Tabla de contenido

Documentación del Proyecto

Introducción

Este proyecto es una aplicación web construida en Python utilizando el framework Flask. Está diseñado siguiendo una arquitectura modular y limpia, con el objetivo de proporcionar una solución efectiva para la gestión de datos relacionados con personas y carreras, incorporando además la funcionalidad de alta y baja lógica en la base de datos.

Diagrama de la Solución:

app.py
└─> app_factory.py

app_factory.py
├─> config.py
├─> models.py
├─> routes.py

```
├─> api.py
└─> gestores.py

routes.py
├─> auth.py
│    └─> login()
├─> personas.py
│    ├─> index()
│    │    └─> gestor_personas.obtener_pagina()
│    ├─> create()
│    │    └─> gestor_personas.crear()
│    ├─> update()
│    │    └─> gestor_personas.editar()
│    └─> delete()
│    └─> gestor_personas.eliminar()
├─> carreras.py
│    ├─> index()
│    │    └─> gestor_carreras.obtener_pagina()
│    ├─> create()
│    │    └─> gestor_carreras.crear_carrera()
│    ├─> update()
│    │    └─> gestor_carreras.editar_carrera()
│    └─> delete()
│         └─> gestor_carreras.eliminar_carrera()
├─> facultades.py
│    ├─> index()
│    │    └─> gestor_facultades.obtener_pagina()
│    ├─> create(),  update() , delete()
│    │    └─> gestor_carreras.crear_editar_eliminar_facultades()
├─> campus.py
│    ├─> index()
│    │    └─> gestor_campus.obtener_pagina()
│    ├─> create(),  update() , delete()
│    │    └─> gestor_campus.crear_editar_eliminar_campus()
├─> programas.py
│    ├─> index()
│    │    └─> gestor_programas.obtener_pagina()
│    ├─> create(),  update() , delete()
│    │    └─> gestor_programas.crear_editar_eliminar_programas()
└─> universidades.py
     ├─> index()
     │    └─> gestor_universidades.obtener_pagina()
     └─> create(),  update() , delete()
          └─> gestor_universidades.crear_editar_eliminar_universidades()



└─> macros.py
```

```
    ├─> paginado()
    ├─> carreras()
    ├─> lugares()
    ├─> género()
    └─> modal()



models.py
└─> Persona
        ├─> guardar()
        └─> borrar()
└─> Carrera
        ├─> guardar()
        └─> borrar()


gestores.py
├─> gestor_personas.py
│    ├─> obtener_pagina()
│    ├─> obtener()
│    ├─> crear()
│    ├─> editar()
│    └─> eliminar()
├─> gestor_lugares.py
│    └─> consultar()
├─> gestor_generos.py
│    └─> obtener_todos()
└─> gestor_comun.py
├─> gestor_email.py
│    └─> enviar()
├─> gestor_tipopersona.py
│    └─> obtener_todos()
│    ├─> crear()
│    ├─> editar()
│    └─> eliminar()
├─> gestor_carreras.py
│    ├─> obtener_pagina()
│    ├─> obtener()
│    ├─> crear()
│    ├─> editar()
│    └─> eliminar()
├─> gestor_personas_carreras.py
│    ├─> obtener_pagina()
│    ├─> obtener()
│    ├─> crear()
│    ├─> editar()
│    └─> eliminar()
```

```
│    └─> obtener_carreras_por_persona()


api.py
├─> PersonasResource
│    └─> gestor_personas
├─> LugaresResource
│    └─> gestor_lugares
├─> CarrerasResource
│    └─> gestor_carreras
├─> GenerosResource
│    └─> gestor_generos
└─> TiposResource
     └─> GestorTiposPersona
```

Descripción

- app.py: Este archivo crea la aplicación Flask y sirve como punto de entrada para la aplicación.

- app_factory.py: Contiene la configuración de la aplicación y se encarga de crear la instancia de la aplicación Flask.

- routes.py: Aquí se encuentran los módulos de rutas de la aplicación, divididos por funcionalidad.

- auth.py: Gestiona la autenticación, incluyendo la función login().

- personas.py: Contiene las rutas CRUD relacionadas con personas, incluyendo index(), create(), update(), y delete().

- macros.py: Contiene macros reutilizables que pueden ser utilizados en diferentes partes del proyecto.

- models.py: Aquí se definen los modelos SQLAlchemy, incluyendo Persona y Carrera, con sus respectivas funciones para guardar() y borrar().
-
- gestores.py: Este módulo encapsula la lógica de negocio de la aplicación y está dividido en varios submódulos:

- gestor_personas.py: Contiene funciones relacionadas con la gestión de personas, como obtener_pagina(), obtener(), crear(), editar(), y eliminar().

- gestor_lugares.py: Proporciona funciones para consultar lugares.

- gestor_generos.py: Ofrece funciones para obtener todos los géneros disponibles.

- gestor_comun.py: Contiene submódulos que gestionan tareas comunes como el envío de correos electrónicos.

- gestor_facultades, gestor_campus, gestor_programas, gestor_universidades: Cada uno por su parte contiene funciones relacionadas con la gestión de su entidad correspondiente, como obtener_pagina(), crear(), editar(), y eliminar().

- api.py: Define los recursos de la API REST de la aplicación, incluyendo PersonasResource y LugaresResource.

- Integración de Alembic
  Se ha implementado la biblioteca Alembic para la gestión de fechas de alta y baja lógica en la base de datos. Esto permite un control más eficiente de las fechas de creación y eliminación de registros en la base de datos.

Explicación

- app.py crea la aplicación Flask.
- app_factory.py tiene la configuración.
- routes contiene los módulos de rutas.
- auth.py maneja la autenticación.
- personas.py las rutas CRUD de personas.
- carreras.py las rutas CRUD de carreras.
- macros.py contiene macros reutilizables.
- models tiene los modelos SQLAlchemy.
- gestores encapsula lógica de negocio.
- api los recursos API REST.

De esta forma se tiene:

- app.py como punto de entrada
- app_factory.py para crear la app
- Rutas agrupadas por funcionalidad
- Modelos y lógica en módulos separados
- Macros para reutilizar código
- APIs REST

Tabla de contenido por módulo:

Archivo	**Descripción**
app.py	Creación de la app Flask
app_factory.py	Configuración de la app
config.py	Configuraciones globales
modules/apis/lugares.py	API REST de Lugares
modules/apis/carreras.py	API REST de Carreras
modules/apis/personas.py	API REST de Personas
modules/apis/generos.py	API REST de Generos
modules/apis/tipos.py	API REST de Tipos
modules/apis/carreras_noRelacionadas.py	API REST que administra carreras no relacionadas a personas
alembic/versions	Actualizacion de alembic
modules/common/gestor_comun.py	Lógica común a gestores
modules/common/gestor_email.py	Lógica de negocio de email
modules/common/gestor_generos.py	Lógica de negocio de generos
modules/common/gestor_lugares.py	Lógica de negocio de Lugares
modules/common/gestor_personas.py	Lógica de negocio de Personas
modules/common/gestor_TipoPersonas.py	Lógica de negocio de TipoPersonas
modules/common/gestor_carreras.py	Lógica de negocio de Carreras
modules/common/gestor_carreras_personas.py	Lógica de negocio CarrerasPersonas
modules/common/gestor_campus.py	Lógica de negocio de Campus
modules/common/gestor_programas.py	Lógica de negocio de Programas
modules/common/gestor_universidades.py	Lógica de negocio de universidades
modules/common/gestor_facultades.py	Lógica de negocio de facultades
modules/models/entities.py	Modelos de la app

modules/models/base.py	Modelos de la base
modules/auth.py	Autenticación de usuarios
modules/routes.py	Rutas generales
modules/routes_personas.py	Rutas CRUD Personas
modules/routes_carreras.py	Rutas CRUD Carreras
modules/routes_facultades.py	Rutas CRUD facultades
modules/routes_programas.py	Rutas CRUD programas
modules/routes_campus.py	Rutas CRUD campus
modules/routes_universidades.py	Rutas CRUD universidades
templates/404.html	Página de error 404
templates/about.html	Página Acerca De
templates/contact.html	Formulario de contacto
templates/index.html	Página de inicio
templates/login.html	Formulario de login
templates/paginado.html	Macro paginación
templates/personas/crear_persona.html	Formulario crear persona
templates/personas/editar_persona_carrera.html	Form editar carrera-persona
templates/personas/personas.html	Listado de personas
templates/personas/editar_persona.html	Formulario editar persona
templates/carreras/editar_carrera.html	Formulario de editar carrera
templates/carreras/crear_carrera.html	Formulario de crear carrera
templates/carreras/carrera.html	Listado de carreras
templates/comun/carreras.html	Macro de carreras
templates/comun/modal_eliminar.html	Macro modal eliminar
templates/comun/base.html	Template base
templates/comun/lugares.html	Macro de lugares

templates/comun/genero.html	Macro de genero
templates/comun/modal_agregar.html	Macro modal agregar
templates/comun/modal_editar.html	Macro modal editar
templates/comun/tipo.html	Macro de tipos de personas
templates/comun/paginado.html	Macro para render del paginado
templates/comun/carreras_noRelacionadas.html	Macro de carreras no relacionadas a persona
templates/campus/campus.html	Listado de campus
templates/facultades/facultades.html	Listado de facultades
templates/programas/programas.html	Listado de programas
templates/universidades/universidades.html	Listado de universidades

Archivo app.py:

- Importa la función create_app() desde app_factory.py
- Crea la aplicación Flask llamando a create_app()
- Ejecuta la aplicación en modo debug cuando se ejecuta directamente

Este es el archivo principal para crear y ejecutar la aplicación. Utiliza el patrón de aplicación factory para crear la app.

Archivo app_factory.py

- Importa dependencias como Flask, Flask-SQLAlchemy, Flask-Login, etc.
- Define la función create_app() que creará y configurará la aplicación Flask
- Inicializa la base de datos
- Registra los blueprints de rutas, authenticación, API's, etc.
- Configura Gestor de Login y CSRF
- Retorna la instancia de la aplicación Flask configurada

Este archivo encapsula toda la creación y configuración de la aplicación Flask. Sigue el patrón de aplicación factory.

Archivo config.py

- Contiene configuraciones globales como conexión a la BD, parámetros, etc.

- Se utiliza en app_factory.py para configurar la aplicación Mantiene configuraciones reutilizables para la aplicación Flask.

Archivo index.html

- Extiende el template base
- Muestra un título de bienvenida
- Enlace a la lista de personas paginada

Template HTML que muestra la página de inicio. Usa herencia de templates.

Archivo about.html

- Extiende el template base
- Muestra un título sobre la página Acerca De
- Botón para regresar al Inicio

Template HTML que muestra la página Acerca De. Usa herencia de templates.

Archivo contact.html

- Extiende el template base
- Muestra formulario para contacto
- Maneja envío y validación con Flask-WTF

Template HTML que muestra el formulario de contacto. Usa herencia de templates y Flask-WTF.

Archivo comun/base.html

- Template base que provee la estructura común
- Navbar, block content y scripts JS comunes
- Permite heredar y extender los otros templates

Template base con la estructura HTML común a todas las páginas.

Archivo paginado.html

- Define macro para renderizar paginación en templates
- Recibe el endpoint y los items paginados

- Muestra los enlaces de paginación

Template que contiene lógica reutilizable para renderizar la paginación. Define la macro “render_paginado” para renderizar la paginación de resultados de forma reusable.

Macro render_paginado:

Recibe dos parámetros:
- endpoint: la ruta o endpoint donde hacer la petición para obtener una página específica.
- items: el objeto Pagination de Flask-SQLAlchemy con los items paginados. Usa los

métodos y atributos de Pagination para renderizar la paginación:
- items.page: obtener el número de página actual.
- items.pages: obtener el total de páginas.
- items.has_prev: verificar si hay página anterior.
- items.has_next: verificar si hay página siguiente.
- items.iter_pages(): iterar sobre los números de página para renderizar.
- url_for(): genera las URLs dinámicamente para cada número de página.

Renderiza los links Previous, 1, 2, 3, Next etc. de forma dinámica según los datos pagination.

Archivo modal_eliminar.html:

- Define macro para mostrar modal de confirmación
- Recibe id, ruta y mensaje para mostrar
- Muestra modal con botones para confirmar o cancelar

Template que encapsula lógica reutilizable para mostrar modales de confirmación. Define la macro “confirmar_eliminar” para mostrar un modal de confirmación antes de eliminar un item.

Macro confirmar_eliminar:

Recibe tres parámetros:
- id: el id del item a eliminar.
- delete_route: la ruta o endpoint para eliminar el item.
- confirm_message: el mensaje a mostrar para confirmar.

Usa Bootstrap para renderizar el modal con los botones Confirmar y Cancelar.
El formulario dentro del modal hace POST a delete_route para eliminar el item si se confirma.
Envía el csrf_token para proteger contra ataques CSRF.
Permite crear fácilmente modales de confirmación reusables antes de eliminar datos.

Archivo 404.html

- Extiende el template base
- Muestra mensaje de error 404 Página No Encontrada
- Link para regresar al Inicio

Template HTML que muestra mensaje de error 404.

Archivo login.html

- Muestra formulario de login con Flask-WTF
- Campos usuario y contraseña

- Maneja envío y autenticación

Template que muestra el formulario de login. Usa Flask-WTF.

Archivo personas/crear_persona.html

Template que muestra el formulario para crear una nueva Persona.

- Extiende el template base
- Envía el formulario al endpoint de creación
- Muestra los campos necesarios para crear una persona
- Puede rellenar los campos con datos previos

Permite llenar un formulario para crear una nueva persona en la base de datos.

Archivo personas/personas.html

Template que muestra la lista paginada de personas con varias funciones.

- Extiende el template base
- Importa macros para paginación y modal eliminar
- Muestra tabla con los campos de Persona
- Usa paginado.render_paginado para mostrar paginación

- Usa macro modal_eliminar antes de eliminar
- Links para editar y eliminar personas Permite listar, editar y eliminar personas.

Archivo personas/editar_persona.html:

Template que muestra el formulario para editar una Persona.

- Extiende el template base
- Obtiene la persona a editar y la envía al template
- Muestra los campos del formulario rellenados con los datos de la persona
- Envía el formulario al endpoint de edición con el ID Permite editar los campos de una persona existente.

Archivo personas/editar_carrera_persona.html:

Template que muestra el formulario para editar una Persona y sus carreras asociadas.

- Extiende el template base
- Obtiene la persona a editar y sus carreras asociadas y la envía al template
- Muestra los campos del formulario rellenados con los datos de la persona y las carreras asociadasa a esa persona
- Envía el formulario al endpoint de edición con el ID Permite editar los campos de una persona y sus carreras asociadas existentes.

Archivo carreras/crear_carrera.html:

Template que muestra el formulario para crear una nueva Carrera.

- Extiende el template base
- Envía el formulario al endpoint de creación
- Muestra los campos necesarios para crear una carrera

Permite llenar un formulario para crear una nueva carrera en la base de datos.

Archivo carreras/carreras.html:

Template que muestra la lista paginada de carreras con varias funciones.

- Extiende el template base
- Importa macros para paginación y modal eliminar
- Muestra tabla con los campos de Carreras
- Usa paginado.render_paginado para mostrar paginación
- Usa macro modal_eliminar antes de eliminar
- Links para editar y eliminar
- Carreras Permite listar, editar y limina carreras.

Archivo carreras/editar_carrera.html:

Template que muestra el formulario para editar una Carrera

- Extiende el template base
- Obtiene la carrera a editar y la envía al template
- Muestra los combos
- Envía el formulario al endpoint de edición
- Permite editar una carrera existente

Archivo comun/carreras.html:

Template que es un macros que permite visualizar un formulario con cuatro campos de selección: Universidad,Facultad, Campus y programa.

- Muestra los combos con las opciones de los cuatros campos que compone carrera
- Obtiene las opciones de los cuatros campos a través de llamadas a un API
- Envía el formulario al endpoint que lo requiera

Archivo comun/lugares.html:

Template que es un macros que permite visualizar un formulario con cuatro campos de selección: País, provincia, ciudad y barrio.

- Muestra los combos con las opciones de los cuatros campos que componen los lugares.
- Obtiene las opciones de los cuatros campos a través de llamadas a un API
- Envía el formulario al endpoint que lo requiera.

Archivo comun/lugares.html:

Template que es un macros que permite visualizar un formulario con el dato del genero de la persona

- Muestra el combos con las opciones de géneros que están en la base de datos.
- Obtiene las opciones a través de llamadas a un API
- Envía el formulario al endpoint que lo requiera.

Archivo comun/modal_agregar.html

- Define macro para mostrar modal para agregar una nueva entidad.
- Muestra modal con botones para confirmar o cancelar
- add_route: la ruta o endpoint para crear el item.

Usa Bootstrap para renderizar el modal con los botones Confirmar y Cancelar.
El formulario dentro del modal hace POST a add_route para agregar el item si se confirma.
Envía el csrf_token para proteger contra ataques CSRF.
Permite crear fácilmente modales de confirmación reusables antes de agregar datos.

Archivo comun/tipo.html:

Template que es un macros que permite visualizar un formulario con el dato del tipo de la persona

- Muestra el combos con las opciones delos que están en la base de datos, que son dos: alumno y profesor.
- Obtiene las opciones a través de llamadas a un API
- Envía el formulario al endpoint que lo requiera.

Archivo comun/carreras_noRelacionadas.html:

Template que es un macros que permite visualizar un formulario con cuatro campos de selección: Universidad,Facultad, Campus y programa.

- Muestra los combos con las opciones de los cuatros campos que componen carrera, pero permitiendo elegir entre cualquier entidad entre las 4 posibles, ya que no hay un join que las obligue a estar unidas.
- Obtiene las opciones de los cuatros campos a través de llamadas a un API
- Envía el formulario al endpoint que lo requiera.

Archivos campus/campus.html, facultades/facultades.html, programas.html y universidades/universidades.html

Templates que muestran la lista paginada de campus, facultades, programas y universidades respectivamente con varias funciones.

- Extiende el template base
- Importa macros para paginación, modal eliminar, modal editar y modal agregar.
- Cada template muestra una tabla con los datos en la base de su entidad correspondiente.
- Usa paginado.render_paginado para mostrar paginación
- Usa macro modal_eliminar antes de eliminar.
- usa macro modal_editar para editar y macro modal_agregar para agregar datos a la base.
- Permite enlistar, eliminar, crear y editar su respectiva entidad (campus, programa, universidad o facultad.

Archivo auth.py:

- Contiene lógica para registrar y autenticar usuarios
- Endpoints para login y logout
- Maneja inicio de sesión con Flask-Login

Módulo que implementa la autenticación de usuarios con Flask-Login.

Archivo routes.py:

- Define rutas para home, acerca de, contacto, etc.

- Renderiza los templates relacionados
- Protege rutas con @login_required

Módulo que implementa las rutas básicas con sus vistas asociadas.

Archivo routes_personas.py:

- Implementa las rutas CRUD para el modelo Persona

- Endpoints para crear, editar, borrar y listar personas
- Usa gestor_personas para la lógica de negocio

Módulo que implementa las operaciones CRUD para el modelo Persona. Contiene las rutas o endpoints que manejan las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para el modelo Persona.
Importa los módulos necesarios como Flask, gestor_personas, etc. Define el blueprint “routes_personas” que agrupa estas rutas.

Endpoint obtener_lista_paginada

- Ruta GET /personas
- Obtiene el parámetro page de la URL para la paginación.
- Utiliza gestor_personas para obtener una página de personas.
- Renderiza la plantilla personas.html pasándole la lista paginada.

Endpoint editar_persona

- Ruta POST /personas//editar
- Obtiene el ID de persona a editar de la URL.
- Si es POST, obtiene los datos del formulario.
- Utiliza gestor_personas para actualizar la persona.
- Redirige a la lista o muestra errores.

Endpoint eliminar_persona

- Ruta POST /personas/
- Obtiene el ID de persona a eliminar.
- Utiliza gestor_personas para eliminar la persona.
- Redirige a la lista.

Endpoint crear_persona

- Ruta GET y POST /personas/crear
- Si es GET, muestra el formulario para crear.
- Si es POST, obtiene los datos del formulario.
- Utiliza gestor_personas para crear la nueva persona.
- Redirige a la lista o muestra errores.

Endpoint crear_persona

- Ruta GET y POST /personas/crear
- Si es GET, muestra el formulario para crear.
- Si es POST, obtiene los datos del formulario.
- Utiliza gestor_personas para crear la nueva persona.
- Redirige a la lista o muestra errores.

Endpoint generar_excel:

- Ruta POST /personas/generar_excel
- Toma todas las personas y sus datos de la base y los exporta a un archivo excel descargable.
- Utiliza gestor_personas.obtener_todo para obtener todos los datos de la base.

Archivo routes_carreras.py:

- Implementa las rutas CRUD para el modelo Carrera
- Endpoints para crear, editar, borrar y listar personas
- Usa gestor_carreras para la lógica de negocio

Módulo que implementa las operaciones CRUD para el modelo Carrera. Contiene las rutas o endpoints que manejan las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para el modelo Carrera.
Importa los módulos necesarios como Flask, gestor_carreras, etc. Define el blueprint “routes_carrera” que agrupa estas rutas.

Endpoint obtener_lista_paginada:

- Ruta GET /carreras
- Obtiene el parámetro page de la URL para la paginación.
- Utiliza gestor_carrera para obtener una página de carreras.
- Renderiza la plantilla carrera.html pasándole la lista paginada.

Endpoint crear_carrera

- Ruta GET y POST /carreras/crear
- Si es GET, muestra el formulario para crear.
- Si es POST, obtiene los datos del formulario.
- Utiliza gestor_carreras para crear la nueva carrera .
- Redirige a la lista o muestra errores.

Endpoint editar_carrera

- Ruta POST /carrera//editar
- Obtiene el ID de carrera a editar de la URL.
- Si es POST, obtiene los datos del formulario.
- Utiliza gestor_carrera para actualizar la carrera .
- Redirige a la lista o muestra errores.

Endpoint eliminar_carrera

- Ruta POST /carrera /
- Obtiene el ID de carrera a eliminar.

- Utiliza gestor_carrera para eliminar la carrera.
- Redirige a la lista.

Endpoint generar_excel:

- Ruta POST /carrera/generar_excel
- Toma todas las carreras de la base y lo exporta a un archivo excel descargable.
- Utiliza gestor_carrera.obtener_todo para obtener todos los datos de la base.

Archivo routes_facultades.py, routes_programas.py, routes_campus.py, routes_universidades.py:

- Implementa las rutas CRUD para los modelos Universidad, Facultades, Programa y Campus según corresponda.
- Endpoints para crear, editar, borrar y listar cada entidad.
- Usa gestor_universidades, gestor_campus, gestor_programas, gestor_facultades para la lógica de negocio según corresponda.

Contiene las rutas o endpoints que manejan las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para cada modelo mencionado
Importa los módulos necesarios como Flask, gestores correspondientes, etc. Define el blueprint “routes_universidades”, “routes_programas”, “routes_campus”, “routes_facultades” según corresponda, que agrupa estas rutas.

Endpoint obtener_lista_paginada:

- Ruta GET /programas o GET/campus o GET/universidades o GET/facultades.
- Obtiene el parámetro page de la URL para la paginación.
- Utiliza el gestor correspondiente para obtener una página de la entidad correspondiente.
- Renderiza la plantilla de html de la entidad correspondiente pasándole la lista paginada.

Endpoints crear_editar_eliminar_programa/crear_editar_eliminar_facultad/ crear_editar_eliminar_universidad/crear_editar_eliminar_campus:

- Ruta GET y POST /”entidad”/
- Si es GET, muestra el formulario para crear.
- Si es POST, obtiene los datos del formulario.
- Utiliza gestor_universidades, gestor_facultades, gestor_programa, gestor_campus para crear, eliminar o editar la nueva entidad según corresponda.
- Obtiene el ID de la entidad a editar de la URL.
- Redirige a la lista o muestra errores.

Endpoint generar_excel:

- Ruta POST /carrera/generar_excel
- Toma todas las carreras de la base y lo exporta a un archivo excel descargable.
- Utiliza gestor_carrera.obtener_todo para obtener todos los datos de la base.

Archivo entities.py

- Define los modelos de la aplicación con Flask-SQLAlchemy
- Clases para Persona, Usuario, Lugar, etc.
- Relaciones entre modelos y métodos de base de datos

Módulo que contiene los modelos de la aplicación definidos con Flask-SQLAlchemy.

Archivo base.py

- Configura la instancia de Flask-SQLAlchemy
- Clase base para los modelos
- Métodos auxiliares para paginación y operaciones comunes

Módulo con la configuración de Flask-SQLAlchemy y métodos base para los modelos.

Archivo gestor_personas.py

- Lógica de negocio y validaciones para operaciones con Personas

- Métodos para crear, editar, borrar y listar personas
- Se utiliza en routes_personas.py

Módulo que encapsula la lógica de negocio reutilizable de Personas. Contiene la lógica de negocio reutilizable para las operaciones con Personas.
Define la clase gestor_personas que encapsula esta lógica.

Método obtener_pagina:

- Utiliza el método de clase obtener_paginado del modelo Persona.
- Obtiene una página de personas según número de página y cantidad por página.

Método obtener:

- Obtiene una persona por su ID.
- Retorna la persona o error si no existe.

Método editar

- Obtiene la persona a editar por ID.
- Valida que los campos obligatorios existan.
- Valida email y fecha única si se incluyen.
- Edita los campos enviados de la persona.

- Guarda los cambios con el método guardar().
- Retorna resultado de la operación.

Método eliminar

- Obtiene la persona por ID.
- Utiliza el método borrar() para eliminarla.
- Retorna resultado de la operación.

Método crear

- Valida que los campos obligatorios existan.

- Valida email y fecha únicos.
- Crea una nueva instancia de Persona con los datos.
- Guarda la nueva persona con el método guardar().
- Retorna resultado de la operación.

Archivo gestor_lugares.py

- Lógica de negocio para operaciones con lugares
- Consulta lugares por diferentes criterios
- Se utiliza en la API de lugares

Módulo con lógica de negocio reutilizable para operaciones con lugares.
Contiene la lógica de negocio reusable para lugares.
Define la clase GestorLugares.

Método consultar()

- Crea consulta SQLAlchemy a la tabla Lugar.
- Filtra por país, provincia, etc. según parámetros.
- Une con tablas relacionadas según criterios.
- Ejecuta la consulta y retorna lugares.

Encapsula la consulta de lugares con varios criterios de búsqueda. Se reutiliza desde el API para proveer la funcionalidad de búsqueda.
En resumen, se separa la lógica de negocio de lugares en un gestor reusable, mientras que el API expone esta funcionalidad via HTTP.

Archivo gestor_comun.py

- Clases base para gestores de entidades
- Métodos y lógica común reutilizable
- Validaciones comunes de emails, fechas, etc.

Módulo con lógica y clases base comunes para los gestores de entidades. En resumen, se tienen:

- Archivos app.py y app_factory.py para crear la aplicación Flask.
- Templates HTML para las vistas.
- Módulos Python para rutas, modelos, lógica de negocio, APIs, etc.

- Varias librerías como Flask-SQLAlchemy, Flask-Login, Flask-WTF, etc.

Archivo gestor_carreras.py

- Lógica de negocio para operaciones con carreras.
- Consulta carreras por diferentes criterios
- Se utiliza en la API de carreras

Módulo con lógica de negocio reutilizable para operaciones con carreras. Contiene la lógica de negocio reutilizable para carreras.
Define la clase gestor_carrera.

Método consultar():

- Crea consulta SQLAlchemy a la tabla Carrera.
- Filtra por universidad,campus,facultad y programa según parámetros.
- Une con tablas relacionadas según criterios.
- Ejecuta la consulta y retorna carreras.

Encapsula la consulta de carreras con varios criterios de búsqueda. Se reutiliza desde el API para proveer la funcionalidad de búsqueda.
En resumen, se separa la lógica de negocio de carreras en un gestor reutilizable, mientras que el API expone esta funcionalidad via HTTP.

Método obtener_pagina:

- Utiliza el método de clase obtener_paginado del modelo Carrera.
- Obtiene una página de carrera según número de página y cantidad por página.

Método obtener_todo:

- Obtiene todas las carreras.
- Retorna todas las carreras en la base de datos.

Método editar_carrera:

- Obtiene la carrera a editar por ID.
- Edita los campos enviados de la carrera.
- Guarda los cambios con el método guardar().
- Retorna resultado de la operación.

Método eliminar

- Obtiene la carrera por ID.
- Utiliza el método borrar() para eliminarla.
- Retorna resultado de la operación.

Método crear:

- Crea una nueva instancia de Carrera con los datos.

- Guarda la nueva carrera con el método guardar().
- Retorna resultado de la operación.

Archivo gestor_carreras_personas.py

- Lógica de negocio y validaciones para operaciones con Personas y carreras.
- Métodos para crear, editar, borrar y listar personas con sus carreras.

Módulo que encapsula la lógica de negocio reutilizable de Personas con sus carreras.
Define la clase gestor_carreras_personas que encapsula esta lógica.

Método obtener_pagina:

- Utiliza el método de clase obtener_paginado del modelo Persona-carreras.
- Obtiene una página de personas-carreras según número de página y cantidad por página.

Método obtener_carreras_por_persona:

- Obtiene una persona y sus carreras por los datos de la persona
- Retorna la persona y sus carreras o error si no existe.

Método obtener:

- Obtiene una persona-carreras por su ID.
- Retorna la persona y sus carreras o error si no existe.

Método editar:

- Obtiene la persona-carrera a editar por ID.
- Valida que los campos obligatorios existan.
- Valida email y fecha única si se incluyen.
- Edita los campos enviados de la persona.
- Guarda los cambios con el método guardar().
- Retorna resultado de la operación.

Método eliminar

- Obtiene la persona-carrera por ID.
- Utiliza el método borrar() para eliminarla.
- Retorna resultado de la operación.

Método crear

- Valida que los campos obligatorios existan.
- Valida email y fecha únicos.
- Crea una nueva instancia de Persona-Carreras con los datos.
- Guarda la nueva persona-carrera con el método guardar().
- Retorna resultado de la operación.

Archivo gestor_tipopersona.py

- Lógica de negocio para operaciones con los tipos de personas
- Contiene métodos para crear, editar, eliminar y consultar tipos personas.
- Se utiliza en la API de tipos de personas.

Módulo con lógica de negocio reutilizable para operaciones con tipos de persona.
Define la clase GestorTiposPersona.

Archivo gestor_generos.py

- Lógica de negocio para operaciones con generos
- Contiene solo un método obtener_todo que retorna todos los géneros en la tabla Géneros de la base de datos.
- Se utiliza en la API de generos.

Módulo con lógica de negocio reutilizable para operaciones con generos.
Define la clase gestor_generos.

Archivo gestor_email.py

- Lógica de negocio para operaciones con email.
- Contiene solo un método enviar_email que contiene toda la lógica para el envío de un email

Módulo con lógica de negocio reutilizable para operaciones con email.
Define la clase gestor_email.

Metodo enviar_email:

- Verifica que la estructura del email se válida, y si la dirección de correo es válida.
- Configura los parámetros necesarios para la conexión SMTP (servidor, puerto,dirección de correo, etc)
- Crea el objeto MIMEMultipart para construir el correo electrónico.
- Si la conexión y la autenticación son exitosas se envía el correo y devuelve mensaje de éxito, sino envía un mensaje de error.

Archivos gestor_campus.py, gestor_programas.py, gestor_universidades.py y gestor_facultades.py:

- Lógica de negocio para operaciones con campus, programas, universidades o facultades respectivamente.
- Se utiliza en la API de carreras.

Módulo con lógica de negocio reutilizable para operaciones con cada una de las entidades mencionadas.
Define la clase gestor_facultades/gestor_universidades/gestor_programas/gestor_campus.

Método consultar():

- Crea consulta SQLAlchemy a la tablas de programas,universidades,campus y facultades respectivamente.
- Filtra por universidad,campus,facultad y programa según parámetros.
- Ejecuta la consulta y retorna cada entidad en su respectiva ventana.

Se reutiliza desde el API para proveer la funcionalidad de búsqueda.
En resumen, se separa la lógica de negocio de todas las entidades que construyen las carreras en un gestor reutilizable, mientras que el API expone esta funcionalidad via HTTP.

Metodo obtener_programa/universidad/campus/facultad:

- Realiza una consulta a la base y retorna los valores de la entidad correspondiente.

Método obtener_pagina:

- Utiliza el método de clase obtener_paginado del modelo de cada entidad de carrera.
- Obtiene una página de cada entidad según número de página y cantidad por página.

Método obtener_todo:

- Obtiene todas las universidades,campus,programas o facultades.
- Retorna todas las universidades, campus, programas o facultades de la base de datos.

Método editar:

- Obtiene la entidad a editar por ID.
- Edita los campos enviados de las entidades.
- Guarda los cambios con el método guardar().
- Retorna resultado de la operación.

Método eliminar

- Obtiene la entidad por ID.
- Utiliza el método borrar() para eliminarla.
- Retorna resultado de la operación.

Método crear:

- Crea una nueva instancia de la entidad con los datos.
- Guarda la nueva entidad con el método guardar().
- Retorna resultado de la operación.

Archivo personas.py

- Implementa el recurso REST de personas
- GET, POST, DELETE para el modelo Persona
- Usa gestor_personas para la lógica Recurso API REST para el modelo Persona.

Archivo lugares.py

- Implementa recurso REST de lugares
- GET para consultar lugares con criterios
- Usa gestor_lugares para la lógica

Recurso API REST para consultar lugares con criterios. Este módulo define el recurso API REST para lugares.

Importa:

- Flask-RESTful: para crear recursos API REST.
- flask_login: para autenticación.
- gestor_lugares: lógica de negocio de lugares.

Define la clase LugaresResource que hereda de Resource de Flask-RESTful.

Método GET

- Obtiene parámetros de consulta como país, provincia, etc.
- Utiliza gestor_lugares para consultar los lugares según criterios.
- Convierte los lugares a JSON con serialize()
- Retorna JSON con los lugares consultados.

Permite buscar lugares por diferentes campos utilizando el gestor_lugares.

Archivo carreras.py

- Implementa el recurso REST de carreras
- GET, POST, DELETE para el modelo Carrera
- Usa gestor_carrera para la lógica Recurso API REST para el modelo Carrera.

Archivo genero.py y tipo.py

- Implementa el recurso REST de genero y tipo de persona
- Método GET para el modelo de genero y tipo de persona
- Usa gestor_genero para la lógica Recurso API REST para el modelo genero.

Archivo carreras_noRelacionadas.py

- Implementa el recurso REST de las carreras de una forma que pueda usarse en los mòdulos de carreras.
- Método GET para el modelo de carreras no relacionadas a personas
- Usa gestor_carrera para la lógica Recurso API REST para el modelo de carreras no relacionadas.
- Usa GestorTiposPersonas para la lógica Recurso API REST para el modelo genero.

Define la clase CarreasNoRelacionadasResource que hereda de Resource de Flask-RESTful.

Archivo alembic/versions:

- Configuraciones para la conexión de alembic con la base de datos en el archivo env.py
- En la carpeta versions están las versiones con las modificaciones a las tablas que se van haciendo.

Protección CSRF (Cross-Site Request Forgery)

La protección CSRF (Cross-Site Request Forgery) sirve para prevenir ataques en los que se envían peticiones no autorizadas desde otro sitio web hacia nuestra aplicación web.
Algunos ejemplos de ataques que previene:

- Un sitio web malicioso que envía una petición POST a nuestro sitio para crear un nuevo usuario sin consentimiento del usuario.
- Un formulario oculto en otra página que envía una petición POST para transferir fondos en un sitio bancario sin que el usuario lo sepa.
- Un botón oculto que aprovecha que el usuario ya inició sesión y envía peticiones GET para cambiar su email o contraseña.

Para prevenir estos ataques, la protección CSRF requiere que cualquier petición POST/PUT/DELETE venga con un token CSRF que solo la aplicación conoce y valida.
Este token se genera en el formulario cuando se renderiza y debe coincidir con el que espera la aplicación al recibir la petición.

Así se asegura que la petición viene de una página controlada por la aplicación y no de un sitio externo malicioso.
CSRF protege contra la falsificación de peticiones desde otros sitios y previene que se ejecuten acciones no autorizadas en nuestra aplicación web.

Python

Se utiliza la extensión Flask-WTF para protección CSRF.
Se crea una instancia de CSRFProtect y se inicializa con la app:

```
from flask_wtf.csrf import CSRFProtect

csrf = CSRFProtect()

csrf.init_app(app)
```

Esto activa la protección CSRF automáticamente en todos los formularios.

HTML

En los templates con formularios se agrega:

```
<form>
      <!-- Campos del formulario -->

      <input type="hidden" name="csrf_token" value="{{ csrf_token()
}}">

</form>
```

Esto renderiza un campo oculto con el token CSRF actual.

Validación

Al recibir el formulario, Flask-WTF valida que:

- El campo csrf_token exista y sea válido.
- El valor coincida con el token esperado. Si no coinciden, se produce un error CSRF 400.

Esto protege contra ataques que envíen formularios falsificados desde otros sitios.

Ventajas

- Protección CSRF automática en todos los formularios.
- No se requiere código adicional en las vistas.
- Fácil de implementar en HTML con {{ csrf_token() }}
- Mayor seguridad sin comprometer facilidad de uso.

Flask-WTF y csrf_token() proveen una protección CSRF simple y efectiva, previniendo ataques comunes en aplicaciones web.

Uso en la solución:

Aquí está la explicación del uso de CSRF basada en el código específico de la solución propuesta:

app_factory.py

Se crea una instancia de CSRFProtect y se inicializa con la aplicación Flask:

```
from flask_wtf.csrf import CSRFProtect

...
csrf = CSRFProtect()
csrf.init_app(app)
```

Esto activa la protección CSRF en toda la aplicación.

routes_personas.py

Se importa CSRFProtect:

```
from flask_wtf.csrf import CSRFProtect

...

csrf = CSRFProtect()
```

templates/personas/crear_persona.html

El formulario de crear persona tiene un campo hidden con el token:

```
<form method="POST">

     <input type="hidden" name="csrf_token" value="{{ csrf_token()
}}">

</form>
```

templates/personas/editar_persona.html

El formulario de editar persona también tiene el campo CSRF:

```
<form method="POST">

<input type="hidden" name="csrf_token" value="{{ csrf_token() }}">

</form>
```

De esta manera, todos los formularios quedan protegidos contra ataques CSRF.
Cuando se reciben los formularios en los endpoints de crear y editar persona, Flask- WTF validará que el csrf_token sea válido antes de procesar la petición, evitando así la falsificación de solicitudes.
Esta implementación aprovecha la integración directa entre Flask-WTF y CSRFProtect para una protección CSRF simple y efectiva.

Diagrama de la base de datos: