritas 17
Imagen 26 - endpoint /viviendasSinVerificar 17
Imagen 27 - endpoint /verificarVivienda 17
Imagen 28 - endpoint /eliminarVIvienda 18
Imagen 29 - endpoint /static/<path:filename> 18
Imagen 30 - endpoint /logout 18
Imagen 31 - endpoint /infoVIvienda/<int:vivienda_id> 19
Imagen 32 - enpoint /infoViviendaSinVerificar/<int:vivienda_id> 19
Imagen 33 - endpoint /editarVivienda 19
Imagen 34 - ednpoint /editarViviendaVerificar 20
Imagen 35 - Datos de acceso a la base de datos 21
Imagen 36 - Función iniciarSesion (user, password) 22
Imagen 37 - Función registro (name, email, username, password) 22
Imagen 38 - Función datosUsuario (id) 23

Imagen 39 - cambiarPassword (id, oldPass, newPass) 23
Imagen 40 - Función ascenderUsuario (username) 23
Imagen 41 - Función agregarFavorita (vivienda_id, user_id) 24
Imagen 42 - Función eliminarFavorita (vivienda_id, user_id) 24
Imagen 43 - Función obtenerViviendasFavoritas (user_id) 25
Imagen 44 - Función verificarVivienda (vivienda_id) 26
Imagen 45 - Función eliminarVivienda (vivienda_id) 26
Imagen 46 - Función mostrarViviendas (page, precio_min, precio_max, ciudad) I 27
Imagen 47 - Función mostrarViviendas (page, precio_min, precio_max, ciudad) II 27
Imagen 48 - Función obtenerCIudades () 27
Imagen 49 - Función actualizarVIviendas (viviendas, user_id) 27
Imagen 50 - obtenerViviendasSinFiltrar (page) 28
Imagen 51 - Función load_user (user_id) 28
Imagen 52 - Función buscarUsuario (nombreBuscar) 29
Imgen 53 - Función getViviendaInfo (vivienda_id) 29
Imagen 54 - Función editarViviendaVerificar (vivienda_id, titulo, descripcion, precio) 29
Imagen 55 - Función editarVivienda (vivienda_id) 30
Imagen 56 - Función getViviendaSinVerificarInfo (vivienda_id) 30
Imagen 57 - Función verificarViviendasCRON () 30
Imagen 58 - Función viviendaSpider (vivienda, ciudad = «no encontrada» 31
Imagen 59 - agregarImagenes (links_imagenes, vivienda_id) 31
Imagen 60 - Función getFotosVivienda (vivienda_id) 31
Imagen 61 - Función obtenerUsuarios () 32
Imagen 62 - Función obtenerUsuario (user_id) 32
Imagen 63 - Funciones cargarUltimoNumero() y guardarUltimoNumero 33
Imagen 64 - Extracción de datos por parte de la araña 33
Imagen 65 - Proceso de transformación de las descripciones 34
Imagen 66 - Función limpar_texto (texto) 34
Imagen 67 - Función filtrar_stopword_digitos (tokens) 34
Imagen 68 - Creación stemmer y función stem_palabras (tokens) 35
Imagen 69 - Función topicoAccesible (vivienda) 35
Imagen 70 - VerificarViviendas.py 36
Imagen 71 - BuscarImagenes.py 36
Imagen 72 - Archivo cronjob 36
Imagen 73 - Dockerfile 37

1. Introducción

En este anexo, correspondiente a la documentación técnica de programación se van a explicar las principales funciones de la aplicación, así como las configuraciones necesarias tanto en el entorno de desarrollo como en el entorno virtualizado. El objetivo principal de este documento es facilitar el trabajo a futuros desarrolladores que puedan continuar el proyecto y quieran ampliar funcionalidades o mejorar el rendimiento.

Se va a recoger información relativa a la estructura del proyecto, tanto módulos generales como cada uno de sus componentes, especificando la funcionalidad que aportan y cómo lo hacen. En el propio código se incluyen docstrings que hace que las funciones definidas sean autoexplicativas.

Además se especificará el modelo de despliegue utilizado, en este caso se ha decidido utilizar contenedores de software.

2. Estructura general del proyecto

El proyecto consta de 3 módulos principales, que se comunican entre si para ofrecer la funcionalidad completa al usuario. La estructura general de los módulos es la siguiente:

- Modulo de vista: Esta compuesto por todas las interfaces del sistema. Ofrecen una representación de los datos almacenados en el sistema. Se comunican con el módulo de gestión mediante el uso de peticiones siguiendo el protocolo REST.
- Módulo de gestión: Se corresponde con la API-REST definida por Flask y el servidor que atiende las peticiones para la API. Define varios puntos de acceso a la información.
- Módulo de persistencia: Se corresponde con el almacenamiento de datos en el sistema. Está compuesto por la propia base de datos, un sistema de acceso a la misma, archivos estáticos y el software scraper que recoge la información de las viviendas.

La estructura concreta de cada uno de los módulos se especificará más adelante. Es importante recalcar que el módulo de gestión accede al módulo de persistencia a través del sistema de acceso a la base de datos y es la única forma de acceder al mismo.

3. Configuración del sistema

El backend de la aplicación está implementado en Python. Es por ello que para el correcto funcionamiento de la aplicación, es necesario que se tenga Python instalado en el sistema. Se utilizan múltiples bibliotecas a lo largo del desarrollo de la aplicación. Estas bibliotecas vienen especificadas en el archivo requirements.txt que es utilizado por el modelo del despliegue para asegurarse que todas están instaladas antes de la ejecución del mismo.

```
beautifulsoup4==4.8.2
cryptography==41.0.0
cryptography>=2.8
Flask==2.3.2
Flask_Login==0.6.2
gensim==4.3.1
geonamescache==1.6.0
itemadapter==0.8.0
mysql_connector_repackaged==0.3.1
nltk==3.8.1
Requests==2.31.0
Scrapy==2.9.0
```

Imagen 1 - Requirements.txt

Más adelante, se explicará detalladamente donde se utilizan cada una de las bibliotecas mostradas.

4. Módulo de vista

El módulo de vista se compone por un conjunto de interfaces que interactúan con la API definida en el módulo de gestión. Las interfaces del sistema son:

- AscenderUsuario : Muestra la interfaz asociada a la funcionalidad de ascender usuarios como administradores. El usuario administrador verá una tabla con los datos de los usuarios, a través de la cual se puede buscar por nombre. Al seleccionar cualquier nombre de usuario se redirecciona a través de la API a la interfaz ConfirmarAscenso con los datos del usuario seleccionado.
- CambiarContraseña: Proporciona la interfaz para el cambio de contraseña del usuario. Proporciona un formulario con 3 campos: Contraseña antigua, contraseña nueva y repetir contraseña.
- ConfirmasAscenso: Muestra los datos del usuario seleccionado en AscenderUsuario y da dos opciones: Cancelar la operación o confirmar la acción. La opción de cancelar devuelve al usuario a la interfaz AscenderUsuario y la opción confirmar lo asciende a administrador.
- InfoVivienda: Interfaz que muestra la información detallada de una vivienda en concreto. Se puede acceder al link de la vivienda original pulsando en el título de la vivienda.
- InfoViviendaAdmin: Similar a InfoVivienda, pero con la capacidad de editar los campos de información de las viviendas. El acceso a la página original de la vivienda se hace a través de un botón en lugar de pulsando el título.
- InfoViviendaVerificar: Similar a InfoViviendaAdmin, pero con las viviendas que no han sido verificadas. Aparecen dos opciones para verificar la vivienda o borrarla del sistema.
- IniciarSesion: Interfaz para el inicio de sesión del usuario. Muestra un cuestionario donde se pide el nombre de usuario o email utilizado en el resgistro y la contraseña asociada a éste. Si los datos no son correctos, se redirige al usuario a esta interfaz de nuevo con un código de error proporcionado por la API.
- MostrarViviendas: Interfaz en la que se muestran las viviendas que se corresponden con los filtros introducidos. Se muestran divididas por páginas de 20 viviendas. Cada vivienda tiene un contenedor que contiene su información: título, descripción, ciudad y precio. Al pulsar en el título, se redirige a InfoVivienda de la vivienda en cuestión. En la parte superior hay un formulario en el que aparecen los filtros de la búsqueda. Pueden cambiarse los filtros y darle al botón para buscar y volver a esta interfaz con los filtros introducidos.
- MostrarViviendasFavoritas: Muestra las viviendas que el usuario tiene asignadas como favoritas.
- MostrarViviendasRegistrado: Muy similar a MostrarViviendas, salvo que aparece un botón para añadir o borrar las viviendas a la lista de favoritos.
- PagInicioRegistrado: Página principal de los usuarios registrados. Tiene varios elementos. En la parte central hay un formulario con 3 campos, todos ellos opcionales: Ciudad, precio mínimo y precio máximo. Al lado, hay un botón con la imagen de una lupa que se utiliza para redirigir a la interfaz MostrarViviendasRegistrado, donde las viviendas que se muestran casan con los filtros introducidos en el formulario. En la parte superior izquierda hay un botón que despliega un menú, en el que aparecen las opciones del usuario registrado.
- PagInicioRegistradoAdmin: Similar a PagInicioRegistrado, pero las opciones del menú son las correspondientes a las del usuario administrador.

- PagPrincipal: Interfaz similar a PagInicioRegistrado. La diferencia es que aquí no aparece un menú, y en la parte superior derecha hay dos botones que redirigen a las interfaces register e IniciarSesion
- Register: Interfaz de registro de usuario en el sistema. Presenta un formulario que recoge los datos necesarios para el registro. Tiene un botón para confirmar el registro y tiene un atajo para la interfaz IniciarSesion.
- VerDatos: Interfaz con los datos del usuario. Desde aquí se puede acceder a la interfaz CambiarContraseña con un botón.
- ViviendasSinVerificar: Similar a mostrar viviendas. No tiene barra de filtros. Las viviendas que se muestran no han sido verificadas por ningún usuario administrador.

En muchas de las interfaces descritas, aparecen elementos que favorecen la navegación por la app, normalmente para acceder a otras interfaces sin la necesidad de editar la URL. La mayoría de estos elementos son botones para redirigir al usuario a la interfaz anterior.

En las descripciones anteriores, cuando se dice redirigir, se refiere a que la interfaz realiza una petición o request a la API-REST del módulo de gestión. Esta solicitud es manejada por el servidor de dicho módulo. La estructura de las peticiones es muy similar en todas las interfaces, cambiando el punto de acceso de la API a través de la URI. Para evitar un documento repetitivo, se van a mostrar algunas peticiones de ejemplo. Más adelante se explicarán todas los puntos de acceso de la API-REST.

```
<div class="container">
    <form action="/mostrar_viviendas" method="GET">
        <div class="search-bar">
            <input class="search-input" name="ciudad" id="ciudad" type="text" placeholder="Ciudad">
            <input class="search-input" name="precio_min" id="precio_min" type="number" placeholder="Precio min">
            <input class="search-input" name="precio_max" id="precio_max" type="number" placeholder="Precio max">
              <button type="submit" class="search-button">
               <img src="{{ url_for('serve_static', filename='PNG/lupa.jpg') }}" alt="Buscar" class="boton_lupa" />
            </button>
        </div>
    </form>
</div>
```

Imagen 2 -Construcción Request /mostrar_viviendas

```
<ul class="menu-options">
    <li class="menu-option"><a href="{{ url_for('mostrarViviendasFavoritas') }}">Lista de viviendas favoritas</a></li>
    <li class="menu-option"><a href="{{ url_for('viviendasSinVerificar') }}">Verificar viviendas accesibles</a></li>
    <li class="menu-option"><a href="{{ url_for('ascenderUsuario') }}" method="GET">Ascender usuario a administrador</a></li>
    <li class="menu-option"><a href="{{ url_for('verDatos') }}">Mis Datos</a></li>
    <li class="menu-option"><a href="{{ url_for('logout') }}">Cerrar sesión</a></li>
</ul>
```

Imagen 3 - Construcción Requests para el acceso de las diferentes opciones del menú

```html
<div class="form-container">
    <h2>Registro</h2>
    <form action="/register" method="POST">
        <label for="name">Nombre:</label>
        <input type="text" id="name" name="name" required>

        <label for="email">Correo electrónico:</label>
        <input type="text" id="email" name="email" required>

        <label for="username">Nombre de usuario:</label>
        <input type="text" id="username" name="username" required>

        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" required>

        <label for="confirm_password">Confirmar contraseña:</label>
        <input type="password" id="confirm_password" name="confirm_password" required>

        <button type="submit">Registrarse</button>
    </form>
```

Imagen 4 - Formulario de registro

Como puede verse en las imágenes 2, 3 y 4, para construir la petición desde la interfaz se debe especificar la acción, representada por la uri de la API. En los casos en las que las consultas utilizan el método get y no proporcionan ningún tipo de información, puede usarse url_for, que establece este método directamente, como se hace en la Imagen 2. Utilizando esta herramienta se debe especificar el nombre de la función y no la url. Para el resto de consultas debe usarse form, como se utiliza en el formulario de registro, Imagen 3. De esta forma se construye una consulta para la URI indicada en actionutulizando el método especificado en method. La información dentro del formulario se incluye en el body de la consulta.

En el resto de interfaces se usa uno de los dos métodos comentados, variando en el caso de form, variando los métodos y parámetros del body, y en ambos casos cambiando la información de acceso a la funcionalidad concreta.

4.1. CSS y JavaScript

Para la hoja de estilo de las interfaces, se han incluido los estilos en el propio archivo html, para facilitar la comprensión de cada uno de los estilos y dónde se utilizan. Es por ello que en todos los ficheros html correspondientes a las interfaces anteriormente explicadas, aparece al inicio de la misma un campo style que define los estilos utilizados en esa misma interfaz.

```
<style>
  body {
      margin: 0;
      padding: 0;
      display: flex;
      min-height: 100vh;
  }
  .content {
      flex: 1;
      background-color: white;
      padding: 20px;
  }
  .green-bar {
      background-color: #4CAF50;
      flex: 0 0 10%;
  }
  .back-button {
      padding: 10px;
      border: none;
      font-size: 18px;
      cursor: pointer;
      position: absolute;
      top: 10px;
      left: 10px;
      text-emphasis-style: none;
      z-index: 1;
      height: 70px;
      width: 70px;
      background-color: #4CAF50;
      border: none;
    }

  .image-style {
    max-width: 100%;
    max-height: 100%;
  }

</style>
```

Imagen 5 - Ejemplo Style en Html

En un futuro, podría ser interesante reunir todos los estilos definidos en un único archivo css y almacenarlo en la carpeta static del proyecto. En el caso de este proyecto, se ha decidido dejarlos así para facilitar la comprensión de los estilos.

De una forma similar ocurre con las funciones JavaScript, se incluyen en las propias interfaces incluyéndolas en un campo script. También podrían incluirse todas juntas en un archivo .js almacenado en static. En este caso tiene aún más sentido incluirlo en el html, ya que se entiende mejor dónde se usan las funciones y qué hacen. No son muchas las funciones utilizadas, y se han utilizado para mejorar la experiencia de usuario en la página.

La primera de las funciones JavaScript utilizadas, se utiliza para actualizar en tiempo de ejecución los botones para agregar a favoritos o eliminar de favoritos las viviendas en la interfaz mostrarViviendasRegistrado. En ella, se ejecuta la solicitud correspondiente, eliminarFavorita o agregarFavorita en función del valor de la vivienda, y se actualiza el botón para mostrar el valor actual.

```
<script>
  // Obtener todos los botones con las clases "agregar-favorita" y "eliminar-favorita"
  const botones = document.querySelectorAll('.agregar-favorita, .eliminar-favorita');

  // Iterar sobre cada botón y agregar un evento de clic
  botones.forEach((boton) => {
      boton.addEventListener('click', (event) => {
          event.preventDefault(); // Evitar el comportamiento predeterminado del botón

          const viviendaId = boton.dataset.vivienda;

          // Crear una instancia de XMLHttpRequest
          const conexion = new XMLHttpRequest();

          // Determinar la acción según la clase del botón
          let url, nuevaClase, nuevaImagen;
          if (boton.classList.contains('agregar-favorita')) {
              url = '/agregarFavorita';
              nuevaClase = 'eliminar-favorita';
              nuevaImagen = "{{ url_for('serve_static', filename='PNG/corazonLLeno.png') }}";
          } else {
              url = '/eliminarFavorita';
              nuevaClase = 'agregar-favorita';
              nuevaImagen = "{{ url_for('serve_static', filename='PNG/corazonVacio.png') }}";
          }

          // Configurar la solicitud
          conexion.open('POST', url);
          conexion.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

          // Configurar la función de respuesta
          conexion.onload = function() {
              if (conexion.status === 200) {
                  // La solicitud se completó exitosamente, puedes realizar acciones adicionales si es necesario

                  // Cambiar la clase y la imagen del botón
                  boton.classList.remove('agregar-favorita', 'eliminar-favorita');
                  boton.classList.add(nuevaClase);
```

Imagen 6 - Actualizar botones JS I

```
                  const imagen = boton.querySelector('img');
                  imagen.src = nuevaImagen;

                  console.log('Acción realizada correctamente');
              }
          };

          // Enviar la solicitud con el ID de la vivienda
          conexion.send(`vivienda_id=${viviendaId}`);
      });
  });
</script>
```

Imagen 7 - Actualizar botones JS II

El otro caso donde se utiliza JS en la aplicación es a la hora de buscar usuarios en la interfaz AscenderUsuario. En esta interfaz, se muestra una tabla en la que aparecen los datos de los usuarios del sistema. Existe una barra de búsqueda que permite filtrar los datos mostrados en la tabla. Para ello se ha definido una función en JS que cada vez que se es-

cribe una letra en esta barra, solo se muestran en la tabla los datos de los usuarios cuyo nombre de usuario coincide total o parcialmente con el texto introducido en la barra de búsqueda.

```
<script>
    function buscarUsuarios() {
        var input = document.getElementById("busqueda");
        var filtro = input.value.toUpperCase();
        var tabla = document.getElementById("tablaUsuarios");
        var filas = tabla.getElementsByTagName("tr");

        for (var i = 0; i < filas.length; i++) {
            var celdaUsuario = filas[i].getElementsByTagName("td")[0];
            if (celdaUsuario) {
                var textoUsuario = celdaUsuario.textContent || celdaUsuario.innerText;
                if (textoUsuario.toUpperCase().indexOf(filtro) > -1) {
                    filas[i].style.display = "";
                } else {
                    filas[i].style.display = "none";
                }
            }
        }
    }
</script>
```

Imagen 8 - buscarUsuarios () JS

5. Módulo de gestión

El modulo de gestión se encarga de llevar la lógica de la aplicación. Conecta las solicitudes del usuario realizadas a través del módulo de vista con el módulo de persistencia. Para ello se ha desarrollado una API-REST, que define uri de acceso a determinadas funciones. A través de estos puntos de acceso es que el usuario es capaz de realizar las peticiones.

Además de la API, este módulo esta conformado por otro elemento, que es el servidor WSGI de prueba que implementa flask. Es el encargado de hace llegar las solicitudes desde la interfaz hasta la API. Sería conveniente en un futuro implementar otro servidor, ya que este servidor está pensado para tareas de desarrollo, pero debido a las limitaciones del proyecto no se ha llevado a cabo. Es una línea de trabajo en el futuro.

Por esta razón, en este apartado solo se explicará la configuración de la API, siendo el servidor conceptualmente un sistema de caja negra.

5.1. Configuración de la API

Para crear un proyecto de flask existen dos archivos que son esenciales. En la estructura del proyecto se deben crear:

1. Un archivo python que se corresponda con el código relativo a la API
2. Una carpeta templates, donde se almacenen las interfaces gráficas. Todas las interfaces descritas en el módulo de vista se incluyen dentro de esta carpeta.
3. Opcionalmente, se puede crear una carpeta static. En el proyecto esta carpeta es utilizada para almacenar imágenes que se usan en la aplicación, pero también se pueden guardar los archivos css y javaScript. Como se ha descrito en el apartado Modulo de vista, en el caso de este proyecto los estilos css y el código javaScript se encuentra dentro del propio html.

En el archivo python correspondiente a la API se usan una serie de bibliotecas que son fundamentales para la implementación. Destaca, como es obvio, flask y todas sus funcionalidades, que son la base del desarrollo de la APi, pero también se utilizan otras bibliotecas complementarias. Más adelante se explicará la función de cada una de ellas.

```
from flask import Flask, render_template, request, redirect, url_for,send_from_directory
from flask_login import LoginManager, UserMixin, login_user, logout_user, login_required,current_user
from cryptography.fernet import Fernet
import os
from flask_login import logout_user
import accesoBaseDatos
```

Imagen 9 - Bibliotecas utilizadas en la API

Todos los puntos de acceso definidos por la API que se comunican con el módulo de persistencia lo hacen a través de funciones definidas por el archivo accesoBaseDatos.py. Este archivo y sus funciones son explicados en su correspondiente apartado. Además, la mayoría de puntos de acceso utilizan una función cuyo valor de retorno es la renderización de una de las interfaces, pasando ciertos parámetros en algunos casos para que sean mostrados en la misma. Esto lo hacen gracias a la función render_template proporcionada por Python. Además, en algunos casos, se redirecciona a otro endpoint diferente. Para ello se utiliza la función redirect, también proporcionada por flask.

UserMixin es una clase proporcionada por la extensión flask_login que se utiliza para simplificar la implementación de la autenticación de usuarios es en una aplicación web. Se usa como mezcla de la clase de Usuario de la aplicación. Para identificar a los usuarios de la aplicación se ha creado una clase Usuario con los valores de id y role, para identificar a los usuarios y establecer el sistema de privilegios.

```python
class User(UserMixin):
    """
    Define la clase usuario de mi aplicación
    """
    def __init__(self, id,role):

        """
        :id id del usuario
        :role rol del usuario
        """

        self.id = id
        self.role = role
```

Imagen 10 - Clase User

Las variablecurrent_useridentifia al usuario actual del sistema. A través de esta variable se puede acceder a los parámetros id y role del usuario, para identificarlo y establecer las restricciones o permisos de su rol. En el proyecto hay dos roles almacenados en las variables ADMIN_ROLE y USER_ROLE.

Además, algunos de los endpoints definidos tienen el decorador @login_required, que solo permite el acceso a usuarios que hayan iniciado sesión en el sistema.

Se deben de crear dos variables, App y Secret_key para completar la configuración de la API y que esta pueda funcionar. Estas variables representan:

- 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 11 - 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 12 - Uso de las variables app y secret_key

Es necesario asociar el administrador de inicio de sesión con la aplicación, mediante la instanciación de la clase Login-Manager proporcionada por flask_login.

Además de las funciones ya descritas y las definidas para los puntos de acceso de la API, se define una función para cargar los datos del usuario. Se asigna esta función al login_manager de la aplicación para cargar los usuarios

```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
```

Imagen 13 - Función load_user

5.2. Puntos de acceso

A continuación, se procede a describir cada uno de los puntos de acceso de la API. En muchos casos, la documentación incluida en el código describe completamente la funcionalidad.

- mostrar_viviendas: Método get. En el body puede haber hasta 3 parámetros opcionales referentes al precio máximo, precio mínimo y ciudad. Son parámetros para aplicar al filtro de la búsqueda. Se accede desde las interfaces principales, tanto de usuarios no registrados como registrados.

```python
@app.route('/mostrar_viviendas', methods=['GET'])
def mostrar_viviendas():

    """
    Define el endpoint para mostrar las viviendas en función de los filtros introducidos en la interfaz
    """

    # Obtener los parámetros de la URL
    page = int(request.args.get('page', 1))
    precio_min = request.args.get('precio_min', '')
    precio_max = request.args.get('precio_max', '')
    ciudad = request.args.get('ciudad', '')
    viviendas, total_paginas=accesoBaseDatos.obtenerViviendas(page,precio_min,precio_max,ciudad)
    ciudades=accesoBaseDatos.obtenerCiudades()


    if current_user.is_authenticated:
        viviendasActualizadas=accesoBaseDatos.actualizarViviendas(viviendas,current_user.id)
        return render_template('MostrarViviendasRegistrado.html',viviendas=viviendasActualizadas, ciudades=ciudades, ciudad_actual=ciudad, page=page, total_paginas=total_
    else:
        return render_template('MostrarViviendas.html', viviendas=viviendas, ciudades=ciudades, ciudad_actual=ciudad, page=page, total_paginas=total_paginas, precio_min_a
```

Imagen 14 - Endpoint /mostrar_viviendas

- Register: Permite los métodos get y post. El método GET se utiliza para desde la interfaz principal de usuario no registrado para renderizar la interfaz de register. El método POST se utiliza para realizar el registro del usuario, incluyendo en el cuerpo de la petición los parámetros necesarios para realizarlo. En caso de que no se encuentren todos los parámetros necesarios, se renderiza de nuevo la interfaz register con un error.

```python
@app.route('/register', methods=['GET', 'POST'])
def register():


    """
    Define el endpoint para registrar a un usuario en función de los parámetros introducidos en la interfaz
    """
    if request.method == 'POST':
        name = request.form['name']
        email = request.form['email']
        username = request.form['username']
        password = request.form['password']
        confirm_password = request.form['confirm_password']
        rol=USER_ROLE
        # Validar que las contraseñas coincidan
        if password != confirm_password:
            error = 'Las contraseñas no coinciden'
            return render_template('register.html', error=error)

        # Crear una conexión a la base de datos
        control= accesoBaseDatos.registro(name, email, username, password)
        if control != 0:
            return render_template('register.html', error=control)
        else:
            return redirect(url_for('Login'))

    return render_template('register.html')
```

Imagen 15 - endpoint /register

- MainRegistered: Define el punto de acceso para renderizar la interfaz principal de un usuario registrado en función de su role

```python
@app.route('/MainRegistered')
@login_required
def MainRegistered():

    """
    Define el endpoint principal de un usuario registrado
    """
    if current_user.role == USER_ROLE:
        return render_template('PagInicioRegistrado.html',user=current_user)
    else:
        return render_template('PagInicioRegistradoAdmin.html',user=current_user)
```

Imagen 16 - endpoint /MainRegistered

- Login: Permite los métodos get y post. Funcionamiento similar a /register pero con los datos del formulario de inciar sesión y su interfaz.

```python
@app.route('/Login', methods=['GET', 'POST'])
def Login():

    """
    Define el endpoint para el inicio de sesión de un usuario
    """

    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        existing_user=accesoBaseDatos.iniciarSesion(username,password)
        if existing_user != 0:
            user_id, role= existing_user
            user = User(user_id, role)
            login_user(user)
            return redirect(url_for('MainRegistered',user=user))
        else:
            error = "Invalid username or password."
            return render_template('IniciarSesion.html', error=error)

    return render_template('IniciarSesion.html')
```

Imagen 17 - endpoint /Login

- / : Permite el método get. Se corresponde con el endpoint principal de la aplicación. Renderiza la interfaz principal de un usuario no registrado.

```python
@app.route('/')
def main():
    """
    Endpoint principal de la aplicación
    """
    return render_template('PagPrincipal.html')
```

Imagen 18 - endpoint /

- VerDatos: Permite el método GET. Renderiza la interfaz VerDatos con los datos del usuario actual.

```python
@app.route('/VerDatos')
@login_required
def verDatos():

    """
    Define el endpoint para mostrar los datos del usuario actual
    """
    # Obtener los datos del usuario desde la base de datos



    userData=accesoBaseDatos.datosUsuario(current_user.id)
    return render_template('VerDatos.html', user=userData)
```

Imagen 19 - endpoint /VerDatos

- CambiarContraseña: Permite los métodos GET y POST. Utilizando el método GET renderiza la interfaz CambiarContraseña. Con el método POST se intenta cambiar la contraseña del usuario con los datos del body de la petición. Si hay algún error, se renderiza la interfaz CambiarContraseña. Si el cambio de contraseña se realiza de manera exitosa se redirecciona al endpoint /verDatos.

```python
@app.route('/cambiarContraseña',methods=['GET', 'POST'])
@login_required
def cambiarContraseña():
    """
    Define el endpoint para el cambio de contraseña del usuario actual
    """

    user=current_user

    if request.method=='POST':
        oldPassword=request.form['oldPassword']
        newPassword=request.form['newPassword']
        repeatPassword=request.form['repeatPassword']
        if repeatPassword != newPassword:
            return render_template('CambiarContraseña.html',user=user,user_id=user.id, error="La contraseña no ha sido confirmada con exito.")
        else:
            control=accesoBaseDatos.cambiarPass(user.id,oldPassword,newPassword)
            if control:
                return redirect(url_for('verDatos', user_id=user.id))
            else:
                return render_template('CambiarContraseña.html', user=user,error="La contraseña antigua no es correcta")
    return render_template('CambiarContraseña.html',user=user)
```

Imagen 20 - endpoint /cambiarContraseña

- AscenderUsuario: Permite el método Get. En caso de que sea un administrador el que intenta acceder al endpoint, renderiza la interfaz AscenderUsuario con los datos de los usuarios del sistema. De no ser así, redirecciona al endpointMainRegistered

```python
@app.route('/ascenderUsuario',methods=["GET","POST"])
@login_required
def ascenderUsuario():
    """
    Define el endpoint para la funcionalidad de ascender usuarios. En este caso se muestra una lista de los usuarios existentes
    """
    if current_user.role==ADMIN_ROLE:

        if request.method=="GET":
            usuarios=accesoBaseDatos.obtenerUsuarios()
            return render_template('AscenderUsuario.html',usuarios=usuarios)
        return render_template('AscenderUsuario.html')

    else:
        return redirect(url_for('MainRegistered'))
```

Imagen 21 - endpoint /ascenderUsuario

- ConfirmarAscenso: Permite el método POST. Dependiendo del body de la consulta, se renderiza la interfaz ConfirmarAscenso con los datos del usuario incluidos en el body de la consulta o se realiza el ascenso al usuario y se redirecciona a /ascenderUsuario.

```python
@app.route('/confirmarAscenso', methods= ["POST"])
@login_required
def confirmarAscenso():

        """
        Se define el endpoint para confirmar el ascenso de un usuario
        """
        if request.method=="POST":
            user = request.form['user']
            if request.form['action']=="ascenso":
                accesoBaseDatos.ascenderUsuario(user)
                return redirect(url_for('ascenderUsuario'))
            elif request.form['action']=="mostrar":
                usuario=accesoBaseDatos.obtenerUsuario(user)
                return render_template('ConfirmarAscenso.html',user=usuario)
```

Imagen 22 - endpoint /confirmarAscenso

- AgregarFavorita: Permite el método POST. Se añade a la lista de favoritos de el usuario actual la vivienda cuyo id se ha pasado el cuerpo de la petición. Tiene un valor de retorno simbólico.

```python
@app.route('/agregarFavorita',methods=["POST"])
@login_required
def agregarFavorita():

    """
    Define el endpoint para agregar una vivienda a favoritos por parte del usuario actual
    """
    vivienda_id = request.form['vivienda_id']
    user_id= current_user.id
    accesoBaseDatos.agregarFavorita(vivienda_id,user_id)
    return "Vivienda añadida a favoritos"
```

Imagen 23 - endpoint /agregarFavorita

- EliminarFavorita: Idem /agregarFavorita, pero para eliminar la vivienda de la lista.

```python
@app.route('/eliminarFavorita',methods=["POST"])
@login_required
def eliminarFavorita():

    """
    Define el endpoint para eliminar una vivienda de favoritos por parte del usuario actual
    """
    vivienda_id = request.form['vivienda_id']
    user_id= current_user.id
    accesoBaseDatos.eliminarFavorita(vivienda_id,user_id)
    return "Vivienda elimnada de favoritos"
```

Imagen 24 - /endpointeliminarFavorita

- MostrarViviendasFavoritas: Permite el método GET. Muestra una lista de las viviendas favoritas del usuario actual.

```python
@app.route('/mostrarViviendasFavoritas',methods=["GET"])
@login_required
def mostrarViviendasFavoritas():
    """
    Define el endpoint para mostrar las viviendas favoritas del usuario actual
    """
    user_id=current_user.id
    viviendas=accesoBaseDatos.obtenerViviendasFavoritas(user_id)
    return render_template("MostrarViviendasFavoritas.html", viviendas=viviendas)
```

Imagen 25 - endpoint /mostrarViviendasFavoritas

- ViviendasSinVerificar: Permite el método GET. Renderiza la interfaz viviendasSinVerificar con los datos de las viviendas que no han sido verificadas como accesibles. En el caso de que el usuario que intenta acceder no sea un administrador, se redirige al endpoint /MainRegistered.

```python
@app.route('/viviendasSinVerificar', methods=["GET"])
@login_required
def viviendasSinVerificar():
    """
    Define el endpoint para mostrar las viviendas sin verificar a los administradores
    """
    if current_user.role==ADMIN_ROLE:
        if request.method=="GET":
            page = request.args.get('page', default=1, type=int)
            viviendas,total_pages=accesoBaseDatos.obtenerViviendasSinFiltrar(page)

        # Pasar los resultados a la plantilla para su visualización
            return render_template('viviendasSinVerificar.html', viviendas=viviendas, total_pages=total_pages,page=page)

    else:
        return redirect(url_for('MainRegistered'))
```

Imagen 26 - endpoint /viviendasSinVerificar

- VerificarVivienda: Permite el método POST. En el caso de que sea un administrador el que intente acceder, se verifica la vivienda cuya id se incluye en el cuerpo de la petición y se redirige al endpoint /viviendasSinVerificar. En el caso de que no sea administrador, se redirige al enndpoint /MainRegistered

```python
@app.route("/verificarVivienda", methods=['POST'])
@login_required
def verificarVivienda():

    """
    Define el endpoint para verificar una vivienda concreta por parte de un administrador
    """
    if current_user.role == ADMIN_ROLE:
        vivienda_id = request.form['vivienda_id']
        accesoBaseDatos.verificarVivienda(vivienda_id)
        return redirect(url_for("viviendasSinVerificar"))
    else:
        return redirect(url_for('MainRegistered'))
```

Imagen 27 - endpoint /verificarVivienda

- EliminarVivienda: Idem /verificarVivienda, pero eliminando la vivienda de la base de datos del sistema.

```python
@app.route("/eliminarVivienda", methods=['POST'])
@login_required
def eliminarVivienda():

    """
    Define el endpoint para eliminar una vivienda del sistema
    """
    if current_user.role == ADMIN_ROLE:
        vivienda_id = request.form['vivienda_id']
        accesoBaseDatos.eliminarVivienda(vivienda_id)
        return redirect(url_for('viviendasSinVerificar'))
    else:
        return redirect(url_for('MainRegistered'))
```

Imagen 28 - endpoint /eliminarVIvienda

- Static. Permite el método GET. Se debe pasar el argumento filename. Devuelve el archivo pedido, siempre y cuando se encuentre en la carpeta static del proyecto. Se pueden enviar archivos gracias a la función send_from_directory proporcionada por flask.

```python
@app.route('/static/<path:filename>')
def serve_static(filename):

    """
    Define un endpoint para el acceso a archivos estáticos del sistema, como imágenes
    :filename nombre del archivo al que se quiere acceder
    """
    root_dir = os.path.dirname(os.getcwd())
    return send_from_directory(os.path.join(root_dir, 'static'), filename)
```

Imagen 29 - endpoint /static/<path:filename>

- Logout: Permite el método GET. Cierra la sesión del usuario en el sistema y lo redirige al endPoint /, representado por la función Main. Para cerrar la sesión del usuario en el sistema se utiliza la función logout_user proporcionada por flask_login

```python
@app.route('/logout')
@login_required
def logout():

    """
    Define el endpoint para cerrar la sesión de usuario
    """
    logout_user()
    return redirect(url_for('main'))
```

Imagen 30 - endpoint /logout

- InfoVivienda: Permite el método GET. Se le debe pasar un argumento correspondiente al id de la vivienda de la cual se quiere obtener la información. Realiza la lógica para renderizar la interfaz adecuada en función de si el

usuario es administrador o no, renderizando infoViviendaAdmin o infoVivienda respectivamente con los datos de la vivienda en cuestión.

```python
@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 31 - endpoint /infoVIvienda/<int:vivienda_id>

- InfoViviendaSinVerificar: Idem /infoVivienda pero para las viviendas que han sido verificadas. Si el usuario no es administrador se redirige al endpoint /MainRegistered.

```python
@app.route("/infoViviendaSinVerificar/<int:vivienda_id>")
@login_required
def infoViviendaSinVerificar(vivienda_id):
    """
    Define el endpoint para mostrar la información de la vivienda sin verificar seleccionada. Solo accesible por administradores
    :vivienda_id id de la vivienda sin verificar de la que se quiere obtener la información
    """
    if current_user.role==ADMIN_ROLE:
        vivienda=accesoBaseDatos.getViviendaSinVerificarInfo(vivienda_id)
        return render_template('infoViviendaVerificar.html', vivienda=vivienda)
    else:
        return render_template('MainRegistered')
```

Imagen 32 - enpoint /infoViviendaSinVerificar/<int:vivienda_id>

- EditarVivienda: Permite el método POST. Si el usuario no es administrador se le redirige al endpointMainRegistered. Si lo es, se recoge la nueva información de la vivienda almacenada en el cuerpo de la petición, se actualiza la base de datos con esa información y se redirige al endpointInfoVivienda pasándole el id de la vivienda cambiada.

```python
@app.route("/editarVivienda", methods=["POST"])
@login_required
def editarVivienda():

    """
    Define el endpoint para editar la información de la vivienda. La información de la vivienda se encuentra en el request realizado
    """
    if current_user.role == ADMIN_ROLE:
        vivienda_id = request.form.get('vivienda_id')
        titulo = request.form.get('titulo')
        descripcion = request.form.get('descripcion')
        precio = request.form.get('precio')
        ciudad = request.form.get('ciudad')
        accesoBaseDatos.editarVivienda(vivienda_id,titulo,descripcion,ciudad,precio)

        return redirect(url_for('infoVivienda', vivienda_id=vivienda_id))
    else:
        return redirect(url_for('MainRegistered'))
```

Imagen 33 - endpoint /editarVivienda

- EditarViviendaVerificar: Idem /editarVivienda pero para las viviendas que no han sido verificadas aún.

```python
@app.route("/editarViviendaVerificar", methods=["POST"])
@login_required
def editarViviendaVerificar():

    """
    Endpoint para editar la información de una vivienda sin verificar. La información de la vivienda se encuentra en la request.
    """
    if current_user.role==ADMIN_ROLE:
        vivienda_id = request.form.get('vivienda_id')
        titulo = request.form.get('titulo')
        descripcion = request.form.get('descripcion')
        precio = request.form.get('precio')
        ciudad = request.form.get('ciudad')
        accesoBaseDatos.editarViviendaVerificar(vivienda_id,titulo,descripcion,ciudad,precio)

        return redirect(url_for('infoViviendaSinVerificar', vivienda_id=vivienda_id))
```

Imagen 34 - ednpoint /editarViviendaVerificar

6. Módulo de persistencia

El módulo de persistencia es el encargado de garantizar la persistencia de los datos, es decir, debe ser capaz de almacenar y recuperar los datos de una forma duradera. Es el módulo que tiene más componentes del sistema.

6.1. Base de datos del sistema

Para la implementación de la base de datos del sistema se ha utilizado un servidor de bases de datos MariaDB. Se han desarrollado varias tablas:

- Tabla users: Contiene la información de los usuarios.
- Tabla infoViviendas: Contiene la información de las viviendas
- Tabla imagenesViviendas: Almacena la información de las imágenes de las viviendas
- Tabla viviendasFavoritas: Almacena la información de las viviendas favoritas de los usuarios.
- Tabla viviendasSinFiltrar: Almacena la información de las viviendas que no han sido verificadas.

Si se quiere encontrar información más concreta de la estructura de las tablas, se explica con detalle en el Anexo III - Especificación de diseño, donde se hace un diseño de los datos y se explican detalladamente los atributos de cada tabla.

6.2. Acceso base de datos

Para realizar consultas a la base de datos del sistema, se ha implementado un script que define las funciones para el acceso a la base de datos. Este script accesoBaseDatos.py, separa la estructura interna de la base de datos de su funcionalidad, lo que permite a la API acceder a la base de datos sin conocer la estructura concreta de la tabla.

Para cumplir su función, se definen varias funciones que responden a las necesidades de acceso principalmente de la API, pero también de cualquier otro componente que lo necesite. Es la única forma de acceder a la base de datos del sistema.

Para el acceso a la base de datos, se utiliza la biblioteca mysql.connector, que permite mediante unos parámetros de conexión, conectarse al servidor de mariaDB.

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

Imagen 35 - Datos de acceso a la base de datos

Utilizando estos parámetros de acceso, se conecta a la base de datos y utiliza las sentencias típicas de acceso en las bases de datos basadas en mySQL.

6.2.1. Funciones de accesoBaseDatos.py

A continuación, se presentan y explican las funciones definidas por el componente de acceso a la base de datos. Se definen para satisfacer todas las necesidades del sistema. Todas las funciones utilizan los parámetros mostrados anteriormente para el acceso a la base de datos.

- iniciarSesión: Comprueba los datos de inicio de sesión del usuario. Para la contraseña se utiliza la función hash de la misma, ya que se guarda así en la tabla. Comprueba si el dato introducido se corresponde con el email o con el nombre de usuario. En caso de que algo no funcione bien retorna 0. Si no, retorna el id y el role del usuario en cuestión.

```python
def iniciarSesion(user, password):

    """
    Comprueba si los datos utilizados para el inicio de sesión son correctos
    :user nombre de usuario o correo electrónico del usuario
    :password contraseña asociada a user
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()

    cursor.execute('SELECT id, role FROM users WHERE username = %s and password = %s', [user, hashlib.sha256(password.encode()).hexdigest()])
    existing_user_matching = cursor.fetchone()
    cursor.execute('SELECT id, role FROM users WHERE email = %s and password = %s', [user, hashlib.sha256(password.encode()).hexdigest()])
    existing_email_matching=cursor.fetchone()
    if existing_user_matching or existing_email_matching:
            if existing_user_matching:
                return existing_user_matching
            else:
                return existing_email_matching
    else:
        return 0
```

Imagen 36 - Función iniciarSesion (user, password)

- registro: Realiza el registro de un nuevo usuario en el sistema. Retorna 0 si se realiza correctamente. En caso de que el nombre de usuario ya exista, se retorna el error correspondiente.

```python
def registro(name, email, username, password):

    """
    Registra a un usuario en el sistema
    :name nombre del usuario
    :email correo electrónico del usuario
    :username nombre de usuario elegido por el usuario, debe ser único en el sistema
    :password contraseña elegida por el usuario
    """

    connection = mysql.connector.connect(**db_config)

    # Verificar si el usuario ya existe
    cursor = connection.cursor()
    cursor.execute('SELECT id FROM users WHERE username = %s', [username])
    existing_user = cursor.fetchone()
    cursor.close()

    if existing_user:
        error = 'El nombre de usuario ya está en uso'
        return error
    hashed_password = hashlib.sha256(password.encode()).hexdigest()

        # Insertar el nuevo usuario en la base de datos

    cursor = connection.cursor()
    cursor.execute('INSERT INTO users (username, name, email, password,role) VALUES (%s, %s, %s, %s,%s)',
                   (username,name,email, hashed_password,USER_ROLE))
    connection.commit()
    cursor.close()
    return 0
```

Imagen 37 - Función registro (name, email, username, password)

- datosUsuario: Devuelve los datos de un ususario, identificado por el id pasado como parámetro.

```python
def datosUsuario(id):

    """
    Devuelve los datos del usuario solicitado
    :id id del usuario
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM users WHERE id=%s',[id])
    userData=cursor.fetchone()
    return userData
```

Imagen 38 - Función datosUsuario (id)

- cambiarPassword: Cambia la contraseña de un usuario. Es necesario pasar como parámetro la contraseña antigua y comprobar que es correcta antes de realizar el cambio. Si es correcta, se realiza el cambio sin problema.

```python
def cambiarPassword (id, oldPass, newPass):

    """
    Cambia la contraseña del usuario especificado
    :id id del usuario
    :oldPass contraseña antigua
    :newPass nueva contraseña
    """
    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM users WHERE id=%s and password=%s',[id,hashlib.sha256(oldPass.encode()).hexdigest()])
    userPass=cursor.fetchone()
    if userPass:
        cursor.execute('UPDATE users SET password = %s WHERE id = %s', [hashlib.sha256(newPass.encode()).hexdigest(), id])
        connection.commit()
        return True
    else:
        return False
```

Imagen 39 - cambiarPassword (id, oldPass, newPass)

- ascenderUsuario: Cambia el role del usuario especificado por el nombre de usuario pasado como parámetro a administrador.

```python
def ascenderUsuario (username):

    """
    Asciende al usuario pasado como parámetro al rol de administrador
    :username nombre de usuario del usuario que se quiere ascender
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute("UPDATE users SET role=%s WHERE username=%s ",[ADMIN_ROLE,username])
    connection.commit()
    return
```

Imagen 40 - Función ascenderUsuario (username)

- agregarFavorita: Añade a la lista de favoritas del usuario identificado por el id del parámetro user_id la vivienda identificada por vivienda_id.

```python
def agregarFavorita(vivienda_id, user_id):

    """
    Agrega la vivienda especificada a la lista de favoritas del usuario especificado
    :vivienda_id id de la vivienda que se quiere agregar a favoritas
    :user_id id del usuario que realiza la acción
    """


    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()

    cursor.execute("SELECT *  from viviendasFavoritas where vivienda_id=%s and usuario_id=%s",[vivienda_id,user_id])
    favorita=cursor.fetchone()
    if not favorita:
        cursor.execute("INSERT INTO viviendasFavoritas (vivienda_id, usuario_id) VALUES (%s, %s)",[vivienda_id,user_id])
        connection.commit()
        return
```

Imagen 41 - Función agregarFavorita (vivienda_id, user_id)

- eliminarFavorita: IdemagregarFavorita, pero elimina la vivienda de la lista de favoritos del usuario

```python
def eliminarFavorita(vivienda_id, user_id):

    """
    Se elimina de la lista de favoritos del usuario especificado la vivienda especificada
    :vivienda_id id de la vivienda a eliminar de la lista
    :user_id id del usuario que realiza la acción
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()

    cursor.execute("SELECT * FROM viviendasFavoritas WHERE vivienda_id=%s AND usuario_id=%s", [vivienda_id, user_id])
    existe = cursor.fetchall()

    if existe:

        cursor.execute("DELETE FROM viviendasFavoritas WHERE vivienda_id=%s AND usuario_id=%s", [vivienda_id, user_id])
        connection.commit()

    cursor.close()
```

Imagen 42 - Función eliminarFavorita (vivienda_id, user_id)

- obtenerViviendasFavoritas: Obtiene la información de las viviendas favoritas del usuario identificado por el id especificado en el parámetro user_id

```
def obtenerViviendasFavoritas(user_id):

    """
    Se obtiene todas las viviendas que el usuario tiene guardadas como favoritas
    :user_id id del usuario
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()

    cursor.execute("SELECT vivienda_id FROM viviendasFavoritas WHERE  usuario_id=%s", [ user_id])
    favoritas = cursor.fetchall()
    viviendas=[]
    for vivienda in favoritas:
        cursor.execute("SELECT * FROM infoViviendas WHERE id=%s", [vivienda[0]])
        viviendaInfo=cursor.fetchone()
        viviendaAppend=viviendaInfo + (1,)
        viviendas.append(viviendaAppend)
    return viviendas
```

Imagen 43 - Función obtenerViviendasFavoritas (user_id)

- verificarVivienda: Traslada la información de la vivienda sin verificar representada por el id pasado en el parámetro vivienda_id. Además, busca las imágenes a través de la función buscarImagenes del script BuscarImagenes.py. Una vez obtiene los links de las imágenes de esta función, utiliza la función agregarImagenes para introducirlas en la tabla imágenes, relacionando las imágenes con el id de la vivienda en la tabla infoViviendas (viviendas ya verificadas)

```
def verificarVivienda(vivienda_id):

    """
    Se verifica la vivienda especificada como accesible, por lo que se traslada su información de la tabla viviendasSinFiltrar a infoViviendas
    :vivienda_id id de la vivienda en la tabla viviendasSinFiltrar
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()

    cursor.execute("SELECT * FROM viviendasSinFiltrar WHERE id=%s ", [vivienda_id])
    existe = cursor.fetchall()

    if existe:
        precio = existe[0][2].replace("€", "")
        precio = re.sub(r'[^0-9]', '', precio)
        precio_entero = int(precio)

        # Realizar la consulta de inserción en la tabla
        cursor.execute('INSERT INTO infoViviendas (titulo, link, precio, ciudad, descripcion) VALUES (%s, %s, %s, %s, %s)',
                       (existe[0][3], existe[0][1], precio_entero, existe[0][5], existe[0][4]))
        connection.commit()

        cursor.execute("DELETE FROM viviendasSinFiltrar WHERE id=%s", [vivienda_id])
        connection.commit()
        cursor.close()

        #Encontramos las imagenes de la vivienda que acabamos de verificar
        link=existe[0][1]
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()

        cursor.execute("SELECT id FROM infoViviendas WHERE link=%s ", [link])
        id_vivienda = cursor.fetchone()

        links_imagenes=BuscarImagenes.buscarImagenes(link)
        agregarImagenes(links_imagenes, id_vivienda)

        return
```

Imagen 44 - Función verificarVivienda (vivienda_id)

- eliminarVivienda: Elimina la vivienda identificada por el id pasado en el parámetro vivienda_id de la tabla viviendasSinFiltrar. Esta operación es únicamente para las viviendas que no han sido verificadas.

```python
def eliminarVivienda(vivienda_id):

    """
    Se elimina la vivienda de la tabla de la base de datos viviendasSinFiltrar
    :vivienda_id id de la vivienda en viviendasSinFiltrar
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()

    cursor.execute("SELECT * FROM viviendasSinFiltrar WHERE id=%s ", [vivienda_id])
    existe = cursor.fetchall()

    if existe:
        cursor.execute("DELETE FROM viviendasSinFiltrar WHERE id=%s",[vivienda_id])

        connection.commit()

    cursor.close()
    return
```

Imagen 45 - Función eliminarVivienda (vivienda_id)

- obtenerViviendas: Devuelve las viviendas de la base de datos que se correspondan con los parámetros pasados.

```python
def obtenerViviendas(page,precio_min, precio_max, ciudad):

    """
    Se devuelven todas las viviendas que pasen los filtros pasados por parámetro
    :page número de paginación, usado en la interfaz
    :precio_min precio mínimo de las viviendas
    :precio_max precio máximo de las viviendas
    :ciudad ciudad donde buscar las viviendas
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()


    # Consulta SQL para obtener las viviendas filtradas por precio y ciudad
    query = "SELECT * FROM infoviviendas WHERE dispo='s'"
    params = []
    if precio_min:
        query += " AND precio >= %s"
        params.append(precio_min)
    if precio_max:
        query += " AND precio <= %s"
        params.append(precio_max)
    if ciudad:
        query += " AND ciudad = %s"
        params.append(ciudad)

    # Obtener el número total de viviendas
    cursor.execute(query, params)
    viviendas = cursor.fetchall()  # Leer completamente los resultados

    total_viviendas = len(viviendas)

    # Calcular el número de páginas
    viviendas_por_pagina = 20
    total_paginas = (total_viviendas + viviendas_por_pagina - 1) // viviendas_por_pagina

    # Calcular el desplazamiento para la paginación
    offset = (page - 1) * viviendas_por_pagina
```

Imagen 46 - Función mostrarViviendas (page, precio_min, precio_max, ciudad) I

```
    # Consulta SQL para obtener las viviendas paginadas
    query += " LIMIT %s, %s"
    params.extend([offset, viviendas_por_pagina])
    cursor.execute(query, params)
    viviendas = cursor.fetchall()
    return viviendas, total_paginas
```

Imagen 47 - Función mostrarViviendas (page, precio_min, precio_max, ciudad) II

- obtenerCiudades: Devuelve una lista con los nombres de todas las ciudades de las que existe alguna vivienda en la base de datos.

```
def obtenerCiudades():

    """
    Devuelve una colección de ciudades, donde aparecen todas las ciudades de las que tenemos alguna vivienda en la base de datos
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute("SELECT DISTINCT ciudad FROM infoviviendas")
    ciudades = cursor.fetchall()

# Ordenar la lista tratando los valores nulos como un caso especial
    ciudades = sorted(ciudades, key=lambda x: x[0] if x[0] is not None else "")
    return ciudades
```

Imagen 48 - Función obtenerCIudades ()

- actualizarViviendas: Comprueba si las viviendas pasadas en el parámetro viviendas están en la lista de favoritas del usuario identificado por el id pasado en el parámetro user_id. Se utiliza para actualizar la interfaz de mostrarViviendasRegistrado. Se añade un nuevo campo a la información de las viviendas que representa si son o no favoritas.

```
def actualizarViviendas(viviendas,user_id):

    """
    Comprueba si las viviendas pasadas como parámetro aparecen como favoritas del usuario
    :viviendas lista de viviendas que se quieren comprobar
    :user_id id del usuario
    """

    viviendasActualizadas = []
    for vivienda in viviendas:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute("SELECT * from viviendasFavoritas where vivienda_id=%s and usuario_id=%s", [vivienda[0], user_id])
        favorita = cursor.fetchall()
        cursor.close()
        viviendaActualizada = vivienda + (1,) if favorita else vivienda + (0,)
        viviendasActualizadas.append(viviendaActualizada)
    return viviendasActualizadas
```

Imagen 49 - Función actualizarVIviendas (viviendas, user_id)

- obtenerViviendasSinFiltrar: IdemobtenerViviendas, pero sin ningún tipo de filtro en la consulta.

```python
def obtenerViviendasSinFiltrar(page):

    """
    Devuelve las viviendas sin pasar ningún filtro
    :page número de página
    """
    connection= mysql.connector.connect(**db_config)
    cursor=connection.cursor()


    items_per_page = 20  # Número de viviendas por página

# Calcular el índice de inicio y fin para la consulta SQL
    start_index = (page - 1) * items_per_page
    end_index = start_index + items_per_page

# Obtener las viviendas para la página actual
    cursor.execute("SELECT * FROM viviendasSinFiltrar LIMIT %s, %s", [start_index, items_per_page])
    viviendas = cursor.fetchall()

# Obtener el número total de viviendas en la base de datos
    cursor.execute("SELECT COUNT(*) FROM viviendasSinFiltrar")
    total_viviendas = cursor.fetchone()[0]

# Calcular el número total de páginas disponibles
    total_pages = (total_viviendas // items_per_page) + (1 if total_viviendas % items_per_page != 0 else 0)
    return viviendas, total_pages
```

Imagen 50 - obtenerViviendasSinFiltrar (page)

- load_user: Función definida user_loader del Login_Manager de flask. Devuelve los parámetros especificados en la clase usuario definida en flask, en este caso id y role.

```python
def load_user(user_id):

    """
    Devuelve la información del usuario pasado como parámetro
    :user_id id del usuario
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT id, role FROM users WHERE id = %s', [user_id])
    existing_user = cursor.fetchone()
    cursor.close()

    if existing_user:
        return existing_user


    return
```

Imagen 51 - Función load_user (user_id)

- buscarUsuario: Devuelve los datos del usuario identificado por el nombre de usuario pasado como parámetro.

```
    return
def buscarUsuario(nombreBuscar):

    """
    Devuelve los datos del usuario pasado como parámetro
    :nombreBuscar nombre de usuario que se debe buscar
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT username, email FROM users WHERE username=%s',[nombreBuscar])
    userFound=cursor.fetchone()
    return userFound
```

Imagen 52 - Función buscarUsuario (nombreBuscar)

- getViviendaInfo: Devuelve la información de la vivienda identificada por el id pasado como parámetro.

```
def getViviendaInfo(vivienda_id):

    """
    Devuelve la información de la vivienda pasada como parámetro
    :vivienda_id id de la vivienda a buscar
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM infoViviendas WHERE id=%s',[vivienda_id])
    vivienda=cursor.fetchone()
    return vivienda
```

Imgen 53 - Función getViviendaInfo (vivienda_id)

- editarViviendaVerificar: Edita la información de la vivienda sin verficar identificada por el id pasado en el parámetro vivienda_id con los valores del resto de parámetros.

```
def editarViviendaVerificar(vivienda_id,titulo, descripcion,ciudad,precio):

    """
    Edita la información de la vivienda sin verificar seleccionada
    :vivienda_id id de la vivienda en la tabla viviendasSinFiltrar
    :titulo nuevo titulo de la vivienda
    :descripcion nueva descripción de la vivienda
    :ciudad nueva ciudad de la vivienda
    :precio nuevo precio de la vivienda
    """
    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('UPDATE viviendasSinFiltrar SET titulo = %s, descripcion=%s, ciudad=%s, precio=%s WHERE id = %s', [titulo, descripcion,ciudad,precio, vivienda_id])
    connection.commit()
    cursor.close()
    return
```

Imagen 54 - Función editarViviendaVerificar (vivienda_id, titulo, descripcion, precio)

- editarVivienda: Idem editarVIviendaVerificar pero para viviendas ya verificadas.

```
def editarVivienda(vivienda_id,titulo, descripcion,ciudad,precio):
    """
    Edita la información de la vivienda  seleccionada
    :vivienda_id id de la vivienda en la tabla infoVivienda
    :titulo nuevo titulo de la vivienda
    :descripcion nueva descripción de la vivienda
    :ciudad nueva ciudad de la vivienda
    :precio nuevo precio de la vivienda
    """
    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('UPDATE infoviviendas SET titulo = %s, descripcion=%s, ciudad=%s, precio=%s WHERE id = %s', [titulo, descripcion,ciudad,precio, vivienda_id])
    connection.commit()
    cursor.close()
    return
```

Imagen 55 - Función editarVivienda (vivienda_id)

- getViviendaSinVerificarInfo: IdemviviendaInfo pero con una vivienda sin verificar.

```
def getViviendaSinVerificarInfo(vivienda_id):

    """
    Devuelve la información de la vivienda sin verificar especificada
    :vivienda_id id de la vivienda en la tabla viviendasSinFiltrar
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM viviendasSinFiltrar WHERE id=%s',[vivienda_id])
    vivienda=cursor.fetchone()
    return vivienda
```

Imagen 56 - Función getViviendaSinVerificarInfo (vivienda_id)

- verificarViviendasCRON: Verifica la disponibilidad de todas las viviendas almacenadas en el sistema, tanto verificadas como no verificadas. A las verificadas que ya no estén disponibles les cambian el atributo dispo para así indicarlo. Las no verificadas son borradas directamente. Esta pensada para ser ejecuta recursivamente.

```
def verificarViviendasCRON():
    """
    Verifica la disponibilidad de las viviendas. Esta función es accedida únicamente desde un archivo de ejecución recursiva
    """
    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT link FROM infoViviendas')
    links = cursor.fetchall()


    for link in links:
        url=link[0]
        response=requests.get(url)
        if response.status_code!=(200):
            cursor.execute("UPDATE infoViviendas set dispo='N'WHERE link = %s", [url])
            connection.commit()

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('SELECT link FROM viviendasSinFiltrar')
    links = cursor.fetchall()
    cursor.close()
    connection.close()

    for link in links:
        url=link[0]
        response=requests.get(url)
        if response.status_code!=(200):
            cursor.execute("DELETE FROM infoViviendas WHERE link = %s", [url])
            connection.commit()

    return
```

Imagen 57 - Función verificarViviendasCRON ()

- viviendaSpider: Define el punto de acceso para el proceso de scraping. Se añade a la tabla viviendasSinFiltrar la información recogida por el scraper.

```
def viviendaSpider(vivienda,ciudad="No encontrada"):

    """
    Función de acceso para la araña de Scrapy, incluye la información encontrada en la tabla viviendasSinFiltrar
    :vivienda colección de datos de la vivienda que debe de incluir los campos link, title, description y price
    :ciudad parámetro opcional que corresponde a la ciudad de la vivienda. El valor por omisión en No encontrada
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute('INSERT INTO viviendasSinFiltrar (link, titulo, descripcion, precio, ciudad) VALUES (%s, %s, %s, %s, %s)', [vivienda["link"], vivienda['title'], vivie
    connection.commit()
    return
```

Imagen 58 - Función viviendaSpider (vivienda, ciudad = «no encontrada»

- agregarImagenes: Añade a la tabla imagenes las imágenes relacionadas con links_imagenes. Cada tupla es añadida con el parámetro vivienda_id, para relacionar las imágenes con una vivienda en concreto.

```
def agregarImagenes (links_imagenes, vivienda_id):
    """
    Agrega las imágenes pasadas como parámetro a la base de datos
    :links_imagenes colección de los links de las imágenes
    :vivienda_id id de la vivienda en la tabla infoViviendas a la que corresponden las imágenes
    """
    for link in links_imagenes:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute("INSERT INTO imagenesViviendas (link, vivienda_id) VALUES (%s, %s)", (link, vivienda_id[0]))

        connection.commit()
        cursor.close()
    return
```

Imagen 59 - agregarImagenes (links_imagenes, vivienda_id)

- getFotosVivienda: Devuelve las imágenes relacionadas con la vivienda identificada por el id pasado en vivienda_id

```
def getFotosVivienda(vivienda_id):

    """
    Devuelve los links de las imágenes de la vivienda pasada como parámetro
    :vivienda_id id de la vivienda en la tabla infoViviendas
    """
    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute("SELECT link from imagenesViviendas where vivienda_id=%s",[vivienda_id])
    imagenes=cursor.fetchall()
    return imagenes
```

Imagen 60 - Función getFotosVivienda (vivienda_id)

- obtenerUsuarios: Devuelve la información de todos los usuarios del sistema

```
def obtenerUsuarios():

    """
    Devuelve una lista con los datos de todos los usuarios del sistema
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute("Select id, username, name, email from users where role=%s",[USER_ROLE])
    usuarios=cursor.fetchall()
    return usuarios
```

Imagen 61 - Función obtenerUsuarios ()

- obtenerUsuario: Devuelve la información del usuario identificado por el id pasado en el parámetro user_id

```
def obtenerUsuario(user_id):

    """
    Devuelve los datos del usuario pasado como parámetro
    :user_id id del usuario
    """

    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    cursor.execute("Select username, name, email from users where id=%s",[user_id])
    usuario=cursor.fetchone()
    return usuario
```

Imagen 62 - Función obtenerUsuario (user_id)

6.3. Sistema scraper

Para realizar el sistema scraper de la aplicación se ha utilizado la biblioteca Scrapy. Para utilizar esta herramienta se debe crear un proyecto Scrapy. Para ello se utiliza la orden;

scrapystartprojectnombre_proyecto

En el caso concreto de este proyecto, nombre_proyecto es Scrappy. La ejecución de esta orden generará la estructura del proyecto con todo lo necesario para funcionar, salvo la creación de la araña. Para crear la araña se debe acceder a la carpeta spiders generada por la anterior orden y crear el script necesario para la araña.

En el código correspondiente a la araña es importante definir una clase Spider proporcionada por scrapy. En esta clase se define el conjunto de url que debe visitar la araña scrapy y el comportamiento de la misma ante estas solicitudes.

Para el conjunto de urls que debe visitar la araña, se aprovecha la estructura de las urls del sitio web, el cual es el siguiente:

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

Donde id es un identificador único que identifica a cada vivienda subida a la página. El último valor de id visitado con éxito se almacena en un fichero llamado config.txt, cuya ruta es la misma que la del script de la araña. En cada ejecución de la araña, esta visita las 1000 url siguientes a la última visitada. Para acceder y guardar el valor de la última vivienda visitada se definen dos funciones, cargarUltimoNumero y guardarUltimoNumero

```python
def cargarUltimoNumero():
    """
    Carga cual es la última vivienda visitada. Este valor se almacena en el archivo definido por ruta_config_txt
    """

    with open (ruta_config_txt,'r') as file:
        content= file.read()
        var= content.split('=')[1].strip()
    return int(var)

def guardarUltimoNumero(valor):

    """
    Guarda el valor de la última vivienda visitada en ruta_config_txt
    """


    with open (ruta_config_txt,'w') as file:
        file.write(f'UltimaAccedida={valor}')
```

Imagen 63 - Funciones cargarUltimoNumero() y guardarUltimoNumero

En cuanto al comportamiento de la araña, esta recoge la información de la url visitada para extraer la información de las viviendas, concretamente el precio, título y descripción de la misma. Después lo pasa por el filtro para saber si es accesible, lo cual se explicará más adelante en el apartado correspondiente al modelo LDA, y si es accesible, busca la ciudad a la que pertenece la vivienda y utiliza el punto de acceso a la base de datos definido para la araña.

```python
if response.status == 200:
    # Extraer los datos de la vivienda
    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 64 - Extracción de datos por parte de la araña

6.4. Modelo LDA y diccionario

Para el filtrado de los datos se ha generado un modelo LDA que contiene un tópico que identifica las viviendas accesibles. La creación de este modelo y el diccionario y su importancia están explicadas más detalladamente en el Anexo III - Especificación de diseño. En resumen, se utilizó una colección de información de viviendas para conformar un diccionario de términos y un modelo LDA a través de las funciones proporcionadas por gensim. Una vez generados, los archivos correspondientes al modelo LDA y el diccionario se almacenan en la misma carpeta que el código de la araña. La araña, para poder utilizar estos dos documentos, debe transformar las descripciones de las viviendas encontradas en un conjunto de tokens. Los tokens son las raíces etimológicas de las palabras de la descripción. Con los tokens es suficiente para que el modelo LDA entienda el mensaje que quiere transmitir la descripción. Para transformar la descripción se definen una serie de funciones y se sigue el siguiente proceso:

```
tokens = limpiar_texto(descripcion)
tokens = tokenizer.tokenize(tokens)
tokens = filtrar_stopword_digitos(tokens)
tokens = stem_palabras(tokens)
```

Imagen 65 - Proceso de transformación de las descripciones

Las funciones mostradas se definen al inicio del script y tienen el siguiente aspecto:

```
def limpiar_texto(texto):
    """
    Función para realizar la limpieza de un texto dado.
    """
    # Eliminamos los caracteres especiales
    texto = re.sub(r'\W', ' ', str(texto))
    # Eliminado las palabras que tengo un solo caracter
    texto = re.sub(r'\s+[a-zA-Z]\s+', ' ', texto)
    # Sustituir los espacios en blanco en uno solo
    texto = re.sub(r'\s+', ' ', texto, flags=re.I)
    # Convertimos textos a minusculas
    texto = texto.lower()
    return texto
```

Imagen 66 - Función limpar_texto (texto)

Para tokenizar el texto resultante de limpiar_texto, se utiliza la herramienta ToktokTokenizer, ofrecida por NLTK, la cual permite la creación de un tokenizer, que transformará el texto pasado a un conjunto de tokens; estos tokens aun son palabras «normales», pero sirve para separar la descripción en un conjunto de palabras. Además, también se utiliza la variable stopwords de nltk, concretamente las del idioma español. Estas stopwords son eliminadas junto con los dígitos en la función filtrar_stopwords_digitos.

```
def filtrar_stopword_digitos(tokens):
    """
    Filtra stopwords y digitos de una lista de tokens.
    """
    return [token for token in tokens if token not in STOPWORDS
            and not token.isdigit()]
```

Imagen 67 - Función filtrar_stopword_digitos (tokens)

Finalmente para reducir los tokens a sus raíces etimológicas, se utiliza la herramienta SnowballStemmer, proporcionada también por NLTK. Para utilizarse se debe crear una herramienta stemmer que use el español y más adelante utilizarlo en la función stem_palabras.

```python
stemmer = SnowballStemmer("spanish")

def stem_palabras(tokens):
    """
    Reduce cada palabra de una lista dada a su raíz.
    """
    return [stemmer.stem(token) for token in tokens]
```

Imagen 68 - Creación stemmer y función stem_palabras (tokens)

De esta manera, el conjunto de tokens resultante ya puede usarse conjuntamente con el modelo LDA y el diccionario para saber si la vivienda es accesible. Para ello, se define una función topicoAccesible, en la que dado unos tokens de entrada valora si esos tokens tienen un valor lo suficientemente alto en el tópico seleccionado.

```python
def topicoAccesible (vivienda):
    """
    En esta función se decide si la descripción de una vivienda se corresponde con la de una vivienda accesible
    :vivienda Conjunto de tokens presente en la descripción de la vivienda
    """


    bow_articulo_nuevo = diccionario.doc2bow(vivienda)
    i=0

    for topico in lda[bow_articulo_nuevo]:
        i+=1
        if topico[0] == 67: #Valor del tópico de nuestro modelo LDA que contiene viviendas accesibles


            if topico[1] > 0.015: #Valor de corte

                return True

            else:

                return False

    return False
```

Imagen 69 - Función topicoAccesible (vivienda)

Si la vivienda pasa este filtro, se le busca en el conjunto de tokens una serie de tokens clave para acotar la cantidad de viviendas erróneas que pasan el filtro. Una vez realizada esta segunda parte del filtro, se añade la información al sistema a través del punto de acceso a la base de datos definido para la araña scrapy.

6.5. VerificarViviendas.py

Este script simplemente se utiliza para la ejecución de la función verificarViviendasCRON de forma periódica.

```python
import accesoBaseDatos

accesoBaseDatos.verificarViviendasCRON()

```

Imagen 70 - VerificarViviendas.py

6.6. BuscarImagenes.py

Este script define una función, la cual es únicamente accedida a través de accesoBaseDatos.py al verificar una vivienda como accesible, la cual busca las imágenes relacionadas con la vivienda cuyo link se pasa como parámetro. Para ello, utiliza BeautifulSoup para encontrar los componentes que sean de la clase image en el html de la página. En estos elementos con la clase image, hay un atributo que guarda el link de la imagen, que es lo que la función recoge.

```python
import requests
from bs4 import BeautifulSoup

def buscarImagenes(link):
    """
    Busca las imágenes presentes en el link
    :link link de la página de la vivienda
    """

    response = requests.get(link)

    if response.status_code == 200:

        soup = BeautifulSoup(response.content, 'html.parser')

        divs = soup.find_all('div', class_='image')

        enlaces_imagenes = [div.img['data-lazy'] for div in divs]
        return enlaces_imagenes
    else:
        return []
```

Imagen 71 - BuscarImagenes.py

6.7. Cron

Para la ejecución periódica tanto de la araña como de VerificarViviendas.py se debe de crear un archivo cron que permita la ejecución periódica de los mismos. Concretamente, estos dos scripts se ejecutan cada hora. Debido a las características de despliegue del proyecto, las cuales se explican más adelante, esta información se almacena en un fihcero-cronjob.

```
0 * * * * root python /app/VerificarViviendas.py
0 * * * * root python -m scrapy crawl todopisos
```

Imagen 72 - Archivo cronjob

7. Modelo de despliegue

Para el modelo de despliegue de la aplicación se ha decidido utilizar contenedores de software. Concretamente, se ha decidido utilizar la herramienta Docker. Los contenedores de software almacenan todo lo necesario para su ejecución, incluyendo bibliotecas, herramientas del sistema, código y tiempo de ejecución. Se ha elegido este modelo por varias razones:

- Rapidez. Dockerizar una aplicación web es un proceso relativamente rápido, es fácil de probar y es fácil aprender a usarla. Debido a las limitaciones temporales del proyecto era un modelo ideal en este aspecto.
- Escalabilidad. Las imágenes generadas por Docker pueden implementarse en cualquier ordenador con cualquier sistema operativo. Esto permite ajustar la escalabilidad del proyecto rápidamente según las necesidades que pueda tener el cliente.

Para utilizar Docker, es necesario crear un archivo Dockerfile en el que se especifique las características del que se quiere crear la imagen. Para ello se ha guardado todo lo necesario para el funcionamiento de la aplicación en una carpeta app. Para generar la imagen, el contenido de Dockerfile debe ser el siguiente:

```
# Establece la imagen base de Python
FROM python:3.9

# Establece el directorio de trabajo dentro del contenedor
WORKDIR /app

# Copia todos los archivos de la aplicación al contenedor
COPY ./app .

# Instala las dependencias
RUN pip install --no-cache-dir -r requirements.txt

# Copia el archivo cronjob al contenedor
COPY cronjob /etc/cron.d/cronjob

# Otorga los permisos adecuados al archivo cronjob
RUN chmod 0644 /etc/cron.d/cronjob

# Establece el comando para ejecutar el cron y la aplicación Flask
CMD service cron start && python mostrarViviendas.py
```

Imagen 73 - Dockerfile

Donde:

- ./app es la carpeta creada para almacenar el proyecto
- requirements.txt es el documento mostrado en Imagen 1
- cronjob es el documento mostrado en Imagen 72
- mostrarViviendas.py es el archivo donde se define la API de flask, explicada en el módulo de gestión.

Utilizando este DockerFile se puede crear la imagen para poder ejecutar el programa en un contenedor Docker. Esto se consigue gracias a la orden:

```
dockerbuild -t viviendasaccesibles .
```

Donde viviendasaccesibles es el nombre que se le da a la imagen y . representa la ruta donde se encuentra Dockerfile. Esta orden es ejecutada en el mismo directorio donde se encuentra Dockerfile. De esta forma, se crea la imagen de Docker, que puede ser exportada mediante zip o tar. Para ejecutar la aplicación en un contenedor software de Docker, se debe ejecutar la orden:

```
docker run -p 5000:5000 viviendasaccesibles
```

Esta orden ejecuta un contenedor utilizando la imagen viviendasaccesibles y mapea el puerto 5000 del contenedor al puerto 5000 del sistema operativo host. Este puerto es el que utiliza el servidor WSGI de prueba de flask.

8. Vista general del proyecto

En este apartado se procede a proporcionar una visión general del proyecto, donde se especifican los módulos, los componentes que tiene cada módulo y las funciones presentes en cada uno de ellos, a modo de resumen.

Modulo de vista

- Templates
  - AscenderUsuario
    - buscarUsuario()
  - CambiarContraseña
  - ConfirmarAscenso
  - infoVivienda
  - infoViviendaAdmin
  - infoViviendaSinVerificar
  - IniciarSesion
  - MostrarViviendas
  - MostrarViviendasFavoritas
  - MostrarViviendasRegistrado
    - ActualizarBotones
  - PagInicioRegistrado
  - PagInicioRegistradoAdmin
  - PagPrincipal
  - register
  - VerDatos
  - viviendasSinVerificar
- Static
  - corazonLleno.png
  - corazonVacio.png
  - lupa.jpg
  - opciones.png
  - volver.png

Modulo de gestión

- mostrarViviendas.py - Flask
  - generate_secret_key()

Class user(UserMixin)

init (self , id ,role)

load_user (user_id)

mostrar_viviendas()

register()

MainRegistered()

Login()

main()

verDatos()

cambiarContraseña()

ascenderUsuario()

confirmarAscenso()

agregarFavorita()

eliminarFavorita()

mostrarViviendasFavoritas()

verificarVivienda()

eliminarVivienda()

serve_static (filename)

logout()

infoVivienda(vivienda_id)

infoViviendaSinVerificar(vivienda_id)

editarVivienda()

editarViviendaVerificar()

Servidor WSGI proporcionado por Flask

Modulo de persistencia

Servidor base datos MariaDB (tablas)

infoViviendas

viviendasSinFiltrar

users

imagenesViviendas

viviendasFavoritas

AccesoBaseDatos.py

- iniciarSesion(user, password)
- registro(name, email, username, password)
- datosUsuario(id)
- cambiarPassword (id, oldPass, newPass)
- ascenderUsuario(username)
- agregarFavorita(vivienda_id, user_id)
- eliminarFavorita(vivienda_id, user_id)
- obtenerViviendasFavoritas (user_id)
- verificarVivienda(vivienda_id)
- eliminarVivienda (vivienda_id)
- obtenerViviendas(page, precio_min, precio_max, ciudad)
- obtenerCiudades()
- actualizarViviendas(viviendas, user_id)
- obtenerViviendasSinFiltrar(page)
- load_user(user_id)
- buscarUsuario(nombreBuscar)
- getViviendaInfo(vivienda_id)
- editarViviendaVerificar (vivienda_id, titulo, descripcion, ciudad, precio)
- editarVivienda (vivienda_id, titulo, descripcion, ciudad, precio)
- getViviendasSinVerificarInfo(vivienda_id)
- verificarViviendasCRON()
- viviendaSpider (vivienda, ciudad= «No encontrada»
- agregarImagenes(links_imagenes, vivienda_id)
- getFotosVivienda(vivienda_id)
- obtenerUsuarios()
- obtenerUsuario(user_id)

prueba.py - Araña Scrapy

- limpiar_texto(texto)
- filtrar_stopwords_digitos(tokens)
- stem_palabras(tokens)
- topicoAccesible(tokens)
- cargarUltimoNumero()

guardarUltimoNumero(valor)

classTodoPisosSpider(scrapy.Spider)

parse(self, response)

ldamodel

casas.dictionary

VerificarViviendas.py

config.txt