17 Django-marcador

¿Que es FastAPI?

FastAPI es un framework web moderno y de alto rendimiento para Python, diseñado para construir APIs de forma rápida y eficiente, aprovechando las anotaciones de tipo (type hints) de Python para validación de datos automática, serialización y generación de documentación interactiva (Swagger/OpenAPI). Se basa en **Starlette** y **Pydantic**,

- ➔ Starlette es un framework web ASGI ligero para servicios asíncronos en Python
- ➔ Pydantic es una biblioteca para la validación y serialización de datos basada en tipos de Python

ofreciendo

- ➔ **velocidad** comparable a Node.js y Go,
- ➔ **facilidad de uso**,
- ➔ **autocompletado** y
- ➔ soporte para **programación asíncrona** (async/await).

Características clave:

- ➔ **Alto Rendimiento**: Extremadamente rápido gracias a su base en Starlette y su ejecución sobre servidores ASGI.
- ➔ **Tipado y Validación**: Usa anotaciones de tipo de Python y Pydantic para validar automáticamente los datos de entrada y salida, reduciendo errores.
- ➔ **Documentación Automática**: Genera documentación interactiva (Swagger UI y ReDoc) de forma automática a partir del código, permitiendo probar los endpoints.
- ➔ **Asincronía**: Soporta operaciones asíncronas y concurrentes de forma nativa (async/await).
- ➔ **Fácil y Rápido de Desarrollar**: Permite escribir menos código y acelerar el desarrollo, enfocándose en la lógica de negocio.
- ➔ **Basado en Estándares**: Compatible con OpenAPI y JSON Schema.

¿Para qué se usa?:

Es ideal para crear backends de aplicaciones web, microservicios, servicios de IA/Machine Learning y APIs RESTful que requieran velocidad y eficiencia, siendo utilizado por empresas como Netflix, Microsoft y Uber.

Comparativa Django vs FastAPI

Si vienes de **Django**, entender **FastAPI** es como pasar de conducir un camión de carga (potente y con todo incluido) a una moto de carreras (ligera, veloz y personalizable).

17 Django-marcador

Tras haber trabajado con **Django**, has experimentado la filosofía *"Batteries Included"* (Pilas incluidas). Ahora, con **FastAPI**, entramos en el mundo del alto rendimiento y la flexibilidad extrema.

1. Comparativa de Filosofías

Característica	Django	FastAPI
Concepto	**Monolito:** Lo trae todo de serie (Admin, Auth, ORM).	**Micro-framework:** Tú eliges qué piezas añadir.
Arquitectura	**MVT** (Model-View-Template).	**Basada en APIs:** Pensada para el backend puro.
ejecución	Síncrona (procesa una tarea tras otra).	**Asíncrona (async/await):** Procesa miles de tareas a la vez.
Validación	Django Forms / Serializers (DRF).	**Pydantic:** Validación automática mediante tipos de Python.
Documentación	Manual o vía librerías externas.	**Automática:** Swagger y ReDoc integrados nativamente.

2. Las Metáforas de Uso

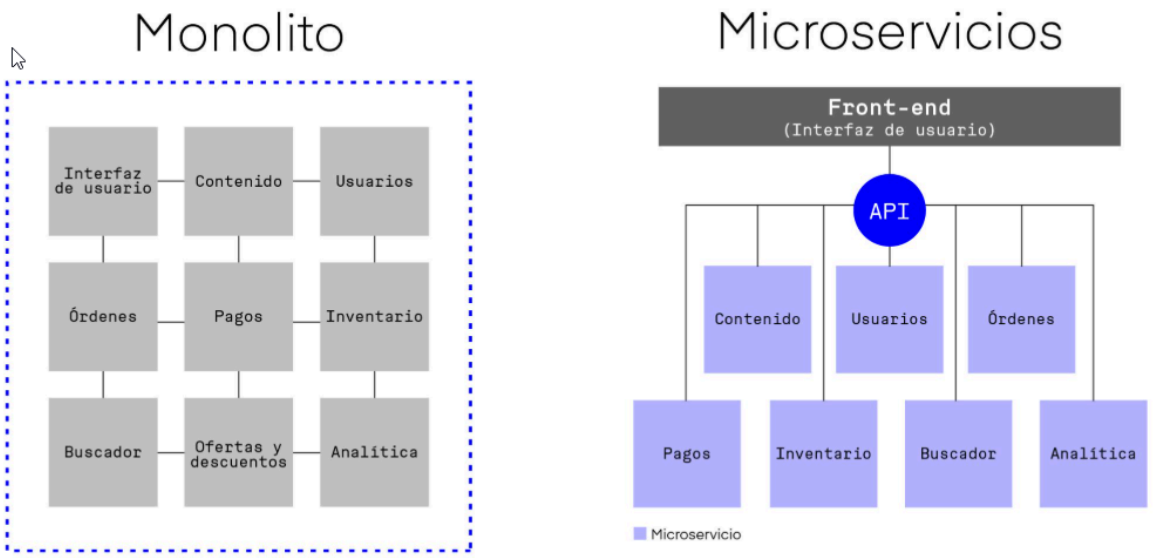
- ➔ **Django** es como una Casa Llave en Mano: Cuando entras, ya tienes la cocina, los baños y los muebles instalados. No puedes mover los muros fácilmente, pero puedes empezar a vivir en ella hoy mismo. Es ideal para aplicaciones de gestión, CMS o paneles de administración complejos.

- ➔ **FastAPI** es como un Set de LEGO: Te dan una base sólida y piezas de alta precisión. Tú decides si construyes un coche, un avión o un cohete. Es la opción preferida para microservicios, aplicaciones de Inteligencia Artificial y sistemas que necesitan manejar mucho tráfico con pocos recursos.

¿Qué es un microservicio?

17 Django-marcador

Los microservicios son un tipo de arquitectura de software que separa todos los servicios de una aplicación en módulos pequeños e independientes.



3. ¿Por qué aprender FastAPI?

- ➔ **Rendimiento**: FastAPI es uno de los frameworks más rápidos del mundo (a la par que Go y Node.js), gracias a su naturaleza asíncrona.
- ➔ **Estándares Modernos**: Utiliza las últimas funciones de Python (Type Hints), lo que hace que el código sea más limpio y haya menos errores.
- ➔ **El mercado de la IA**: Casi todos los servicios modernos de IA y Machine Learning se despliegan con FastAPI debido a su ligereza.
- ➔ **Productividad**: La documentación interactiva (Swagger) permite probar los endpoints mientras los programas, sin necesidad de usar herramientas externas, como Postman, desde el minuto uno.

4. Resumen: ¿Cuándo usar cada uno?

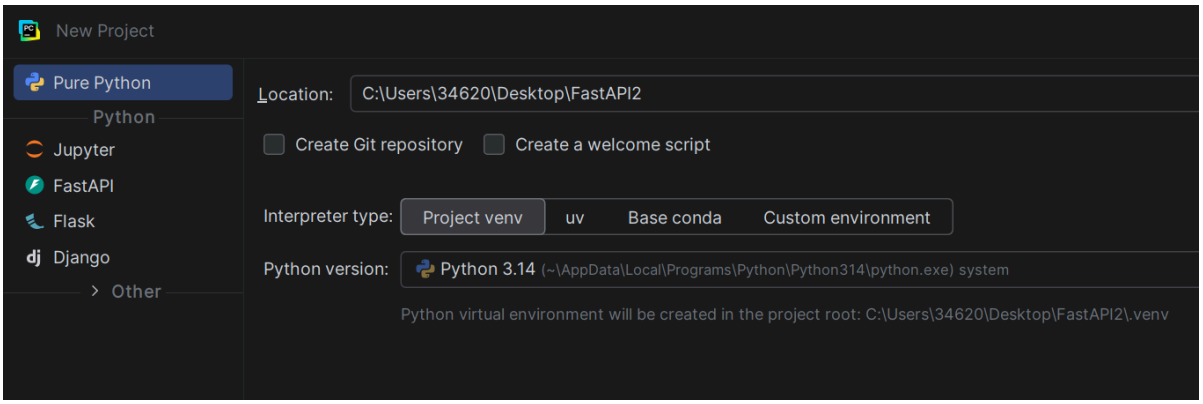
- ➔ Usa Django cuando... necesites un panel de administración potente de inmediato, manejo de usuarios complejo ya resuelto y una estructura muy rígida y segura.
- ➔ Usa FastAPI cuando... busques velocidad extrema, necesites crear una API para una aplicación móvil o frontend moderno (React/Vue), o estés integrando modelos de Inteligencia Artificial.

17 Django-marcador

Consejo final: "No se trata de cuál es mejor, sino de qué herramienta es la adecuada para el problema. Django os ha enseñado a construir aplicaciones robustas; FastAPI os enseñará a construir sistemas de alto rendimiento".

Instalación de FastAPI

Preparamos un **entorno virtual** con PyCharm.



Ahora, instala FastAPI:

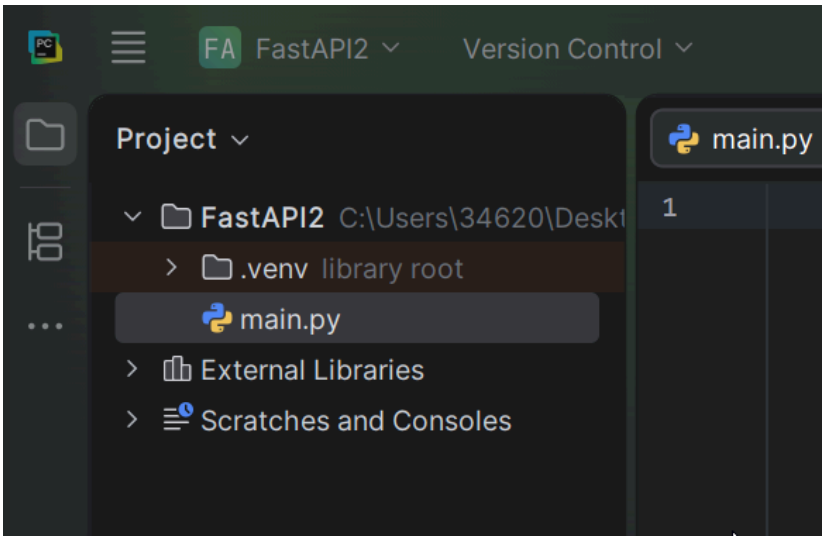
```
pip3 install fastapi
```

FastAPI es un marco para construir APIs, pero para probar tus APIs necesitarás un servidor web local. Uvicorn es un servidor web de interfaz de puerta de enlace de servidor asíncrono (ASGI) para Python, muy rápido, que es ideal para el desarrollo. Para instalar Uvicorn, ejecuta este comando:

```
pip3 install "uvicorn[standard]"
```

Tras la instalación, crea un archivo llamado main.py en el directorio de trabajo de tu proyecto. Este archivo será el punto de entrada de tu aplicación.

17 Django-marcador



Un Ejemplo Rápido de FastAPI, Para probar tu instalación configura rápidamente un endpoint de ejemplo. En tu archivo main.py, pega el siguiente código y guarda el archivo:

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
async def root():
    return {"greeting": "Hello world"}
```

El fragmento anterior crea un endpoint FastAPI básico. Un resumen de lo que hace cada línea:

- ➔ **from fastapi import FastAPI**: La funcionalidad de tu API la proporciona la clase FastAPI de Python.
- ➔ **app = FastAPI()**: Esto crea una instancia de FastAPI.
- ➔ **@app.get("/")**: Este es un decorador que especifica a FastAPI que la función que hay debajo es la encargada de gestionar las peticiones. Esto crea un método GET en la ruta del sitio. El resultado es devuelto por la función envuelta.

Otras posibles operaciones que se utilizan para comunicarse son

@app.post(), **@app.put()**, **@app.delete()**, y menos usuales **@app.options()**, **@app.head()**, **@app.patch()** y **@app.trace()**.

En el directorio de archivos, ejecuta el siguiente comando en tu terminal para iniciar el servidor API:

```
$ uvicorn main:app --reload
```

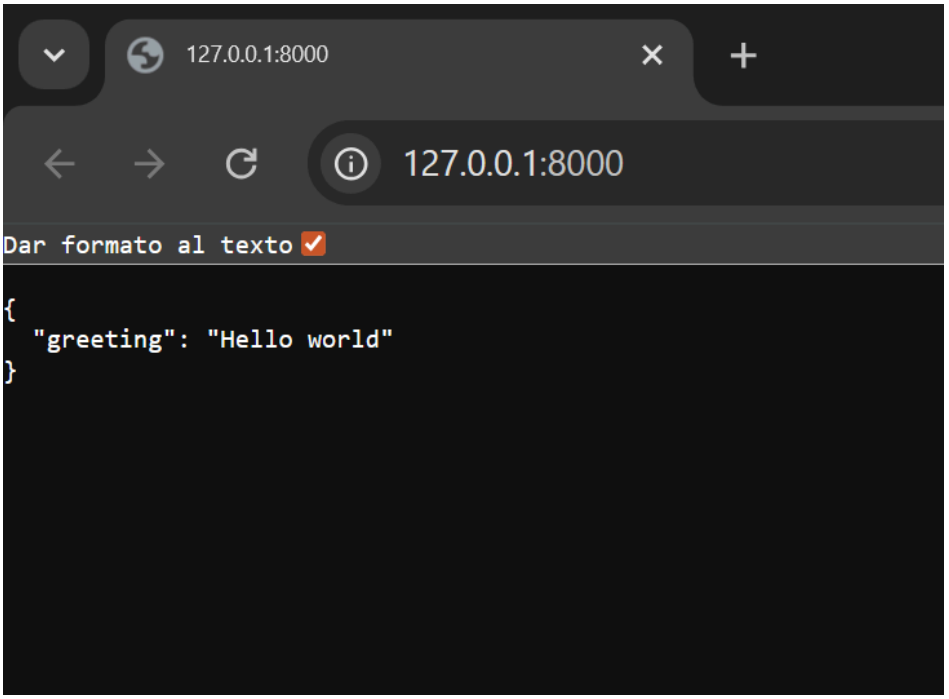
17 Django-marcador

En este comando, **main** es el nombre de tu módulo. El objeto **app** es una instancia de tu aplicación, y se importa en el servidor ASGI. La bandera **--reload** indica al servidor que se recargue automáticamente cuando hagas algún cambio.

Deberías ver algo así en tu terminal:

```
 $ uvicorn main:app --reload

INFO: Will watch for changes in these directories: ['D:\WEB
DEV\Eunit\Tests\fast-api']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [26888] using WatchFiles
INFO: Started server process [14956]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

En tu navegador, navega hasta http://localhost:8000 para confirmar que tu API está funcionando. Deberías ver «Hello»: «World» como objeto JSON en la página. Esto ilustra lo fácil que es crear una API con FastAPI. Lo único que has tenido que hacer es definir una ruta y devolver tu diccionario Python.



Utiliza Type Hints

17 Django-marcador

Si utilizas Python, estás acostumbrado a anotar variables con tipos de datos básicos como **int**, **str**, **float** y **bool**. Sin embargo, a partir de la versión 3.9 de Python, se introdujeron estructuras de datos avanzadas.

Esto te permite trabajar con estructuras de datos como **dictionaries**, **tuples**, y **lists**. Con los type hints de FastAPI puedes estructurar el esquema de tus datos utilizando modelos pydánticos y luego, utilizar los modelos pydánticos para hacer sugerencias de tipo y beneficiarte de la validación de datos que se proporciona.

En el siguiente ejemplo, se demuestra el uso de type hints en Python con una sencilla calculadora de precios de comidas:

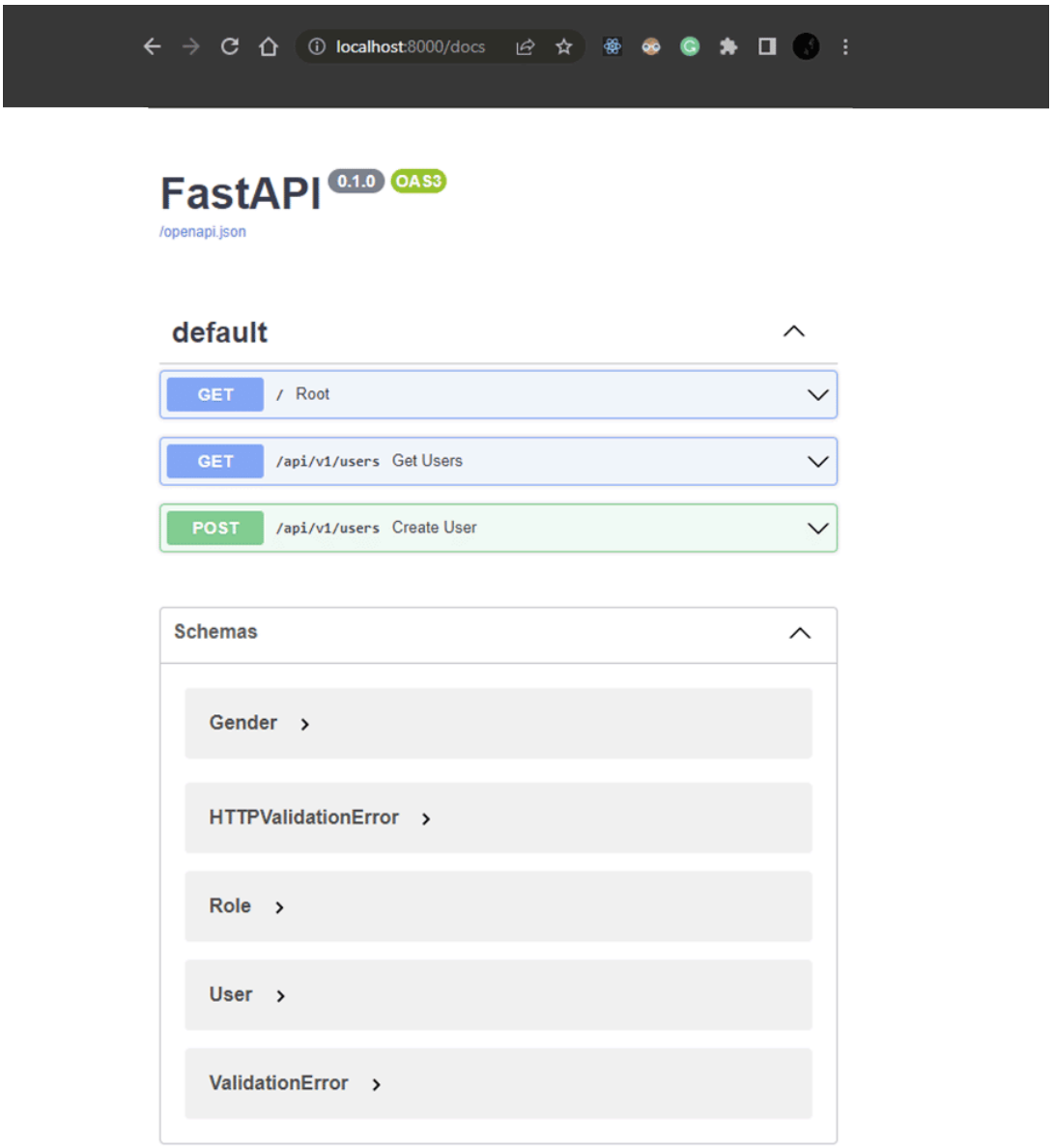
```
def calculate_meal_fee(beef_price: int, meal_price: int) -> int:
    total_price: int = beef_price + meal_price
    return total_price

print("Calculated meal fee", calculate_meal_fee(75, 19))
```

Ten en cuenta que los type hints no cambian la forma en que se ejecuta tu código.

Documentación Interactiva de la API de FastAPI

FastAPI utiliza Swagger UI para proporcionar documentación interactiva automática de la API. Para acceder a ella, navega a http://localhost:8000/docs y verás una pantalla con todos tus endpoints, métodos y esquemas.

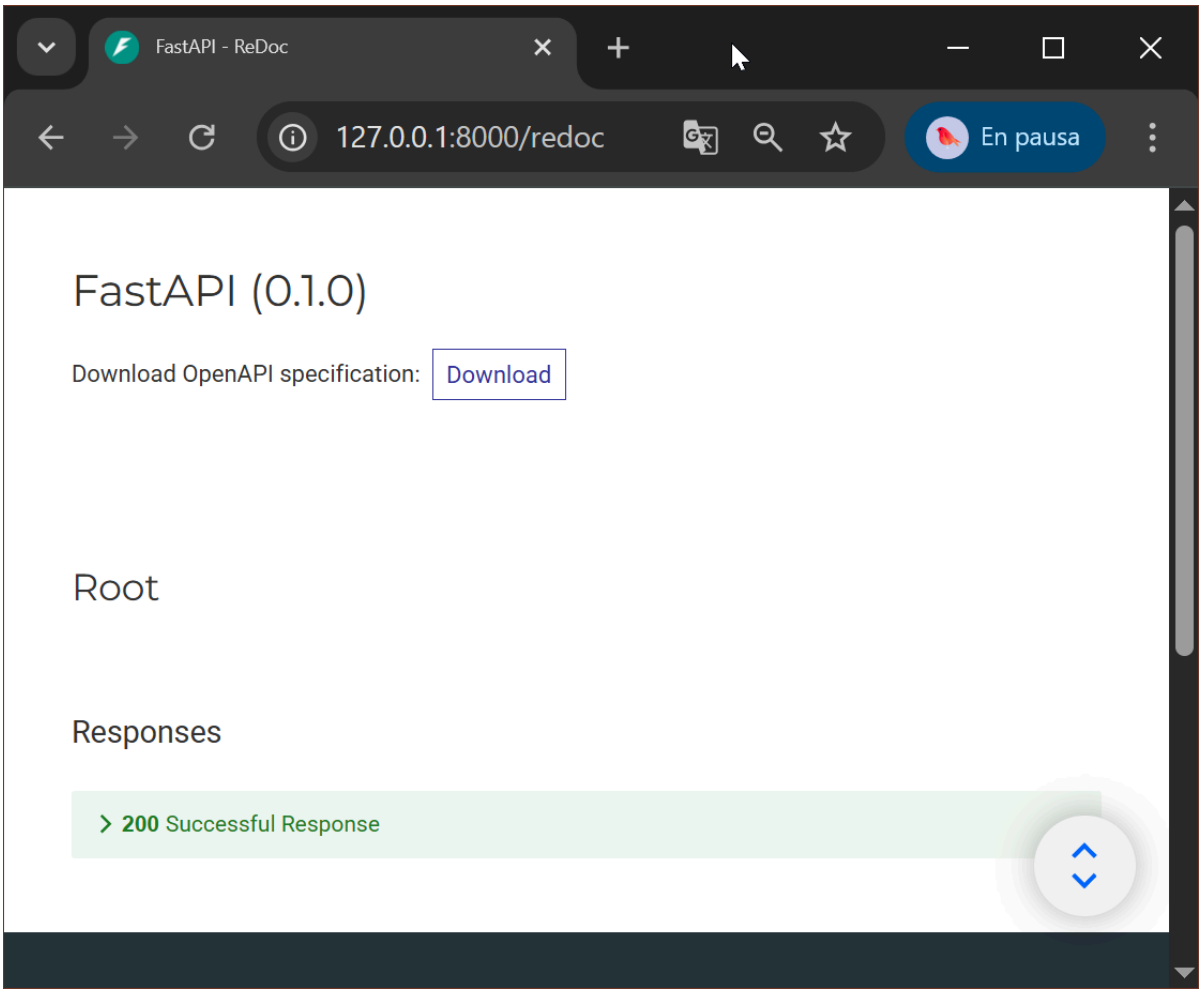
17 Django-marcador



Esta documentación automática de la API basada en el navegador la proporciona FastAPI, y no necesitas hacer nada más para aprovecharla.

Una documentación alternativa de la API basada en el navegador, también proporcionada por FastAPI, es **Redoc**. Para acceder a Redoc, navega a http://localhost:8000/redoc, donde se te presentará una lista de tus puntos finales, los métodos y sus respectivas respuestas.

17 Django-marcador



Configurar Rutas en FastAPI

El decorador @app te permite especificar el método de la ruta, como @app.get o @app.post, y admite GET, POST, PUT y DELETE, así como las opciones menos comunes, HEAD, PATCH y TRACE.

Construye Tu Aplicación con FastAPI

Vamos a construir una aplicación CRUD con FastAPI. La aplicación podrá:

- ➔ Crear un usuario
- ➔ Leer el registro de un usuario en la base de datos
- ➔ Actualizar un usuario existente
- ➔ Eliminar un usuario concreto

Para ejecutar estas operaciones CRUD, crearás métodos que expongan los endpoints de la API. El resultado será una base de datos en memoria que puede almacenar una lista de usuarios.

17 Django-marcador

Utilizarás la biblioteca pydantic para realizar la validación de los datos y la gestión de la configuración mediante anotaciones de tipo en Python.

Vamos a tener la base de datos en memoria. Esto es para iniciarte rápidamente en el uso de FastAPI para construir tus APIs. Sin embargo, para la versión en vivo, puedes utilizar cualquier base de datos que elijas, como PostgreSQL, MySQL, SQLite o incluso Oracle.

Construir la Aplicación

Comenzarás creando tu modelo de usuario. El modelo de usuario tendrá los siguientes atributos:

- id: Un identificador único universal (UUID)
- first_name: El nombre del usuario
- last_name: El apellido del usuario
- gender: El género del usuario
- roles que es una lista que contiene los roles *admin* y *user*

Empieza creando un nuevo archivo llamado models.py en tu directorio de trabajo, y luego pega el siguiente código en models.py para crear tu modelo:

models.py

```
from typing import List, Optional
from uuid import UUID, uuid4
from pydantic import BaseModel
from enum import Enum

class Gender(str, Enum):
    male = "male"
    female = "female"

class Role(str, Enum):
    admin = "admin"
    user = "user"

class User(BaseModel):
    id: Optional[UUID] = uuid4()
    first_name: str
    last_name: str
    gender: Gender
    roles: List[Role]
```

17 Django-marcador

En el código anterior

- ➔ Tu clase User extiende BaseModel, que a su vez se importa de pydantic.
- ➔ Has definido los atributos del usuario, como se ha comentado anteriormente.

El siguiente paso es crear tu base de datos. Sustituye el contenido de tu archivo main.py por el siguiente código:

```
# main.py
from typing import List
from uuid import uuid4
from fastapi import FastAPI
from models import Gender, Role, User

app = FastAPI()
db: List[User] = [
    User(
        id=uuid4(),
        first_name="John",
        last_name="Doe",
        gender=Gender.male,
        roles=[Role.user],
    ),
    User(
        id=uuid4(),
        first_name="Jane",
        last_name="Doe",
        gender=Gender.female,
        roles=[Role.user],
    ),
    User(
        id=uuid4(),
        first_name="James",
        last_name="Gabriel",
        gender=Gender.male,
        roles=[Role.user],
    ),
    User(
        id=uuid4(),
        first_name="Eunit",
        last_name="Eunit",
        gender=Gender.male,
        roles=[Role.admin, Role.user],
    ),
```

```
]
```

Nota: El uuid4 en Python es una función del módulo estándar uuid que genera un identificador único universal (UUID) de la versión 4, basado completamente en números aleatorios. Produce una cadena de 36 caracteres alfanuméricos (32 dígitos hexadecimales y 4 guiones) con una probabilidad de colisión extremadamente baja, ideal para claves primarias en bases de datos.

En main.py:

- ➔ Has inicializado db con un tipo List, y has pasado el modelo
- ➔ Creaste una base de datos en memoria con cuatro usuarios, cada uno con los atributos necesarios como first_name, last_name, gender, y roles. Al usuario Eunit se le asignan los roles de admin y user, mientras que a los otros tres usuarios sólo se les asigna el rol de user.

Leer Registros de la Base de Datos

Has configurado con éxito tu base de datos en memoria y la has llenado de usuarios, así que el siguiente paso es configurar un punto final que devuelva una lista de todos los usuarios. Aquí es donde entra FastAPI.

En tu archivo main.py, pega el siguiente código justo debajo de tu punto final Hello World:

```
# main.py
@app.get("/api/v1/users")
async def get_users():
    return db
```

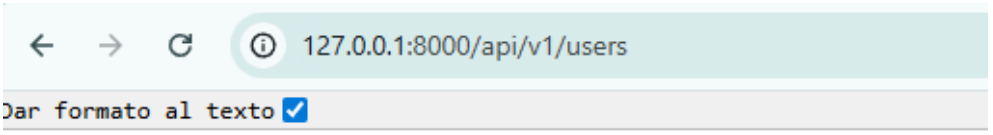
Este código define el punto final /api/v1/users, y crea una función asíncrona, get_users, que devuelve todo el contenido de la base de datos, db.

Guarda tu archivo y podrás probar tu endpoint de usuario. Ejecuta el siguiente comando en tu terminal para iniciar el servidor API:

```
uvicorn main:app --reload
```

En tu navegador, navega hasta http://localhost:8000/api/v1/users. Esto debería devolver una lista de todos tus usuarios, como se ve a continuación:

17 Django-marcador




```
[
  {
    "id": "9e0d48a5-5346-40d1-b048-49c3fb563b22",
    "first_name": "John",
    "last_name": "Doe",
    "gender": "male",
    "roles": [
      "user"
    ]
  },
  {
    "id": "ab54ed35-76b0-44c1-babf-ea8463d418ac",
    "first_name": "Jane",
    "last_name": "Doe",
    "gender": "female",
    "roles": [
      "user"
    ]
  },
  {
    "id": "7caaede8-cbb5-4fd2-8f83-e8803907cda0",
    "first_name": "James",
    "last_name": "Gabriel",
    "gender": "male",
    "roles": [
      "user"
    ]
  },
  {
    "id": "a6ea81f3-8798-4f18-baf6-779c651ccab0",
    "first_name": "Eunit",
    "last_name": "Eunit",
    "gender": "male",
    "roles": [
      "admin",
      "user"
    ]
  }
]
```


En esta fase, tu archivo main.py tendrá el siguiente aspecto:

```python
# main.py
from typing import List
from uuid import uuid4
from fastapi import FastAPI
from models import Gender, Role, User

app = FastAPI()
db: List[User] = [
    User(
```

```
        id=uuid4(),
        first_name="John",
        last_name="Doe",
        gender=Gender.male,
        roles=[Role.user],
    ),
    User(
        id=uuid4(),
        first_name="Jane",
        last_name="Doe",
        gender=Gender.female,
        roles=[Role.user],
    ),
    User(
        id=uuid4(),
        first_name="James",
        last_name="Gabriel",
        gender=Gender.male,
        roles=[Role.user],
    ),
    User(
        id=uuid4(),
        first_name="Eunit",
        last_name="Eunit",
        gender=Gender.male,
        roles=[Role.admin, Role.user],
    ),
]


@app.get("/")
async def root():
    return {"Hello": "World", }


@app.get("/api/v1/users")
async def get_users():
    return db
```

Crear Registros de la Base de Datos

El siguiente paso es crear un endpoint para crear un nuevo usuario en tu base de datos. Pega el siguiente fragmento en tu archivo main.py:

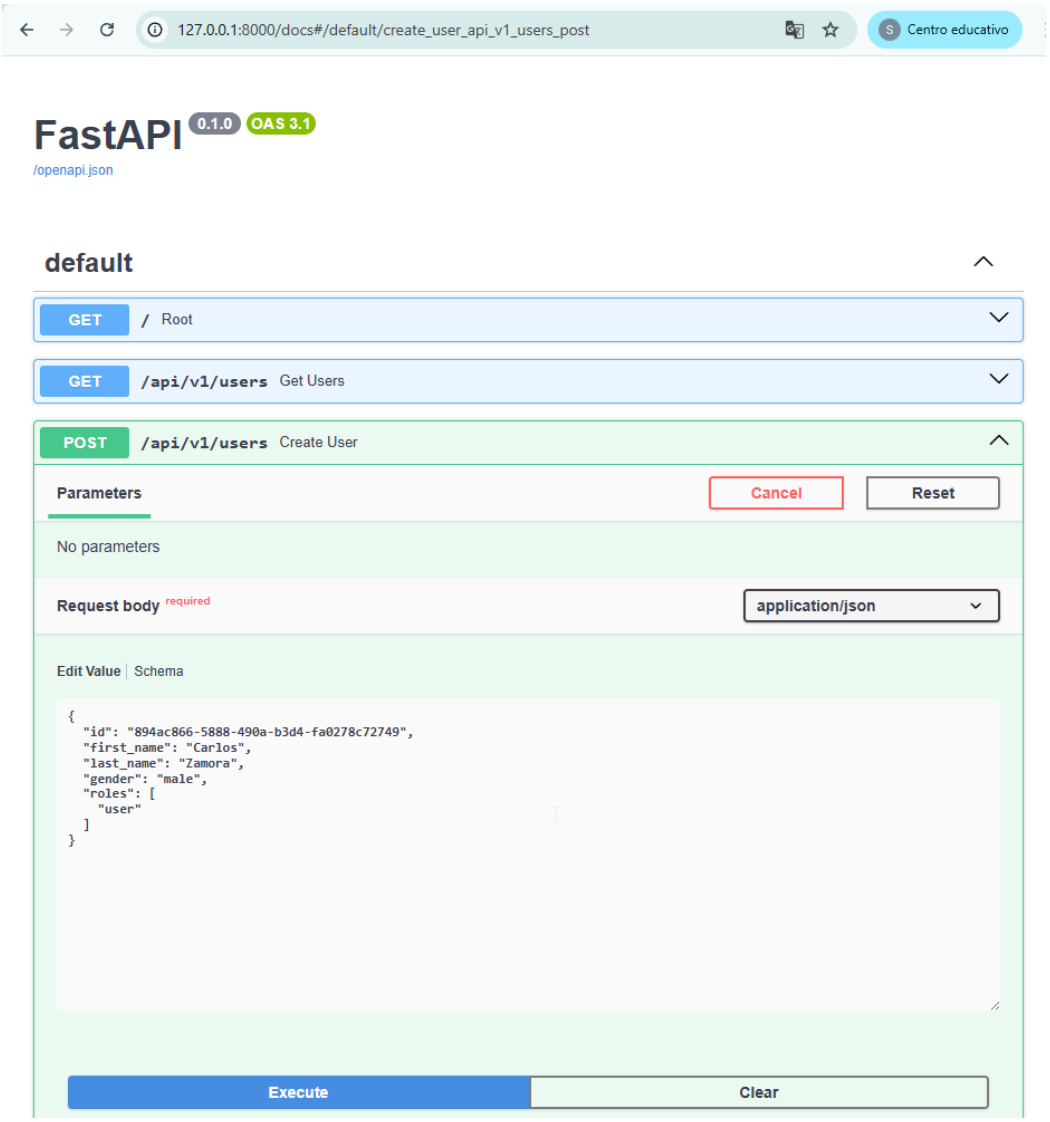
17 Django-marcador

```python
# main.py
@app.post("/api/v1/users")
async def create_user(user: User):
    db.append(user)
    return {"id": user.id}
```

En este fragmento, has definido el endpoint para enviar un nuevo usuario y has utilizado el decorador `@app.post` para crear un método `POST`.

También has creado la función `create_user`, que acepta `user` del modelo `User`, y has añadido el recién creado `user` a la base de datos. Finalmente, el endpoint devuelve un objeto JSON del usuario recién creado id.

Tendrás que utilizar la documentación automática de la API proporcionada por FastAPI para probar tu endpoint, como se ha visto anteriormente. Esto se debe a que no puedes hacer una solicitud de post con el navegador web. Navega a http://localhost:8000/docs para probar utilizando la documentación proporcionada por SwaggerUI.



Eliminar Registros de la Base de Datos

Como estás construyendo una aplicación CRUD, tu aplicación necesitará tener la capacidad de eliminar un recurso específico. Para este tutorial, crearás un endpoint para eliminar un usuario.

Pega el siguiente código en tu archivo main.py:

```
from fastapi import HTTPException
from uuid import UUID

@app.delete("/api/v1/users/{id}")
```

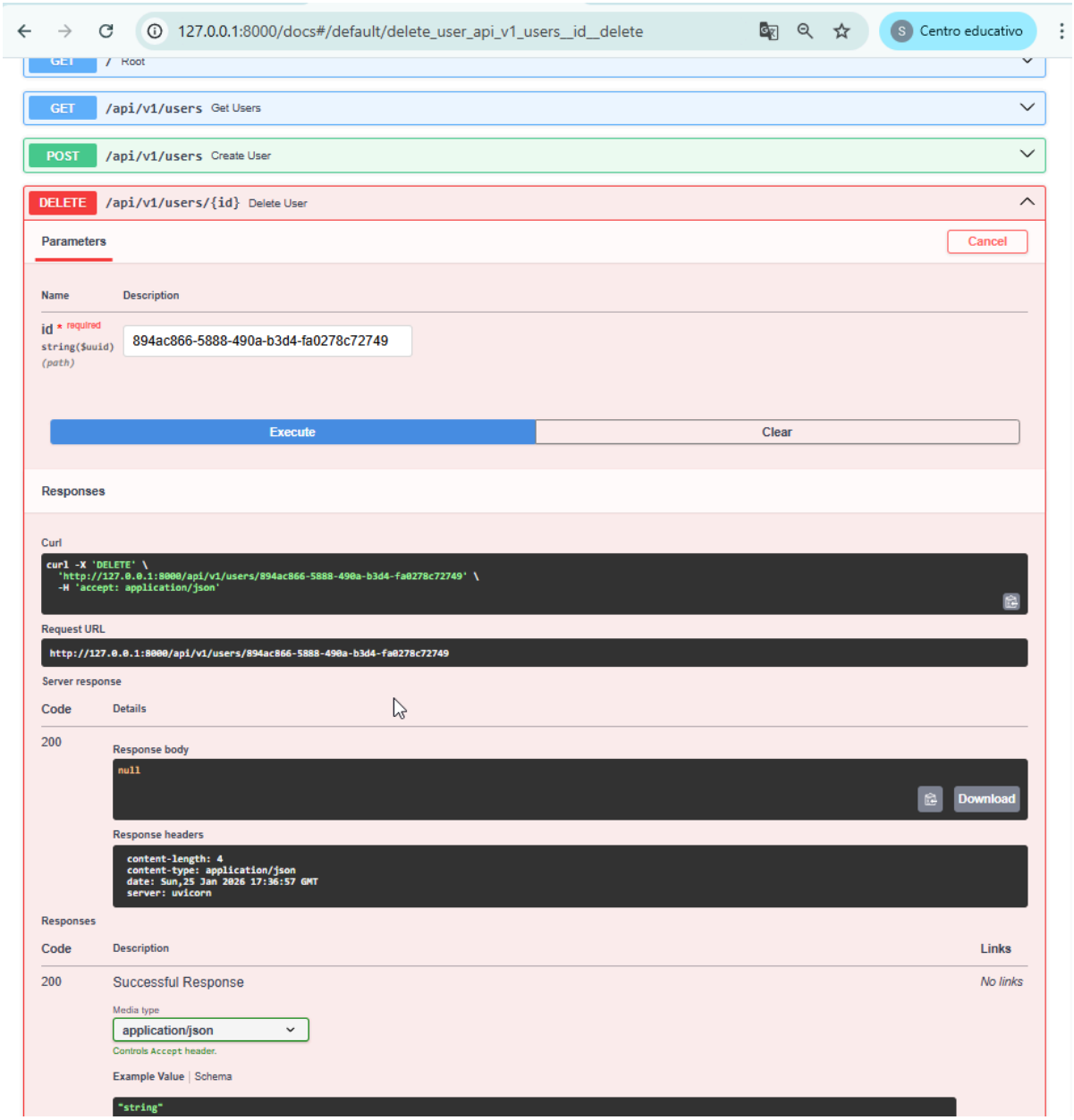
17 Django-marcador

```
async def delete_user(id: UUID):
    for user in db:
        if user.id == id:
            db.remove(user)
            return
    raise HTTPException(status_code=404, detail=f"Delete user failed, id {id} not found.")
```

Aquí tienes un desglose línea por línea de cómo funciona ese código:

- ➔ `@app.delete("/api/v1/users/{id}")`: Has creado el endpoint de eliminación utilizando el decorador `@app.delete()`. La ruta sigue siendo `/api/v1/users/{id}`, pero entonces recupera la id, que es una variable de ruta correspondiente al id del usuario.
- ➔ `async def delete_user(id: UUID):`: Crea la función delete_user, que recupera el id de la URL.
- ➔ `for user in db:`: Esto le dice a la aplicación que haga un bucle a través de los usuarios de la base de datos, y compruebe si el `id` pasado coincide con un usuario de la base de datos.
- ➔ `db.remove(user)`: Si el id coincide con un usuario, éste será eliminado; en caso contrario, se lanzará un `HTTPException` con un código de estado `404`.

17 Django-marcador



Actualizar Registros de la Base de Datos

Vas a crear un endpoint para actualizar los detalles de un usuario. Los detalles que se pueden actualizar incluyen los siguientes parámetros: `first_name`, `last_name`, y `roles`.

En tu archivo `models.py`, pega el siguiente código bajo tu modelo User, es decir, después de la clase `User(BaseModel):`:

```
# models.py
class UpdateUser(BaseModel):
    first_name: Optional[str]
```

```
    last_name: Optional[str]
    roles: Optional[List[Role]]
```

En este fragmento, la clase UpdateUser extiende BaseModel. A continuación, establece que los parámetros de usuario actualizables, como first_name, last_name, y roles, sean opcionales.

Ahora crearás un endpoint para actualizar los datos de un usuario concreto. En tu archivo main.py, pega el siguiente código después del decorador @app.delete:

```
# main.py
@app.put("/api/v1/users/{id}")
async def update_user(user_update: UpdateUser, id: UUID):
    for user in db:
        if user.id == id:
            if user_update.first_name is not None:
                user.first_name = user_update.first_name
            if user_update.last_name is not None:
                user.last_name = user_update.last_name
            if user_update.roles is not None:
                user.roles = user_update.roles
            return user.id
    raise HTTPException(status_code=404, detail=f"Could not find user
with id: {id}")
```

En el código anterior, has hecho lo siguiente

- ➔ Creado @app.put("/api/v1/users/{id}"), el endpoint de actualización. Tiene un parámetro variable id que corresponde al id del usuario.
- ➔ Creó un método llamado update_user, que toma la clase UpdateUser y id.
- ➔ Utiliza un bucle for para comprobar si el usuario asociado al id pasado está en la base de datos.
- ➔ Comprueba si alguno de los parámetros del usuario es is not None (no nulo). Si algún parámetro, como first_name, last_name, o roles, no es nulo, se actualiza.
- ➔ Si la operación tiene éxito, se devuelve el identificador del usuario.
- ➔ Si no se localiza al usuario, se lanza una excepción HTTPException con un código de estado 404 y un mensaje Could not find user with id: {id}.

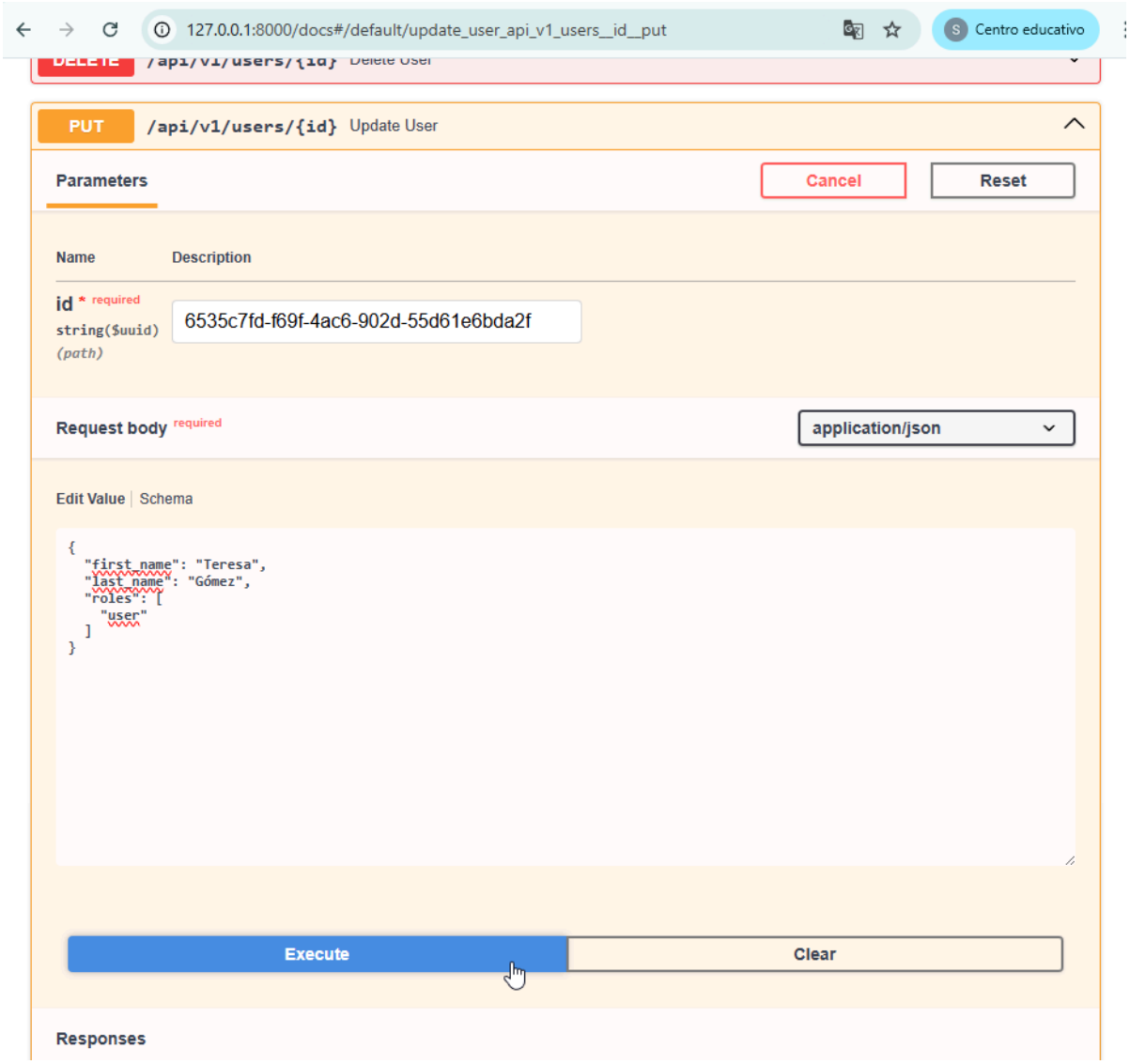
Para probar este endpoint, asegúrate de que tu servidor Uvicorn se está ejecutando.

Si no se está ejecutando, introduce este comando:

17 Django-marcador

```
uvicorn main:app --reload
```

A continuación se muestra una captura de pantalla de la prueba.



Resumen

Has conocido el framework FastAPI para Python y has comprobado por ti mismo lo rápido que puedes poner en marcha una aplicación potenciada por FastAPI. Has aprendido a construir endpoints de la API CRUD utilizando el framework: crear, leer, actualizar y eliminar registros de la base de datos.

17 Django-marcador

FastAPI + Frontend estático

Para servir un Frontend estático dentro de FastAPI, seguiremos estos pasos:

Paso 1. Organización de archivos

Lo ideal es que tu proyecto se vea así para mantener el orden:

```
.
├── main.py
├── models.py
├── static/
│   ├── style.css
│   └── script.js
└── templates/
    └── index.html
```

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>User Management</title>
    <link rel="stylesheet" href="{{ url_for('static', path='/style.css')
}}">
</head>
<body>
<div class="app-container">
    <header>
        <h1>Dashboard de Usuarios</h1>
        <button id="refresh-btn">Actualizar Lista</button>
    </header>

    <div class="card" style="margin-bottom: 30px;">
        <h2>Añadir Nuevo Usuario</h2>
        <form id="userForm">
            <div style="display: flex; gap: 10px; margin-bottom: 10px;">
                <input type="text" id="firstName" placeholder="Nombre"
required
                       style="flex:1; padding: 10px; border-radius: 5px;
border: 1px solid #ddd;">
                <input type="text" id="lastName" placeholder="Apellido"
required
                       style="flex:1; padding: 10px; border-radius: 5px;
```

```
border: 1px solid #ddd;">
            </div>
            <div style="display: flex; gap: 10px; margin-bottom: 15px;">
                <select id="gender" required style="flex:1; padding:
10px; border-radius: 5px; border: 1px solid #ddd;">
                    <option value="male">Masculino</option>
                    <option value="female">Femenino</option>
                </select>
                <select id="role" required style="flex:1; padding: 10px;
border-radius: 5px; border: 1px solid #ddd;">
                    <option value="user">User</option>
                    <option value="admin">Admin</option>
                    <option value="student">Student</option>
                </select>
            </div>
            <button type="submit" id="submit-btn"
                    style="width: 100%; background: #00d1b2; color:
white; border: none; padding: 12px; border-radius: 5px; cursor: pointer;
font-weight: bold;">
                Guardar Usuario
            </button>
        </form>
    </div>

    <div class="table-card">
        <table id="userTable">
            <thead>
            <tr>
                <th>Nombre</th>
                <th>Género</th>
                <th>Roles</th>
            </tr>
            </thead>
            <tbody id="userBody">
            </tbody>
        </table>
    </div>
</div>
<script src="{{ url_for('static', path='/script.js') }}"></script>
</body>
</html>
```

style.css

```
body {
```

17 Django-marcador

```
    background-color: #f0f2f5;
    font-family: 'Inter', sans-serif;
    display: flex;
    justify-content: center;
    padding: 40px;
}
.app-container { width: 100%; max-width: 900px; }
.table-card {
    background: white;
    border-radius: 10px;
    padding: 20px;
    box-shadow: 0 4px 12px rgba(0,0,0,0.05);
}
table { width: 100%; border-collapse: collapse; }
th { text-align: left; padding: 12px; border-bottom: 2px solid #eee;
color: #666; }
td { padding: 12px; border-bottom: 1px solid #eee; }
#refresh-btn {
    background: #4f46e5;
    color: white;
    border: none;
    padding: 10px 20px;
    border-radius: 6px;
    cursor: pointer;
    margin-bottom: 20px;
}
```

script.js

```
async function loadUsers() {
    const response = await fetch('/api/v1/users');
    const users = await response.json();
    const container = document.getElementById('userBody');

    container.innerHTML = users.map(u => `
        <tr>
            <td><strong>${u.first_name} ${u.last_name}</strong></td>
            <td>${u.gender}</td>
            <td>${u.roles.join(', ')}</td>
        </tr>
    `).join('');
}

document.getElementById('refresh-btn').addEventListener('click',
loadUsers);
```

```
document.addEventListener('DOMContentLoaded', loadUsers);

document.getElementById('userForm').addEventListener('submit', async (e)
=> {
    e.preventDefault();

    // Capturamos los datos
    const userData = {
        first_name: document.getElementById('firstName').value,
        last_name: document.getElementById('lastName').value,
        gender: document.getElementById('gender').value,
        roles: [document.getElementById('role').value]
    };

    try {
        const response = await fetch('/api/v1/users', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(userData)
        });

        if (response.ok) {
            // Limpiamos el formulario y recargamos la tabla
            document.getElementById('userForm').reset();
            loadUsers();
        } else {
            alert("Error al guardar el usuario");
        }
    } catch (error) {
        console.error("Error:", error);
    }
});
```

Paso 2. Actualización de main.py

```
from typing import List
from uuid import UUID, uuid4
from fastapi import FastAPI, HTTPException, Request
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from models import Gender, Role, User, UpdateUser

app = FastAPI()
```

17 Django-marcador

```
# Montamos la carpeta de archivos estáticos
app.mount("/static", StaticFiles(directory="static"), name="static")

# Configuramos la carpeta de plantillas
templates = Jinja2Templates(directory="templates")

db: List[User] = [
    User(id=uuid4(), first_name="John", last_name="Doe",
gender=Gender.male, roles=[Role.user]),
    User(id=uuid4(), first_name="Jane", last_name="Doe",
gender=Gender.female, roles=[Role.user])
]

# --- RUTA PARA CARGAR EL FRONTEND ---
@app.get("/")
async def read_index(request: Request):
    # Esto busca en /templates/index.html y le pasa el request
obligatorio
    return templates.TemplateResponse("index.html", {"request":
request})

# --- RESTO DE LA API ---
@app.get("/api/v1/users")
async def get_users():
    return db

@app.post("/api/v1/users")
async def create_user(user: User):
    # Si el ID no viene, lo generamos
    if not user.id:
        user.id = uuid4()
    db.append(user)
    return user

@app.delete("/api/v1/users/{id}")
async def delete_user(id: UUID):
    for user in db:
        if user.id == id:
            db.remove(user)
            return {"detail": "User deleted"}
    raise HTTPException(status_code=404, detail="User not found")

@app.put("/api/v1/users/{id}")
async def update_user(user_update: UpdateUser, id: UUID):
```

17 Django-marcador

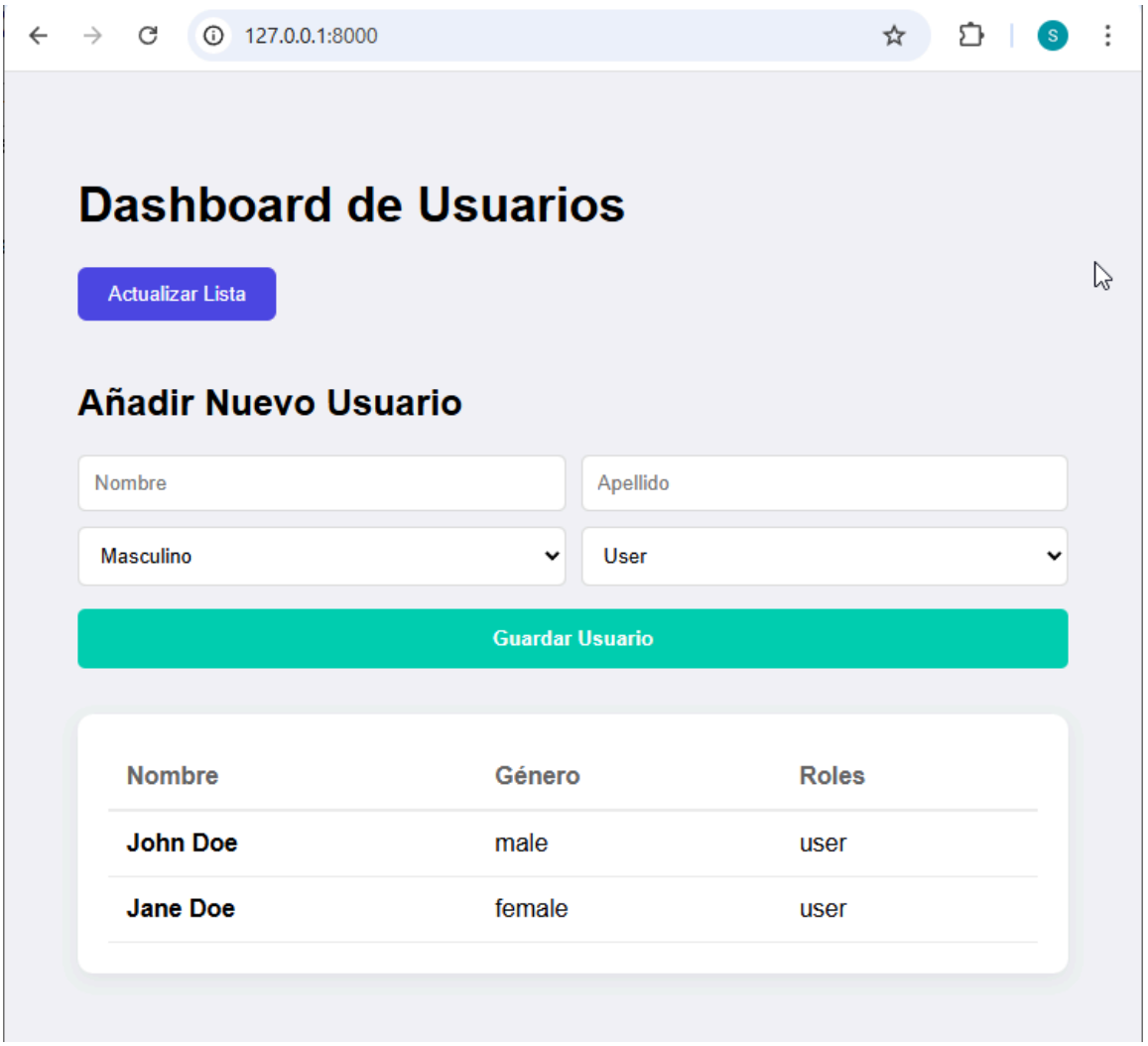
```
    for user in db:
        if user.id == id:
            if user_update.first_name: user.first_name =
user_update.first_name
            if user_update.last_name: user.last_name =
user_update.last_name
            if user_update.roles: user.roles = user_update.roles
            return user
    raise HTTPException(status_code=404, detail="User not found")
```

models.py

```
from typing import List, Optional
from uuid import UUID, uuid4
from pydantic import BaseModel
from enum import Enum

class Gender(str, Enum):
    male = "male"
    female = "female"

class Role(str, Enum):
    admin = "admin"
    user = "user"

class User(BaseModel):
    id: Optional[UUID] = uuid4()
    first_name: str
    last_name: str
    gender: Gender
    roles: List[Role]

class UpdateUser(BaseModel):
    first_name: Optional[str]
    last_name: Optional[str]
    roles: Optional[List[Role]]
```

Ejecuta uvicorn main:app --reload. Prueba a editar y borrar usuarios desde tu navegador.

17 Django-marcador



Conexión a una base de datos real (como SQLite)

Usaremos SQLite, que es perfecta para empezar porque no requiere instalación (es solo un archivo .db en tu carpeta). Para esto utilizaremos SQLAlchemy, el estándar de oro en Python para manejar bases de datos.

Paso 1. Instalación

Necesitas instalar la librería de base de datos:

```
pip install sqlalchemy
```

Paso 2. Configuración de la base de datos (database.py)

17 Django-marcador

Crea este nuevo archivo para gestionar la conexión:

```
#database.py
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# El archivo se creará automáticamente como 'usuarios.db'
SQLALCHEMY_DATABASE_URL = "sqlite:///./usuarios.db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)

Base = declarative_base()

# Dependencia para obtener la DB en cada ruta
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Paso 3. Modelos de Base de Datos (models.py)

Actualizaremos el archivo para tener tanto el modelo de la Base de Datos (SQLAlchemy) como los esquemas de Validación (Pydantic).

```
from sqlalchemy import Column, String, JSON
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from pydantic import BaseModel
from typing import List, Optional
from uuid import UUID, uuid4

# --- CONFIGURACIÓN SQLITE ---
SQLALCHEMY_DATABASE_URL = "sqlite:///./users.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL,
connect_args={"check_same_thread": False})
```

```
SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)
Base = declarative_base()

# --- MODELO DE BASE DE DATOS (SQLAlchemy) ---
class UserDB(Base):
    __tablename__ = "users"
    id = Column(String, primary_key=True, index=True)
    first_name = Column(String)
    last_name = Column(String)
    gender = Column(String)
    roles = Column(JSON) # SQLite soporta JSON para guardar la lista de
roles

# --- ESQUEMAS PARA LA API (Pydantic) ---
class UserBase(BaseModel):
    first_name: str
    last_name: str
    gender: str
    roles: List[str]

class UserCreate(UserBase):
    id: Optional[str] = None

class UserResponse(UserBase):
    id: str
    class Config:
        from_attributes = True
```

Paso 4. Integración en main.py

Ahora conectamos todo. Fíjate cómo el db ya no es una lista, sino una sesión de base de datos.

```
from fastapi import FastAPI, Depends, Request, HTTPException
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from sqlalchemy.orm import Session
import models
from models import SessionLocal, engine, UserDB, UserCreate

# Crear las tablas al iniciar
models.Base.metadata.create_all(bind=engine)
```

17 Django-marcador

```
app = FastAPI()

app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")

# Dependencia para obtener la DB en cada petición
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.get("/")
async def read_index(request: Request):
    return templates.TemplateResponse("index.html", {"request": request})

@app.get("/api/v1/users")
async def get_users(db: Session = Depends(get_db)):
    return db.query(UserDB).all()

@app.post("/api/v1/users")
async def create_user(user: UserCreate, db: Session = Depends(get_db)):
    db_user = UserDB(
        id=str(models.uuid4()),
        first_name=user.first_name,
        last_name=user.last_name,
        gender=user.gender,
        roles=user.roles
    )
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user


@app.delete("/api/v1/users/{user_id}")
async def delete_user(user_id: str, db: Session = Depends(get_db)):
    # Buscamos al usuario en la base de datos
    db_user = db.query(models.UserDB).filter(models.UserDB.id == user_id).first()

    if not db_user:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")
```

```
    db.delete(db_user)
    db.commit()
    return {"detail": "Usuario eliminado correctamente"}


@app.put("/api/v1/users/{user_id}")
async def update_user(user_id: str, user_update: models.UserBase, db: Session =
Depends(get_db)):
    # Buscamos al usuario existente
    db_user = db.query(models.UserDB).filter(models.UserDB.id == user_id).first()

    if not db_user:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")

    # Actualizamos los campos
    db_user.first_name = user_update.first_name
    db_user.last_name = user_update.last_name
    db_user.gender = user_update.gender
    db_user.roles = user_update.roles

    db.commit()
    db.refresh(db_user)
    return db_user
```

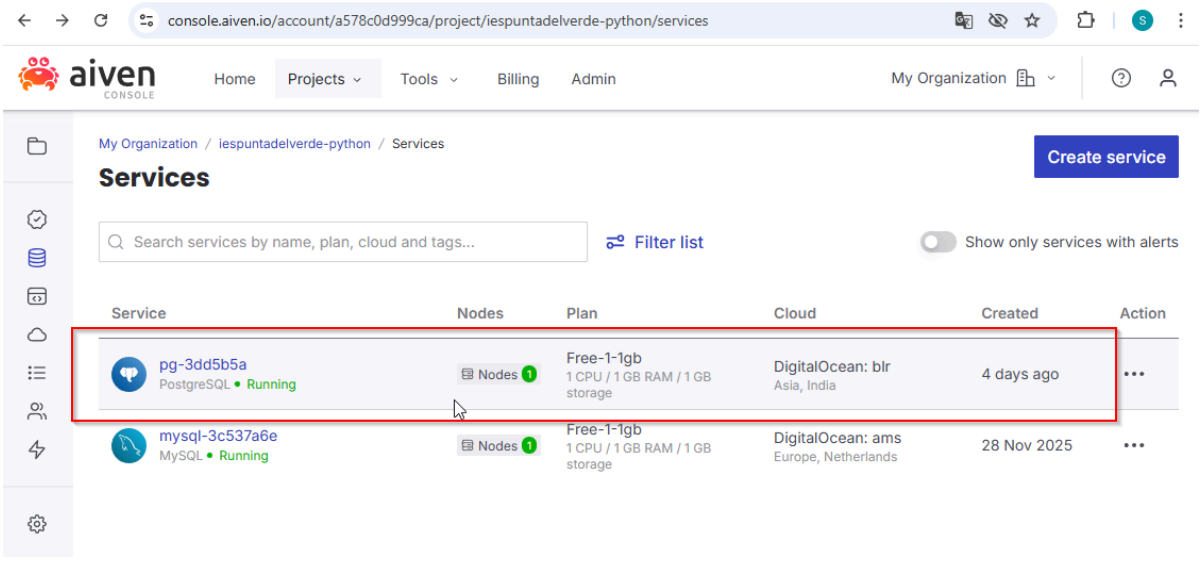
Despliegue de una API

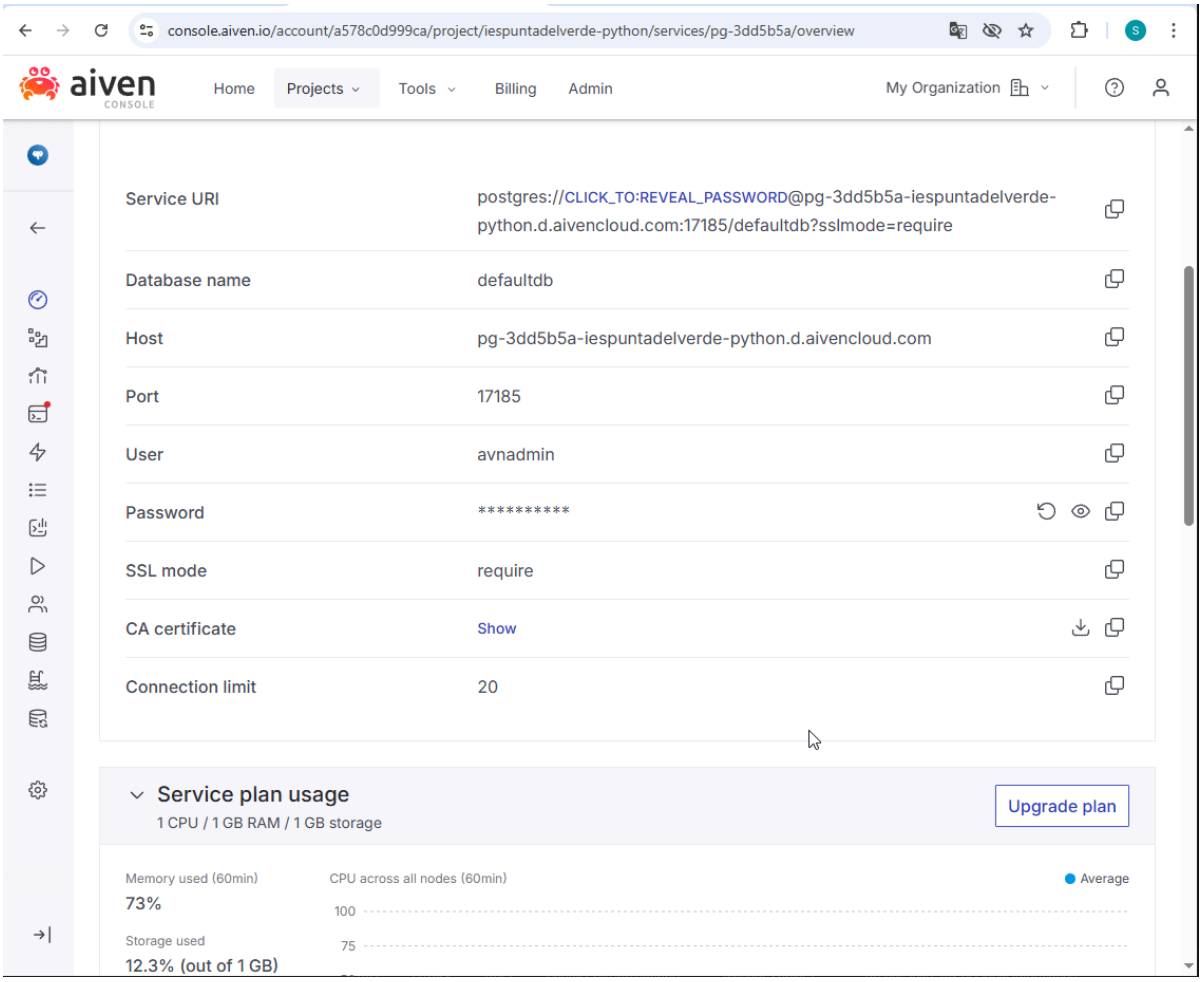
Vamos a desplegar una API que persiste en una BD PostgreSQL alojada en Aiven en un servidor web que provee Railway. El proyecto en Python FastAPI está en Github:

https://github.com/salvadormartos/FastAPI_Persistencia.git

La BD la alojamos en Aiven:

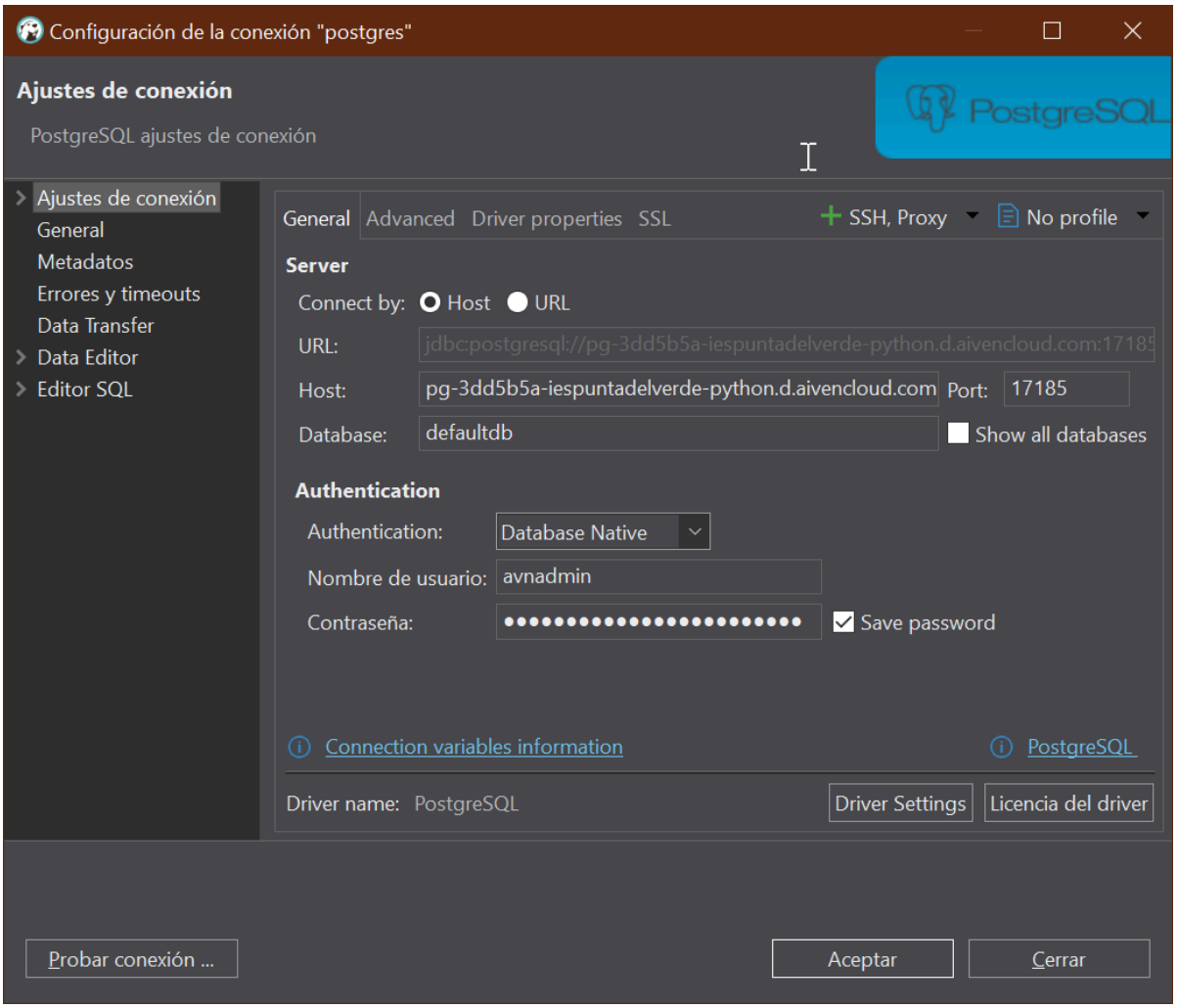
17 Django-marcador



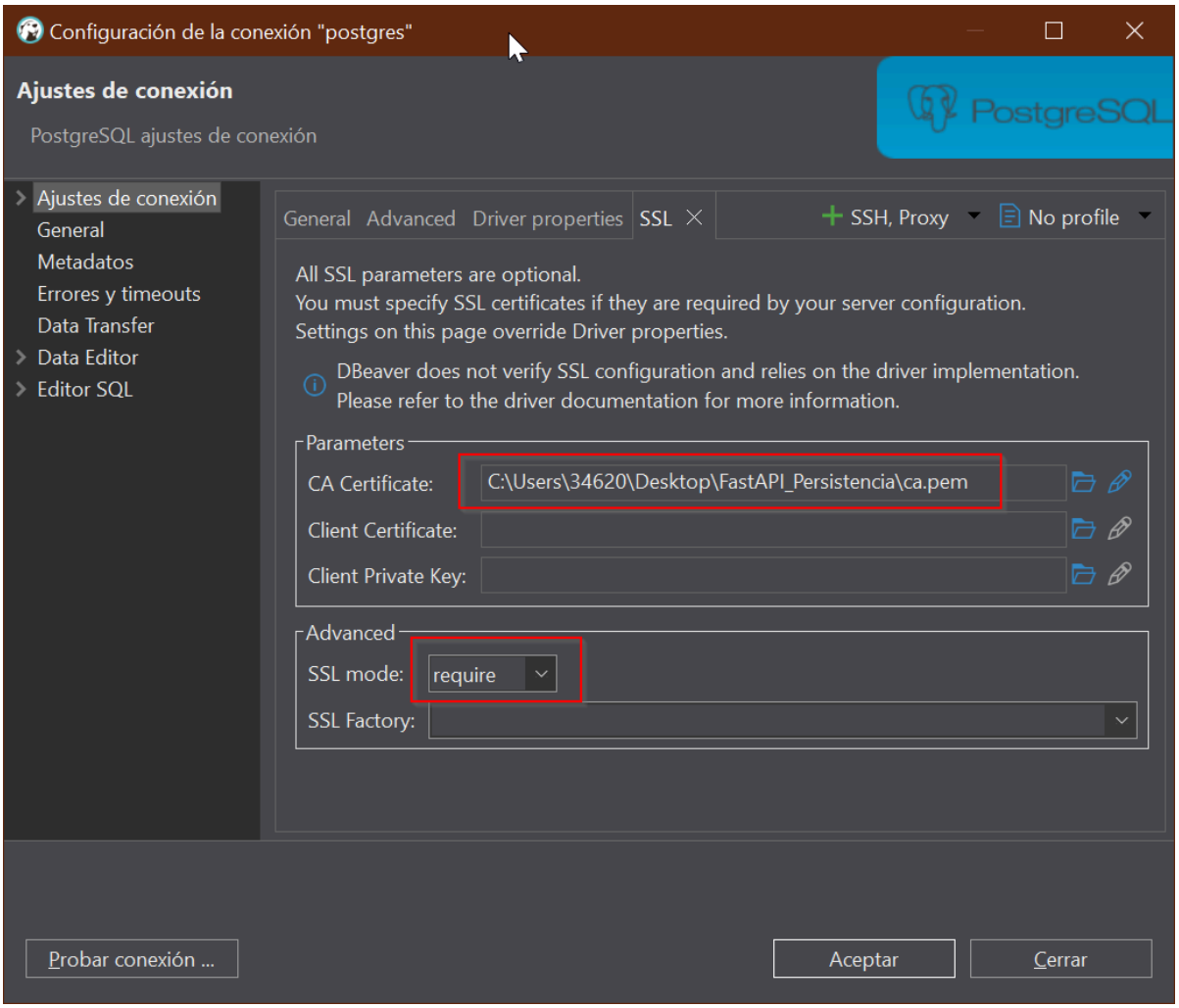


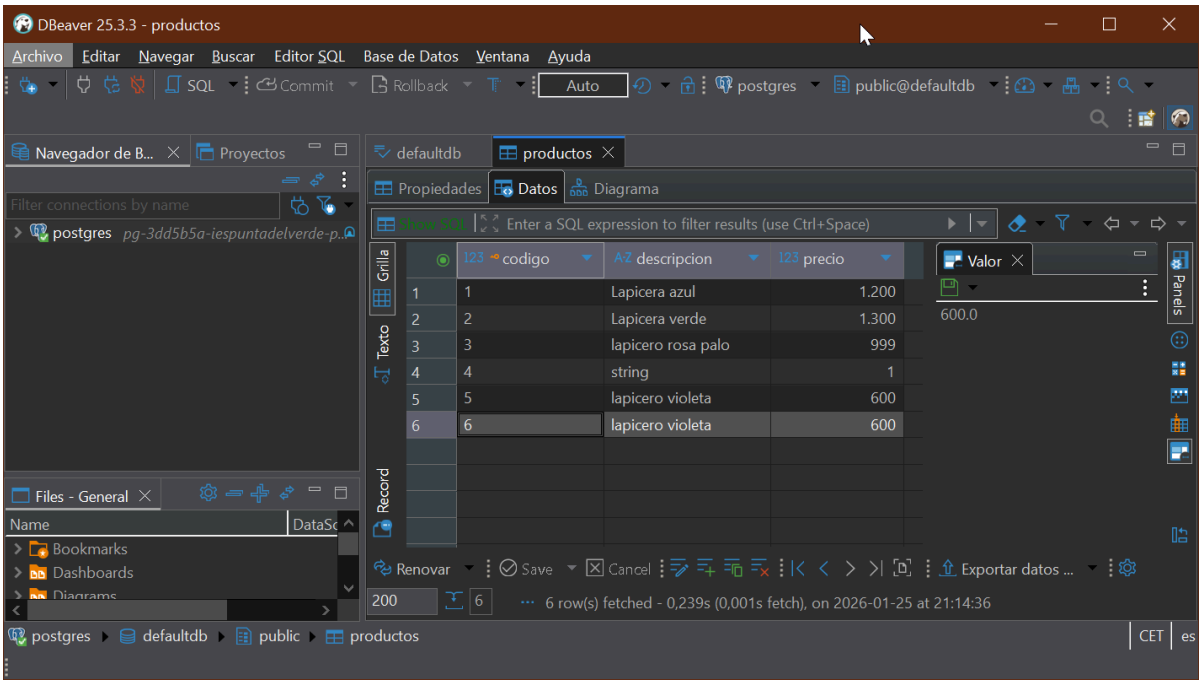
Con DBeaver podemos acceder a la tabla de la BD:

17 Django-marcador



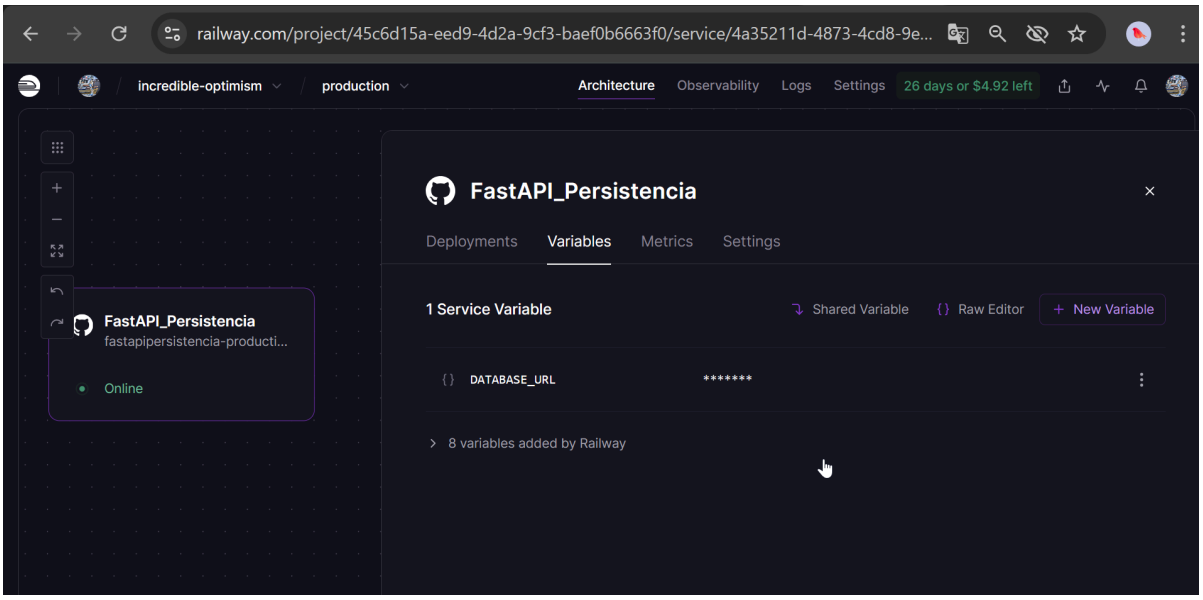
17 Django-marcador

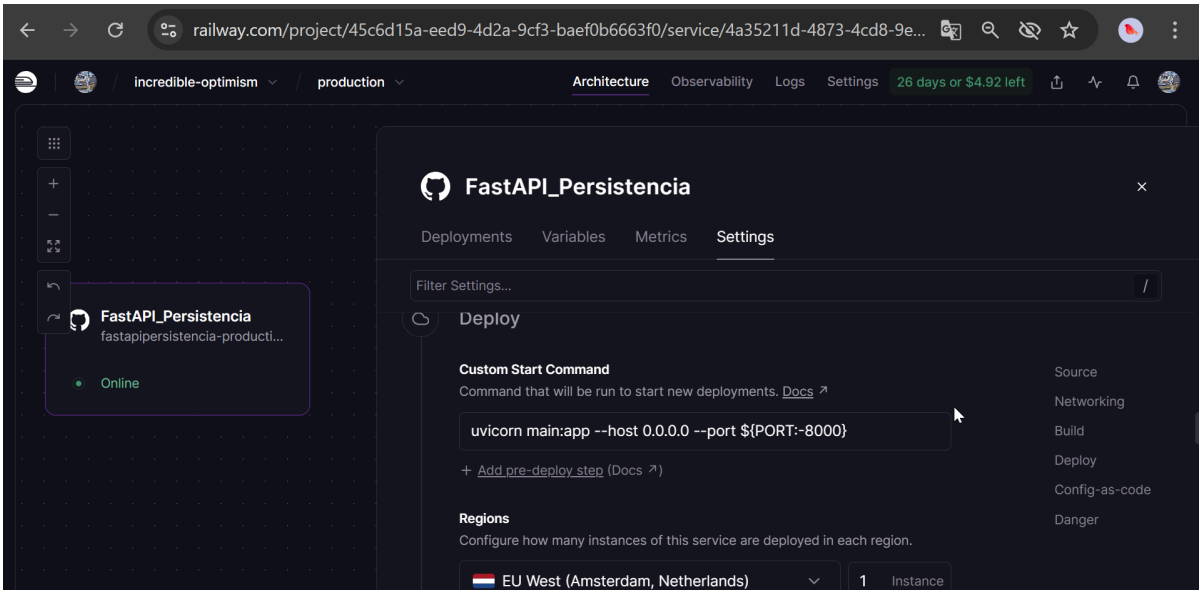




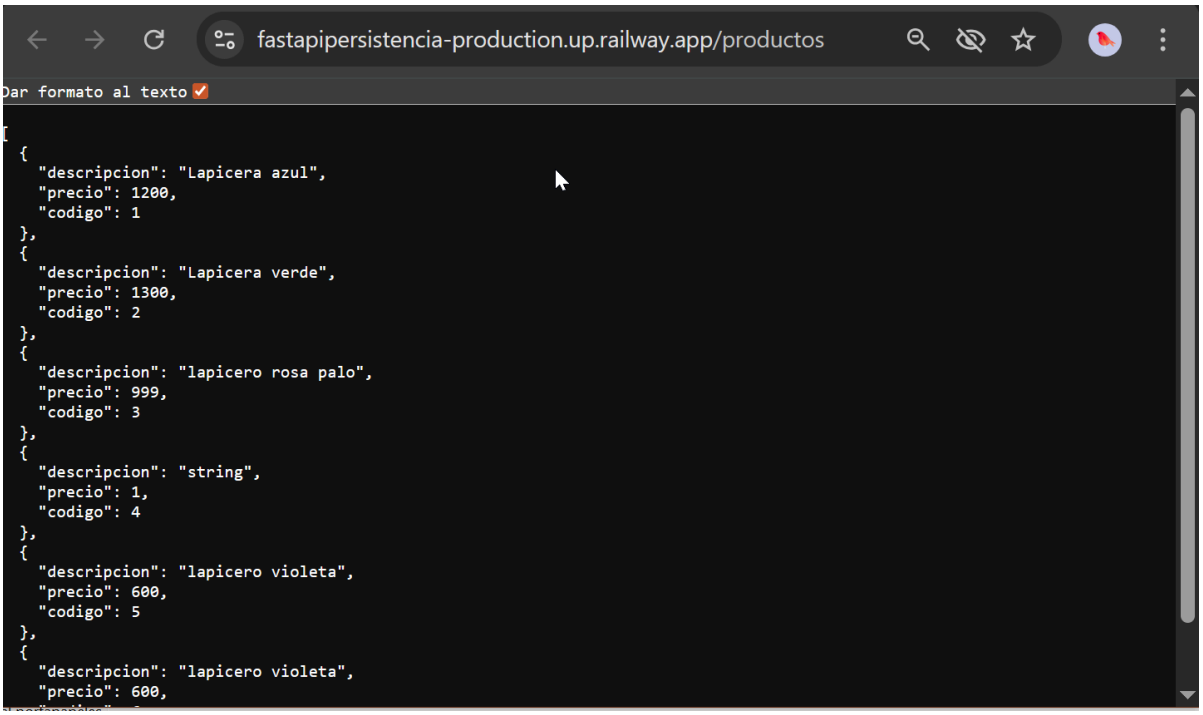
17 Django-marcador

En Railway hacemos el despliegue:





17 Django-marcador



https://fastapipersistencia-production.up.railway.app/productos