Manual Técnico del Proyecto Sistema de Inventarios

documentación para el manual técnico

fecha de realización junio 28 SISTEMA DE INVENTARIOS v1.0

V.1 Salgado Martínez Yuren Joshua

Contenido

Introducción 4
Objetivo del Proyecto 5
Herramientas utilizadas: 6
Requisitos para iniciar el proyecto. 7
Hardware: 7
Software: 8
Otros Consideraciones: 8
Configuración: 8
Instalación 8
Instalar Django 9
instalación de requerimientos. 9
Instalar dependencias desde requirements.txt 10
Verificar la instalación 11
Instalar PostgreSQL 11
Creación de la Base de Datos 12
Configurar Gunicorn para el deploy. 13
Estructura del Proyecto 14
Estructura 16
Descripción de Archivos Clave 18
urls.py 19
views.py 19
models.py 21
settings.py 22
Uso de la Aplicación 24
Ejecución de la Aplicación 24
Funcionalidades Principales 24
Personalización y Ampliación 25
Gestión de Usuarios 25
Recuperación de Contraseña 27
Acerca de 29
Gestión de Perfil del Usuario 29
Perfil del Usuario 30
Cambio de Contraseña 31
Gestión de Inventario 32

Marcas, Proveedores, Ubicaciones y Tipos 33
Gestión de Recetas 33
Uso de Receta y Resumen 35
Bitácora de Actividades 40
Gestión de Permisos y Grupos de Usuarios 41

Fig. 1 Dependencias del Proyecto 10
Fig. 2 Base de datos del sistema 13
Fig. 3 Diagrama de casos de uso 13
Fig. 4 Configuración de la base de datos 18
Fig. 5 Path para la navegación dentro del sistema 19
Fig. 6 Ejemplo del archivo views.py (eliminación) 20
Fig. 7 Forms.py ejemplo 20
Fig. 8 Imports del forms.py 21
Fig. 9 Import del formulario al views.py 21
Fig. 10 Models.py (Tablas de la base de datos) 22
Fig. 11 settings.py (apps para que funcione el sistema) 22
Fig. 12 settings.py (lenguaje de la aplicación) 23
Fig. 13 settings.py (archivos css, js e imagenes) 23
Fig. 14 settings.py (redirección del inicio, zona horaria) 23
Fig. 15 Registro de usuario 26
Fig. 16 Registro de usuario codigo 26
Fig. 17 Inicio de sesion 27
Fig. 18 Recuperacion de contrasena 28
Fig. 19 Codigo de recuperacion 28
Fig. 20 Acerca de 29
Fig. 21 Inicio del sistema 30
Fig. 22 Perfil del usuario 30
Fig. 23 codigo de perfil del usuario 31
Fig. 24 cambio de contraseña del usuario 31
Fig. 25 codigo de cambio de clave 32
Fig. 26 Tabla items 32
Fig. 27 Tabla de modulos 33
Fig. 28 vista de los diferentes modulos 33
Fig. 29 Registro de nueva receta 34
Fig. 30 código de registro de recetas 34
Fig. 31 Resumen de la receta 36
Fig. 32 Resumen de recetas 36
Fig. 33 Cotizacion de la receta 38
Fig. 34 Cotizar receta 39
Fig. 35 Bitacora de actividades 40

Fig. 36 codigo de la bitacora 41
Fig. 37 Permiso de usuarios 42
Fig. 38 Traduccion de permisos 43
Fig. 39 Codigo de permisos 44
Fig. 40 Codigo de permisos 2 45
Fig. 41 Bitácora de uso de recetas 47
Fig. 42 Uso de recetas 47
Fig. 43 Reporte de recetas 48
Fig. 44 Generacion de reporte 49
Fig. 45 Estilo del reporte 50
Fig. 46 Reporte 51

Introducción

El proyecto "Generación de un sistema de inventarios" consiste en concebir, desarrollar e instalar un Sistema de Gestión de Inventario, fundamentado en un conjunto de operaciones CRUD (Crear, Leer, Actualizar, Eliminar), que se ajuste plenamente a los lineamientos de calidad estipulados por la normativa ISO 9001:2015.
Con relación a las delimitaciones se tiene:

- Alcance funcional: El sistema de inventario cubrirá las operaciones CRUD para gestionar la información de materiales como reactivos químicos, materiales plásticos y suministros diversos utilizados en las actividades diarias de la unidad. Permitirá el registro, seguimiento y actualización de la información relacionada con la recepción, almacenamiento, consumo y caducidad de estos materiales.

- Usuarios y roles: Los usuarios del sistema estarán divididos en tres roles principales: administradores, encargados de inventario y usuarios estándares. Los administradores tendrán acceso completo al sistema para conFig.r parámetros, gestionar usuarios y generar informes. Los encargados de inventario podrán realizar todas las operaciones CRUD relacionadas con los materiales. Los usuarios estándares solo tendrán acceso de lectura para consultar la información del inventario.

- Tipos de materiales a gestionar: El sistema gestionará exclusivamente los materiales mencionados anteriormente: reactivos químicos, materiales plásticos y otros suministros relevantes para las operaciones de la unidad.

- Unidades o áreas involucradas: El sistema estará dirigido al Departamento de Laboratorio y al área de almacenamiento de la unidad, siendo utilizado por el personal responsable de la gestión de inventarios en estas áreas.

- Integración con otros sistemas: El sistema de inventario no requerirá integración con otros sistemas existentes en la organización en esta etapa inicial del proyecto.

- Limitaciones tecnológicas: El sistema de inventario se desarrollará utilizando tecnologías web modernas. Se utilizará una base de datos relacional para almacenar la información del inventario.

- Cronograma y recursos: El proyecto se llevará a cabo en un período de seis meses, con un equipo de desarrollo compuesto por un analista de sistemas, un programador y un diseñador de interfaces. Se asignará un presupuesto específico para cubrir los costos de desarrollo, pruebas y capacitación del personal.

Objetivo del Proyecto

General:

- Concebir, desarrollar e instalar un sistema de gestión de inventario, fundamentado en un conjunto de operaciones CRUD (Crear, Leer, Actualizar, Eliminar), que se ajuste plenamente a los lineamientos de calidad estipulados por la normativa ISO 9001:2015.

Específicos:

- Registro Detallado:
  - Diseñar un formato estandarizado para el registro de materiales que incluya información detallada como nombre del material, número de lote, fecha de adquisición, proveedor, fecha de caducidad (si aplica), y cantidad disponible.
- Optimización de Localización y Seguimiento:
  - Establecer un sistema de etiquetado claro y eficiente en las áreas de almacenamiento para facilitar la rápida identificación y localización de los materiales.
- Generación de Informes y Análisis:
  - Desarrollar un sistema que permita generar informes automáticos sobre el uso de recetas y el costo agrupándolas por fechas.
- Interfaz de Usuario Intuitiva y Segura:
  - Diseñar una interfaz de usuario amigable que requiera un mínimo de formación para su uso.
  - Implementar medidas de seguridad robustas, como roles de usuario y acceso restringido, para proteger la información confidencial y garantizar la integridad de los datos.
- Sistema de alertas:
  - Alertas mediante la cual mostrara cuando el 75% del stock de cada reactivo se haya consumido.
  - Alerta cuando el reactivo vaya a caducar para la compra de nuevas existencias.

Herramientas utilizadas:

- **amqp (5.2.0)**: Implementa el protocolo AMQP (Advanced Message Queuing Protocol). Es utilizado por bibliotecas de mensajería como Celery para comunicarse con servidores de mensajes.
- **asarPy (1.0.1)**: Paquete de Python para manejar archivos ASAR, que son empaquetados de aplicaciones (comúnmente utilizados por Electron).
- **asgiref (3.8.1)**: Implementa ASGI (Asynchronous Server Gateway Interface) que es una especificación para aplicaciones web asincrónicas, usada por Django y otros frameworks.
- **billiard (4.2.0)**: Es un reemplazo de multiprocessing de Python y es una dependencia de Celery.
- **celery (5.4.0)**: Un sistema de colas de tareas distribuido que soporta tareas en tiempo real y programadas. Se integra bien con Django para manejar tareas asincrónicas.
- **cffi (1.16.0)**: Permite la llamada de código C desde Python de una manera eficiente y sencilla. Es usado por muchas bibliotecas que necesitan hacer interfaces con código C.
- **chardet (5.2.0)**: Detección de la codificación de caracteres en archivos o secuencias de bytes.
- **click (8.1.7)**: Un paquete para crear interfaces de línea de comandos (CLI) en Python de manera simple y elegante.
- **click-didyoumean (0.3.1)**: Una extensión para Click que sugiere correcciones para comandos mal escritos en CLI.
- **click-plugins (1.1.1)**: Soporte de plugins para Click.
- **click-repl (0.3.0)**: Añade una interfaz de línea de comandos interactiva (REPL) para aplicaciones de Click.
- **Django (5.0.4)**: Un framework de alto nivel para el desarrollo web en Python que promueve un desarrollo rápido y un diseño limpio y pragmático.
- **kombu (5.3.7)**: Una biblioteca de mensajería para Python que abstrae varios sistemas de mensajes. Es utilizado por Celery para comunicarse con brokers de mensajes.
- **pillow (10.3.0)**: Una biblioteca de procesamiento de imágenes en Python que soporta la apertura, manipulación y guardado de muchos formatos de imagen diferentes.
- **pip (24.0)**: El instalador de paquetes para Python. Permite la instalación y gestión de paquetes y dependencias de Python.
- **prompt_toolkit (3.0.47)**: Una biblioteca para construir aplicaciones de línea de comandos con interfaces de usuario avanzadas, como editores de texto y otros controles interactivos.

- **psycopg2-binary (2.9.9)**: Un adaptador de base de datos PostgreSQL para Python. Es la biblioteca más utilizada para conectar aplicaciones Python con bases de datos PostgreSQL.
- **pycparser (2.22)**: Un analizador sintáctico (parser) de C puro en Python. Es utilizado como dependencia por muchas bibliotecas que interactúan con código C.
- **python-dateutil (2.9.0.post0)**: Proporciona extensiones potentes y flexibles para la manipulación de fechas y tiempos en Python.
- **pytz (2024.1)**: Manejo de zonas horarias. Permite un trabajo sencillo y correcto con zonas horarias en Python.
- **rabbitmq (0.2.0)**: Un cliente RabbitMQ para Python, utilizado para interactuar con el servidor de mensajes RabbitMQ.
- **redis (5.0.6)**: Cliente Redis para Python, utilizado para interactuar con la base de datos en memoria Redis, que se utiliza para almacenamiento en caché y colas de mensajes.
- **reportlab (4.2.0)**: Permite la generación de documentos PDF directamente desde Python.
- **setuptools (65.5.0)**: Una herramienta para gestionar paquetes y sus dependencias en Python. Se utiliza ampliamente para la instalación y distribución de paquetes Python.
- **six (1.16.0)**: Una biblioteca de compatibilidad entre Python 2 y 3. Facilita el soporte de ambos versiones en el mismo código base.
- **sqlparse (0.5.0)**: Un analizador sintáctico (parser) y formateador de SQL en Python.
- **tzdata (2024.1)**: Base de datos de zonas horarias, utilizada para mantener actualizadas las zonas horarias en aplicaciones.
- **vine (5.1.0)**: Implementación de promesas para Python, utilizada como dependencia por Celery.
- **wcwidth (0.2.13)**: Mide la cantidad de espacio de pantalla que una cadena de caracteres ocupará. Es útil para aplicaciones de línea de comandos.

Requisitos para iniciar el proyecto.

Hardware:

1. **Procesador (CPU)**:
    - **Mínimo**: Intel Core i3 / AMD Ryzen 3 o equivalente
    - **Recomendado**: Intel Core i5 / AMD Ryzen 5 o superior
    - Tener múltiples núcleos ayudará en tareas paralelas y procesos asincrónicos.
2. **Memoria (RAM)**:
    - **Mínimo**: 8 GB
    - **Recomendado**: 16 GB o más
    - Suficiente memoria es crucial para manejar múltiples procesos, bases de datos y servidores web.
3. **Gráficos (GPU)**:

    - No es crucial para este tipo de aplicaciones, cualquier GPU moderna servirá.
4. **Conexión a internet:**
    - Se necesita una conexión a internet para poder correr el sistema adecuadamente.

Software:

1. **Sistema Operativo**:
    - **Recomendado**: Ubuntu 20.04 LTS o posterior.
    - Linux es comúnmente preferido para servidores debido a su estabilidad y eficiencia.
2. **Entorno de Desarrollo**:
    - **Python**: Versión 3.8 o superior
    - **Django**: Asegúrate de tener la versión 5.0.4 instalada según tus dependencias.
    - **PostgreSQL**: Base de datos recomendada para producción.
    - **Redis**: Para almacenamiento en caché y colas de mensajes.
    - **RabbitMQ**: Para colas de tareas con Celery.

Otros Consideraciones:

- **Entorno Virtual**: Utiliza virtualenv o pipenv para aislar las dependencias de tu proyecto.
- **Docker**: Considera usar Docker para manejar servicios como PostgreSQL, Redis, y RabbitMQ en contenedores, lo que simplifica la configuración y asegura que tu entorno de desarrollo sea consistente con el de producción.
- **Herramientas de Desarrollo**: Un buen IDE como PyCharm, Visual Studio Code, o Sublime Text puede mejorar la productividad.

Configuración:

1. **PostgreSQL**: Configura PostgreSQL para que utilice una cantidad adecuada de memoria según la cantidad de RAM disponible.
2. **Celery**: Configura los trabajadores (workers) de Celery para que no consuman más recursos de los disponibles.
3. **RabbitMQ y Redis**: Asegúrate de que están configurados correctamente y no están sobrecargados con demasiadas tareas simultáneas.

Instalación

Clonar el Repositorio de GitHub

Instala Git:

- sudo apt-get install git

para comprobar si se instalo correctamente usamos el siguiente comando para ver la versión.

- git --version

Clona el repositorio:

- sudo git clone https://github.com/xexeyt5/residencia-proyect.git

veremos si se clono correctamente el repositorio con el siguiente comando:

- ls

si se clono correctamente debería aparecer una carpetea de nombre “residencia-protect”

Navega al directorio del proyecto:

- cd residencia-proyect

Instalar Python 3.8 o superior

Al instalar la versión de Python debemos cambiar la x por algún numero que sea 8 o superior.

- sudo apt install python3.x

para ver su versión de Python es:

- python3 --version

Instalar Django

Primero, crea un entorno virtual:

- python3.8 -m venv env
- source env/bin/activate

Luego, instala Django 5.0.4:

- pip install django==5.0.4 (debemos de tener el entorno virtual activado)
- en caso de no tener instalado pip usar el siguiente comando: sudo apt install python3-pip
- para ver la version de django si es correcta deberemos usar el siguiente comando:
    - django-admin –version

instalación de requerimientos.

Dentro del proyecto existe un archivo llamado requeriments.txt en donde vienen todas las dependencias usadas en el proyecto anteriormente mencionadas.

```
1   amqp==5.2.0
2   asarPy==1.0.1
3   asgiref==3.8.1
4   billiard==4.2.0
5   celery==5.4.0
6   cffi==1.16.0
7   chardet==5.2.0
8   click==8.1.7
9   click-didyoumean==0.3.1
10  click-plugins==1.1.1
11  click-repl==0.3.0
12  Django==5.0.4
13  kombu==5.3.7
14  pillow==10.3.0
15  prompt_toolkit==3.0.47
16  psycopg2-binary==2.9.9
17  pycparser==2.22
18  python-dateutil==2.9.0.post0
19  pytz==2024.1
20  rabbitmq==0.2.0
21  redis==5.0.6
22  reportlab==4.2.0
23  six==1.16.0
24  sqlparse==0.5.0
25  tzdata==2024.1
26  vine==5.1.0
27  wcwidth==0.2.13
```

Fig. 1 Dependencias del Proyecto

Instalar dependencias desde requirements.txt

Tener activado el entorno virtual

- source myenv/bin/activate

Instala las dependencias desde requirements.txt: Navega al directorio donde se encuentra tu archivo requirements.txt y ejecuta:

- pip install -r requirements.txt

Verificar la instalación

Después de ejecutar el comando anterior, pip instalará todas las dependencias listadas en el archivo requirements.txt

- pip list

Instalar PostgreSQL

Ejecuta este comando para actualizar el repositorio de Ubuntu:

- sudo apt-get update

Ejecuta este comando para instalar la última versión disponible, que la recomendable es la 14, si se requiere cambiar de version cambie el numero por la version que vaya a utilizar.

- sudo apt install postgresql-14 postgresql-contrib

Para comprobar si la instalación de PostgreSQL se ha realizado correctamente, ejecuta lo siguiente

- sudo -u postgres psql

A continuación, ejecuta este comando para comprobar la versión actual de PostgreSQL

- SELECT version();

Comprueba el estado de PostgreSQL

Tras instalar PostgreSQL, comprueba si está habilitado y activo. Utiliza la utilidad de comandos systemctl o service para verificar el estado del servicio

- sudo service postgresql status
- sudo systemctl status postgresql

Conectate a PostgreSQL

- sudo su – postgres

Tu línea de comandos debe comenzar ahora con postgres. Para conectarte a la base de datos PostgreSQL, utiliza el intérprete de comandos PostgreSQL con lo siguiente:

- sudo psql

Crea un nuevo rol y una base de datos en PostgreSQL:

- sudo -i -u postgres
- psql
    - CREATE USER uusmb1 WITH PASSWORD 'G3ZU6e9/';

    - CREATE DATABASE sistema_inventariose;
    - GRANT ALL PRIVILEGES ON DATABASE sistema_inventarios TO uusmb1;
- \q
- Exit

Verificación de la instalación

Creación de la Base de Datos

diagrama de entidad-relación concebido para el sistema de inventarios. Este diagrama representa las entidades (o tablas) en la base de datos y las relaciones entre ellas. Proporciona una vista clara y concisa de la estructura de la base de datos, lo que facilita la comprensión

Descripción breve de las tablas principales:

- usuarios: Almacena información de usuarios.
- permisos: Define tipos de permisos en el sistema.
- proveedor: Contiene datos de proveedores.
- marca: Información sobre marcas de productos.
- bitacora: Registra eventos o acciones en el sistema.
- receta: Almacena recetas.
- item: Contiene información de productos.
- Registro: tabla para registro de items.
- Tipo: muestra los diferentes tipos que puede ser el item.
- Ubicación: muestra las ubicaciones donde puede estar el item

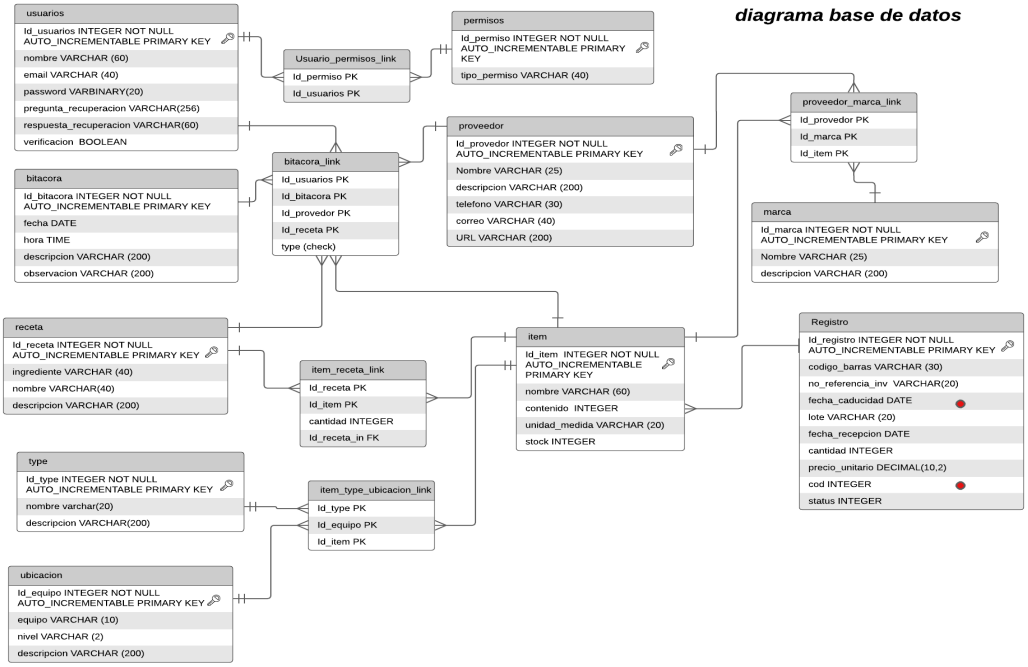


Fig. 2 Base de datos del sistema

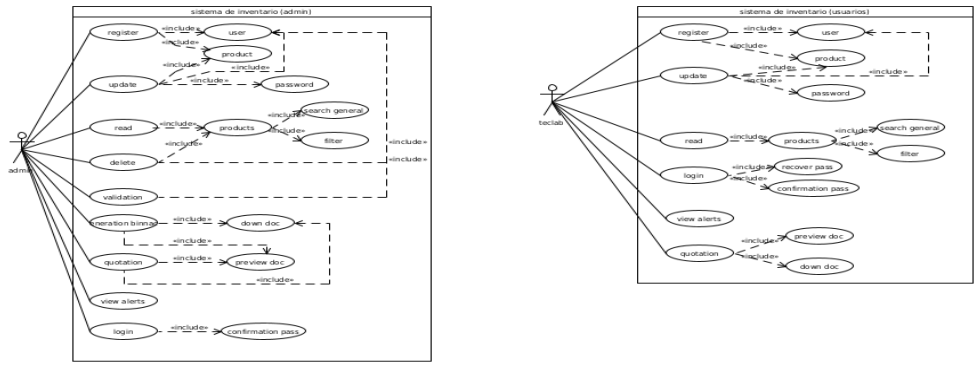


Fig. 3 Diagrama de casos de uso

Configurar Gunicorn, WSGI y Nginx.

Empezamos con la instalación de nginx con el siguiente comando:

- Sudo apt install nginix
- Y vemos si se instaló correctamente con el siguiente comando: nginx -v

Una vez instalado tendremos que meternos al proyecto de django y activar el entorno virtual

- source myenv/bin/activate

estando con el entorno virtual activado tendremos que instalar gunicorn con el siguiente comando:

- pip install gunicorn
- y para ver su version usamos el siguiente comando: gunicorn –version
  o lo vemos con un pip list.

Ahora tendremos que entrar al archivo settings.py de django que se encuentra en la siguiente ruta.

- nano residencia-proyect/Inventario/settings.py

una vez que estemos en el archivo editándolo tendremo que irnos a la parte de HOST en donde colocaremos nuestra dirección IP:

```
DEBUG = True

ALLOWED_HOSTS = ['']
```

```
ALLOWED_HOSTS = ['167.99.192.225']
```

Y ahora debemos crear dentro de residencia-proyect una carpeta llamada conf con el siguiente comando.

- Mkdir conf

Dentro de esa carpeta crearemos un archivo de nombre gunicorn_config.py con el siguiente comando

- Nano conf/gunicorn_config.py

Y una vez dentro del archivo tendremos que colocar las siguientes líneas.

- command = '/home/ubuntut/residencia-proyect/venv/bin/gunicorn'
- pythonpath = '/home/ubuntu/residencia-proyect'
- bind = '167.99.192.225:8000' (dirección IP)
- workers = 3

y guardamos el archivo

después debemos colocar el siguiente comando que sirve para iniciar Gunicorn, utilizando una configuración específica y apuntando al archivo WSGI delproyecto Django:

- /home/ubuntu/residencia-proyect/venv/bin/gunicorn -c /home/ubuntu/residencia-proyect/conf/gunicorn_config.py Inventario.wsgi:application.

Una vez que apliquemos ese comando deberemos iniciar el servicio de nginx con el siguiente comando:

- Sudo service nginx start

Una vez puesto el comando debemos crear un archivo dentro de etc/nginx/sites-available/residencia-proyect

- Sudo nano /etc/nginx/sites-available/residencia-proyect

Y colocamos las siguientes líneas

```
Server {
        Listen 80;
        server_name 167.99.192.225; (dirección IP)

location /static/ {
        root /residencia-proyect/static/;
}
Location{
        proxy_pass http://167.99.192.225:8000; (dirección IP)
}
}
```

Una vez que se haga esto se debe de ir a la siguiente carpeta con el siguiente comando:

- Cd /etc/nginx/sites-enable/

Y de ahí debemos crear una liga con el siguiente comando:

- Sudo ln -s /etc/nginx/sites-available/residencia-proyect

Y verificamos si se hizo de manera correcta con:

- ls -l

ahora reiniciamos nginx.

- Sudo systemctl restart nginx

Y ahora si debemos meternos a la dirección IP asignado.

Estructura del Proyecto

La estructura del proyecto está organizada de manera que sigue las mejores prácticas de Django, facilitando la escalabilidad y sostenibilidad del código.

- **Directorio Principal**: Contiene el archivo manage.py, que es una herramienta de línea de comandos para interactuar con el proyecto Django, y el archivo db.sqlite3, que es la base de datos SQLite donde se almacenan los datos de la aplicación.

- **Carpeta** Inventario: Contiene los archivos de configuración del proyecto, incluyendo settings.py para la configuración global, urls.py para las rutas URL, y wsgi.py y asgi.py para la configuración del servidor web.
- **Carpetas** app **y** user: Estas carpetas contienen el código de las aplicaciones individuales. Cada una de estas carpetas sigue una estructura similar con archivos como models.py para los modelos de la base de datos, views.py para la lógica de las vistas, forms.py para los formularios, y admin.py para la configuración del panel de administración de Django pero especialmente la carpeta user se usó para la creación de un formulario personalizado para el usuario.
- **Carpeta** media: Utilizada para almacenar los archivos subidos por los usuarios, organizados en subdirectorios como user.
- **Carpeta** venv: Contiene el entorno virtual que aisla las dependencias del proyecto, asegurando que las bibliotecas y paquetes utilizados no interfieran con otros proyectos en el mismo sistema.

Estructura

La estructura del proyecto Django está organizada de manera que facilita el desarrollo modular y escalable de aplicaciones web.

```
residencia-proyect/          # Directorio principal del proyecto (contiene todo el proyecto)
├── app/                     # Aplicación principal del proyecto (el cerebro de la aplicación)
│   ├── admin.py             # Configuración del panel de administración
│   ├── apps.py              # Configuración de la aplicación
│   ├── forms.py             # Definición de formularios (es en conjunto de models.py)
│   ├── __init__.py          # Archivo que marca el directorio como un paquete Python
│   ├── migrations/          # Migraciones de la base de datos (cambios en el archivo models.py)
│   ├── models.py            # Definición de modelos (tablas) de la base de datos
│   ├── __pycache__/         # Archivos de caché de Python
│   ├── static/              # Archivos estáticos (CSS, JavaScript, imágenes)
│   ├── templates/           # Plantillas HTML
│   ├── tests.py             # Pruebas unitarias
│   └── views.py             # Definición de vistas (lo que se verá dentro de los templetes)
│
```

```
├── db.sqlite3                  # Base de datos SQLite (local)
│
├── Inventario/                 # Configuración del proyecto Django
│   ├── asgi.py                 # Punto de entrada para servidores ASGI
│   ├── __init__.py             # Archivo que marca el directorio como un paquete Python
│   ├── __pycache__/            # Archivos de caché de Python
│   ├── settings.py             # Configuración global del proyecto (base de datos, horario, lenguaje)
│   ├── urls.py                 # Configuración de rutas URL (para mostrar los templetes)
│   └── wsgi.py                 # Punto de entrada para servidores WSGI
│
├── manage.py                   # Gestor de comandos Django
│
├── media/                      # Archivos multimedia (subidos por usuarios)
│   └── user/                   # Directorio específico para archivos de usuarios
│
├── user/                       # Aplicación secundaria (manejo de usuarios)
│   ├── admin.py
│   ├── apps.py
│   ├── forms.py
│   ├── __init__.py
│   ├── migrations/
│   ├── models.py
│   ├── __pycache__/
│   ├── tests.py
│   └── views.py
│
└── venv/                       # Entorno virtual Python (aisla dependencias)
    ├── bin/                    # Ejecutables para el entorno virtual
    ├── include/                # Archivos de encabezado para el entorno virtual
```

```
├── lib/                    # Bibliotecas instaladas en el entorno virtual
├── lib64 -> lib/           # Enlace simbólico al directorio `lib/`
└── pyvenv.cfg              # Archivo de configuración para el entorno virtual
```

Configura la base de datos:

Abre el archivo *settings.py* del proyecto con la configuración de tu base de datos PostgreSQL.

Por defecto django coloca una base de datos SQLite para que la puedas usar, pero con la siguiente configuración puedes modificarlo para conectarlo a tu base de datos.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'sistema_inventarios',
        'USER': 'uusmb1',
        'PASSWORD': 'G3ZU6e9/',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Fig. 4 Configuración de la base de datos

Y para que se actualice migre correctamente la base de datos de Django a PostgreSQL usa:

python manage.py makemigrations

python manage.py migrate

Ejecuta el servidor de desarrollo: Finalmente, ejecuta el servidor de desarrollo para verificar que todo esté configurado correctamente:

python manage.py runserver

Descripción de Archivos Clave

En esta sección se explicarán los principales archivos y módulos del proyecto Django, detallando su propósito y contenido. Esto incluye los archivos views.py, urls.py, forms.py, models.py, y settings.py, así como la configuración de archivos estáticos dentro del archivo settings.py.

urls.py

El archivo urls.py define las rutas URL de la aplicación y las asocia con las vistas correspondientes. Esto permite a Django saber qué vista debe llamar para una URL determinada.

Donde su estructura es path (“la URL que tendra en la web”, “la funcion que tomara del archivo views.py”, “el nombre para hacer referencia dentro de los templates”)

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name="index"),
    path('accounts/', include('django.contrib.auth.urls')),
    path('salir/', views.salir, name="salir"),
    path('recuperar-contraseña/', views.recuperar_contraseña, name='recuperar_contraseña'),
    path('inventario/', views.inventario),
    path('bitacora/', views.bitacora, name='bitacora'),
    path('marca/', views.marca, name='marca'),
    path('registro-marca/', views.marcaregistro, name='marcaregistro'),
    path('editar-marca/<int:marca_id>/', views.editar_marca, name='editar_marca'),
    path('eliminar-marca/<int:marca_id>/', views.eliminar_marca, name='eliminar_marca'),
    path('proveedores/', views.provedores),
    path('proveedores-registro/', views.proveedoresregistro, name="r_proveedor"),
    path('editar-proveedor/<int:proveedor_id>/', views.editar_proveedor, name='editar_proveedor'),
    path('eliminar-proveedor/<int:proveedor_id>/', views.eliminar_proveedor, name='eliminar_proveedor'),
    path('insertar/', views.registrar_item, name='insertar_r'),
    path('eliminar_registro/<int:registro_id>/', views.eliminar_registro, name='eliminar_registro'),
    path('editar_registro/<int:registro_id>/', views.editar_registro, name='editar_registro'),
    path('asignar-permisos/', views.asignar_permisos, name='asignar_permisos'),
    path('api/group-permissions/<int:group_id>/', views.get_group_permissions, name='get_group_permissions')
    path('crear-grupo/', views.crear_grupo, name='crear_grupo'),
    path('perfil/', views.perfil, name='perfil'),
```

Fig. 5 Path para la navegación dentro del sistema

views.py

El archivo views.py contiene las vistas de la aplicación web, esta vista es una función que toma solicitud web y devuelve una respuesta web. Esta respuesta puede ser el contenido HTML de una página web. Una redirección, un error 404, un documento XML, una imagen o cualquier otro tipo de cosa que se necesite.

Las vistas son lógica detrás de la aplicación y se colocan en el archivo views.py, un ejemplo la creación de formularios dentro de la aplicación en donde por el método POST recibirá los datos dentro de la funcion, tambien dentro de la vista puedes iterar datos y dentro de los templanes llamarlos para visualizar estos datos.

```python
@permission_required('app.view_auth_permission')
@login_required
def eliminar_usuario(request, username):
    try:
        usuario = User.objects.get(username=username)
    except User.DoesNotExist:
        messages.error(request, f"El usuario '{username}' no existe.")
        return redirect('asignar_permisos')

    if request.method == 'POST':
        if request.POST.get('confirmar') == 'true':
            usuario.delete()
            messages.success(request, f"El usuario '{username}' ha sido eliminado.")
        else:
            messages.warning(request, "No se ha eliminado al usuario.")
        return redirect('asignar_permisos')

    return render(request, 'eliminar_usuario.html', {'usuario': usuario})
```

Fig. 6 Ejemplo del archivo views.py (eliminación)

Dentro del views.py se creó una función por si un usuario ya no forma parte del sistema y pueda ser eliminado obteniendo al usuario y verificando si este existe, en dado caso que no mandara un mensaje de error mostrando que el usuario no existe, pero si el usuario existe podrá ser eliminado y mostrara un mensaje de eliminación exitosa.

Todas estas funciones creadas se usarán para el uso de recetas, el registro de los diferentes módulos, la creación de grupos con permisos, para asignar permisos para los usuarios, etc.

forms.py

El archivo forms.py se utiliza para definir formularios en Django. Los formularios pueden ser utilizados para la entrada y validación de datos por parte del usuario. Django proporciona un módulo llamado forms que facilita la creación y gestión de formularios.

```python
class marcaform(forms.ModelForm):
    class Meta:
        model = Marca
        fields = ['nombre', 'descripcion']


class proveedorForm(forms.ModelForm):
    class Meta:
        model = Proveedor
        fields = ['nombre','descripcion','telefono','correo','url']
```

Fig. 7 Forms.py ejemplo

Formularios que creas en forms.py especificando las celdas que quieres que muestre.

```
from .forms import marcaform, proveedorForm, ItemForm, TypeForm, LocationForm
```

Fig. 8 Imports del forms.py

Dentro del views.py para usarlo debemos hacer referencia a ellos con un import y el nombre del formulario que estarás usando.

```
@login_required
@permission_required('app.add_location', raise_exception=True)
def crear_location(request):
    if request.method == 'POST':
        form = LocationForm(request.POST)
        if form.is_valid():
            location = form.save(commit=False)
            location.usuario = request.user
            location.save()
            messages.success(request, 'Se agregó la ubicación correctamente')
            return redirect('listar_locations')
    else:
        form = LocationForm()
    return render(request, 'crear_locations.html', {'form': form})
```

Fig. 9 Import del formulario al views.py

Dentro de las diferentes funciones (backent de la app) estaremos llamando a los formularios que creamos (dentro del archivo views.py de la app) y mediante el método post los datos llenados dentro del formulario los va a solicitar para después guardarlos. Y después te mandara a por medio del return al html/template que hayas colocado.

models.py

El archivo models.py contiene los modelos de la aplicación, que son clases que representan las estructuras de datos y las tablas de la base de datos. Django utiliza un ORM (Object-Relational Mapping) para interactuar con la base de datos a través de estos modelos.

```python
class Type(models.Model):
    nombre = models.CharField(max_length=100)
    descripcion = models.CharField(max_length=255)
    usuario = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True)

    def delete(self, *args, **kwargs):
        self._usuario = kwargs.pop('usuario', None)
        self._descripcion_personalizada = kwargs.pop('descripcion_personalizada', '')
        super().delete(*args, **kwargs)

    def __str__(self):
        return self.nombre

class Location(models.Model):
    equipo = models.CharField(max_length=100)
    nivel = models.CharField(max_length=50)
    descripcion = models.CharField(max_length=255)
    usuario = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True)
```

Fig. 10 Models.py (Tablas de la base de datos)

Dentro de los modelos se estarán definiendo las diferentes tablas de la base de datos en donde mediante el class “nombre de la tabla” (models.model) definiremos el nombre de la tabla, y abajo de se pondrá cada dato que contendrá como el nombre, la descripción y en este caso el usuario que se usa en la bitácora.

settings.py

El archivo settings.py contiene la configuración global del proyecto Django. Aquí se definen ajustes como la configuración de la base de datos, las aplicaciones instaladas, y la configuración de archivos estáticos.

```python
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'app',
    'user',
]
```

Fig. 11 settings.py (apps para que funcione el sistema)

Aplicaciones instaladas que por defecto vendrán unas que funcionen con django a excepción de app (aplicación principal) y user (aplicación para el registro, login de usuarios)

```
LANGUAGE_CODE = 'es'

TIME_ZONE = 'UTC'

USE_I18N = True

USE_TZ = True
```

Fig. 12 settings.py (lenguaje de la aplicación)

en este caso también unas configuraciones ya vienen por defecto con django pero puedes modificar el código de lenguaje para que salga en español.

```
STATIC_URL = 'static/'

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
]
```

Fig. 13 settings.py (archivos css, js e imagenes)

URL para los archivs "static" que de este directorio la aplicación Django es de donde tomara aquellos archivos de estilos (css o scss), archivos js, imágenes, etc.

```
LOGIN_REDIRECT_URL = '/'

MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, "media")

AUTH_USER_MODEL = 'user.User'

TIME_ZONE = 'America/Mexico_City'
```

Fig. 14 settings.py (redirección del inicio, zona horaria)

configuración para que la dentro de la aplicación muestre la zona horaria de mexico, donde se redirija cuando el usuario inicie sesión y el modelo de user que se usara para el tema de los usuarios.

Uso de la Aplicación

En este capítulo se explicará cómo interactuar y utilizar la aplicación desarrollada con Django. Se detallarán los pasos necesarios para ejecutar la aplicación, así como las funcionalidades principales que ofrece a los usuarios.

Ejecución de la Aplicación

Para ejecutar la aplicación Django en un entorno de desarrollo, sigue estos pasos:

1. **Activación del Entorno Virtual:** Si no has activado el entorno virtual previamente, hazlo con el siguiente comando:

```
source venv/bin/activate
```

2. **Iniciar el Servidor de Desarrollo:** Utiliza manage.py para iniciar el servidor de desarrollo de Django:

```
python manage.py runserver
```

Esto iniciará el servidor en http://127.0.0.1:8000/. Abre tu navegador y navega a esta dirección para ver la aplicación en funcionamiento.

Funcionalidades Principales

Algunas de las funcionalidades principales que puedes encontrar en la aplicación:

- **Registro y Autenticación de Usuarios:** La aplicación puede incluir formularios de registro y autenticación de usuarios, utilizando vistas y formularios definidos en views.py y forms.py.
- **Visualización de Datos:** Utiliza las vistas definidas en views.py para procesar y presentar datos desde la base de datos. Puedes utilizar plantillas HTML en el directorio templates/ para renderizar estas vistas.

- **Administración de Datos:** Utiliza el panel de administración de Django (/admin/) para gestionar datos de la base de datos. Configura modelos en models.py y administra registros usando admin.py.
- **Interacción con URL:** Define rutas URL en urls.py para asociar URLs específicas con vistas correspondientes. Esto permite la navegación y acceso a diferentes partes de la aplicación.
- **Gestión de Archivos Estáticos y Multimedia:** Configura archivos estáticos (CSS, JavaScript, imágenes) en settings.py. Utiliza directorios como static/ y media/ para almacenar y servir estos archivos.

Personalización y Ampliación

Para personalizar y ampliar la aplicación:

- **Crear Nuevas Funcionalidades:** Define nuevas vistas, modelos y formularios según los requisitos de la aplicación en app/.
- **Integración de Funcionalidades Externas:** Integra bibliotecas externas y aplicaciones Django de terceros utilizando pip install y configurándolas en settings.py.
- **Optimización y Escalabilidad:** Implementa buenas prácticas de desarrollo, como el uso eficiente de consultas a la base de datos, caché de datos, y técnicas de optimización de rendimiento.

Gestión de Usuarios

Registro de Usuario

- **Descripción:** Permite a los usuarios crear una cuenta nueva en la aplicación.
- **Acción:** Los usuarios pueden completar un formulario con información básica como nombre, correo electrónico y contraseña para registrarse.

Fig. 15 Registro de usuario

```python
def registro(request):
    if request.method == 'POST':
        formulario = CustomUserCreationForm(data=request.POST)
        if formulario.is_valid():
            user = formulario.save(commit=False)  # Crear la instancia del usuario pero no guardar aún
            user.is_active = False  # Desactivar el usuario
            user.save()  # Guardar el usuario con is_active=False

            username = formulario.cleaned_data['username']
            password = formulario.cleaned_data['password1']
            # No es necesario autenticar aquí porque el usuario no está activo
            messages.success(request, "Te has registrado correctamente, pero tu cuenta necesita ser activada por un administrador.")
            return redirect('login')
        else:
            messages.error(request, "El formulario de registro no es válido")
    else:
        formulario = CustomUserCreationForm()

    return render(request, 'registration/registro.html', {'form': formulario})
```

Fig. 16 Registro de usuario codigo

Función del usuario donde por el método POST obtiene los datos del formulario customusercreationform y si el usuario es valido crea una instancia para luego desactivarlo para que no entre dentro del sistema y posteriormente por user.save() los datos del usuario se guardan dentro de la base de datos para después limpiar los campos y mandar un mensaje que se registro correctamente y con return redirect('login') mandaria al usuario al login para iniciar sesión. Django puede gestionar la autenticación para varios usos. También proporciona un sistema de hash de contraseñas configurable y herramientas para restringir el contenido en formularios y vistas.

Login

- **Descripción:** Permite a los usuarios autenticarse en la aplicación.
- **Acción:** Los usuarios ingresan sus credenciales (usuario y contraseña) para acceder a su cuenta.

Fig. 17 Inicio de sesion

Recuperación de Contraseña

- **Descripción:** Permite a los usuarios restablecer su contraseña en caso de olvido.
- **Acción:** Los usuarios pueden solicitar con un su pregunta de verificación para restablecer su contraseña.

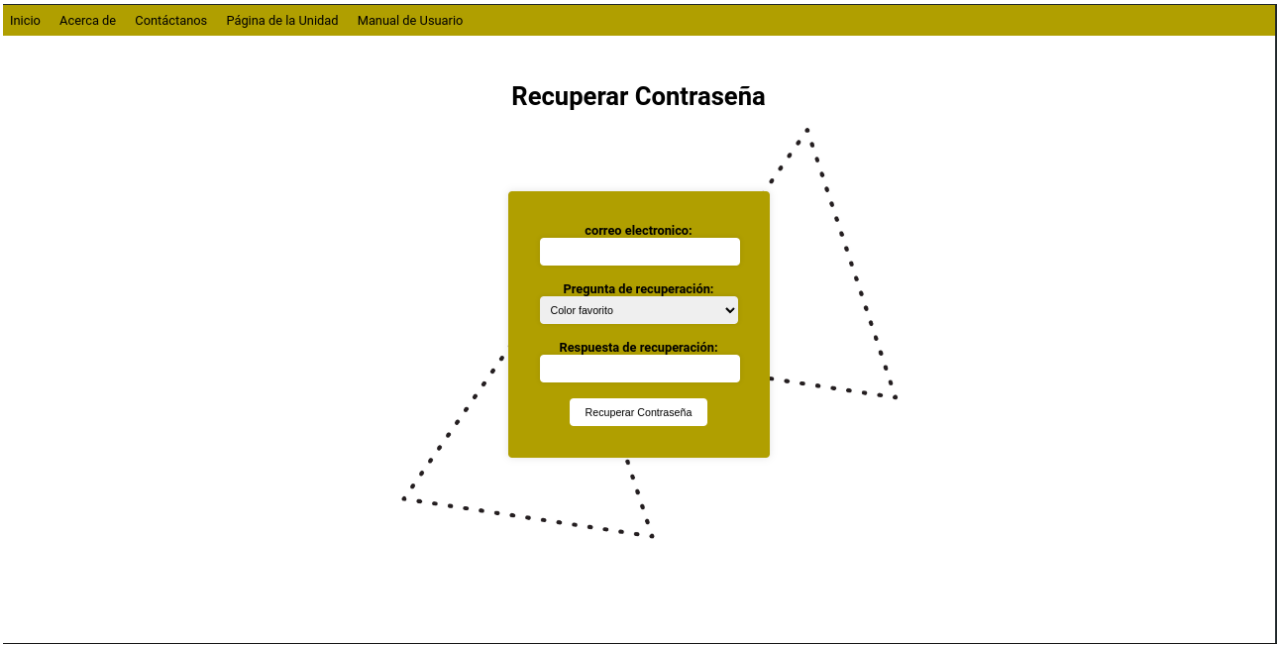


Fig. 18 Recuperacion de contrasena

```
User = get_user_model()
def recuperar_contraseña(request):
    contraseña_desencriptada = None
    mensaje = None
    opciones_pregunta_recuperacion = User.PREGUNTA_RECUPERACION_CHOICES

    if request.method == 'POST':
        email = request.POST.get('email')
        pregunta = request.POST.get('pregunta_recuperacion')
        respuesta = request.POST.get('respuesta_recuperacion')

        try:
            user = User.objects.get(email=email, pregunta_recuperacion=pregunta)

            if respuesta == user.respuesta_recuperacion:
                contraseña_desencriptada = user.plaintext_password

                if not contraseña_desencriptada:
                    mensaje = "No se puede mostrar la contraseña porque está vacía o no está establecida."
                else:
                    mensaje = f"La contraseña de {user.email} es: {contraseña_desencriptada}"
            else:
                mensaje = "La respuesta de recuperación no es correcta."
        except User.DoesNotExist:
            mensaje = "No se encontró ningún usuario con ese correo."

    return render(request, 'recuperar_contraseña.html', {'mensaje': mensaje, 'opciones_pregunta_recuperacion': opciones_pregunta_recuperacion})

def acerca(request):
            return render(request, 'acerca.html')
```

Fig. 19 Codigo de recuperacion

Recuperación de contraseña en donde mediante el método post obtiene el email, pregunta y respuesta, y mediante sentencias verifica si el campo esta vacío, la contraseña es correcta, la respuesta de verificación esta mal o si no se encontró algún correo asociado a ese usuario.

Acerca de

- **Descripción:** Información sobre la aplicación y su propósito.
- **Acción:** Página estática que describe lo que se hace dentro de la unidad (uusmb).

Fig. 20 Acerca de

Gestión de Perfil del Usuario

Inicio

- **Descripción:** Página principal después del login que puede mostrar un resumen o dashboard personalizado.
- **Acción:** Muestra información relevante para el usuario, barra de navegación y una bienvenida al usuario.

Fig. 21 Inicio del sistema

Perfil del Usuario

- **Descripción:** Permite a los usuarios ver y actualizar su información personal.
- **Acción:** Los usuarios pueden editar detalles como nombre, pregunta, información, etc.

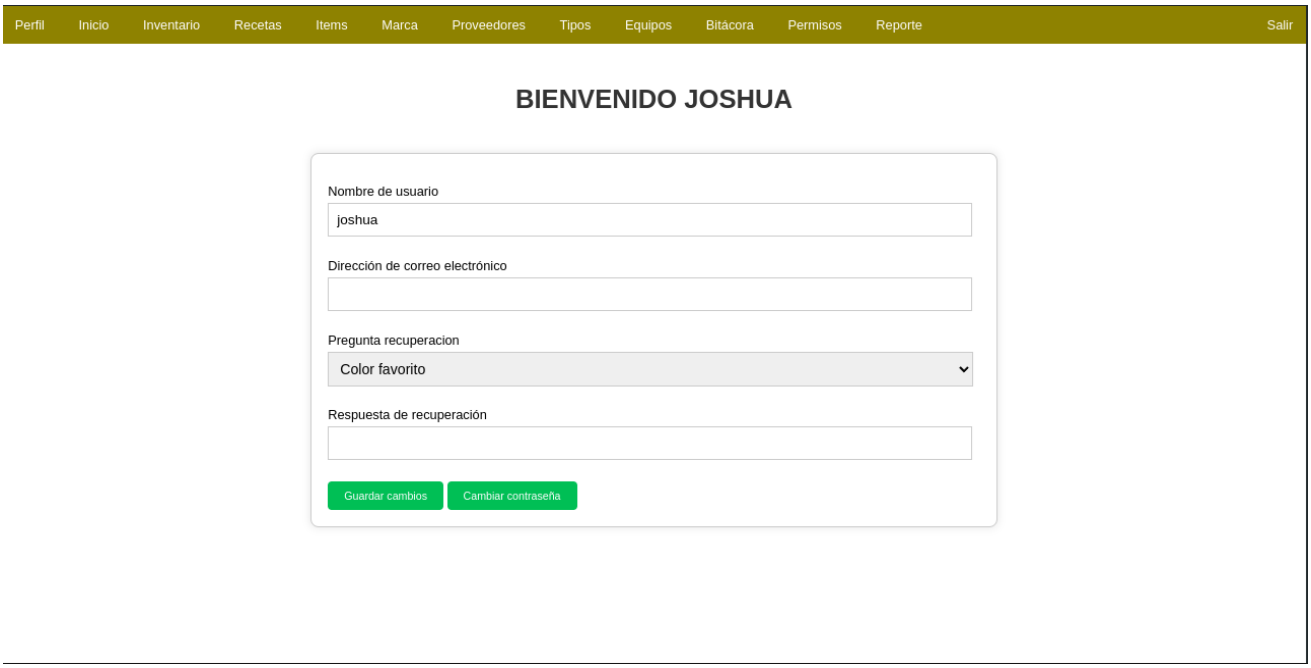


Fig. 22 Perfil del usuario

```python
@login_required
def perfil(request):
    user = request.user
    if request.method == 'POST':
        form = CustomUserChangeForm(request.POST, instance=user)
        if form.is_valid():
            form.save()
            messages.success(request, "Se modificaron tus datos")
            return redirect('perfil')
    else:
        form = CustomUserChangeForm(instance=user)

    return render(request, 'perfil.html', {'form': form})
```

Fig. 23 codigo de perfil del usuario

Una vez que inicias sesión y vas al apartado de perfil mostrara tus datos, en el views se maneja por un formulario que viene del archivo forms.py, donde puedes hacer varios cambios dentro del perfil dependiendo los campos que contenga y una vez cambiado manda un mensaje que se modificaron los datos correctamente.

Cambio de Contraseña

- **Descripción:** Permite a los usuarios cambiar su contraseña actual.
- **Acción:** Los usuarios deben proporcionar su contraseña actual y una nueva contraseña para actualizarla.

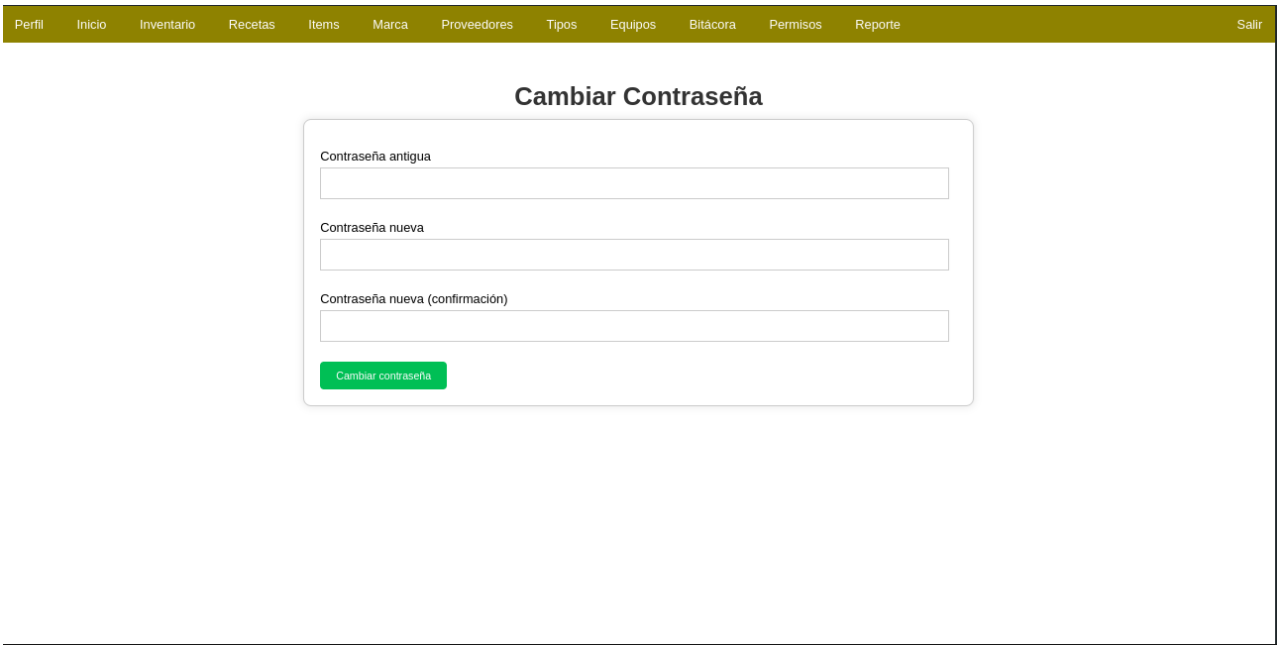


Fig. 24 cambio de contraseña del usuario

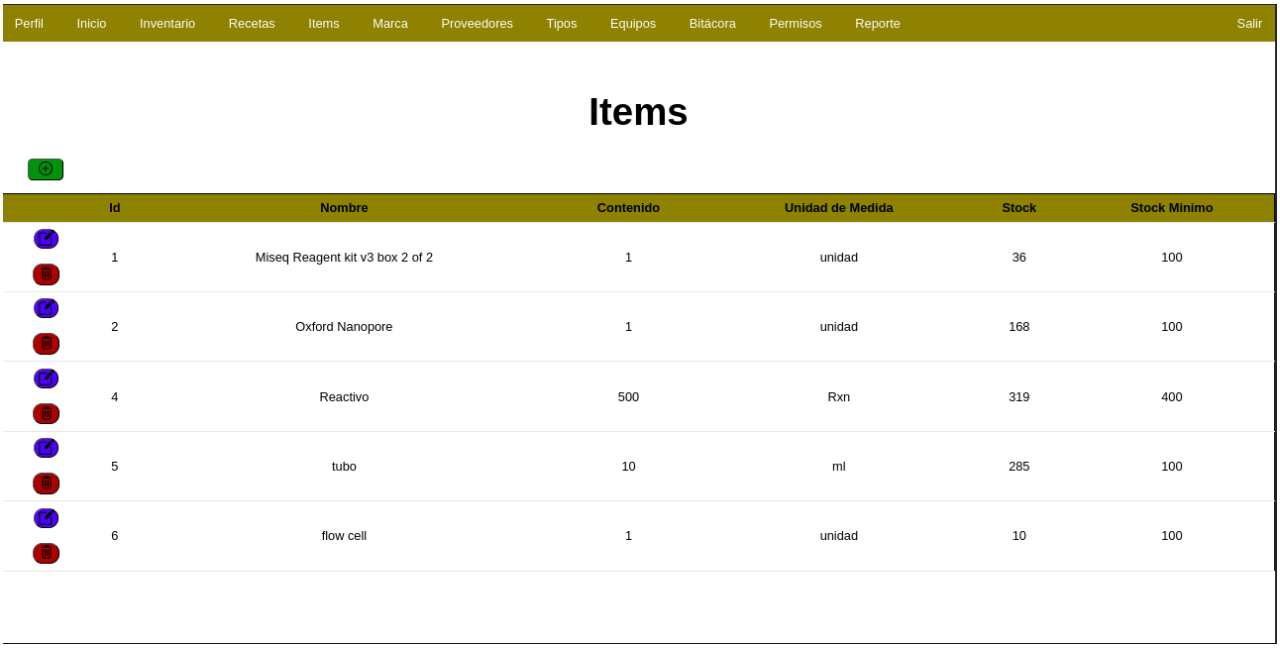
```python
@login_required
def cambiar_contrasena(request):
    if request.method == 'POST':
        password_form = PasswordChangeForm(request.user, request.POST)
        if password_form.is_valid():
            password_form.save()
            messages.success(request, "Contraseña actualizada correctamente")
            update_session_auth_hash(request, password_form.user)
            return redirect('perfil')
    else:
        password_form = PasswordChangeForm(request.user)

    return render(request, 'cambiar_contrasena.html', {'password_form': password_form})
```

Fig. 25 codigo de cambio de clave

Gestión de Inventario

Items

- **Descripción:** Administración de los artículos en el inventario.
- **Acción:** Incluye funciones para agregar, editar y eliminar artículos, junto con la visualización de detalles de cada artículo.



Id	Nombre	Contenido	Unidad de Medida	Stock	Stock Minimo
1	Miseq Reagent kit v3 box 2 of 2	1	unidad	36	100
2	Oxford Nanopore	1	unidad	168	100
4	Reactivo	500	Rxn	319	400
5	tubo	10	ml	285	100
6	flow cell	1	unidad	10	100

Fig. 26 Tabla items

Marcas, Proveedores, Ubicaciones y Tipos

- **Descripción:** Gestión de entidades relacionadas con el inventario (marcas, proveedores, ubicaciones, tipos).
- **Acción:** Cada entidad tiene operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para gestionar registros relacionados.

Fig. 27 Tabla de modulos

```
permission_required('app.view_marca')
@login_required
def marca(request):
    Marcas = Marca.objects.all()
    return render(request, 'marca.html', {'Marcas':Marcas})
```

Fig. 28 vista de los diferentes modulos

Vista en los diferentes modulos (ejemplo con marca) en donde obtiene todos los objetos que tiene la marca y los puede mostrar.

Gestión de Recetas

Registro y Edición de Recetas

- **Descripción:** Definición y modificación de recetas de productos o procesos.
- **Acción:** Los usuarios pueden crear nuevas recetas, editar recetas existentes, especificar cantidades necesarias de ingredientes u otros pasos.

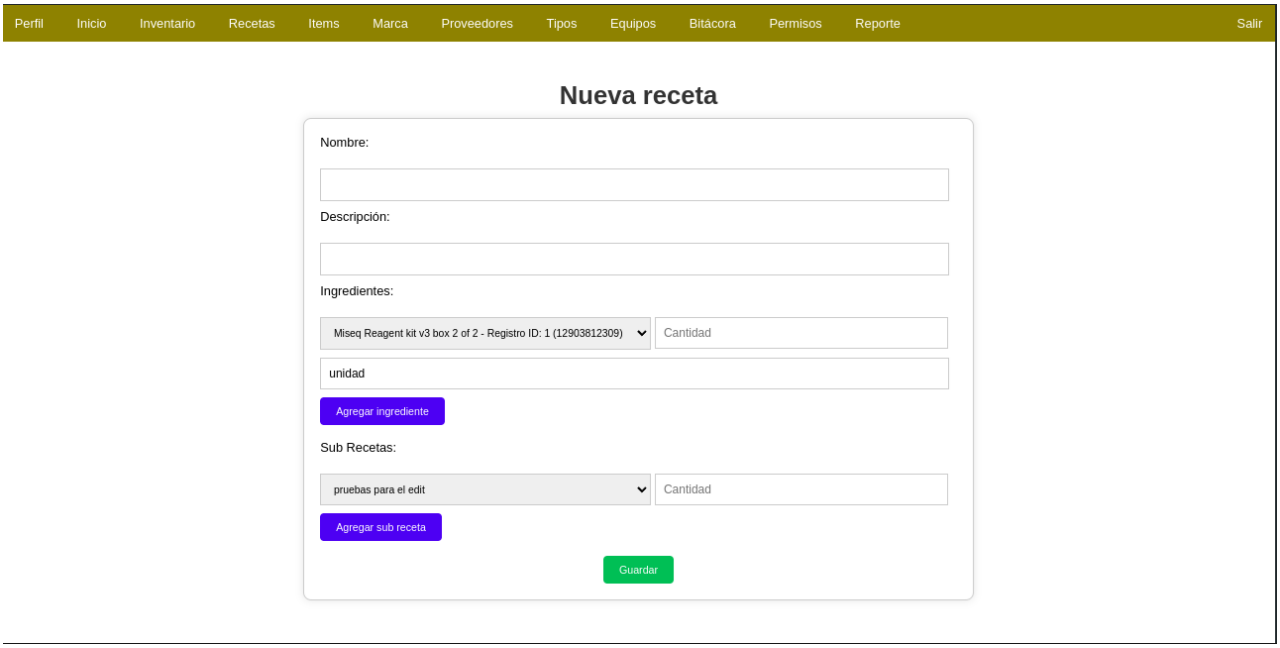


Fig. 29 Registro de nueva receta

```python
@permission_required('app.add_receta')
@login_required
def Recetas_registrar(request):
    if request.method == 'POST':
        nombre = request.POST.get('nombre')
        descripcion = request.POST.get('descripcion')
        ingredientes = request.POST.getlist('ingredientes')
        cantidades = request.POST.getlist('cantidades')
        subrecetas = request.POST.getlist('subrecetas')
        subcantidades = request.POST.getlist('subcantidades')

        receta = Receta(nombre=nombre, descripcion=descripcion)
        receta.save()

        for i in range(len(ingredientes)):
            try:
                registro = Registro.objects.get(pk=ingredientes[i])
                item = registro.item
                cantidad = int(cantidades[i])
                RecetaItem.objects.create(receta=receta, item=item, cantidad=cantidad, registro=registro)
            except Registro.DoesNotExist:
                messages.error(request, f'Error: El Registro con ID {ingredientes[i]} no existe.')

        for i in range(len(subrecetas)):
            if subrecetas[i] and subcantidades[i]:  # Verifica que subrecetas[i] y subcantidades[i] no estén vacíos
                try:
                    subreceta = Receta.objects.get(pk=subrecetas[i])
                    cantidad = int(subcantidades[i])
                    RecetaReceta.objects.create(receta=receta, subreceta=subreceta, cantidad=cantidad)
                except Receta.DoesNotExist:
                    messages.error(request, f'Error: La Subreceta con ID {subrecetas[i]} no existe.')

        messages.success(request, 'Receta registrada exitosamente')
        return redirect('/recetas/')  # Redirige a la vista de recetas después de registrar

    registros = Registro.objects.all()  # Obtén todos los registros disponibles
    recetas = Receta.objects.all()

    context = {
        'registros': registros,
        'recetas': recetas
    }

    return render(request, 'registrar_receta.html', context)
```

Fig. 30 código de registro de recetas

En este codigo obtiene mediante un formulario personalizado el nombre, descripción, ingredientes, cantidades, subrecetas y subcantidades; Se crea una nueva instancia de Receta con nombre y descripcion, y se guarda en la base de datos.

Manejo de Ingredientes: Para cada ingrediente proporcionado en el formulario:

Se intenta obtener un objeto Registro usando el ID proporcionado.

Si el Registro existe, se crea una instancia de RecetaItem asociando la receta recién creada con el ingrediente correspondiente.

Si el Registro no existe, se agrega un mensaje de error indicando que el Registro con el ID proporcionado no existe.

Manejo de Subrecetas: Para cada subreceta proporcionada en el formulario:

Se verifica que tanto la subreceta como la cantidad asociada no estén vacías.

Se intenta obtener un objeto Receta usando el ID proporcionado.

Si la Receta existe, se crea una instancia de RecetaReceta asociando la receta recién creada con la subreceta correspondiente.

Si la Receta no existe, se agrega un mensaje de error indicando que la Receta con el ID proporcionado no existe.

Mensajes y Redirección: Si la receta se registra exitosamente, se agrega un mensaje de éxito y se redirige a la vista de recetas (/recetas/).

Uso de Receta y Resumen

- **Descripción:** Aplicación práctica de una receta en operaciones diarias o en producción.
- **Acción:** Los usuarios pueden aplicar una receta para realizar procesos según las especificaciones definidas.

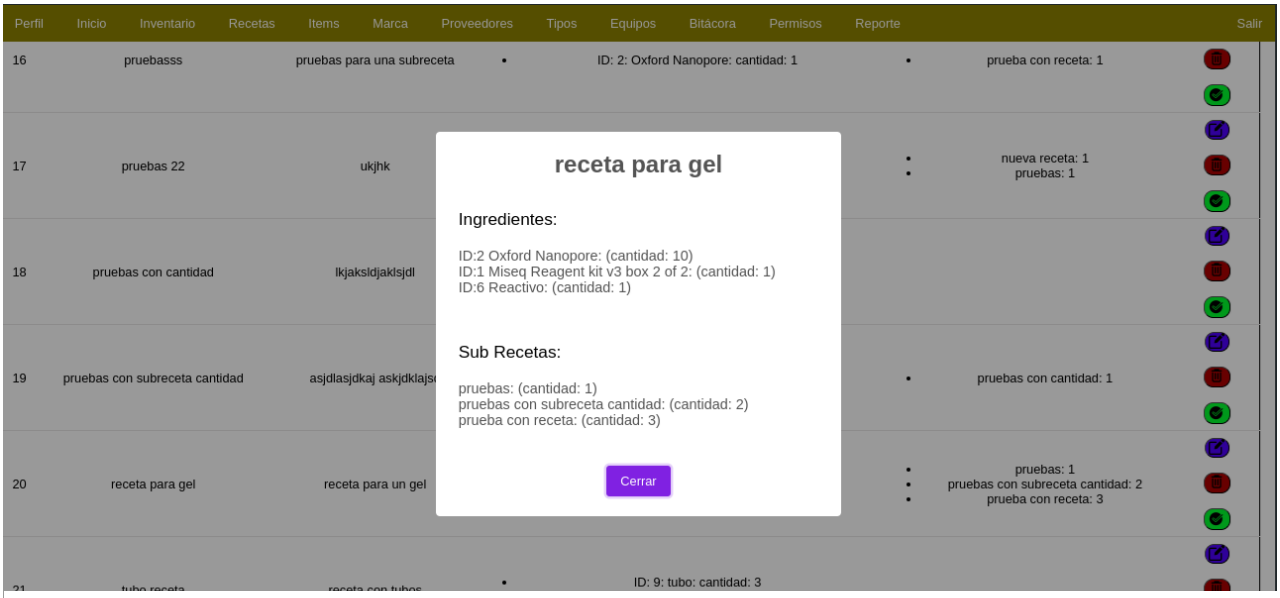


Fig. 31 Resumen de la receta

```python
def resumen_receta(request, receta_id):
    try:
        receta = Receta.objects.get(id=receta_id)
        ingredientes = [
            {
                "nombre": ri.item.nombre,
                "registroId": ri.registro.id,
                "cantidad": ri.cantidad
            }
            for ri in receta.recetaitem_set.all()
        ]
        subrecetas = [
            {
                "nombre": rr.subreceta.nombre,
                "cantidad": rr.cantidad
            }
            for rr in receta.receta_principal.all()
        ]
        data = {
            "success": True,
            "receta": {
                "id": receta.id,
                "nombre": receta.nombre,
                "descripcion": receta.descripcion,
                "ingredientes": ingredientes,
                "subrecetas": subrecetas
            }
        }
    except Receta.DoesNotExist:
        data = {"success": False, "error": "Receta no encontrada"}
    return JsonResponse(data)
```

Fig. 32 Resumen de recetas

Intentar obtener la receta:

La función recibe el receta_id de la solicitud y usa este identificador para buscar la receta correspondiente en la base de datos.

Si la receta no se encuentra, se maneja la excepción Receta.DoesNotExist y se devuelve un mensaje de error en formato JSON.

Si la receta es encontrada:

Se recuperan todos los ingredientes asociados a la receta mediante recetaitem_set.all(), se crean diccionarios con el nombre del ingrediente, el ID del registro y la cantidad, y se almacenan en la lista ingredientes.

Se recuperan todas las subrecetas asociadas a la receta mediante receta_principal.all(), se crean diccionarios con el nombre de la subreceta y la cantidad, y se almacenan en la lista subrecetas.

Se construye un diccionario data que contiene la estructura de la receta, incluyendo su ID, nombre, descripción, ingredientes y subrecetas.

Devolver la respuesta JSON:

Si la receta fue encontrada, el diccionario data contiene un campo success con valor True y los detalles de la receta.

Si la receta no fue encontrada, el diccionario data contiene un campo success con valor False y un mensaje de error.

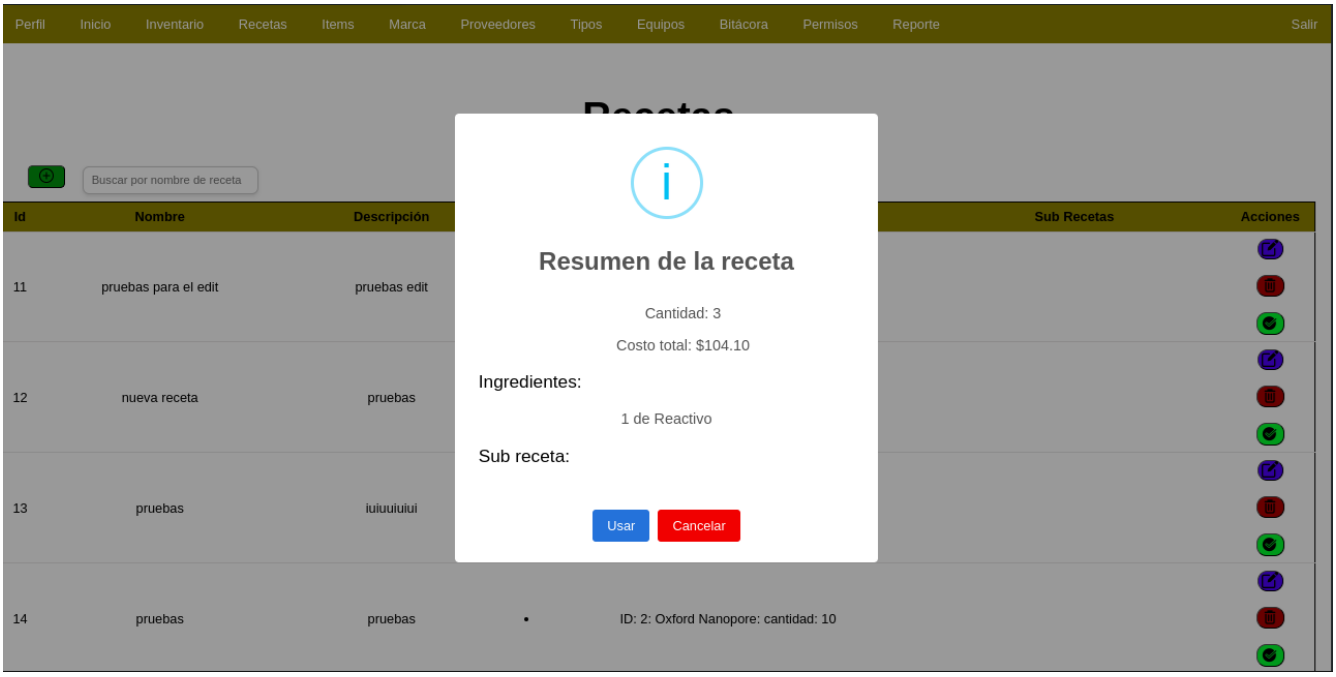


Fig. 33 Cotizacion de la receta

```python
@permission_required('app.view_receta')
@login_required
@require_POST
def cotizar_receta(request, receta_id):
    try:
        data = json.loads(request.body)
        cantidad = int(data.get('cantidad', 1))  # Convertir la cantidad a entero

        receta = Receta.objects.get(id=receta_id)
        total_costo = 0

        # Calcular costo de los ingredientes
        ingredientes = []
        for receta_item in receta.recetaitem_set.all():
            total_costo += receta_item.registro.precio * receta_item.cantidad * cantidad
            ingredientes.append(f"{receta_item.cantidad} de {receta_item.item.nombre}")

        # Calcular costo de las subrecetas
        subrecetas = []
        for receta_receta in receta.receta_principal.all():
            subreceta_costo = 0
            subreceta = receta_receta.subreceta
            for subreceta_item in subreceta.recetaitem_set.all():
                subreceta_costo += subreceta_item.registro.precio * subreceta_item.cantidad * cantidad
            total_costo += subreceta_costo * receta_receta.cantidad
            subrecetas.append(f"{receta_receta.cantidad} de {subreceta.nombre}")

        data = {
            "success": True,
            "total_costo": float(total_costo),
            "ingredientes": ingredientes,
            "subrecetas": subrecetas
        }
    except Receta.DoesNotExist:
        data = {"success": False, "error": "Receta no encontrada"}
    except Exception as e:
        data = {"success": False, "error": str(e)}
    return JsonResponse(data)
```

Fig. 34 Cotizar receta

Leer los datos de la solicitud POST:

Se leen y decodifican los datos JSON enviados en el cuerpo de la solicitud.

Se obtiene la cantidad, convirtiéndola a un entero y usando 1 como valor por defecto si no se proporciona.

Obtener la receta:

Se intenta obtener la receta usando el receta_id. Si la receta no se encuentra, se maneja la excepción Receta.DoesNotExist y se devuelve un mensaje de error en formato JSON.

Calcular el costo total:

Ingredientes:

Se iteran los ingredientes de la receta (recetaitem_set), calculando el costo multiplicando el precio del registro por la cantidad del ingrediente y la cantidad solicitada.

Se agrega una descripción del ingrediente a la lista ingredientes.

Subrecetas:

Se iteran las subrecetas de la receta (receta_principal), calculando el costo total de cada subreceta de manera similar a los ingredientes.

Se agrega una descripción de la subreceta a la lista subrecetas.

Construir y devolver la respuesta JSON:

Si no hay errores, se construye un diccionario data con el estado success, el total_costo, la lista de ingredientes y la lista de subrecetas.

Si hay una excepción diferente, se captura y se devuelve un mensaje de error.

Bitácora de Actividades

Registro de Actividades

- **Descripción:** Registro de todas las operaciones realizadas en el sistema relacionadas con el inventario y otras entidades.
- **Acción:** Cada vez que se realiza una operación de creación, edición o eliminación en el inventario, marcas, proveedores, ubicaciones, tipos o artículos, se registra en la bitácora.

Acción	Usuario	Fecha	Hora	Tabla	ID Instancia	Descripción
Crear	joshua	19/06/2024	18:11	Registro	11	Se crear el Registro con ID 11 y con codigo de barras 12308912
Crear	joshua	19/06/2024	05:23	Registro	10	Se crear el Registro con ID 10 y con codigo de barras 812371
Actualizar	joshua	18/06/2024	19:56	Registro	6	Se actualizar el registro 6 y con codigo de barras 1232193819028
Actualizar	joshua	18/06/2024	19:44	Registro	6	Se actualizar el registro 6 y con codigo de barras 1232193819028
Actualizar	joshua	18/06/2024	19:44	Registro	9	Se actualizar el registro 9 y con codigo de barras 19091431
Actualizar	joshua	18/06/2024	19:37	Item	5	
Actualizar	joshua	18/06/2024	19:35	Item	4	
Actualizar	joshua	18/06/2024	19:35	Item	1	
Actualizar	joshua	18/06/2024	19:34	Item	2	
Actualizar	joshua	18/06/2024	19:27	Item	2	
Actualizar	joshua	18/06/2024	19:27	Item	1	
Actualizar	joshua	18/06/2024	19:26	Item	1	
Actualizar	joshua	18/06/2024	19:22	Item	2	
Actualizar	joshua	18/06/2024	19:21	Item	1	
Crear	joshua	18/06/2024	19:10	Item	6	Se crear el item con ID 6 y nombre flow cell

Fig. 35 Bitacora de actividades

```python
@login_required
@permission_required('app.view_bitacora')
def bitacora(request):
    # Diccionario de traducción
    traduccion_modelos = {
        'Item': 'Producto',
        'Location': 'Ubicación',
        'Type': 'Tipo'
    }

    registros = Bitacora.objects.all().order_by('-fecha_hora')

    # Traducir los nombres de los modelos en los registros
    for registro in registros:
        if registro.modelo in traduccion_modelos:
            registro.modelo = traduccion_modelos[registro.modelo]

    return render(request, 'bitacora.html', {'registros': registros})
```

Fig. 36 codigo de la bitacora

Diccionario de Traducción:

Define un diccionario traduccion_modelos que mapea nombres de modelos a sus traducciones correspondientes.

Obtener y Ordenar Registros:

Obtiene todos los registros de la bitácora usando Bitacora.objects.all() y los ordena en orden descendente por la fecha y hora del registro (order_by('-fecha_hora')).

Traducir Nombres de Modelos:

Itera sobre los registros de la bitácora y traduce los nombres de los modelos usando el diccionario traduccion_modelos.

Renderizar Plantilla:

Renderiza la plantilla bitacora.html con el contexto de los registros traducidos.

Gestión de Permisos y Grupos de Usuarios

Permisos y Creación de Grupos

- **Descripción:** Configuración de permisos específicos para diferentes roles de usuarios.

- **Acción:** Los administradores pueden crear grupos de usuarios y asignar permisos granulares a cada grupo, controlando qué acciones pueden realizar los usuarios.

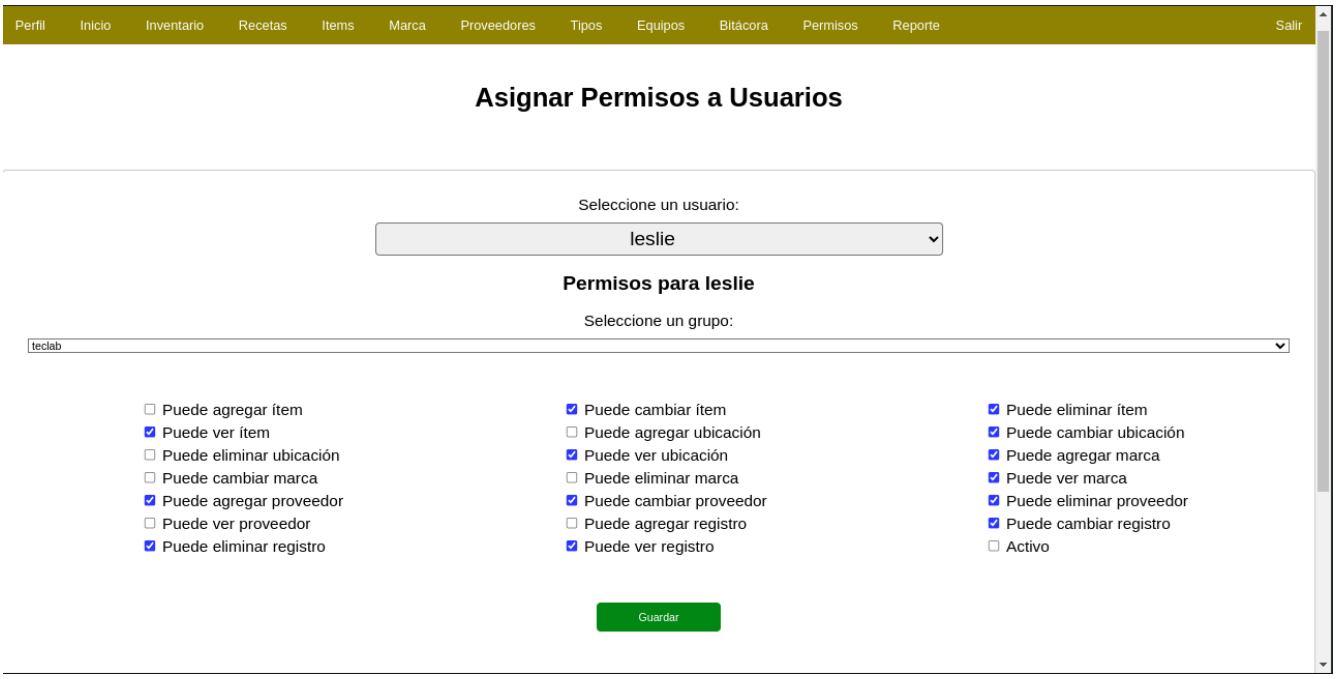


Fig. 37 Permiso de usuarios

```python
@permission_required('app.view_auth_permission')
@login_required
def asignar_permisos(request):
    usuarios = User.objects.all()

    content_types = ContentType.objects.filter(
        app_label='app',
        model__in=['location', 'marca', 'proveedor', 'item', 'registro', 'receta', 'type']
    )

    permisos = Permission.objects.filter(content_type__in=content_types)

    permisos_traduccion = {
        'Can add item': 'Puede agregar ítem',
        'Can change item': 'Puede cambiar ítem',
        'Can delete item': 'Puede eliminar ítem',
        'Can view item': 'Puede ver ítem',
        'Can add location': 'Puede agregar ubicación',
        'Can change location': 'Puede cambiar ubicación',
        'Can delete location': 'Puede eliminar ubicación',
        'Can view location': 'Puede ver ubicación',
        'Can add marca': 'Puede agregar marca',
        'Can change marca': 'Puede cambiar marca',
        'Can delete marca': 'Puede eliminar marca',
        'Can view marca': 'Puede ver marca',
        'Can add proveedor': 'Puede agregar proveedor',
        'Can change proveedor': 'Puede cambiar proveedor',
        'Can delete proveedor': 'Puede eliminar proveedor',
        'Can view proveedor': 'Puede ver proveedor',
        'Can add registro': 'Puede agregar registro',
        'Can change registro': 'Puede cambiar registro',
        'Can delete registro': 'Puede eliminar registro',
        'Can view registro': 'Puede ver registro',
        'Can add receta': 'Puede agregar receta',
        'Can change receta': 'Puede cambiar receta',
        'Can delete receta': 'Puede eliminar receta',
        'Can view receta': 'Puede ver receta',
        'Can add type': 'Puede agregar tipo',
        'Can change type': 'Puede cambiar tipo',
        'Can delete type': 'Puede eliminar tipo',
        'Can view type': 'Puede ver tipo'
    }
```

Fig. 38 Traduccion de permisos

```
    if request.method == 'POST':
        if 'delete_user' in request.POST:
            username = request.POST.get('username')
            print(f"Username received for deletion: {username}")

            try:
                user_to_delete = User.objects.get(username=username)
                user_to_delete.delete()
                messages.success(request, f"El usuario '{username}' ha sido eliminado.")
            except User.DoesNotExist:
                messages.error(request, f"El usuario '{username}' no existe.")

            return redirect('asignar_permisos')

        elif 'make_superuser' in request.POST:
            username = request.POST.get('username')
            print(f"Username received for superuser: {username}")

            try:
                selected_user = User.objects.get(username=username)
                selected_user.is_superuser = True
                selected_user.save()
                messages.success(request, f"Se ha hecho a '{username}' superusuario.")
            except User.DoesNotExist:
                messages.error(request, f"El usuario '{username}' no existe.")

            return redirect('asignar_permisos')

        username = request.POST.get('username')
        print(f"Username received in POST: {username}")

        try:
            selected_user = User.objects.get(username=username)
        except User.DoesNotExist:
            messages.error(request, f"El usuario '{username}' no existe.")
            return redirect('asignar_permisos')

        permission_ids = request.POST.getlist('permissions')
        selected_user.user_permissions.clear()

        for permission_id in permission_ids:
            permission = Permission.objects.get(id=permission_id)
            selected_user.user_permissions.add(permission)

        print(f"is_active: {request.POST.get('is_active')}")
```

Fig. 39 Codigo de permisos

```
        print(f"is_active: {request.POST.get('is_active')}")

        is_active = 'is_active' in request.POST
        selected_user.is_active = is_active
        selected_user.save()

        return redirect('asignar_permisos')

    selected_username = request.GET.get('username')
    print(f"Username received in GET: {selected_username}")

    selected_user = User.objects.filter(username=selected_username).first()
    selected_user_permissions = selected_user.user_permissions.all() if selected_user else []

    grupos = Group.objects.all()

    # Traducir los nombres de los permisos
    permisos_traducidos = []
    for permiso in permisos:
        permiso.name = permisos_traduccion.get(permiso.name, permiso.name)
        permisos_traducidos.append(permiso)

    return render(request, 'asignar_permisos.html', {
        'usuarios': usuarios,
        'permisos': permisos_traducidos,
        'selected_username': selected_username,
        'selected_user_permissions': selected_user_permissions,
        'selected_user': selected_user,
        'grupos': grupos
    })
```

Fig. 40 Codigo de permisos 2

Obtener Usuarios y Permisos:

Obtiene todos los usuarios (User.objects.all()).

Filtra los tipos de contenido (ContentType) relevantes para la aplicación (app) y sus modelos específicos (location, marca, proveedor, etc.).

Obtiene los permisos correspondientes a estos tipos de contenido (Permission.objects.filter(content_type__in=content_types)).

Diccionario de Traducción de Permisos:

Define un diccionario permisos_traduccion para traducir los nombres de los permisos a un idioma más comprensible.

Manejo de Solicitudes POST:

Si la solicitud es POST, verifica si se solicita la eliminación de un usuario, la conversión de un usuario a superusuario o la asignación de permisos.

Para la eliminación de un usuario: obtiene el nombre de usuario, elimina el usuario correspondiente y redirige a la misma vista con un mensaje de éxito o error.

Para convertir a un usuario en superusuario: obtiene el nombre de usuario, establece is_superuser a True y redirige a la misma vista con un mensaje de éxito o error.

Para la asignación de permisos: obtiene el nombre de usuario, los permisos seleccionados, limpia los permisos actuales del usuario, asigna los nuevos permisos, establece el estado de activación del usuario y guarda los cambios.

Manejo de Solicitudes GET:

Obtiene el nombre de usuario seleccionado de la solicitud GET, busca el usuario correspondiente y sus permisos.

Renderización de la Plantilla:

Traduce los nombres de los permisos usando el diccionario permisos_traduccion.

Renderiza la plantilla asignar_permisos.html con el contexto de usuarios, permisos traducidos, usuario seleccionado, permisos del usuario seleccionado y grupos.

Uso de bitácora y reportes de recetas

- **Descripción:** modulo para ver las recetas usadas y generar reportes por fechas de las recetas
- **Acción:** los usuarios pueden ver las recetas usadas y pueden generar reportes para ver el costo total de las recetas usadas.

Receta	Cantidad	Cotización Total	Fecha de Uso
nueva receta	1	$34,70	18 Jun 2024 13:56
tubo receta	5	$912,50	18 Jun 2024 13:44
nueva receta	2	$69,40	18 Jun 2024 10:21
nueva receta	1	$34,70	17 Jun 2024 12:35
nueva receta	2	$69,40	17 Jun 2024 12:21
pruebas	2	$347,00	17 Jun 2024 00:45
nueva receta	3	$520,50	17 Jun 2024 00:45
pruebas con cantidad	2	$3047,80	17 Jun 2024 00:40

Fig. 41 Bitácora de uso de recetas

```
permission_required('app.view_Recetas')
@login_required
def lista_usos_receta(request):
    usos_recetas = UsoReceta.objects.all().order_by('-fecha_uso')
    mexico_tz = pytz.timezone('America/Mexico_City')
    for uso in usos_recetas:
        uso.fecha_uso_mexico = uso.fecha_uso.astimezone(mexico_tz)
    context = {
        'usos_recetas': usos_recetas
    }
    return render(request, 'lista_usos_receta.html', context)
```

Fig. 42 Uso de recetas

Bitacora de uso de recetas en donde muestra el uso de las recetas con la cantidad y muestra la zona hoararia de mexico al usar la receta.

UUSMB
UNIDAD UNIVERSITARIA DE SECUENCIACIÓN MASIVA Y BIOINFORMÁTICA

Reporte de Recetas

Del 17/06/2024 al 19/06/2024

Receta	Cantidad	Cotización Total
nueva receta	9	$728.70
tubo receta	5	$912.50
pruebas	2	$347.00
pruebas con cantidad	2	$3047.80

Total Cantidad: 18 **Total Costo: $5036.00**

Fig. 43 Reporte de recetas

```python
@permission_required('app.view_recetas')
@login_required
def generar_reporte(request):
    if request.method == 'POST':
        fecha_inicio = request.POST.get('fecha_inicio')
        fecha_fin = request.POST.get('fecha_fin')

        if fecha_inicio and fecha_fin:
            fecha_inicio = timezone.datetime.strptime(fecha_inicio, '%Y-%m-%d').date()
            fecha_fin = timezone.datetime.strptime(fecha_fin, '%Y-%m-%d').date()

            usos_recetas = UsoReceta.objects.filter(fecha_uso__date__range=(fecha_inicio, fecha_fin)).order_by('-fecha_uso')

            receta_agrupada = defaultdict(lambda: {'cantidad': 0, 'cotizacion_total': 0})
            for uso in usos_recetas:
                receta_agrupada[uso.receta.nombre]['cantidad'] += uso.cantidad
                receta_agrupada[uso.receta.nombre]['cotizacion_total'] += uso.cotizacion_total

            # Crear el PDF
            buffer = BytesIO()
            p = canvas.Canvas(buffer, pagesize=letter)

            # Función para agregar una nueva página y el encabezado de la tabla
            def agregar_pagina():
                p.showPage()
                p.setFont("Helvetica-Bold", 12)
                p.drawString(0.5 * inch, 9.5 * inch, "Receta")
                p.drawString(2 * inch, 9.5 * inch, "Cantidad")
                p.drawString(3.5 * inch, 9.5 * inch, "Cotización Total")
                p.setFont("Helvetica", 10)
                return 9.25 * inch

            # Añadir la imagen más abajo y más estirada
            image_path = 'app/static/imagenes/uusmb.png'
            p.drawImage(image_path, 0.5 * inch, 9.5 * inch, width=2 * inch, height=1 * inch)

            # Título del reporte
            p.setFont("Helvetica-Bold", 16)
            p.drawCentredString(4.25 * inch, 10 * inch, "Reporte de Recetas")

            # Subtítulo con fechas
            p.setFont("Helvetica", 12)
            p.drawCentredString(4.25 * inch, 9.7 * inch, f"Del {fecha_inicio.strftime('%d/%m/%Y')} al {fecha_fin.strftime('%d/%m/%Y')}")
```

Fig. 44 Generacion de reporte

```
        # Líneas divisorias
        p.setStrokeColor(colors.black)
        p.setLineWidth(1)
        p.line(0.5 * inch, 9.5 * inch, 7.5 * inch, 9.5 * inch)

        # Table headers
        p.setFont("Helvetica-Bold", 12)
        p.drawString(0.5 * inch, 9.0 * inch, "Receta")
        p.drawString(2 * inch, 9.0 * inch, "Cantidad")
        p.drawString(3.5 * inch, 9.0 * inch, "Cotización Total")

        # Líneas divisorias
        p.setLineWidth(0.5)
        p.line(0.5 * inch, 8.95 * inch, 7.5 * inch, 8.95 * inch)

        y = 8.75 * inch
        total_cantidad = 0
        total_costo = 0

        p.setFont("Helvetica", 10)
        for nombre_receta, datos in receta_agrupada.items():
            p.drawString(0.5 * inch, y, nombre_receta)
            p.drawString(2 * inch, y, str(datos['cantidad']))
            p.drawString(3.5 * inch, y, f"${datos['cotizacion_total']:.2f}")

            total_cantidad += datos['cantidad']
            total_costo += datos['cotizacion_total']

            y -= 0.25 * inch

            if y < 1 * inch:
                y = agregar_pagina()

        if y < 1.5 * inch:
            y = agregar_pagina() - 0.25 * inch

        y -= 0.25 * inch
        p.setFont("Helvetica-Bold", 12)
        p.drawString(1 * inch, y, f"Total Cantidad: {total_cantidad}")
        p.drawString(3.5 * inch, y, f"Total Costo: ${total_costo:.2f}")
```

Fig. 45 Estilo del reporte

```
        p.showPage()
        p.save()

        buffer.seek(0)
        filename = f"reporte_usos_recetas_{datetime.datetime.now().strftime('%Y%m%d%H%M%S')}.pdf"

        response = HttpResponse(content_type='application/pdf')
        response['Content-Disposition'] = f'attachment; filename="{filename}"'
        response.write(buffer.getvalue())

        return response

    return render(request, 'generar_reporte.html')
```

Fig. 46 Reporte

Manejo de Solicitudes POST:

Verifica si la solicitud es POST.

Obtiene las fechas de inicio y fin del formulario.

Convierte las fechas de las cadenas de texto a objetos de fecha (datetime.date).

Filtrado de Datos:

Filtra los objetos UsoReceta dentro del rango de fechas proporcionado, ordenados por la fecha de uso en orden descendente.

Agrupación de Datos:

Agrupa los datos de uso de recetas por nombre de receta, sumando las cantidades y las cotizaciones totales.

Creación del PDF:

Utiliza ReportLab para crear un documento PDF en memoria (BytesIO).

Define una función agregar_pagina para agregar nuevas páginas y el encabezado de la tabla cuando sea necesario.

Añade una imagen en la parte superior del PDF.

Añade el título y subtítulo del reporte.

Añade encabezados de tabla y líneas divisorias.

Itera sobre los datos agrupados y los agrega a la tabla en el PDF.

Añade los totales al final del PDF.

Guarda el PDF y lo envía como una respuesta HTTP con un encabezado de Content-Disposition para descargar el archivo.

Renderización de la Plantilla:

Si la solicitud no es POST, renderiza la plantilla generar_reporte.html para mostrar el formulario al usuario.