## Ce qu’on cherche à construire

Une application web qui répond en JSON à des requêtes HTTP.
Les données vivent dans une base SQL et sont lues/écrites de manière fiable.

On veut comprendre le “pourquoi” de chaque brique, pas seulement exécuter des commandes.

## API HTTP : le contrat

Une API, c’est une interface entre un client et un serveur.
Le client envoie une intention via une URL, une méthode, et parfois un corps JSON.

Le serveur répond avec un statut HTTP et un JSON qui représente l’état métier.

## Ce que Flask apporte

Flask reçoit la requête, la route vers une fonction, et construit la réponse.
Le cœur, c’est le mapping “URL → code Python”.

Le reste, ce sont des extensions et des choix d’architecture.

## Où se place SQL

SQL stocke l’état durable.
L’API manipule des ressources, la base garantit la persistance.

La difficulté n’est pas “écrire des données”, c’est “écrire des données sans casser la cohérence”.

## Concepts SQL à garder en tête

Une table décrit une forme de données.
Une clé primaire identifie une ligne de façon stable.

Les contraintes et index sont des garde-fous et des accélérateurs.
Elles évitent de “réparer” au niveau Python ce que la base peut garantir.

# Exemple de schéma minimal

Ici, la base impose l'unicité d'un email et relie les tâches à un utilisateur.

```sql
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  password_hash TEXT NOT NULL
);

CREATE TABLE todos (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL REFERENCES users(id),
  title TEXT NOT NULL,
  done BOOLEAN NOT NULL DEFAULT FALSE
);
```

## Ce qui se passe lors d’un INSERT

La base valide les contraintes avant d’écrire.
Si l’email existe déjà, c’est la base qui refuse.

Résultat important : l’erreur devient un signal métier, pas un bug “mystère”.

## Pourquoi un ORM (SQLAlchemy)

L’ORM transforme des lignes en objets Python et inversement.
Il garde une “session” qui suit ce qu’on a chargé, modifié, puis écrit.

Point clé : ce n’est pas magique.
C’est une traduction contrôlée entre deux mondes.

## Transaction : la règle d’or

Une transaction regroupe plusieurs opérations en une seule unité logique.
Soit tout passe, soit rien ne passe.

C’est la meilleure façon d’éviter les états “à moitié écrits” quand une erreur arrive.

# Migrations : versionner le schéma

Le code évolue, donc le schéma doit évoluer aussi.
Une migration décrit un changement de structure de manière rejouable.

But pédagogique : traiter le schéma comme du code, avec un historique clair.

# Configuration : pourquoi un .env

Le code ne doit pas contenir des secrets ou des URLs de base.

On sépare “ce que fait l’app” de “où et comment elle tourne”.

Le `.env` sert au développement local.

En production, on vise de vraies variables d’environnement.

```
DATABASE_URL=postgresql+psycopg2://user:pass@localhost:5432/demo_api
JWT_SECRET=change-me
JWT_EXPIRES_MIN=15
```

## Ce qui se passe quand on charge .env

Un loader lit le fichier et injecte les variables dans l’environnement du processus.
Le code ne “lit pas un fichier”, il lit des variables.

Ça rend l’application portable entre Windows, Linux, macOS et serveurs.

```
import os
from dotenv import load_dotenv

load_dotenv()
DATABASE_URL = os.environ["DATABASE_URL"]
```

# Brancher Flask à la base

On crée un objet “connexion/ORM” qui sera initialisé avec l’app Flask.
Flask fournit le contexte, SQLAlchemy gère le pool de connexions et les transactions.

Idée à retenir : l’app ne se connecte pas “à chaque ligne de code”, elle utilise un moteur partagé.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
def create_app():
    app = Flask(__name__)
    app.config["SQLALCHEMY_DATABASE_URI"] = DATABASE_URL
    db.init_app(app)
    return app
```

# Modèles : la forme des données côté Python

Un modèle décrit comment un objet se mappe vers une table.
L'intérêt, c'est la cohérence : même définition, partout.

On évite de construire des dictionnaires "à la main" dans tous les sens.

```
class Todo(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, nullable=False)
    title = db.Column(db.String(200), nullable=False)
    done = db.Column(db.Boolean, default=False, nullable=False)
```

## Pourquoi parler de migrations

Au début, on crée des tables. Ensuite, le besoin change.
Le danger, c’est de modifier la base “à la main” et perdre la cohérence entre machines.

Une migration, c’est un historique de changements du schéma.
On traite le schéma comme du code versionné.

## Le rôle d’Alembic

Alembic est le moteur de migrations de SQLAlchemy.
Il génère des scripts qui décrivent “comment passer de V1 à V2”.

Point clé : ce sont des fichiers Python que tu peux relire.
Ce n’est pas une boîte noire.

## Ce qui se passe lors d’une migration

On compare l’état attendu (modèles) et l’état actuel (base).
Un script “upgrade” décrit les changements à appliquer.

Quand on exécute, la base applique une suite d’opérations DDL.
Un journal interne garde la trace de la dernière version appliquée.

## La table de suivi des versions

Alembic crée une table de versionnement.
Elle stocke l’identifiant de la dernière migration appliquée.

Résultat : l’outil sait quoi exécuter et dans quel ordre.

## Comment activer les migrations dans un projet Flask

Dans Flask, on branche Alembic via Flask-Migrate.
Ça expose Alembic en ligne de commande de manière intégrée à l’app.

Le but n’est pas “un outil de plus”,
c’est de garantir que dev, test, prod ont le même schéma.

## Initialiser le système de migrations

Ici, on crée l’infrastructure de migrations dans le projet.
On ne touche pas encore au schéma, on installe le mécanisme.

```
flask db init
```

Première version du schéma

On part d’un modèle simple.
La migration va matérialiser ce modèle en tables SQL.

On passe de “définition Python” à “structure en base”.

```
flask db migrate -m "initial schema"
flask db upgrade
```

# Étape modèle

On ajoute le champ côté Python.

Ce changement décrit l'intention, pas l'exécution.

```
from datetime import datetime

class Todo(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, nullable=False)
    title = db.Column(db.String(200), nullable=False)
    done = db.Column(db.Boolean, default=False, nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow, nullable=False)
```

## Générer la migration d’évolution

L’outil détecte le delta entre modèles et schéma actuel.
Il produit un script “add created_at”.

```
flask db migrate -m "add created_at to todos"
```

## Appliquer la migration

À ce moment, la base est réellement modifiée.
La structure devient celle attendue par le code.

```
flask db upgrade
```

# API : routes, JSON, statuts

Une route est une promesse : “si tu m’appelles comme ça, je réponds comme ça”. Les statuts HTTP portent du sens, pas seulement “ça marche”.

Le JSON est un format d’échange, pas forcément le modèle interne exact.

```
from flask import request, jsonify

@app.get("/api/todos/<int:todo_id>")
def get_todo(todo_id):
    todo = Todo.query.get_or_404(todo_id)
    return jsonify({"id": todo.id, "title": todo.title, "done": todo.done}), 200
```

## Validation et sérialisation

Le client peut envoyer n’importe quoi.
Valider, c’est refuser tôt et clairement.

Sérialiser, c’est choisir ce qu’on expose.
On ne renvoie jamais des champs “internes” juste parce qu’ils existent en base.

## Token : ce que ça représente

Un token est une preuve temporaire.
Le serveur signe un message, le client le transporte.

Le serveur vérifie la signature et la date d’expiration avant d’exécuter l’action.

# JWT : la mécanique en bref

Un JWT contient des “claims” et une signature.

Le serveur ne fait confiance qu’à ce qu’il peut vérifier cryptographiquement.

Important : un JWT n’est pas chiffré par défaut.

On ne met pas de données sensibles dedans.

```
import jwt, datetime

def make_token(user_id, secret, minutes):
    exp = datetime.datetime.utcnow() + datetime.timedelta(minutes=minutes)
    payload = {"sub": str(user_id), "exp": exp}
    return jwt.encode(payload, secret, algorithm="HS256")
```

# Vérifier un token côté Flask

Vérifier, c'est décoder, contrôler l'expiration, puis attacher l'identité au contexte. Si ça échoue, on coupe court avec 401.

```
from functools import wraps
from flask import request, jsonify, g

def require_auth(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        auth = request.headers.get("Authorization", "")
        token = auth.removeprefix("Bearer ").strip()
        try:
            payload = jwt.decode(token, os.environ["JWT_SECRET"], algorithms=["HS256"])
        except Exception:
            return jsonify({"error": "unauthorized"}), 401
        g.user_id = int(payload["sub"])
        return fn(*args, **kwargs)
    return wrapper
```

## Au-delà du token

HTTPS protège le transport.
Le hashing protège les mots de passe en base.

Limiter le débit et journaliser les accès réduisent l’impact des abus.
CORS se règle selon qui a le droit d’appeler l’API depuis un navigateur.

# Tester l’API sur Linux/macOS

`curl` est partout.

Le point sensible, c’est le JSON et les headers.

```
curl -X POST http://127.0.0.1:5000/api/login \
  -H "Content-Type: application/json" \
  -d '{"email":"a@b.com","password":"secret"}'

curl http://127.0.0.1:5000/api/todos \
  -H "Authorization: Bearer <TOKEN>"
```

# Tester l’API sur Windows (PowerShell)

PowerShell gère bien le JSON via des objets.

On évite beaucoup de problèmes d’échappement.

```
$body = @{ email="a@b.com"; password="secret" } | ConvertTo-Json
$r = Invoke-RestMethod "http://127.0.0.1:5000/api/login" -Method Post -ContentType "application/json" -Body $body
$token = $r.token
Invoke-RestMethod "http://127.0.0.1:5000/api/todos" -Headers @{ Authorization = "Bearer $token" }
```

## Tester l’API sur Windows (cmd.exe + curl)

Ici, l’effort principal est l’échappement des guillemets.

C’est le même HTTP, juste une autre syntaxe shell.

```
curl -X POST http://127.0.0.1:5000/api/login ^
  -H "Content-Type: application/json" ^
  -d "{\"email\":\"a@b.com\",\"password\":\"secret\"}"
```

## A) Swagger UI avec `flask-swagger-ui` (simple, clair)

### 1) Installation

```
pip install flask-swagger-ui
```

## 2) Ajouter un fichier OpenAPI

Crée un fichier openapi.json (dans un dossier `static/` par exemple) :

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Mon API Flask",
    "version": "1.0.0",
    "description": "Documentation de l'API"
  },
  "paths": {
    "/health": {
      "get": {
        "summary": "Vérifie que l'API fonctionne",
        "responses": {
          "200": {
            "description": "OK"
          }
    ...
}
```

# 3) Brancher Swagger UI dans Flask

```
from flask import Flask, jsonify
from flask_swagger_ui import get_swaggerui_blueprint
app = Flask(__name__)
@app.get("/health")
def health():
    return jsonify({"status": "ok"})
SWAGGER_URL = "/docs"              # URL de la page Swagger UI
API_URL = "/static/openapi.json"  # URL du fichier OpenAPI

swagger_bp = get_swaggerui_blueprint(
    SWAGGER_URL,
    API_URL,
    config={"app_name": "Mon API Flask"}
)
app.register_blueprint(swagger_bp, url_prefix=SWAGGER_URL)
if __name__ == "__main__":
    app.run(debug=True)
```