howest
hogeschool

# Backend in Python

howest
hogeschool

# FastAPI

#Python #WebAPI #ASGI

# Building Webservices in Python

- We can build webservices ourselves to create APIs to interact with databases
- Flask and FastAPI are two popular choices
- Flask
  - Micro web framework to deploy web applications with minimal amount of code
  - WSGI – Web Server Gateway Interface
  - Create a worker for each request
- **FastAPI**
  - Framework which supports concurrency and asynchronous code
  - ASGI – Asynchronous Server Gateway Interface

- More on ASGI and WSGI in the next chapter

# A short introduction

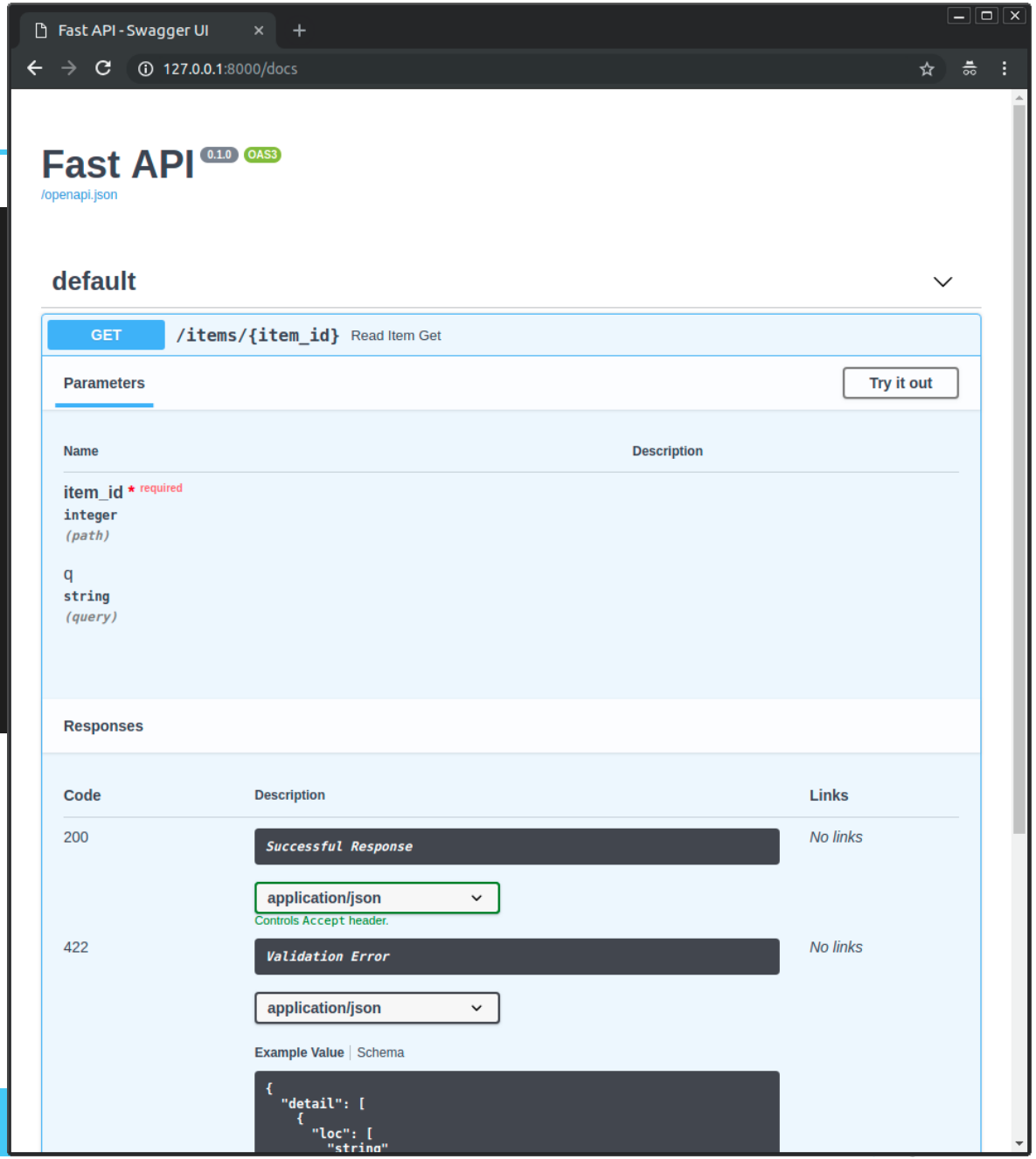
- Built-in **Swagger** docs
- **Middleware** options
- Great integrations with Database scheme's through **Pydantic** and **SQLAlchemy**
- **ASGI-compatible** (more on that later)
- Easy to deploy in **Docker** and **Kubernetes**
- Options with **Websockets, GraphQL**
- Many more features!

# A short introduction

```python
from typing import Optional

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

# A short introduction

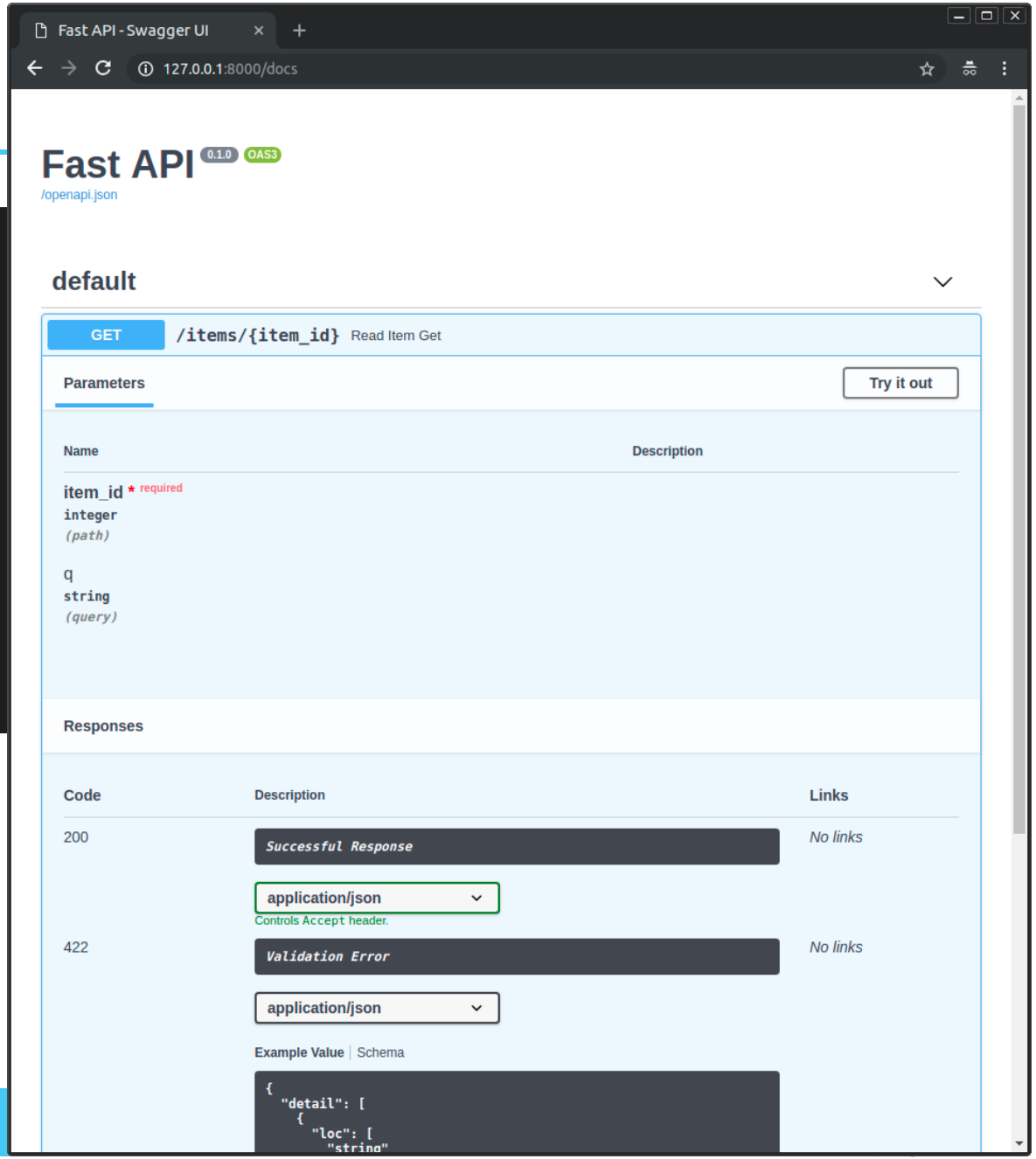
```python
from typing import Optional

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```



# A short introduction

```python
from typing import Optional

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

What's this?

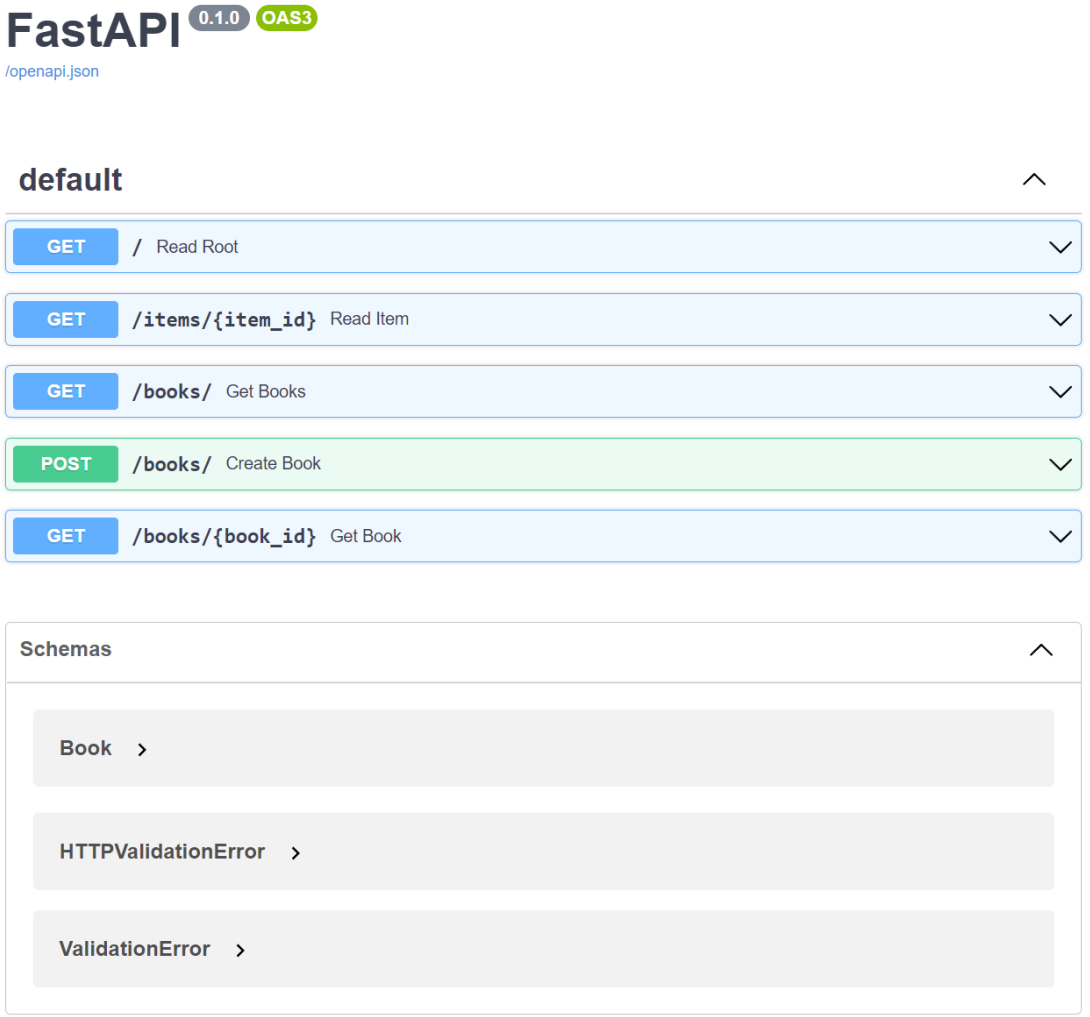


# A short introduction -- Pydantic

```python
from pydantic import BaseModel
from typing import List

# Without Pydantic
class Book():
    def __init__(self, title: str, authors: List[str], year: int, id = None):
        self.id = id
        self.title = title
        self.authors = authors
        self.year = year

# With Pydantic
class Book(BaseModel):
    id: Optional[int]
    title: str
    authors: List[str]
    year: int
```

# A short introduction -- Pydantic

```python
# With Pydantic
class Book(BaseModel):
    id: Optional[int]
    title: str
    authors: List[str]
    year: int


books = []

@app.post("/books/")
def create_book(book: Book):
    book.id = len(books) + 1
    books.append(book)
    return book

@app.get("/books/")
def get_books():
    return books

@app.get("/books/{book_id}")
def get_book(book_id: int):
    return list(filter(lambda b: b.id == book_id, books))[0]
```

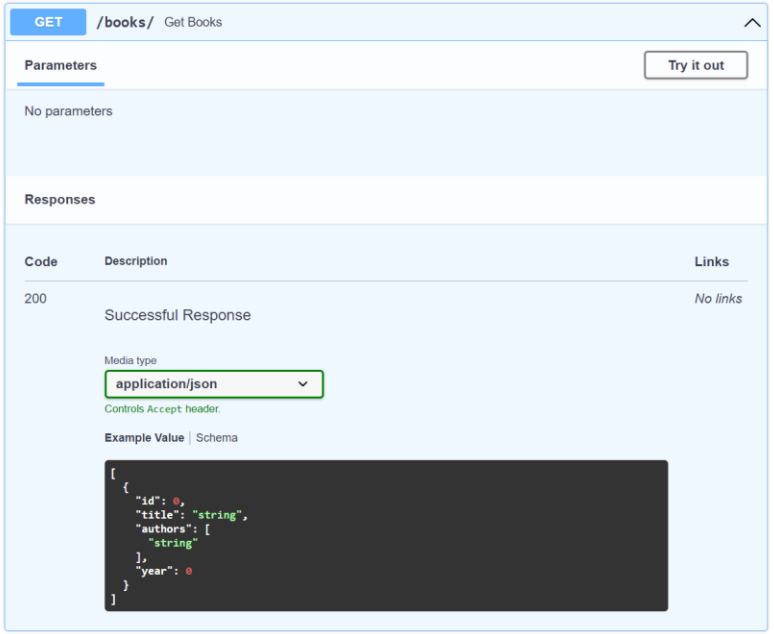


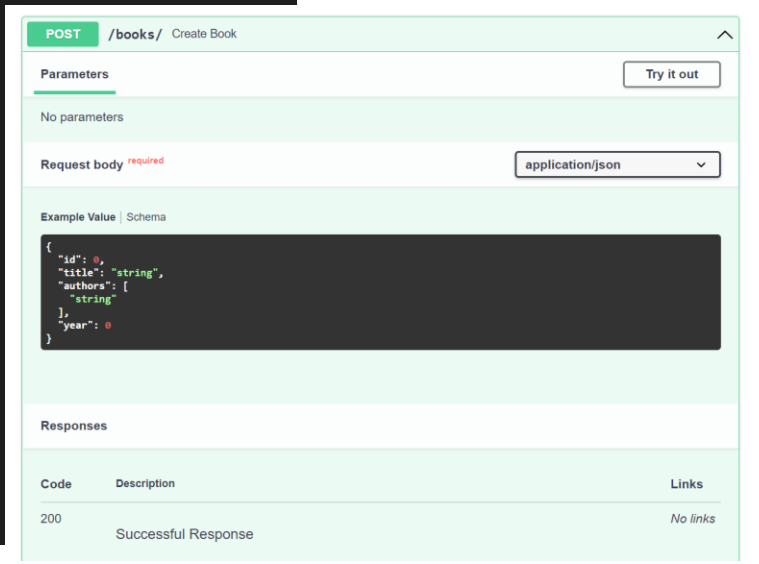



howest hogeschool

# A short introduction -- Pydantic

```python
# With Pydantic
class Book(BaseModel):
    id: Optional[int]
    title: str
    authors: List[str]
    year: int


books = []

@app.post("/books/", response_model=Book)
def create_book(book: Book):
    book.id = len(books) + 1
    books.append(book)
    return book

@app.get("/books/", response_model=List[Book])
def get_books():
    return books

@app.get("/books/{book_id}" , response_model=Book)
def get_book(book_id: int):
    return list(filter(lambda b: b.id == book_id, books))[0]
```






howest hogeschool

# Production ready?

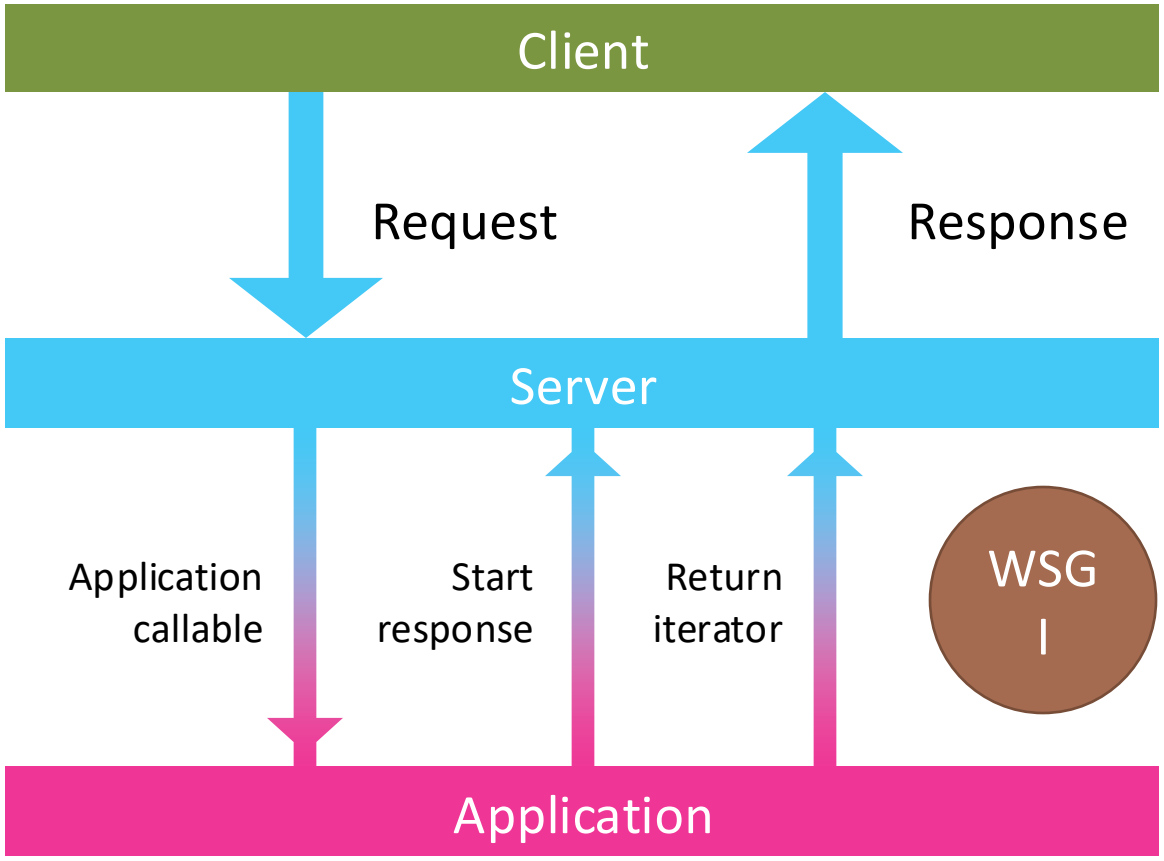
- ALMOST!

- FastAPI is implementing the Asynchronous Gateway Interface
  - Concurrent and asynchronous requests
- It still requires a Python webserver to translate HTTP requests to the right routes
  - And to correctly format the Python responses to HTTP responses

- **Uvicorn**, gunicorn, cherryPy … are options to implement this!

# Old: Web Server Gateway Interface (WSGI)

- Deploying **Flask** in production requires a Web Server Gateway Interface (WSGI)
- It allows for multiple simultaneous requests
- **uWSGI** is one interface implementation
  - Also: gunicorn, cherryPy …
- Allows to easily change Application frameworks





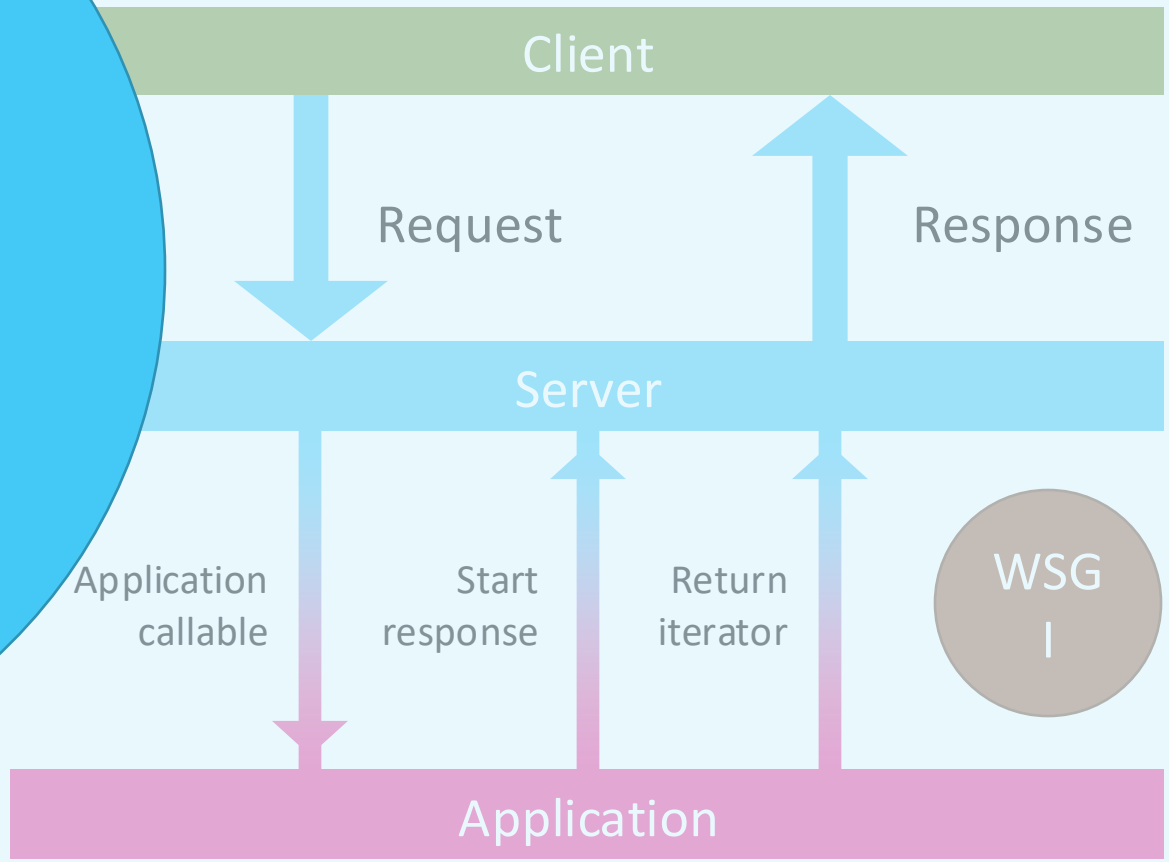

howest hogeschool

# Old: Web Ser… …erface (WSGI)

- Deployi… a Web …
- It allo… reque…
- **uWSGI**
  - Also…
- Allows to … frameworks

**OUTDATED**

Flask is not suitable for API's.
As we will write API's in Backend, we will use **FastAPI**






howest hogeschool

# NEW: Asynchronous Server Gateway Interface (ASGI)

- Compatible with WSGI
- Asynchronous functions
  - Allows for background coroutines

- **FastAPI**
  - “FastAPI is an ASGI web framework (made with Starlette) for building web APIs based on standard Python type annotations and standards like OpenAPI, JSON Schema, and OAuth2. Supports HTTP and WebSockets.”

howest
hogeschool

# Diving into the details!

Can FastAPI do everything we need?

# FastAPI routes (demo 1)

## main.py

```python
@app.post("/books/", response_model=Book)
def create_book(book: Book):
    book.id = len(books) + 1
    books.append(book)
    return book

@app.get("/books/", response_model=List[Book])
def get_books():
    return books

@app.get("/books/{book_id}", response_model=Book)
def get_book(book_id: int):
    return list(filter(lambda b: b.id == book_id, books))[0]

@app.put("/books", response_model=List[Book])
def update_book(book: UpdateBook):
    old_book: Book = get_book(book.id)
    index: int = books.index(old_book)
    books[index].authors = book.authors
    return books

@app.delete("/books/{book_id}")
def delete_book(book_id: int):
    books.pop(book_id - 1)
    return {"message": "Book deleted successfully"}
```

```python
class Book(BaseModel):
    id: Optional[int]
    title: str
    authors: List[str]
    year: int

class UpdateBook(BaseModel):
    id: int
    authors: List[str]
```

# FastAPI routes in seperate file (demo 2)

## main.py

```python
from .routers import books
app = FastAPI()

app.include_router(
    books.router
)
```

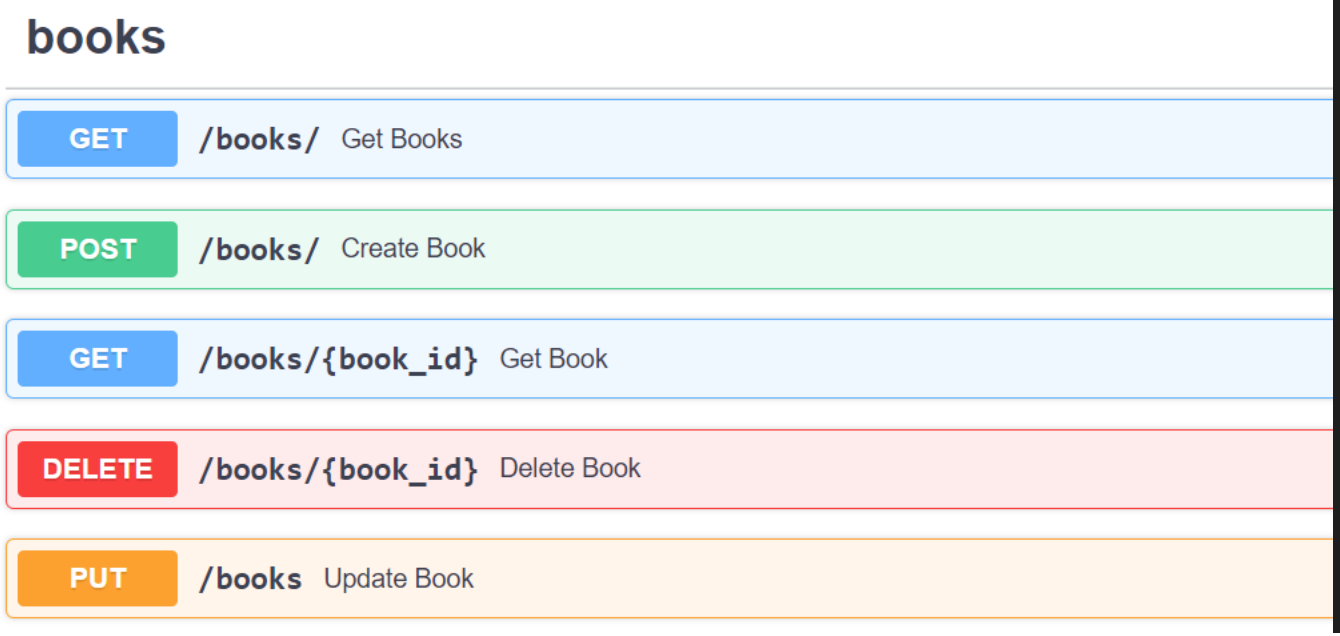
## routers/books.py

```python
from fastapi.routing import APIRouter
router = APIRouter()


@router.post("/books/", response_model=Book)
def create_book(book: Book):
    book.id = len(books) + 1
    books.append(book)
    return book

@router.get("/books/", response_model=List[Book])
def get_books():
    return books

@router.get("/books/{book_id}", response_model=Book)
def get_book(book_id: int):
    return list(filter(lambda b: b.id == book_id, books))[0]

@router.put("/books", response_model=List[Book])
def update_book(book: UpdateBook):
    old_book: Book = get_book(book.id)
    index: int = books.index(old_book)
    books[index].authors = book.authors
    return books

@router.delete("/books/{book_id}")
def delete_book(book_id: int):
    books.pop(book_id - 1)
    return {"message": "Book deleted successfully"}
```

# FastAPI routes in seperate file (demo 2)

## main.py

```python
from .routers import books
app = FastAPI()

app.include_router(
    books.router,
    prefix="/books",
    tags=["books"],
)
```




## routers/books.py

```python
from fastapi.routing import APIRouter
router = APIRouter()


@router.post("/", response_model=Book)
def create_book(book: Book):
    book.id = len(books) + 1
    books.append(book)
    return book

@router.get("/", response_model=List[Book])
def get_books():
    return books

@router.get("/{book_id}", response_model=Book)
def get_book(book_id: int):
    return list(filter(lambda b: b.id == book_id, books))[0]

@router.put("", response_model=List[Book])
def update_book(book: UpdateBook):
    old_book: Book = get_book(book.id)
    index: int = books.index(old_book)
    books[index].authors = book.authors
    return books

@router.delete("/{book_id}")
def delete_book(book_id: int):
    books.pop(book_id - 1)
    return {"message": "Book deleted successfully"}
```

# FastAPI OpenAPI definition (Swagger) (demo 3)

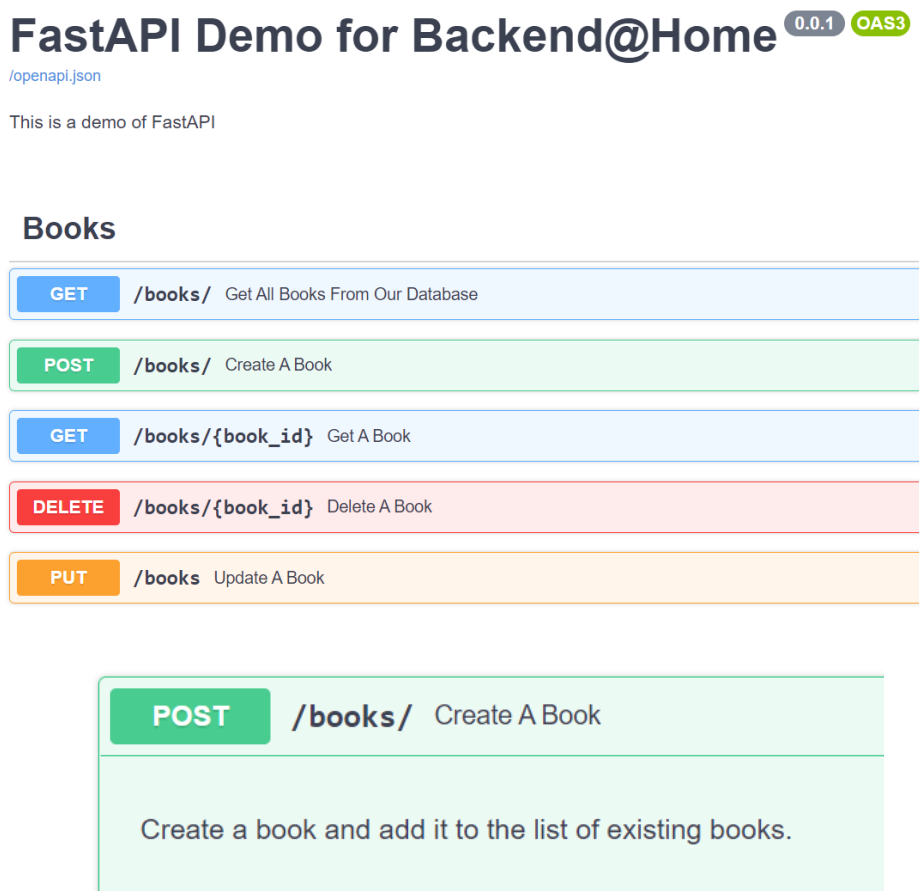
## main.py

```python
app = FastAPI(
    title="FastAPI Demo for Backend@Home",
    description="This is a demo of FastAPI",
    version="0.0.1",
)
```

## routers/books.py

```python
@router.post("/", response_model=Book, tags=["Books"], name="Create a book")
def create_book(book: Book):
    """
    Create a book and add it to the list of existing books.
    """
    book.id = len(books) + 1
    books.append(book)
    return book
```

docstring






howest hogeschool

howest
hogeschool

# Recap into to databases

# The basis of a database … Data

| Number | FirstName | LastName | Phone | Email | Lecturer or Student ? | Favourite Course |
|---|---|---|---|---|---|---|
| 1 | Nathan | Segers | +32 123 456 789 | Nathan.Segers@howest.be | Lecturer | **Backend** |
| 2 | Wouter | Gevaert | +32 123 456 789 | Wouter.Gevaert@howest.be | Lecturer | **Deep Learning** |
| 3 | John | Doe | +32 123 456 789 | John.Doe@student.howest.be | Student | **Data Science** |

What do you note here?



howest hogeschool

# The basis of a database … Data

- Data is mostly interesting when it is linked
- **FavouriteCourse** can be a separate **database**

| CourseID | CourseName |
|---|---|
| 1 | Backend |
| 2 | Data Science |
| 3 | Deep Learning |

| LecturerID | CourseID |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 2 |

| LecturerID | LecturerName | yearsOfExperience |
|---|---|---|
| 1 | Nathan Segers | 3 |
| 2 | Wouter Gevaert | 20 |
| 3 | Gilles Depypere | 2 |

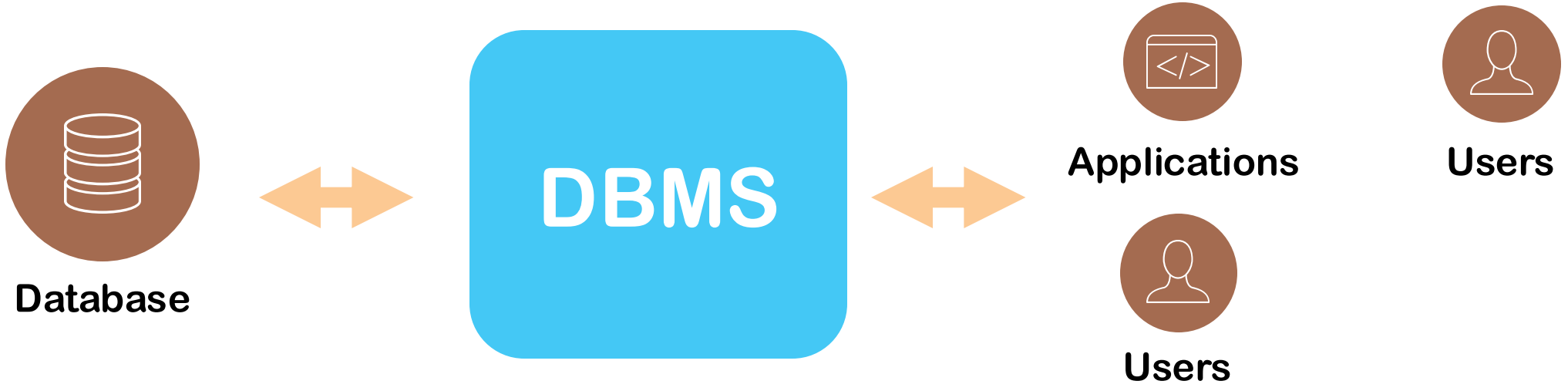


howest hogeschool

# Terminology

A **database** is a name for a system which **collects**, **organises** and **links** data together.

A Database Management System (DBMS) is **software** that allows the **real data** from a **database** to be **read, edited** and **deleted**






howest hogeschool

# More Terminology

**CRUD**

- **C**reate → Insert data into the database
- **R**ead → Read data from the database
- **U**pdate → Update fields in the database
- **D**elete → Delete a field / row from the database

**Migrations** Changes to the structure or data of a database into a new version is called a migration. This also applies to data that has been **merged** together from multiple other databases

# Database types

| | |
|---|---|
| **Relational DB** | Database models with relations and constraints between tables<br>MySQL, MSSQL |
| **Document DB** | Store **documents** in a **tree structure**. Most likely in a JSON format.<br>MongoDB, DynamoDB |
| **Graph DB** | Store in a **multidimensional** structure with **nodes, properties** and **connections (edges)**.<br>**Neo4J** |
| **Time Series DB** | When **time** is the most important feature (logging, search, analysis), use a Time Series database<br>**InfluxDB** |
| **Search Engines** | Documents with **live** and **fast** searches with **fulltext**<br>**Elasticsearch** |

howest
hogeschool

# Relational databases recap

# More Terminology – Relational DB

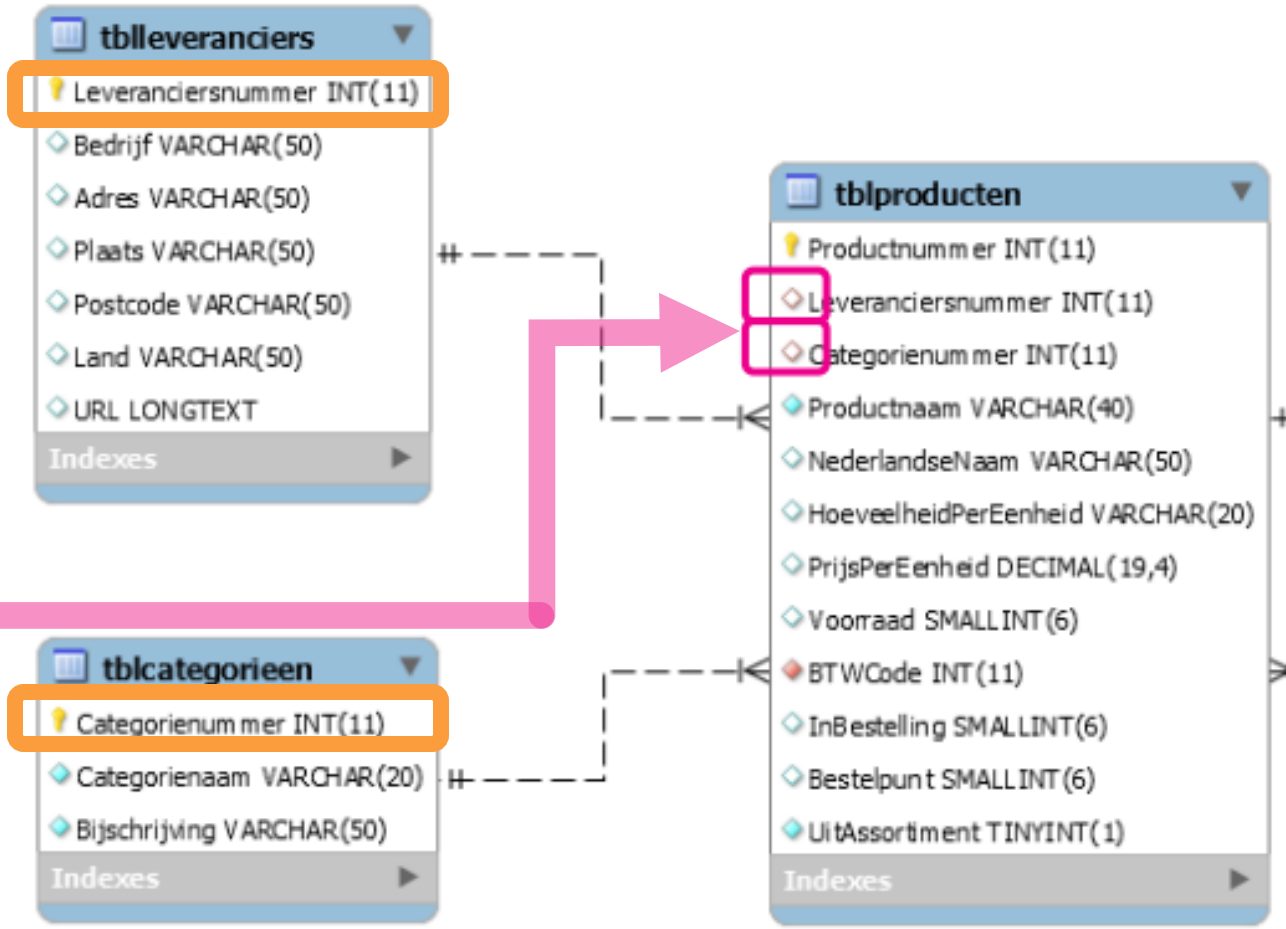
- Tables / Entities
  - Contains data which belongs together
  - **Please ensure** no duplicate or false data is entered / can be entered!

- Fields **or** columns
  - Contains a certain **datatype**
- Record
  - 1 row in a table

| Description | DataType | Example |
|---|---|---|
| Bit | bit | 0 / 1 |
| Whole numbers | int | 42 |
| Decimal number | decimal | 42,05 |
| Real number | float | 3,1415926 |
| Variable length string | Varchar | “Howest” |
| Fixed length string | Char | “8500” |
| Date without time | date | “2022-01-01” |
| Date with time | datetime | “2022-01-01 23:24:00” |

# More Terminology – Relational DB → Keys

- Primary Key (PK)
  - Identification record
  - Unique value in this column
- **Foreign** / Reference Key (FK)
  - Connects to a PK ***in another table*** and forms a **relation**
  - Is not unique in the column




Keys have **same** datatype.

Check **integrity**
Removing PK is not allows if FK exists!

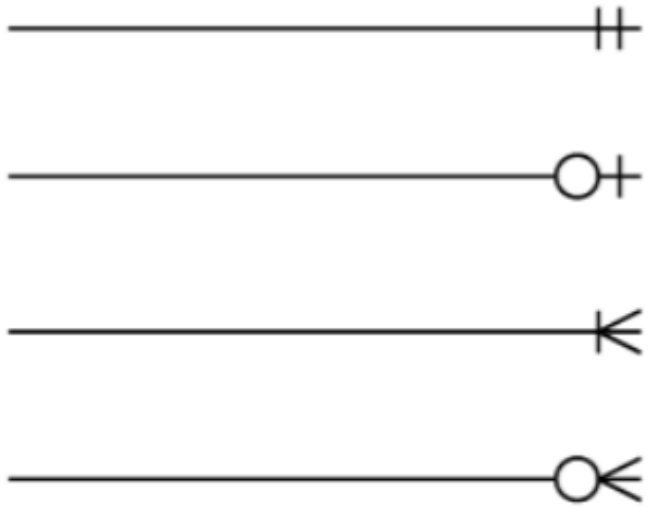


howest hogeschool

# More Terminology – Relational DB → More special fields

**NOTNULL → Field** cannot be a Null value
NULL is not **0** and not **“ ”**

**Relationship**



**One (and only one)**

**Zero or one**

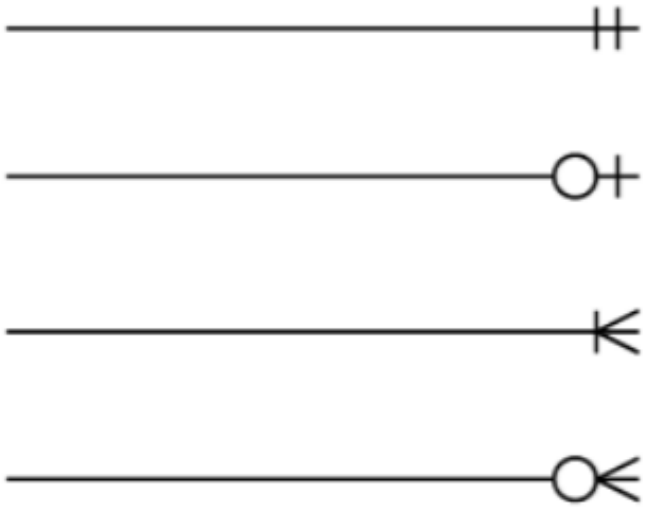
**One or many**

**Zero or many**

# More Terminology – Relational DB → More special fields

**NOTNULL → Field** cannot be a Null value
NULL is not **0** and not **“ ”**

**Relationship**

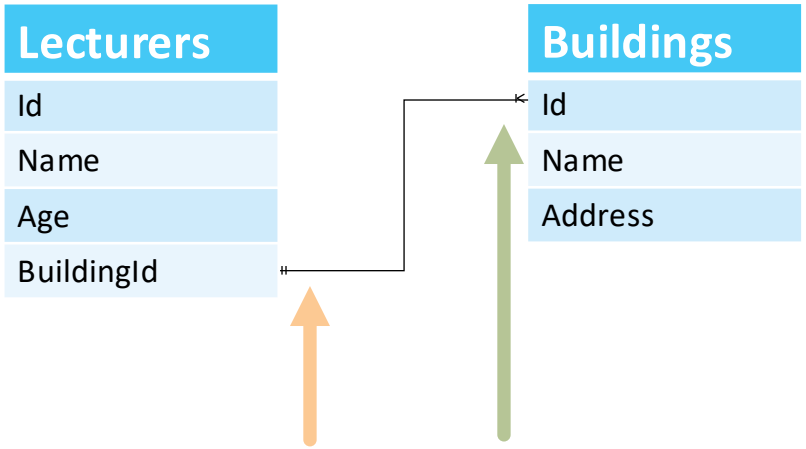


**One (and only one)**

**Zero or one**

**One or many**

**Zero or many**




**One** lecturer resides in **one** building
A lecturer **always** needs **a** building
**One** building is linked to **multiple** lecturers
A building needs **at least one** lecturer



howest hogeschool

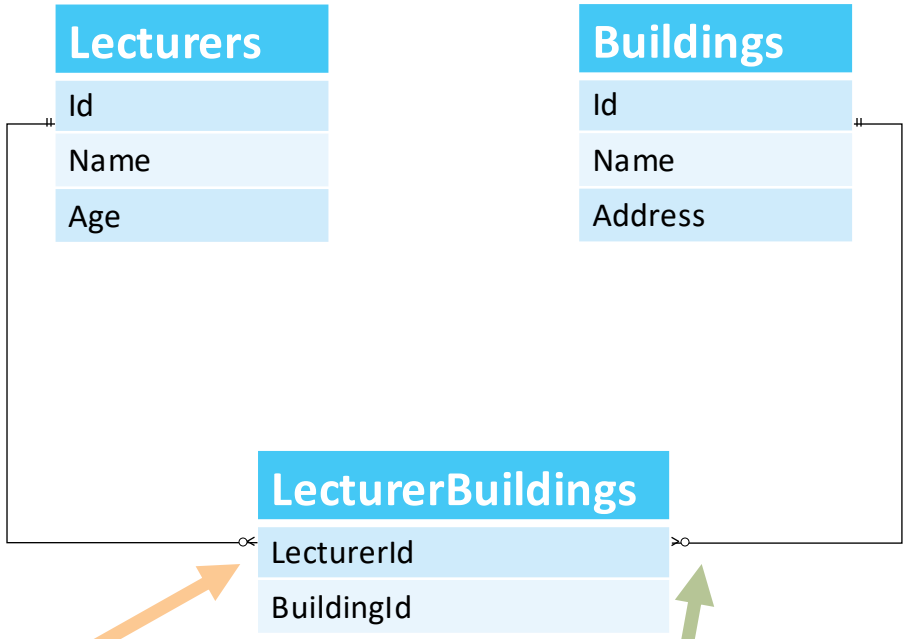
# More Terminology – Relational DB → Many to many relations

**Relationship**

**Two entities** having a **many** to **many** relationship can only work with an **extra table**

**Cardinality**

- 1:1 – One to One
- 1:N – One to many
- N:1 – Many to one
- N:M – Many to many




**One** lecturer resides in **many** buildings
A lecturer **does not always** need a **building**
**One** building contains **many** lecturers
A building **does not always** need a **lecturer**



howest hogeschool

# More Terminology – Relational DB → Transactions

Transactions allow you to roll back a certain **INSERT, UPDATE** or **DELETE.**

You can also **combine** multiple of these statements into one **Transaction** which is ACID-compliant…

ACID? Read on…

# More Terminology – Relational DB → ACID

| | |
|---|---|
| Atomicity | Everything or nothing.<br>A **Transaction** cannot fail partly. It either succeeds or rolls back fully. |
| Consistent | Database can contain **incorrect** data (E.g.: Wrong *age* for a user) but the structure remains **valid**.<br>E.g.: Unique elements stay **unique** … |
| Isolated | Every **transaction** happens isolated from others. They don't even know they exist… |
| Durable | Every **successful transaction** stays persistent, even after a filesystem crash. It doesn't just exist **in-memory**. |

# More Terminology – Relational DB → Normalisation

Normalisation is a **technique** to create a **relational database model**

A few **normalisation forms**

**1 NF → 7 NF**

**3.5 NF** → Dr Boyce Codd (BCNF)

# Installation and management

**MySQL Workbench** can help you install and manage your database …

**…** but, remember Docker?

We can also use that to quickly spin up a working database container using any of the Docker Images that are being distributed freely.

In order to manage our database, we will use the **Adminer** Docker image. This serves as a GUI on top of multiple DBMS.

howest
hogeschool

# SQL and DBML

# Query database using SQL

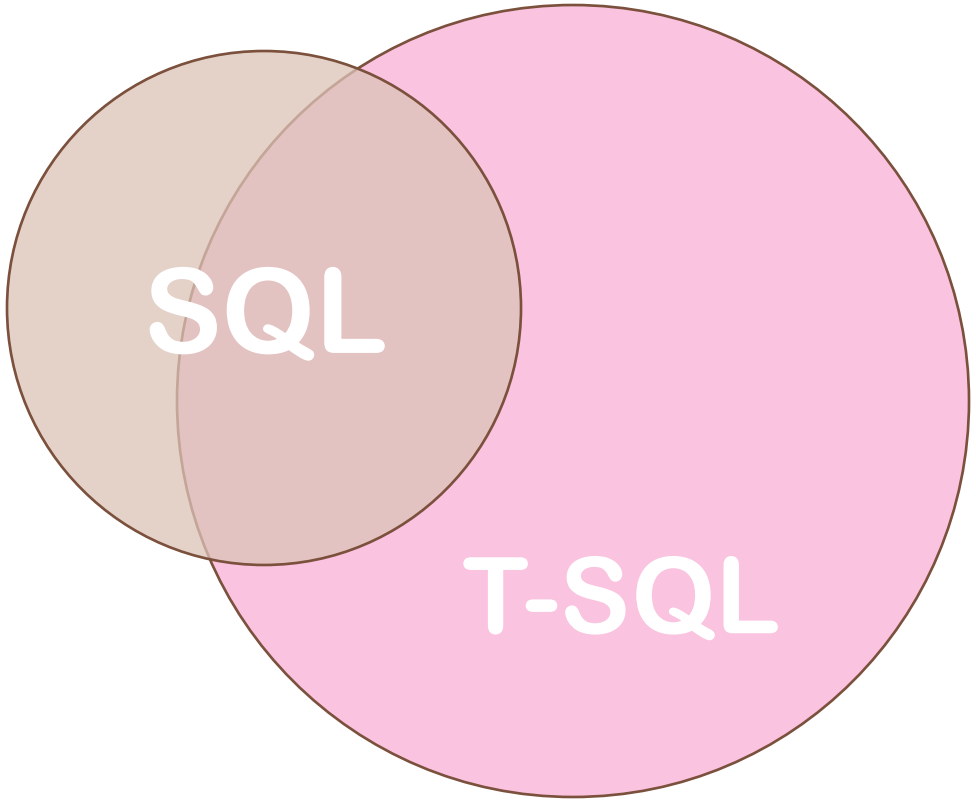
SQL → Structured Query Language

Pronounced: S-Q-L or SeeQuel

T-SQL → Transact-SQL

Created by **Microsoft**

**DML → Data Manipulation Language**






howest hogeschool

# Query database using SQL

Structured → **Structure** your statements!

Query → **Fetch** information (or change information)

Language → 4th generation of the language, starting to feel natural

- Try to understand and explain the question you have
- "Give me all information of all the customers living in Kortrijk"
  - ALL information
  - ALL customers
  - Filter on those living in Kortrijk
- Know the **structure** of your database!

# SQL Syntax

- Multiple column names allow you to fetch multiple fields at once.
  Use * if you want to fetch all
- Use **concat()** if you want to combine two things into one field
- Use **AS** in the **FROM** statement to set an Alias
- Perform calculations on Query results to reformat your results.
- Operators such as =, <, >, <=, >=, **<> or !=**
  Maths: + - * / %
  Boolean: AND, OR, NOT, XOR

**SELECT**
Column_list

**FROM**
table_name

**WHERE**
Filter_condition

# SQL Syntax

There are many more options, which I'm not going to go into detail for now.

Search for some more explanations regarding:

- Joins
- Filters
- Limit
- OrderBy
- …

**SELECT**
Column_list

**FROM**
table_name

**WHERE**
Filter_condition

# SQL, always the best option?

Not always will we use SQL together with relational databases …

Sometimes it’s necessary to have a look at the other types of databases, if they fit our need better. For other types, we might use other dedicated query languages

The motivation for relational databases compared to alternatives depends on a certain number of factors. Let’s dive in to some of those!

howest
hogeschool

# SQLAlchemy and ORM

# FastAPI, Pydantic and SQLAlchemy

- FastAPI: High-performance web framework for building APIs with Python
  - Async for improved performance
- Pydantic: Data validation and parsing using Python type annotations
  - Request and Response validation
- SQLAlchemy: Powerful ORM and SQL Toolkit for Python
  - Code-first database structure
  - Relationships
  - Migrations --> Alembic

Together they enable rapid development of robust APIs with validated data models and database integration.

# Introduction to Queries in SQLAlchemy

SQLAlchemy allows ORM-based querying using Pythonic syntax.

Queries are constructed using the session object and Table classes.

Supports CRUD operations and complex queries including joins and filters.

```python
class Post(Base):
    __tablename__ = 'posts'
    id = Column(Integer, primary_key=True)
    title = Column(String)
    user_id = Column(Integer, ForeignKey('users.id'))
    user = relationship('User', back_populates='posts')
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    posts = relationship('Post', back_populates='user')
```

```python
class PostModel(BaseModel):
    id: int
    title: str
    user_id: int
    user: UserModel

class UserModel(BaseModel):
    id: int
    name: str
    posts: List[PostModel]
```

# Simple Query Example

Example: Fetch a single post by ID

SQLAlchemy:

```
post = db.query(Post).filter(Post.id == 1).first()
```

Equivalent SQL:

```
SELECT * FROM posts WHERE id = 1;
```

# Create Operation (Insert)

Example: Adding a new user

SQLAlchemy:

```
new_user = User(name='John')
db.add(new_user)
db.commit()
```

Equivalent SQL:

```
INSERT INTO users (name) VALUES ('John');
```

# Read Operation (Select)

Example: Get all posts by a specific user

SQLAlchemy:

```
posts = db.query(Post).filter(Post.user_id == 1).all()
```

Equivalent SQL:

```
SELECT * FROM posts WHERE user_id = 1;
```

# Update Operation

Example: Update a post title

SQLAlchemy:

```
post = db.query(Post).filter(Post.id == 1).first()
post.title = 'Updated Title'
db.commit()
```

Equivalent SQL:

```
UPDATE posts SET title = 'Updated Title' WHERE id = 1;
```

# Delete Operation

Example: Delete a user by ID

SQLAlchemy:

```
user = db.query(User).filter(User.id == 1).first()

db.delete(user)

db.commit()
```

Equivalent SQL:

```
DELETE FROM users WHERE id = 1;
```

# Join Queries

Example: Fetch all posts with their user details

SQLAlchemy:

```
posts = db.query(Post, User).join(User, Post.user_id == User.id).all()
```

Equivalent SQL:

```
SELECT * FROM posts JOIN users ON posts.user_id = users.id;
```

# Complex Queries with Filters and Sorting

Example: Get posts by user ID with sorting

SQLAlchemy:

```
posts = db.query(Post).filter(Post.user_id == 1).order_by(Post.created_at.desc()).all()
```

Equivalent SQL:

```
SELECT * FROM posts WHERE user_id = 1 ORDER BY created_at DESC;
```

# Many-to-Many Relationship Query

Example: Get all tags associated with a post

SQLAlchemy:

```
tags = db.query(Tag).join(post_tags)
.filter(post_tags.c.post_id == 1).all()
```

Equivalent SQL:

```
SELECT * FROM tags JOIN post_tags ON tags.id = post_tags.tag_id WHERE
post_tags.post_id = 1;
```

# Using Pydantic Models with Queries

Convert SQLAlchemy objects to Pydantic models

Example:

```
post_data = PostModel.model_validate(post)
```

Ensures validated data for API responses

# Conclusion on CRUD Operations

SQLAlchemy provides a powerful and expressive way to interact with databases.

Code-first approach keeps database management flexible and Pythonic.

CRUD operations are intuitive and map directly to SQL queries when needed.