

## 1. Prototipo de Base de Datos

**Normas de diseño y seguridad aplicadas:** integridad referencial, separación de roles, cifrado de datos sensibles, autenticación robusta.

### Tablas principales

#### 1. usuarios

- id (INT, PK, identity)
- username (NVARCHAR(50), UNIQUE, NOT NULL)
- password\_hash (NVARCHAR(256), NOT NULL) — almacenar hash seguro (bcrypt/Argon2)
- email (NVARCHAR(100), UNIQUE, NOT NULL)
- role\_id (INT, FK → roles.id)
- created\_at, updated\_at (DATETIME2)
- last\_login (DATETIME2, Nullable)

#### 2. roles

- id (INT, PK)
- name (NVARCHAR(50), UNIQUE, NOT NULL)

#### 3. tokens\_revoked

- id (INT, PK)
- jti (UNIQUE IDENTIFIER of JWT)
- revoked\_at (DATETIME2)

#### 4. logs

- id (INT, PK)
- user\_id (INT, FK → usuarios.id, Nullable)
- action (NVARCHAR(100))
- ip\_address (NVARCHAR(45))
- created\_at (DATETIME2)

#### 5. ejemplos (una tabla de ejemplo para tus módulos)

- id (INT, PK)
- usuario\_id (INT, FK → usuarios.id)

- data (NVARCHAR(500))
- created\_at, updated\_at

### Seguridad relacionada a DB:

- Usar cuentas específicas con mínimos privilegios (ej. solo EXECUTE en procedimientos).
- Parámetros siempre con sentencias preparadas (ORM o SQL parametrizado).
- Cifrado en reposo y en tránsito (TLS + Transparent Data Encryption si es posible).

## 2. 📁 Estructura del Proyecto

Organización clara para mantener la seguridad y escalabilidad:

bash

CopiarEditar

mi\_app/

```

├── app/
|   ├── __init__.py    # crea la app, config, registro de blueprints
|   ├── config.py      # config segura: usar variables de entorno
|   ├── extensions.py  # db (SQLAlchemy), login, migrate, jwt
|   ├── models/        # modelos ORM
|   |   ├── __init__.py
|   |   ├── user.py
|   |   └── ejemplo.py
|   ├── routes/        # endpoints organizados por módulo
|   |   ├── auth.py
|   |   └── ejemplo.py
|   ├── services/      # lógica del negocio
|   ├── schemas/       # validación/serialización (Marshmallow/Pydantic)
|   ├── utils/         # utils generales (e.g., hashing, logs)
|   └── static/        # assets tailwind (CSS, JS)
|       └── tailwind.css

```

```
| └─ templates/      # plantillas Jinja2
|   └─ layout.html
|   └─ pages/
└─ migrations/      # alembic o flask-migrate
└─ tests/           # unit + integración
└─ .env             # variables sensibles excluidas de git
└─ requirements.txt
└─ run.py           # punto de entrada
```

---

### 3. Plan de seguridad en cada módulo

#### Configuración (app/config.py)

- Usa variables de entorno para secretos (JWT\_SECRET\_KEY, DB creds).
- Activa modos seguros: SESSION\_COOKIE\_SECURE=True, REMEMBER\_COOKIE\_HTTPONLY=True, JWT\_TOKEN\_LOCATION=["headers"], JWT\_COOKIE\_SECURE=True, JWT\_COOKIE\_SAMESITE='Lax'.

#### Extensiones (app/extensions.py)

- Configurar SQLAlchemy con uri con usuario limitado.
- JWT con rotación de tokens y lista de revocación (tokens\_revoked).
- Flask-Migrate para migraciones seguras.
- Flask-Limiter para limitar peticiones y prevenir brute force.

#### Modelos (models/)

- Usuario: incluye métodos para hashing (bcrypt) y verificación segura.
- Rol básico de roles con verificación de permisos ("admin", "user").

#### Rutas de autenticación (routes/auth.py)

- **Registro:** validación fuerte, letras/caracteres, hash de contraseña con bcrypt.
- **Login:** validación, registro de last\_login, generación de JWT (access + refresh).
- **Logout:** revocar JWT (guardar jti en tokens\_revoked).
- **Refresh token:** autenticación segura y verificación de revocación.

#### Rutas protegidas (ej. routes/ejemplo.py)

- Decorar con `@jwt_required()` y chequear roles si es necesario.
- Validar JSON con schemas/.
- Registrar acciones críticas con IP y usuario.

#### **Validación / Serialización (schemas/)**

- Usar Marshmallow o Pydantic, chequear tipos, longitudes, patrones (emails, passwords).
- Desinfectar inputs si se va a renderizar (XSS).

#### **Lógica / Servicios (services/)**

- Separación lógica de acceso a DB, negocio, control de errores.
- Incluir chequeo de roles/permiso antes de actualizaciones o lecturas sensibles.

#### **Logs (models/logs.py, utils/logger.py)**

- Guardar logs: login, intentos fallidos, operaciones CRUD.
- Distinguir niveles: INFO, WARNING, ERROR; incluir IP, usuario, timestamp.
- En puntos críticos (login, logout, token refresh, accesos a datos SENSITIVE).

#### **Pruebas (tests/)**

- Unitarios: hashing, servicios, validación.
- Integración: endpoints auth y protegido, testing de JWT, revocación, roles.
- Simular ataques: CSRF, XSS, inyección SQL, brute force.

#### **Frontend (Tailwind + Templates)**

- Usar templates seguras, `{{ }}` en Jinja2 escapa por defecto.
- Incluir meta-tags como Content-Security-Policy, X-Frame-Options, X-XSS-Protection.
- Usar `tailwind.config.js` para compilar un CSS optimizado y revisar vulnerabilidades.

#### **Deploy / Infraestructura**

- Usa WSGI server (gunicorn/uWSGI) detrás de reverse proxy (Nginx):
    - Habilita HTTPS con certbot.
    - Configura HSTS, frame-ancestors 'self'.
    - No exponer el puerto DB a Internet.
  - Backups automáticos + cifrado.
  - Análisis de vulnerabilidades periódicos (dependencias, escaneo de código).
-

## ✓ Resumen de flujo

1. Configuración segura desde el comienzo.
2. Diseño de base de datos con integridad y cifrado.
3. Rutas organizadas con validación y control de acceso.
4. JWT con revocación + restricción por roles.
5. Registro detallado de logs.
6. Tests completos para prevenir regresiones y vulnerabilidades.
7. Frontend seguro con políticas HTTP.
8. Deploy profesional con HTTPS, hardening y backups.

Con esta guía tienes el **esqueleto endpoint-módulo**, el modelo de datos, las carpetas bien ordenadas y todas las **medidas de seguridad** esenciales: hashing, validación, roles, logs, políticas HTTP, tokens y centralización de configuración crítica.

¿Te gustaría que prepare un ejemplo real de auth.py y su esquema Marshmallow/JWT para que lo adaptes directamente?

**Dijiste:**

si porfavor

**ChatGPT dijo:**

Perfecto. A continuación te muestro un módulo completo de autenticación (auth.py) usando Flask, JWT, Marshmallow para validación, y todas las buenas prácticas de seguridad implementadas:

---

### ✓ 1. schemas/user\_schema.py — Validación con Marshmallow

python

CopiarEditar

```
from marshmallow import Schema, fields, validate
```

```
class RegisterSchema(Schema):
```

```
    username = fields.Str(required=True, validate=validate.Length(min=4, max=50))
```

```
    email = fields.Email(required=True)
```

```
password = fields.Str(required=True, load_only=True, validate=validate.Length(min=8))
```

```
class LoginSchema(Schema):
```

```
    username = fields.Str(required=True)
```

```
    password = fields.Str(required=True, load_only=True)
```

---

## ✅ 2. models/user.py — Modelo con hash seguro

python

CopiarEditar

```
from app.extensions import db
```

```
from werkzeug.security import generate_password_hash, check_password_hash
```

```
from datetime import datetime
```

```
class Usuario(db.Model):
```

```
    __tablename__ = 'usuarios'
```

```
    id = db.Column(db.Integer, primary_key=True)
```

```
    username = db.Column(db.String(50), unique=True, nullable=False)
```

```
    email = db.Column(db.String(100), unique=True, nullable=False)
```

```
    password_hash = db.Column(db.String(256), nullable=False)
```

```
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

```
    last_login = db.Column(db.DateTime)
```

```
    def set_password(self, password):
```

```
        self.password_hash = generate_password_hash(password, method='pbkdf2:sha256')
```

```
    def check_password(self, password):
```

```
        return check_password_hash(self.password_hash, password)
```

---

### ✅ 3. routes/auth.py — Autenticación JWT protegida

python

CopiarEditar

```
from flask import Blueprint, request, jsonify
from app.models.user import Usuario
from app.extensions import db, jwt
from flask_jwt_extended import (
    create_access_token, create_refresh_token,
    jwt_required, get_jwt_identity, get_jwt
)
from app.schemas.user_schema import RegisterSchema, LoginSchema
from datetime import datetime, timedelta
from app.models.token_revoked import TokenRevoked # si usas revocación de JWT
from marshmallow import ValidationError

auth_bp = Blueprint('auth', __name__, url_prefix="/auth")

@auth_bp.route("/register", methods=["POST"])
def register():
    try:
        data = RegisterSchema().load(request.json)
    except ValidationError as err:
        return jsonify(err.messages), 400

    if Usuario.query.filter((Usuario.username == data["username"]) | (Usuario.email == data["email"])).first():
        return jsonify({"error": "Usuario o email ya existen"}), 409

    user = Usuario(username=data["username"], email=data["email"])
```

```
user.set_password(data["password"])

db.session.add(user)

db.session.commit()

return jsonify({"msg": "Usuario creado correctamente"}), 201
```

```
@auth_bp.route("/login", methods=["POST"])
```

```
def login():
```

```
    try:
```

```
        data = LoginSchema().load(request.json)
```

```
    except ValidationError as err:
```

```
        return jsonify(err.messages), 400
```

```
    user = Usuario.query.filter_by(username=data["username"]).first()
```

```
    if not user or not user.check_password(data["password"]):
```

```
        return jsonify({"error": "Credenciales inválidas"}), 401
```

```
    user.last_login = datetime.utcnow()
```

```
    db.session.commit()
```

```
    access_token = create_access_token(identity=user.id, expires_delta=timedelta(minutes=15))
```

```
    refresh_token = create_refresh_token(identity=user.id)
```

```
    return jsonify({
```

```
        "access_token": access_token,
```

```
        "refresh_token": refresh_token,
```

```
        "msg": "Inicio de sesión exitoso"
```

```
    })
```

```
@auth_bp.route("/refresh", methods=["POST"])
```

```
@jwt_required(refresh=True)
```



```

def refresh_token():
    current_user = get_jwt_identity()

    new_token = create_access_token(identity=current_user,
    expires_delta=timedelta(minutes=15))

    return jsonify({"access_token": new_token})

@auth_bp.route("/logout", methods=["POST"])
@jwt_required()
def logout():
    jti = get_jwt()["jti"]
    revoked = TokenRevoked(jti=jti)
    db.session.add(revoked)
    db.session.commit()

    return jsonify({"msg": "Sesión cerrada correctamente"})

```

---

#### ✅ 4. models/token\_revoked.py — JWT revocado

python

CopiarEditar

```

from app.extensions import db
from datetime import datetime

```

```

class TokenRevoked(db.Model):
    __tablename__ = 'tokens_revoked'

    id = db.Column(db.Integer, primary_key=True)
    jti = db.Column(db.String(36), unique=True, nullable=False)
    revoked_at = db.Column(db.DateTime, default=datetime.utcnow)

```

---

#### ✅ 5. extensions.py — Configurar JWT con revocación

python

CopiarEditar

```
from flask_sqlalchemy import SQLAlchemy
```

```
from flask_jwt_extended import JWTManager
```

```
db = SQLAlchemy()
```

```
jwt = JWTManager()
```

```
from app.models.token_revoked import TokenRevoked
```

```
@jwt.token_in_blocklist_loader
```

```
def check_if_token_revoked(jwt_header, jwt_payload):
```

```
    jti = jwt_payload["jti"]
```

```
    token = TokenRevoked.query.filter_by(jti=jti).first()
```

```
    return token is not None
```

---

## ✅ 6. .env (usado en config.py)

env

CopiarEditar

```
FLASK_ENV=development
```

```
JWT_SECRET_KEY=supersecretkey123456
```

```
SQLALCHEMY_DATABASE_URI=mssql+pyodbc://usuario:clave@servidor/nombreBD?driver=ODBC  
+Driver+17+for+SQL+Server
```

---

## ✅ 7. config.py

python

CopiarEditar

```
import os
```

class Config:

```
SECRET_KEY = os.environ.get("JWT_SECRET_KEY")
```

```
SQLALCHEMY_DATABASE_URI = os.environ.get("SQLALCHEMY_DATABASE_URI")
```

```
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

```
JWT_SECRET_KEY = os.environ.get("JWT_SECRET_KEY")
```

```
JWT_ACCESS_TOKEN_EXPIRES = 900 # 15 minutos
```

```
JWT_REFRESH_TOKEN_EXPIRES = 2592000 # 30 días
```

---

¿Te gustaría que prepare también un frontend de login con Tailwind y fetch al endpoint /auth/login o prefieres avanzar con los módulos privados?