Contents

[FASTAPI 3](#_Toc187224654)

[KEY FEATURE 3](#_Toc187224655)

[SETTING UP FASTAPI 3](#_Toc187224656)

[CREATE A VIRTUAL ENVIRONMENT 3](#_Toc187224657)

[VIRTUAL ENVIRONMENT 4](#_Toc187224658)

[SETTING UP VENV IN WINDOWS (USING PIP) 4](#_Toc187224659)

[INSTALLING PACKAGES 5](#_Toc187224660)

[CREATING HELLO WORLD APP 5](#_Toc187224661)

[SWAGGER UI 5](#_Toc187224662)

[CRUD OPERATION 6](#_Toc187224663)

[GET 6](#_Toc187224664)

[POST 8](#_Toc187224665)

[PUT – (UPDATE) 8](#_Toc187224666)

[DELETE 8](#_Toc187224667)

[REQUEST DATA VALIDATION – PYDANTIC 9](#_Toc187224668)

[HOW TO USE PYDANTIC 9](#_Toc187224669)

[EXAMPLE 9](#_Toc187224670)

[VALIDATION ON FIELD LEVEL 11](#_Toc187224671)

[PYDANTIC CONFIGURATION FOR SWAGGER UI 13](#_Toc187224672)

[CRUD APPLICATION 15](#_Toc187224673)

[PATH VARIABLE – DATA VALIDATION 16](#_Toc187224674)

[QUERY PARAMETER – DATA VALIDATION 17](#_Toc187224675)

[RAISING HTTP EXCEPTIONS 17](#_Toc187224676)

[EXPLICTLY SETTING THE STATUS CODE 19](#_Toc187224677)

[DEPENDENCY INJECTION 21](#_Toc187224678)

[FASTAPI WITH DATABASE 21](#_Toc187224679)

[SETUP THE PROJECT 22](#_Toc187224680)

[SETTING UP ORM – SQLALCHEMY 22](#_Toc187224681)

[DATABASE MODELS 24](#_Toc187224682)

[MODEL – TODOs APP 25](#_Toc187224683)

[APIs 26](#_Toc187224684)

[AUTHORIZATION AND AUTHENTICATION 27](#_Toc187224685)

[ROUTER 28](#_Toc187224686)

[ROUTER UPDATES 28](#_Toc187224687)

[DATABASE UPDATES 29](#_Toc187224688)

[AUTHENTICATING USER 30](#_Toc187224689)

[HASHING PASSWORD 32](#_Toc187224690)

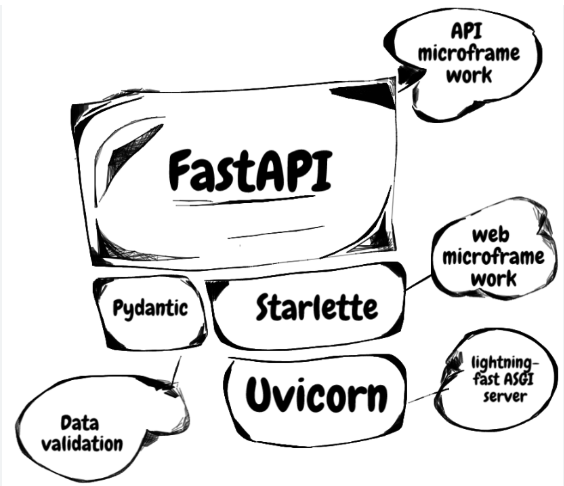
[JWT – JSON WEB TOKEN 32](#_Toc187224691)

[JWT IN FAST API APPS 33](#_Toc187224692)

[ROUTER GROUPING 35](#_Toc187224693)

[TEMPLATING 35](#_Toc187224694)

# FASTAPI



* FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+ based on standard Python type hints.
* It is designed to be fast, easy to use, and to provide automatic validation and serialization of requests and responses.
* FastAPI is built on top of **Starlette**, which is an asynchronous web framework, allowing for high performance and scalability.
* FastAPI leverages Python type hints to provide automatic data validation and serialization, reducing the amount of boilerplate code required.
* It also supports features like **dependency injection, OAuth2 authentication, and automatic generation of API documentation using the OpenAPI standard.**
* Documentation: <https://fastapi.tiangolo.com/>

## KEY FEATURE

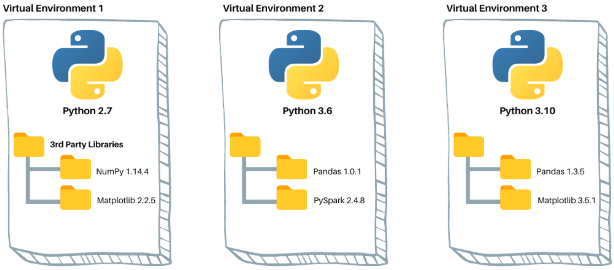
* **High performance**: FastAPI is built on Starlette, an asynchronous web framework, which allows for high-performance and scalability.
* **Easy to use**: FastAPI is designed to be easy to use and reduces the amount of boilerplate code required.
* **Automatic validation and serialization**: FastAPI uses Python type hints to automatically validate incoming requests and serialize outgoing responses.
* **API documentation**: FastAPI automatically generates detailed API documentation using the OpenAPI standard, which can be viewed and interacted with through a web interface.
* **OAuth2 authentication**: FastAPI provides built-in support for OAuth2 authentication, allowing you to secure your APIs easily.

## SETTING UP FASTAPI

### CREATE A VIRTUAL ENVIRONMENT

## VIRTUAL ENVIRONMENT

* A virtual environment in Python is an isolated environment that allows us to have separate Python installations and package dependencies for different projects.
* **It helps in managing and organizing project-specific dependencies, ensuring that each project has its own set of required packages without interfering with other projects or the system-wide Python installation**.
* In a virtual environment, we can install specific versions of Python packages and maintain consistency across different projects.
* By isolating dependencies, we can avoid conflicts that may arise when different projects require different versions of the same package.



### SETTING UP VENV IN WINDOWS (USING PIP)

Step 1: Create a directory where the virtual environment needs to be set up

Step 2: Run the following command in the cmd line (In Corporate machines, create the virtual environments and install the packages with Admin Privilege)

|  |  |
| --- | --- |
| CREATE THE VIRTUAL ENVIRONMENT:   * Virtual environments are created using tools like **virtualenv** or the built-in **venv module** in Python. * These tools create a separate directory with its own Python executable, libraries, and scripts. | python -m venv <**virtual\_env\_name**>  python -m venv fastapienv |
| ACTIVATE THE VIRTUAL ENVIRONMENT: | <**virtual\_env\_name**>\Scripts\activate.bat  fastapiapp\Scripts\activate.bat |
| A black background with white text  Description automatically generated | |
| * Once inside a virtual environment, we can use the Python interpreter and install packages using tools like pip. * Any packages installed in the virtual environment will only be available within that environment and won't interfere with other projects or the system. | |
| For example – Let install the “fastapi” package in the virtual environment (fastapiapp)  **pip install fastapi**   * When we execute – **pip list** command it in the virtual environment it will show the fastapi in installed packages list | A screen shot of a computer  Description automatically generated |

## INSTALLING PACKAGES

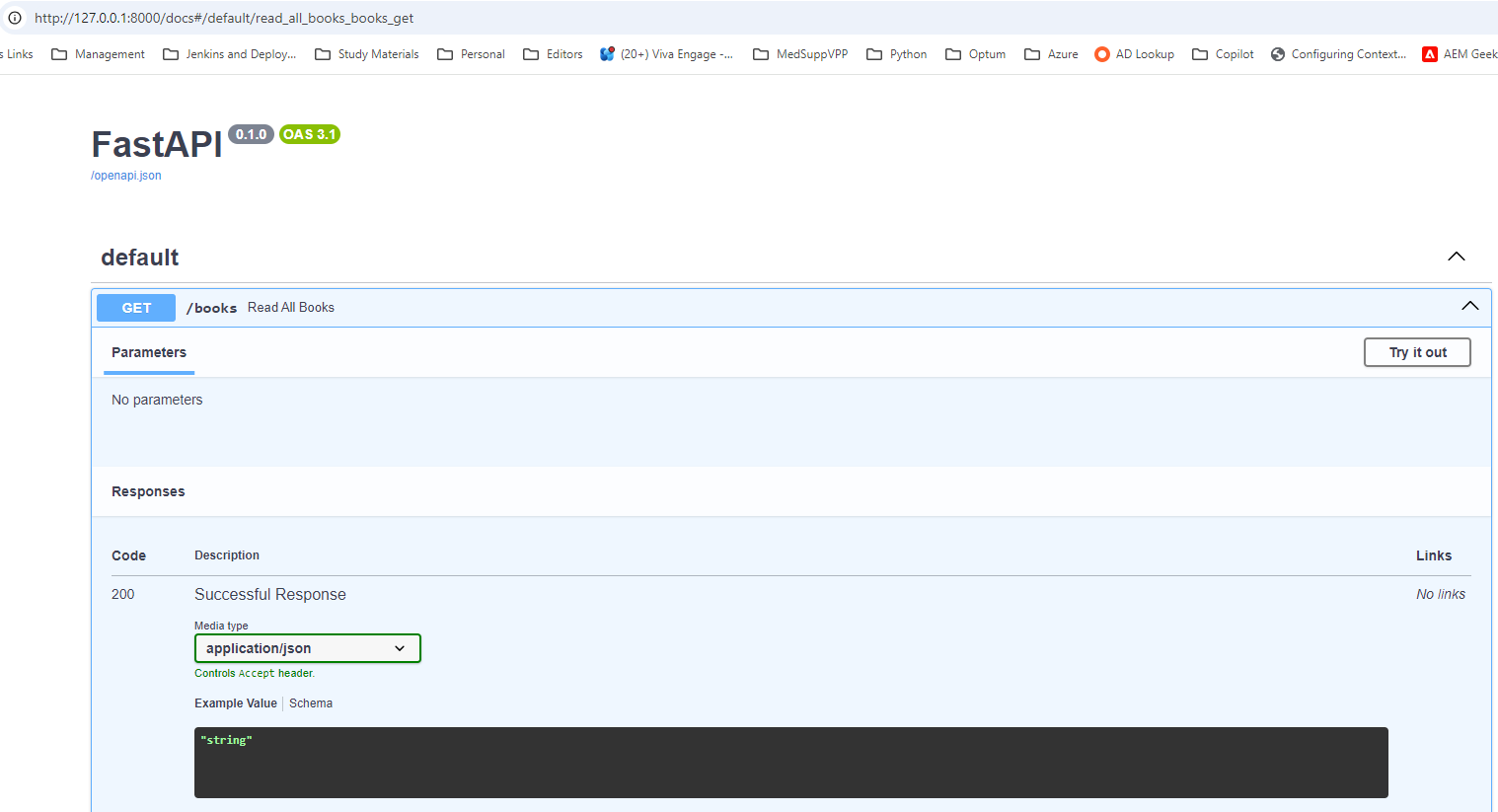
|  |  |
| --- | --- |
| INSTALL FASTAPI | **pip install fastapi** |
| INSTALL uvicorn :  **pip install "uvicorn[standard]"**   * UVicorn is a high-performance ASGI (Asynchronous Server Gateway Interface) server that is commonly used to run Python web applications. * It is specifically designed for running web applications built with frameworks like FastAPI, Starlette, and Quart. | |

## CREATING HELLO WORLD APP

|  |  |
| --- | --- |
| **from fastapi import FastAPI  app = FastAPI()  @app.get("/welcome") async def welcome():  return {'message': "Hello World"}**   1. *The async keyword is* ***not*** *required to be explicitly declared for defining asynchronous functions, it is implicitly added by FastAPI* | RUNNING THE APP: App can be run in multiple ways   1. **UVICORN CMD**: uvicorn books:app --reload 2. This will execute the application at <http://127.0.0.1:8000/> 3. The advantage of using asynchronous functions in FastAPI is that it allows the application to handle a larger number of concurrent requests more efficiently and with better performance. |
| **RUNNING THE FAST APP APPLICATION** | |
| **To run in PROD mode** | **To run in DEV mode** |
| * fastapi run <filename> * Example – fastapi **run** books.py | * fastapi dev <filename> * Example – fastapi **dev** books.py |

### SWAGGER UI

* FastAPI has built-in support for generating Swagger documentation, which is also known as OpenAPI documentation. Swagger documentation allows developers to document their APIs and provide a user-friendly interface for exploring and testing them.
* Local URL for Swaggers UI - <http://127.0.0.1:8000/docs>



## CRUD OPERATION

### GET

|  |
| --- |
| **INITIAL DATA**  **from fastapi import Body, FastAPI app = FastAPI()  BOOKS = [  {'title': 'Title One', 'author': 'Author One', 'category': 'science'},  {'title': 'Title Two', 'author': 'Author Two', 'category': 'science'},  {'title': 'Title Three', 'author': 'Author Three', 'category': 'history'},  {'title': 'Title Four', 'author': 'Author Four', 'category': 'math'},  {'title': 'Title Five', 'author': 'Author Five', 'category': 'math'},  {'title': 'Title Six', 'author': 'Author Two', 'category': 'math'} ]** |

#### PATH PARAMS

|  |
| --- |
| @app.get("/books**/{book\_title}")** def read\_books\_by\_author**(book\_title: str):**  filtered\_books = list(filter(lambda book: book['title'].lower() == book\_title.lower(), BOOKS))  return filtered\_books |

#### QUERY STRING

* For the category parameter, FastAPI interprets it as a query parameter since it is not part of the URL pattern.
* FastAPI automatically converts the query parameter value to the specified type based on the annotation.
* The Query parameter key should match the function parameters (ex. category)

|  |
| --- |
| **@app.get("/books/")**  def read\_books\_by\_author(category: str):  filtered\_books = list(filter(lambda book: book['category'].lower() == category.lower(), BOOKS))  return filtered\_books |

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

#### COMBINING BOTH PATH PARAM AND QUERY PARAM

A computer screen shot of a code

Description automatically generated

|  |
| --- |
| @app.get("/books/{author}") def read\_books\_author\_and\_query\_category(author:str, category: str):  filtered\_books = list(filter(lambda book: book['author'].lower() == author.lower()  and book['category'].lower() == category.lower(), BOOKS))  return filtered\_books |

### POST

|  |  |
| --- | --- |
| **from fastapi import Body, FastAPI**  **@app.post("/books/create")**  def create\_book(book: **dict = Body()):**  BOOKS.append(book)  return book  **REQUEST BODY**  {  "title" : "Title Seven",  "author" : "Author Seven",  "category": "computers"  } | * The book parameter in the create\_book function represents the request body. **FastAPI automatically parses the JSON data in the request body and converts it into a Python dictionary using the pydantic library, which is integrated into FastAPI.** |

### PUT – (UPDATE)

|  |
| --- |
| **from fastapi import Body, FastAPI**  **@app.put("/books/update")** def update\_book(book:dict=Body()):  for i in range(len(BOOKS)):  if BOOKS[i]['title'] == book['title']:  BOOKS[i] = book  return book  return "Book not found" |

### DELETE

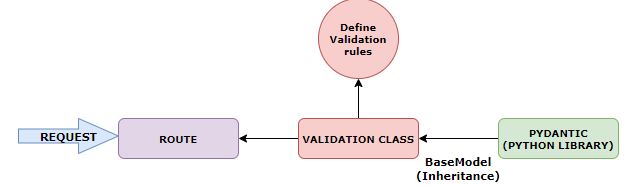
|  |
| --- |
| **@app.delete("/books/delete/{book\_title}")** def delete\_book(book\_title: str):  for index,book in enumerate(BOOKS):  if book['title'].lower() == book\_title.lower():  return BOOKS.pop(index)  return "Book not found" |

## REQUEST DATA VALIDATION – PYDANTIC

* **Pydantic is a Python library that used for** 
  + **DATA MODELING**
  + **DATA PARSING**
  + **ERROR HANDING.**
* It is commonly used with FastAPI to define the structure and validation rules for request and response data.
* Note - Pydantic comes preinstalled with FastAPI installation

### HOW TO USE PYDANTIC

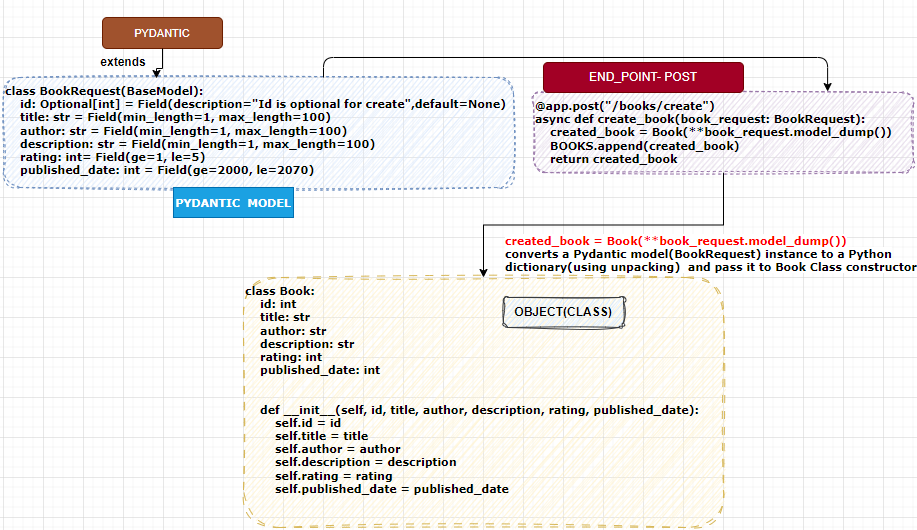
|  |  |
| --- | --- |
| 1. Step1: Create a request model for data validation 2. Step 2: Add field validation on each variable/element | A screen shot of a computer program |



### EXAMPLE

|  |
| --- |
| from fastapi import FastAPI **from pydantic import BaseModel**  **app = FastAPI()**  class Book:  id: int  title: str  author: str  description: str  rating: int  published\_date: int   def \_\_init\_\_(self, id, title, author, description, rating, published\_date):  self.id = id  self.title = title  self.author = author  self.description = description  self.rating = rating  self.published\_date = published\_date  **class BookRequest(BaseModel): 🡨 These rules defines the data types in the request.  id: int  title: str  author: str  description: str  rating: int  published\_date: int**  BOOKS = [  Book(1, 'Computer Science Pro', 'codingwithroby', 'A very nice book!', 5, 2030),  Book(2, 'Be Fast with FastAPI', 'codingwithroby', 'A great book!', 5, 2030),  Book(3, 'Master Endpoints', 'codingwithroby', 'A awesome book!', 5, 2029),  Book(4, 'HP1', 'Author 1', 'Book Description', 2, 2028),  Book(5, 'HP2', 'Author 2', 'Book Description', 3, 2027),  Book(6, 'HP3', 'Author 3', 'Book Description', 1, 2026) ]  @app.get("/books") async def read\_all\_books():  return BOOKS  @app.post("/books/create") async def create\_book(**book\_request: BookRequest**):  **created\_book = Book(\*\*book\_request.model\_dump())**  BOOKS.append(created\_book)  return created\_book |

**new\_book = Book(\*\*book\_request.model\_dump())**



* **book\_request.model\_dump()** has been used can use to convert a Pydantic model instance to a Python dictionary

|  |  |
| --- | --- |
| def greet(name, age):  print(f"Hello {name}, you are {age} years old.")    person = {"name": "John", "age": 30}    greet(\*\*person) | UNPACKING DICTIONARY USING \*\* operator |

* The **\*\* o**perator is used to unpack the dictionary returned by book\_request.model\_dump() as **keyword arguments**. It allows passing the dictionary's key-value pairs as individual arguments to the Book class constructor.
* Book class has attributes that match the keys in the dictionary, the new\_book instance of the Book class is created with the attributes from the book\_request model.

**EXAMPLE**

|  |
| --- |
| class BookRequest(BaseModel):  id: int  title: str  author: str  description: str  rating: int  published\_date: int    book\_request = BookRequest(id=9, title="Making", author="John Doe", description="Book Description", rating=5, published\_date=2026)    **book\_request\_dict = book\_request.model\_dump()**  **print(book\_request\_dict)**  **# Output: {'id': 9, 'title': 'Making', 'author': 'John Doe', 'description': 'Book Description', 'rating': 5, 'published\_date': 2026}**  new\_book = Book(\*\*book\_request\_dict) |

IF WRONG DATA TYPE IS SENT IN REQUEST

|  |  |
| --- | --- |
| {  **"id": "jjj9",**  "title": "Making",  "author": "John Doe",  "description": "Book Description",  "rating": 5,  "published\_date": "2026"  } |  |

### VALIDATION ON FIELD LEVEL

|  |  |
| --- | --- |
| **UPDATED PYDANTIC MODEL**  from pydantic import BaseModel, Field  class BookRequest(BaseModel):  id: int = **Field( ge=1, le=100)**  title: str = **Field(min\_length=1, max\_length=100)**  author: str = **Field(min\_length=1, max\_length=100)**  description: str = **Field(min\_length=1, max\_length=100)**  rating: int= **Field(ge=1, le=5)**  published\_date: int = **Field(ge=2000, le=2030)** | |
| **ON INVALID REQUEST** | |
| **{**  **"id": "10",**  **"title": "Making",**  **"author": "John Doe",**  **"description": "Book Description",**  **"rating": 50,**  **"published\_date": "2026"**  **}** |  |

#### OPTIONAL FIELD

|  |  |
| --- | --- |
| * In the below example – every book has associated “Id”. * If the Id is auto generated we need not it the in request , hence can be marked Optional * Hence “Id” can be omitted from the request | {  "title": "Making",  "author": "John Doe",  "description": "Book Description",  "rating": 5,  "published\_date": "2026"  } |
| from typing import Optional  from fastapi import FastAPI from pydantic import BaseModel, Field  app = FastAPI()  class Book:  id: int  title: str  author: str  description: str  rating: int  published\_date: int   def \_\_init\_\_(self, id, title, author, description, rating, published\_date):  self.id = id  self.title = title  self.author = author  self.description = description  self.rating = rating  self.published\_date = published\_date  class BookRequest(BaseModel):  **id: Optional[int] = None**  title: str = Field(min\_length=1, max\_length=100)  author: str = Field(min\_length=1, max\_length=100)  description: str = Field(min\_length=1, max\_length=100)  rating: int= Field(ge=1, le=5)  published\_date: int = Field(ge=2000, le=2030)  BOOKS = [  Book(1, 'Computer Science Pro', 'codingwithroby', 'A very nice book!', 5, 2030),  Book(2, 'Be Fast with FastAPI', 'codingwithroby', 'A great book!', 5, 2030),  Book(3, 'Master Endpoints', 'codingwithroby', 'A awesome book!', 5, 2029),  Book(4, 'HP1', 'Author 1', 'Book Description', 2, 2028),  Book(5, 'HP2', 'Author 2', 'Book Description', 3, 2027),  Book(6, 'HP3', 'Author 3', 'Book Description', 1, 2026) ]  @app.get("/books") async def read\_all\_books():  return BOOKS  @app.post("/books/create") async def create\_book(book\_request: BookRequest):  created\_book = Book(\*\*book\_request.model\_dump())  BOOKS.append(create\_unique\_book\_id(created\_book))  return created\_book  def create\_unique\_book\_id(book: Book):  book.id = max([book.id for book in BOOKS]) + 1  return book | |

### PYDANTIC CONFIGURATION FOR SWAGGER UI

#### ADDING DESCRIPTION TO FIELD

|  |
| --- |
| class BookRequest(BaseModel):  id: Optional[int] = Field(description="Id is optional for create",default=None)  title: str = Field(min\_length=1, max\_length=100)  author: str = Field(min\_length=1, max\_length=100)  description: str = Field(min\_length=1, max\_length=100)  rating: int= Field(ge=1, le=5)  published\_date: int = Field(ge=2000, le=2030) |
|  |

#### SETTING UP DEFAULT VALUE IN SCHEMA IN SWAGGER UI

|  |  |
| --- | --- |
| If we want to prepopulate the “Example value” for the request in the in Swagger UI   * This can be achieved using “model\_config” in Pydantic Model |  |
| class BookRequest(BaseModel):  id: Optional[int] = Field(description="Id is optional for create",default=None)  title: str = Field(min\_length=1, max\_length=100)  author: str = Field(min\_length=1, max\_length=100)  description: str = Field(min\_length=1, max\_length=100)  rating: int= Field(ge=1, le=5)  published\_date: int = Field(ge=2000, le=2070)   **model\_config = {  "json\_schema\_extra": {  "example": {  "title": "Title",  "author": "Author",  "description": "Description",  "rating": 5,  "published\_date": 2021  }  }  }** | |
|  | |

### CRUD APPLICATION

|  |
| --- |
| from typing import Optional  from fastapi import FastAPI from pydantic import BaseModel, Field  app = FastAPI()  class Book:  id: int  title: str  author: str  description: str  rating: int  published\_date: int   def \_\_init\_\_(self, id, title, author, description, rating, published\_date):  self.id = id  self.title = title  self.author = author  self.description = description  self.rating = rating  self.published\_date = published\_date  class BookRequest(BaseModel):  id: Optional[int] = Field(description="Id is optional for create",default=None)  title: str = Field(min\_length=1, max\_length=100)  author: str = Field(min\_length=1, max\_length=100)  description: str = Field(min\_length=1, max\_length=100)  rating: int= Field(ge=1, le=5)  published\_date: int = Field(ge=2000, le=2070)   **model\_config = {  "json\_schema\_extra": {  "example": {  "title": "Title",  "author": "Author",  "description": "Description",  "rating": 5,  "published\_date": 2021  }  }  }**  BOOKS = [  Book(1, 'Computer Science Pro', 'codingwithroby', 'A very nice book!', 5, 2030),  Book(2, 'Be Fast with FastAPI', 'codingwithroby', 'A great book!', 5, 2030),  Book(3, 'Master Endpoints', 'codingwithroby', 'A awesome book!', 5, 2029),  Book(4, 'HP1', 'Author 1', 'Book Description', 2, 2028),  Book(5, 'HP2', 'Author 2', 'Book Description', 3, 2027),  Book(6, 'HP3', 'Author 3', 'Book Description', 1, 2026) ]  @app.get("/books") async def read\_all\_books():  return BOOKS @app.get("/books/{book\_id}") async def get\_book\_by\_id(book\_id: int):  for book in BOOKS:  if book.id == book\_id:  return book  return None  @app.get("/books/") async def get\_book\_by\_rating(rating: int):  return [book for book in BOOKS if book.rating == rating]  @app.post("/books/create") async def create\_book(book\_request: BookRequest):  created\_book = Book(\*\*book\_request.model\_dump())  BOOKS.append(create\_unique\_book\_id(created\_book))  return created\_book  @app.delete("/books/{book\_id}") async def delete\_book(book\_id: int):  for index, book in enumerate(BOOKS):  if book.id == book\_id:  return BOOKS.pop(index)  return None  @app.put("/books/update\_book") async def update\_book(book: BookRequest):  for index in range(len(BOOKS)):  if BOOKS[index].id == book.id:  BOOKS[index] = book  return book  return None  def create\_unique\_book\_id(book: Book):  book.id = max([book.id for book in BOOKS]) + 1  return book |

### PATH VARIABLE – DATA VALIDATION

|  |  |
| --- | --- |
| from typing import Optional  from fastapi import FastAPI, **Path** from pydantic import BaseModel, Field  app = FastAPI()  @app.get("/books/{book\_id}") async def get\_book\_by\_id(**book\_id: int = Path(gt=0)):**  for book in BOOKS:  if book.id == book\_id:  return book  return None | * The GET request will accept the “book\_id” path parameter greater than 0 * If the path param is less or equal to zero |
|  | |

### QUERY PARAMETER – DATA VALIDATION

|  |
| --- |
| from fastapi import FastAPI, Path, Query from pydantic import BaseModel, Field app = FastAPI()  @app.get("/books/") async def get\_book\_by\_rating(rating: int= Query(gt=0, le=5)):  return [book for book in BOOKS if book.rating == rating] |

### RAISING HTTP EXCEPTIONS



* HTTP exception can be to raised within our method, which will cancel the functionality of the method

|  |
| --- |
| from typing import Optional  from fastapi import FastAPI, Path, Query, **HTTPException** from pydantic import BaseModel, Field  app = FastAPI()  class Book:  id: int  title: str  author: str  description: str  rating: int  published\_date: int   def \_\_init\_\_(self, id, title, author, description, rating, published\_date):  self.id = id  self.title = title  self.author = author  self.description = description  self.rating = rating  self.published\_date = published\_date  class BookRequest(BaseModel):  id: Optional[int] = Field(description="Id is optional for create",default=None)  title: str = Field(min\_length=1, max\_length=100)  author: str = Field(min\_length=1, max\_length=100)  description: str = Field(min\_length=1, max\_length=100)  rating: int= Field(ge=1, le=5)  published\_date: int = Field(ge=2000, le=2070)   model\_config = {  "json\_schema\_extra": {  "example": {  "title": "Title",  "author": "Author",  "description": "Description",  "rating": 5,  "published\_date": 2021  }  }  }  BOOKS = [  Book(1, 'Computer Science Pro', 'codingwithroby', 'A very nice book!', 5, 2030),  Book(2, 'Be Fast with FastAPI', 'codingwithroby', 'A great book!', 5, 2030),  Book(3, 'Master Endpoints', 'codingwithroby', 'A awesome book!', 5, 2029),  Book(4, 'HP1', 'Author 1', 'Book Description', 2, 2028),  Book(5, 'HP2', 'Author 2', 'Book Description', 3, 2027),  Book(6, 'HP3', 'Author 3', 'Book Description', 1, 2026) ]  @app.get("/books") async def read\_all\_books():  return BOOKS  @app.get("/books/{book\_id}") async def get\_book\_by\_id(book\_id: int = Path(gt=0)):  for book in BOOKS:  if book.id == book\_id:  return book  raise HTTPException(status\_code=404, detail="Book not found")  @app.get("/books/") async def get\_book\_by\_rating(rating: int= Query(gt=0, le=5)):  return [book for book in BOOKS if book.rating == rating]  @app.post("/books/create") async def create\_book(book\_request: BookRequest):  created\_book = Book(\*\*book\_request.model\_dump())  BOOKS.append(create\_unique\_book\_id(created\_book))  return created\_book  @app.delete("/books/{book\_id}") async def delete\_book(book\_id: int):  is\_book\_deleted = False  for index, book in enumerate(BOOKS):  if book.id == book\_id:  is\_book\_deleted = True  return BOOKS.pop(index)  if not is\_book\_deleted:  raise HTTPException(status\_code=404, detail="Book not deleted")  @app.put("/books/update\_book") async def update\_book(book: BookRequest):  is\_book\_changed = False  for index in range(len(BOOKS)):  if BOOKS[index].id == book.id:  BOOKS[index] = book  is\_book\_changed = True  return book  if not is\_book\_changed:  raise HTTPException(status\_code=404, detail="Book not updated")   def create\_unique\_book\_id(book: Book):  book.id = max([book.id for book in BOOKS]) + 1  return book |

### EXPLICTLY SETTING THE STATUS CODE

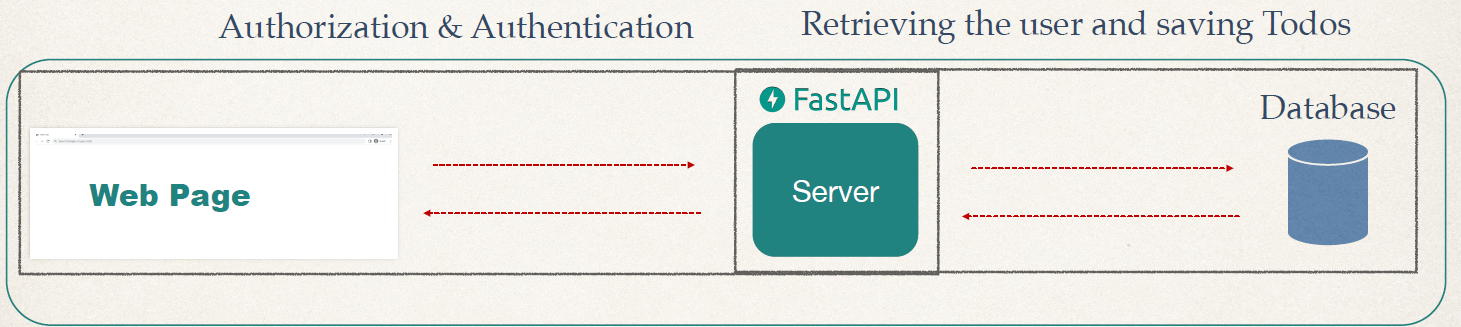
|  |
| --- |
| from typing import Optional  from fastapi import FastAPI, Path, Query, HTTPException from pydantic import BaseModel, Field **from starlette import status**  app = FastAPI()   class Book:  id: int  title: str  author: str  description: str  rating: int  published\_date: int   def \_\_init\_\_(self, id, title, author, description, rating, published\_date):  self.id = id  self.title = title  self.author = author  self.description = description  self.rating = rating  self.published\_date = published\_date   class BookRequest(BaseModel):  id: Optional[int] = Field(description="Id is optional for create",default=None)  title: str = Field(min\_length=1, max\_length=100)  author: str = Field(min\_length=1, max\_length=100)  description: str = Field(min\_length=1, max\_length=100)  rating: int= Field(ge=1, le=5)  published\_date: int = Field(ge=2000, le=2070)   model\_config = {  "json\_schema\_extra": {  "example": {  "title": "Title",  "author": "Author",  "description": "Description",  "rating": 5,  "published\_date": 2021  }  }  }   BOOKS = [  Book(1, 'Computer Science Pro', 'codingwithroby', 'A very nice book!', 5, 2030),  Book(2, 'Be Fast with FastAPI', 'codingwithroby', 'A great book!', 5, 2030),  Book(3, 'Master Endpoints', 'codingwithroby', 'A awesome book!', 5, 2029),  Book(4, 'HP1', 'Author 1', 'Book Description', 2, 2028),  Book(5, 'HP2', 'Author 2', 'Book Description', 3, 2027),  Book(6, 'HP3', 'Author 3', 'Book Description', 1, 2026) ]   @app.get("/books", status\_code = status.HTTP\_200\_OK) async def read\_all\_books():  return BOOKS   @app.get("/books/{book\_id}",status\_code = status.HTTP\_200\_OK) async def get\_book\_by\_id(book\_id: int = Path(gt=0)):  for book in BOOKS:  if book.id == book\_id:  return book  raise HTTPException(status\_code=404, detail="Book not found")   @app.get("/books/",status\_code = status.HTTP\_200\_OK) async def get\_book\_by\_rating(rating: int= Query(gt=0, le=5)):  return [book for book in BOOKS if book.rating == rating]   @app.post("/books/create", status\_code=status.HTTP\_201\_CREATED) async def create\_book(book\_request: BookRequest):  created\_book = Book(\*\*book\_request.model\_dump())  BOOKS.append(create\_unique\_book\_id(created\_book))  return created\_book   @app.delete("/books/{book\_id}", status\_code=status.HTTP\_204\_NO\_CONTENT) async def delete\_book(book\_id: int):  is\_book\_deleted = False  for index, book in enumerate(BOOKS):  if book.id == book\_id:  is\_book\_deleted = True  return BOOKS.pop(index)  if not is\_book\_deleted:  raise HTTPException(status\_code=404, detail="Book not deleted")   @app.put("/books/update\_book", status\_code=status.HTTP\_200\_OK) async def update\_book(book: BookRequest):  is\_book\_changed = False  for index in range(len(BOOKS)):  if BOOKS[index].id == book.id:  BOOKS[index] = book  is\_book\_changed = True  return book  if not is\_book\_changed:  raise HTTPException(status\_code=404, detail="Book not updated")   def create\_unique\_book\_id(book: Book):  book.id = max([book.id for book in BOOKS]) + 1  return book |

# DEPENDENCY INJECTION

* In FastAPI, we can use dependency injection to declare and manage dependencies for the API endpoints or other components. This allows us to easily inject the required dependencies into the functions or classes without explicitly creating or managing them within the function or class itself.
* The most common way to implement dependency injection in FastAPI is **by using function parameters with type hints**. When FastAPI receives a request to an endpoint, it automatically resolves the dependencies based on their declared types and provides them as arguments to your function.

|  |  |
| --- | --- |
| from fastapi import Depends, FastAPI    app = FastAPI()    def get\_db():  # Some code to create and return a database connection    @app.get("/users")  def get\_users(db = Depends(get\_db)):  # Use the db dependency to retrieve users from the database  # ...  return users | * In this example, the `get\_users` function has a dependency on the `db` object, which is resolved by the `Depends` class and the `get\_db` function. * FastAPI automatically creates and provides the `db` object as an argument when the `get\_users` endpoint is called. * You can also use class-based dependencies by using the `Depends` class as a decorator on class methods or by using the `\_\_init\_\_` method of a class to declare dependencies. |

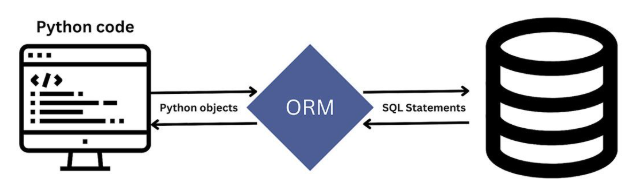
# FASTAPI WITH DATABASE



## SETUP THE PROJECT

|  |  |
| --- | --- |
| 1. Create a Python Package “ToDoApp” 2. As DB we are going to use    1. SQLite- <https://www.sqlite.org/download.html>   *We can be able to browse the database using SQL lite UI tool SQL Browser* |  |

## SETTING UP ORM – SQLALCHEMY

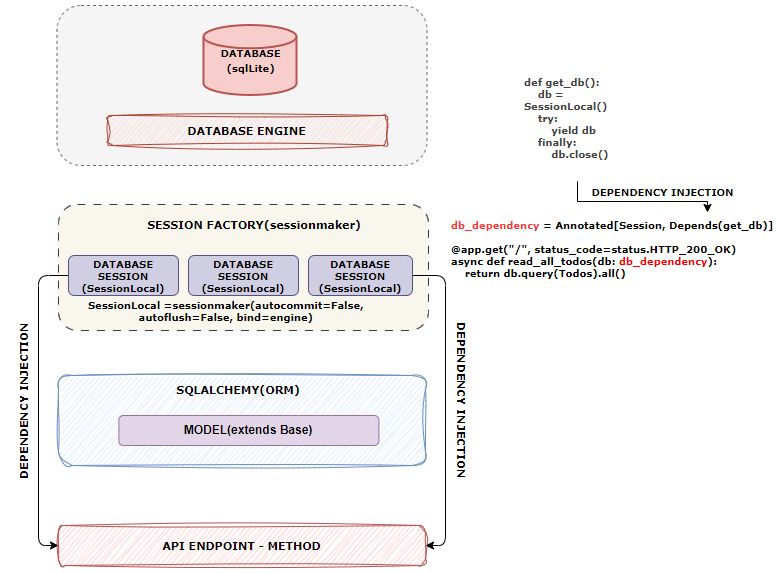


|  |  |
| --- | --- |
| **INSTALL SQL ALCHEMY PACKAGE** | **pip install sqlalchemy** |

* SQLAlchemy is an open-source SQL toolkit and Object-Relational Mapping (ORM) library for Python.
* It provides a set of high-level APIs for working with relational databases, making it easier to interact with databases and perform CRUD (Create, Read, Update, Delete) operations.

**CREATE A PYTHON FILE FOR DB CONNECTION CONFIGURATIONS - database.py**

|  |  |  |
| --- | --- | --- |
| Create a file “**database.py**” in ToDos Application for data base configurations | |  |
| **Step 1: IMPORT LIBRARIES** | **from fastapi import FastAPI**  **from sqlalchemy import create\_engine**  **from sqlalchemy.orm import sessionmaker** | |
| **Step 2: CREATE THE DATABASE CONNECTION**  We need to specify the database URL, which includes the database type (sqlite), and the path to the SQLite database file | **database\_url = "sqlite:///./mydatabase.db"**  **engine = create\_engine(database\_url)** | |
| **STEP 3: CREATE A SESSION FACTORY**   * Create a session factory using the **sessionmaker** function provided by **SQLAlchemy**. * This factory will be used to create individual sessions that interact with the database | **SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)** | |
| **STEP 4: DEFINE DEPENDENCY INJECTION**   * Define a dependency function that will create a new database session for each request and close it after the request is processed. * This will ensure that each request has its own session, and the changes made in one request are not visible to other requests: | **def get\_db():**  **db = SessionLocal()**  **try:**  **yield db**  **finally:**  **db.close()** | |
| **STEP 5: USE THE DATABASE SESSION IN THE ROUTES**   * We can use the **get\_db** function as a dependency in the routes to get a session object for database operations.   @app.get("/books/{book\_id}")  def get\_book(book\_id: int, db: Session **= Depends(get\_db)):**  **book = db.query(Book).filter(Book.id == book\_id).first()**  if book is None:  raise HTTPException(status\_code=404, detail="Book not found")  return book | | |
| from sqlalchemy import create\_engine from sqlalchemy.orm import sessionmaker from sqlalchemy.ext.declarative import declarative\_base  **SQLALCHEMY\_DATABASE\_URL = 'sqlite:///./todos.db' engine = create\_engine(SQLALCHEMY\_DATABASE\_URL, connect\_args={'check\_same\_thread':False}) SessionLocal = sessionmaker(autocommit = False, autoflush= False, bind=engine) Base = declarative\_base()** | | |
| * This code snippet demonstrates the use of SQLAlchemy to create a database engine, session maker, and declarative base. * The `**create\_engine**` function is used to create a database engine. The `SQLALCHEMY\_DATABASE\_URL` variable specifies the URL of the database to connect to. In this case, it is a SQLite database file named "todos.db" located in the current directory. The `**connect\_args**` parameter is used to specify additional arguments to the underlying database driver. In this case, **`check\_same\_thread=False**` is used to allow concurrent access to the database from multiple threads. * The `**sessionmaker**` function is used to create a session factory. The `autocommit=False` and `autoflush=False` parameters are set to disable automatic committing and flushing of changes to the database, respectively. The `bind` parameter is set to the previously created engine, so that the session factory is bound to that engine. * The `**declarative\_base**` function is used to create a base class for declarative models. **This base class will be used as a superclass for all the models defined in the application.** | | |



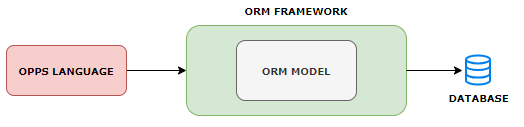
**DATABASE ENGINE**

* A database engine is a software component responsible for **managing the storage, retrieval, and manipulation of data in a database**.
* It provides the necessary functionality to interact with the database, execute queries, and perform various operations.
* A database engine acts as an intermediary between the application and the underlying database system.
* It translates high-level commands and queries from the application into low-level operations that the database system can understand and execute. It handles tasks such as connection management, query optimization, transaction management, and data integrity enforcement.

**SESSION FACTORY**

* A session factory is a class that is responsible for creating and managing database sessions.
* A session represents a single connection to the database and provides a way to interact with the database through SQLAlchemy's ORM (Object-Relational Mapping) capabilities.
* In the context of SQLAlchemy, the session factory is typically created using the sessionmaker class from the sqlalchemy.orm module

## DATABASE MODELS



* It provides a way to interact with the database using object-oriented programming principles instead of raw SQL queries. **The model is typically defined as a class** where:

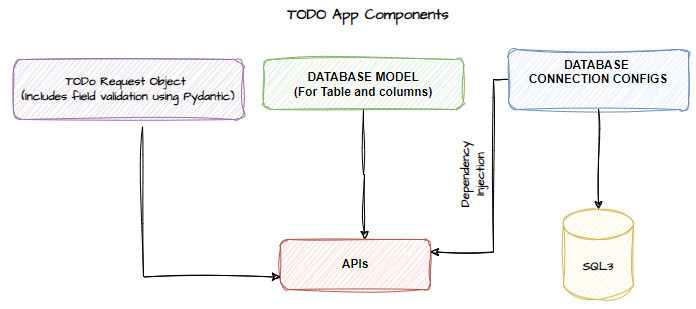
|  |  |
| --- | --- |
| * **Each class represents a database table**:   + The name of the class often corresponds to the name of the table, although it can be customized. * **Each attribute corresponds to a table column**:   + Class attributes represent the columns in the database table. These attributes specify the data types, constraints, and relationships. * **Encapsulation of business logic**:   + Models can include methods for manipulating data and implementing business logic directly related to the table or entity they represent. * **Automatic CRUD operations**:   + The model provides methods for common database operations, such as Create, Read, Update, and Delete (CRUD). * **Relationships**:   + Models define relationships (e.g., one-to-one, one-to-many, many-to-many) between tables using attributes or specific methods. | **EXAMPLE IN SQLALCHEMY (PYTHON)**  from sqlalchemy import Column, Integer, String, Float, create\_engine  from sqlalchemy.ext.declarative import declarative\_base  Base = declarative\_base()  **class Product(Base):**  **\_\_tablename\_\_ = 'products'**  **id = Column(Integer, primary\_key=True)**  **name = Column(String(100))**  **price = Column(Float)**  **# Connecting to a database**  **engine = create\_engine('sqlite:///:memory:')**  **Base.metadata.create\_all(engine)** |

### MODEL – TODOs APP

**The table for which we will create model for**

A table with text on it

Description automatically generated



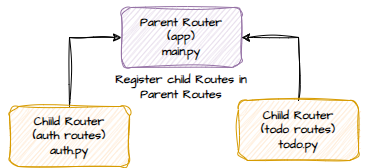
|  |
| --- |
| from database import Base from sqlalchemy import Column, Integer, String, Boolean  class Todos(**Base**):  \_\_tablename\_\_ = "todos"   id = Column(Integer,primary\_key=True, index=True)  title = Column(String)  description = Column(String)  priority = Column(Integer)  complete = Column(Boolean, default=False) |

### APIs

|  |
| --- |
| from typing import Annotated  from fastapi import FastAPI, Depends, HTTPException, Path from pydantic import BaseModel, Field from sqlalchemy.orm import Session from starlette import status import models from models import Todos from database import engine, SessionLocal  app = FastAPI()  models.Base.metadata.create\_all(bind=engine)   **def get\_db():  db = SessionLocal()  try:  yield db  finally:  db.close()**   db\_dependency = Annotated[Session, **Depends(get\_db**)]   class **ToDoRequest**(BaseModel):  title: str = Field(min\_length=3)  description: str = Field(min\_length=3, max\_length=50)  priority: int = Field(gt=0, lt=6)  complete: bool   @app.**get**("/", status\_code=status.HTTP\_200\_OK) async def read\_all\_todos(db: db\_dependency):  return db.query(Todos).all()  @app.**get**("/todo/{todo\_id}", status\_code=status.HTTP\_200\_OK) async def read\_todos(db: db\_dependency, todo\_id: int = Path(gt=0)):  todo\_model = db.query(Todos).filter(Todos.id == todo\_id).first()  if todo\_model is not None:  return todo\_model  raise HTTPException(status\_code=404, detail="No Todo found")   @app.**post**("/todo", status\_code=status.HTTP\_201\_CREATED) async def create\_todos(db: db\_dependency, todos\_request: ToDoRequest):  todo\_model = Todos(\*\*todos\_request.model\_dump())  db.add(todo\_model)  db.commit()  @app.**put**("/todo/{todo\_id}", status\_code=status.HTTP\_200\_OK) async def update\_todos(db: db\_dependency, todos\_request: ToDoRequest, todo\_id: int = Path(gt=0)):  todo\_model = db.query(Todos).filter(Todos.id == todo\_id).first()  if todo\_model is not None:  todo\_model.title = todos\_request.title  todo\_model.description = todos\_request.description  todo\_model.priority = todos\_request.priority  todo\_model.complete = todos\_request.complete  db.add(todo\_model)  db.commit()  return todo\_model  raise HTTPException(status\_code=404, detail="No Todo found")  @app.**delete**("/todo/{todo\_id}", status\_code=status.HTTP\_204\_NO\_CONTENT) async def delete\_todos(db: db\_dependency, todo\_id: int = Path(gt=0)):  todo\_model = db.query(Todos).filter(Todos.id == todo\_id)  if todo\_model is not None:  todo\_model.delete()  db.commit()  return {"detail": "Todo deleted successfully"}  raise HTTPException(status\_code=404, detail="No Todo found") |

# AUTHORIZATION AND AUTHENTICATION

## ROUTER



A screenshot of a diagram

Description automatically generated

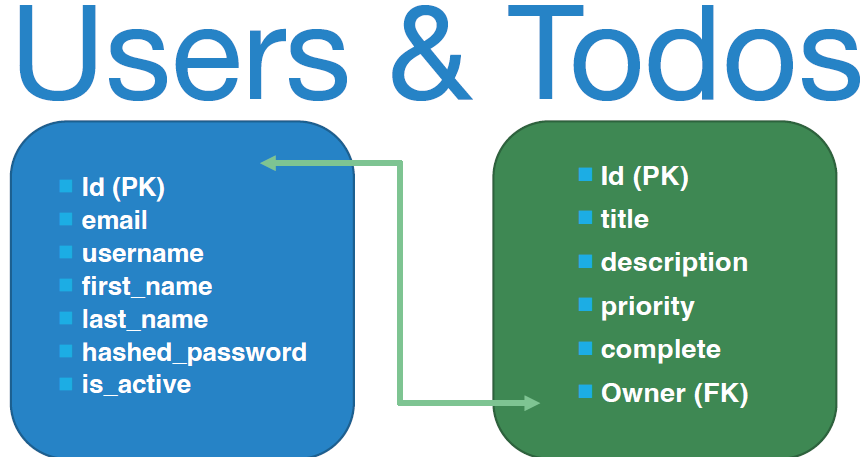
### ROUTER UPDATES

* **We will modify the Todo application and modularize the application based on router**

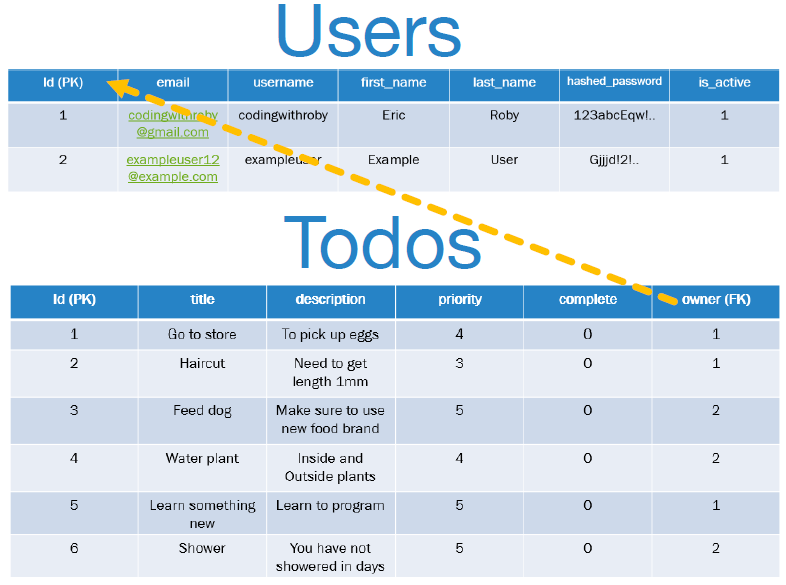
|  |  |
| --- | --- |
| **todo.py (child Router)** | **auth.py(child router)** |
| **from fastapi import APIRouter  router = APIRouter()  @router.post("/user") async def create\_user():  return {"message": "User Created"}** | **from typing import Annotated  from fastapi import APIRouter, Depends, HTTPException, Path from pydantic import BaseModel, Field from sqlalchemy.orm import Session from starlette import status import models from models import Todos from database import engine, SessionLocal  router = APIRouter()  def get\_db():  db = SessionLocal()  try:  yield db  finally:  db.close()  db\_dependency = Annotated[Session, Depends(get\_db)]  class ToDoRequest(BaseModel):  title: str = Field(min\_length=3)  description: str = Field(min\_length=3, max\_length=50)  priority: int = Field(gt=0, lt=6)  complete: bool   @router.get("/", status\_code=status.HTTP\_200\_OK) async def read\_all\_todos(db: db\_dependency):  return db.query(Todos).all()**  **…..**  **OTHER ROUTES FOLLOW** |
| **main.py (Parent Route)** |
| **from fastapi import FastAPI  import models  from database import engine from routers import auth, todos  app = FastAPI()  models.Base.metadata.create\_all(bind=engine)   app.include\_router(auth.router) app.include\_router(todos.router)** |

### DATABASE UPDATES

* **Step 1:** We need to create user table in **todosapp** database. This implies we need to create a **User Model**

****

* **Step 2:**  Add a new column “Owner” in todos table which will act as foreign key, as shown above/ below



#### MODEL UPDATES

|  |
| --- |
| **from database import Base from sqlalchemy import Column, Integer, String, Boolean, ForeignKey  class Users(Base):  \_\_tablename\_\_ = "users"  id = Column(Integer, primary\_key=True, index=True)  email = Column(String, unique=True)  username = Column(String)  first\_name = Column(String)  last\_name = Column(String)  hashed\_password = Column(String)  is\_active = Column(Boolean, default=True) # New created user is active by default  role = Column(String, default="user")  class Todos(Base):  \_\_tablename\_\_ = "todos"   id = Column(Integer, primary\_key=True, index=True)  title = Column(String)  description = Column(String)  priority = Column(Integer)  complete = Column(Boolean, default=False)  owner\_id = ForeignKey("users.id"); 🡸 FOREIGN KEY RELATIONSHIP with “id” column of Users Table** |

A screenshot of a computer

Description automatically generated

## AUTHENTICATING USER

**WHAT IS MULTIPART REQUEST?**

* A multipart request is an HTTP request that contains multiple parts or sections, typically used for sending complex or binary data, **such as file uploads or form data with multiple fields**. It allows us to send a combination of text and binary data within a single HTTP request.
* To authenticate users – the request sends multipart request as a form data.

**STEP 1: INSTALL python-multipart**

* **python-multipart** is a Python library that provides support for parsing and encoding multipart form data. It allows us to work with HTTP requests or responses that contain multipart form data, which is commonly used for file uploads and submitting complex form data.

To install python multipart

|  |  |
| --- | --- |
| **INSTALL MULTIPART** | pip install python-multipart |

This will enable us to submit a form to the FastAPI application

**STEP 2: Make use of OAuth2PasswordRequestForm from fastapi.security**

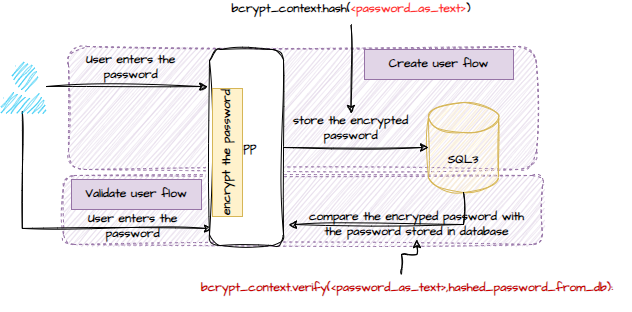
* OAuth2PasswordRequestForm is a utility provided by FastAPI, a Python web framework, that helps handle a specific type of authentication **called OAuth 2.0 password grant**.
* **OAuth 2.0 is a standard protocol for secure authentication and authorization on the web. The password grant type allows a user to enter their username and password to obtain an access token, which can then be used to access protected resources.**
* In FastAPI, we can use OAuth2PasswordRequestForm to receive the username and password from a form submission in a secure manner. It automatically takes care of parsing the form data and providing the username and password values.
* Once we have the username and password, we can use them to authenticate the user, typically by checking if they are valid against a database or external service.
* If the authentication is successful, we can generate an access token, which is a unique identifier that represents the user's authorization to access certain resources.

A diagram of a password request

Description automatically generated

* The access token can be returned in the response to the user, who can then use it to make requests to protected endpoints by including the token in the request headers.

### HASHING PASSWORD



* For the encryption of password we need to install below package

|  |  |
| --- | --- |
| INSTALL passlib | pip install passlib |
| INSTALL bcrypt(Specific version – compatible with passlib) | pip install bcrypt==4.0.1 |

### JWT – JSON WEB TOKEN

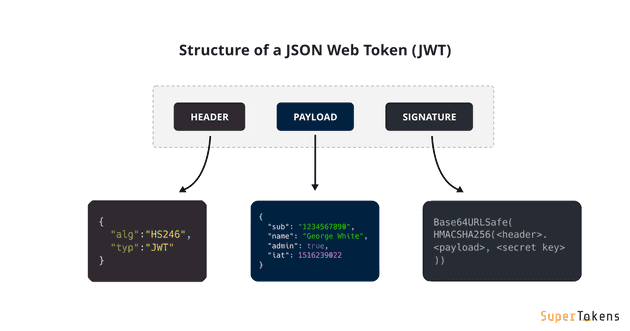
* JSON Web Token, commonly referred to as JWT, is an open standard (RFC 7519) for **securely transmitting information between parties as a JSON object**.
* The token is digitally signed, ensuring its authenticity and integrity. JWTs are primarily used to authenticate users, authorize access to certain resources, and exchange information securely.

#### WORKING PRINCIPLE

A diagram of a server

Description automatically generated

* When a user logs in or attempts to access a protected resource, the server generates a JWT after successful authentication.
* The client then stores this token, usually in local storage or a cookie. For every subsequent request that requires authentication, the client sends the JWT in the request headers.
* The server validates the token by checking the signature and decoding the payload to ensure the user’s authenticity and authorization.



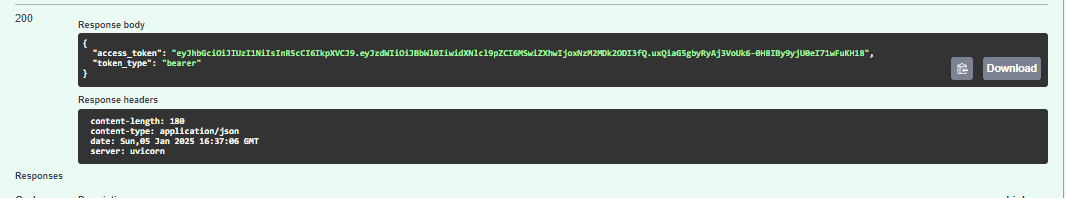
## JWT IN FAST API APPS

|  |
| --- |
| **pip install "python-jose[cryptography]"**   * python-jose[cryptography] is a Python library that provides support for JSON Web Tokens (JWT) and the cryptographic operations required for working with them. It is built on top of the cryptography library, which is a popular cryptography toolkit for Python. * To use python-jose[cryptography], we need to have both python-jose and cryptography installed. * The [cryptography] part in the command specifies that we want to install the additional dependencies required for cryptography support in python-jose. |

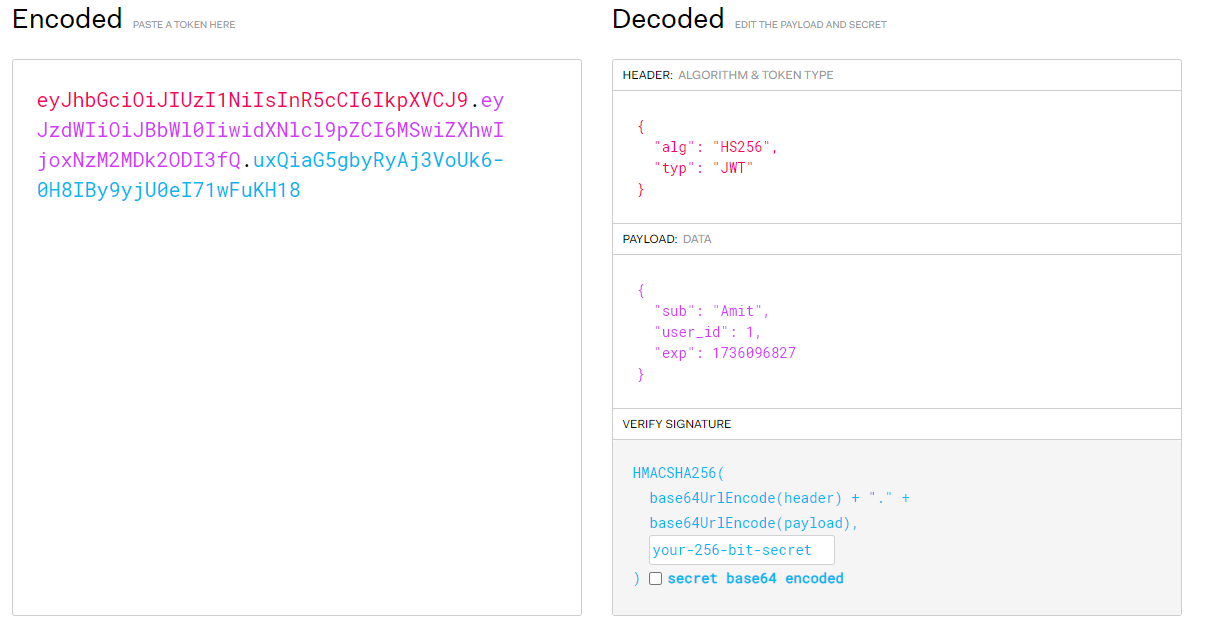
JWT need a secret and an algo

A screenshot of a computer

Description automatically generated



<https://jwt.io/>



DECODING JWT

When a user logs in, we pass in the username and password, we return a JWT.

And now each API endpoint that needs security or authorization.

We're going to verify the JWT that the client is going to pass in to make sure that it's secure and

it hasn't been messed with.

Well, just like we did from our fast API security, we imported our OAuth two password request form.

## ROUTER GROUPING

* APIRouter` allows us to group related endpoints together and apply common prefixes, tags, and other configurations.

### EXAMPLE

**HERE'S AN EXAMPLE OF HOW TO USE `APIROUTER` WITH PREFIXES AND TAGS:**

|  |  |
| --- | --- |
| from fastapi import APIRouter    **router = APIRouter(**  **prefix="/users",**  **tags=["users"],**  **responses={404: {"description": "Not found"}},**  **)**    @router.get("/") 🡨 API ENDPOINT - /users  async def get\_users():  # Logic to retrieve and return all users    @router.get("/{user\_id}") 🡨 API ENDPOINT - /users/{user\_id}  async def get\_user(user\_id: int):  # Logic to retrieve and return a specific user    @router.post("/") 🡨 API ENDPOINT - /users }  async def create\_user(user\_data: dict):  # Logic to create a new user  @router.put("/{user\_id}")  async def update\_user(user\_id: int, user\_data: dict):  # Logic to update a specific user    @router.delete("/{user\_id}") 🡨 API ENDPOINT - /users/{user\_id}  async def delete\_user(user\_id: int):  # Logic to delete a specific user | * The `prefix="/users"` sets the common prefix for all the endpoints defined in the router to "/users". So, the complete URLs for the endpoints will start with "/users". * The `tags=["users"]` assigns the tag "users" to all the endpoints defined in the router. Tags are used for grouping and organizing endpoints in the API documentation. * The `responses={404: {"description": "Not found"}}` sets a default response description for the HTTP status code 404 (Not Found) for all the endpoints in the router. |

### UPDATES IN TODO APP

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

* The API is divided into sections(tags)
* Each URL is now prefixed by prefix mentioned on APIRouter(**todos/auth**)

A group of colored lines

Description automatically generated

# TEMPLATING

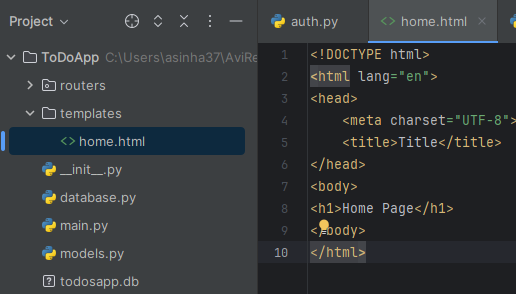
* We use Jinja2 for templating.

## SETTING UP TEMPLATING

1. Step 1: Create a **template** folder, which will contain all the html files
2. **INSTALL FOLLOWING DEPENDENCIES**

|  |  |
| --- | --- |
| **AIO FILES** | **pip install aiofiles** |
| **JINJA 2** | **pip install jinja2** |

1. Create html file in the template folder e.g. **home.html**



### UPDATES IN main.py

|  |  |
| --- | --- |
| from fastapi import FastAPI, **Request**  import models from database import engine from routers import auth, todos **from fastapi.templating import Jinja2Templates**  app = FastAPI()  models.Base.metadata.create\_all(bind=engine)  **templates = Jinja2Templates(directory="templates")**  @app.get("/") async def home(**request: Request**):  return templates.TemplateResponse("home.html", **{"request": request})**  app.include\_router(auth.router) app.include\_router(todos.router) | * We need to import Request and Jinja2Teamplates. * Jinja2 templates are configured to be served from the templates directory * A route is defined for the root URL (/). This route is asynchronous and returns a template response using Jinja2, rendering home.html and passing the request object to the template. * **When rendering a template using Jinja2Templates, we need to pass the request object to the TemplateResponse. This is required for the template to have access to the request context.** |

### SETTING UP STATIC FILES(JS/ CSS)

|  |  |
| --- | --- |
|  | * Create a folder – “static” in root directory * Copy the CSS and JS files in the static folder   **UPDATES IN main.py**  from fastapi import FastAPI, Request  import models from database import engine from routers import auth, todos from fastapi.templating import Jinja2Templates from fastapi.staticfiles import StaticFiles app = FastAPI()  models.Base.metadata.create\_all(bind=engine) templates = Jinja2Templates(directory="templates")  app.mount("/static", StaticFiles(directory="static"), name="static")  @app.get("/") async def home(request: Request):  return templates.TemplateResponse("home.html", {"request": request})  app.include\_router(auth.router) app.include\_router(todos.router) |

#### Mount method

* **app.mount()** is used to **mount a specific path to a directory or another application**.
* The `app.mount()` function allows us to serve static files, such as CSS files, JavaScript files, or images, from a specific directory in the FastAPI application.

**SYNTAX** : **app.mount(path, app, name=None)**

|  |  |
| --- | --- |
| **path** | is the URL path at which we want to serve the static files |
| **app** | is the application or directory we want to mount. In this case, it is a `StaticFiles` object that represents the static files directory. |
| **name** | is an optional parameter that allows us to provide a name for the mounted path |

In the above example:

* We use `app.mount()` to mount the `/static` path to the `StaticFiles` object.
* The `StaticFiles` object is created using the `directory` parameter, which specifies the directory from which the static files will be served. In this case, it is the "static" directory relative to the current working directory.
* After mounting the path, we can place the static files (e.g., CSS, JavaScript, images) in the specified directory, and they will be accessible via the **`/static`** URL path in the application. For example, if we have a file named "style.css" in the "static" directory, it can be accessed at **`http://localhost:8000/static/style.css`.**

#### ACCESSING THE STATIC FILES

|  |
| --- |
| **home.html**  **!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>Title</title>  <link rel="stylesheet" href="{{url\_for('static', path='css/bootstrap.css')}}"> </head> <body> <h1>Home Page {{url\_for('static', path='css/bootstrap.css')}}</h1> </body> </html>** |

##### url\_for

* The `url\_for` function in FastAPI (using Jinja2 templates) is used to generate URLs for static files or endpoints dynamically.
* This will generate the correct URL for the CSS file, ensuring it works correctly in different deployment environments

**syntax: {{ url\_for('static', path='path/to/file') }}**

**Here, `'static'` is the name of the static endpoint, and `path` is the path to the file relative to the static directory.**

|  |  |
| --- | --- |
|  |  |

### LAYOUTING- INHERITANCE

|  |  |
| --- | --- |
|  | **base.html** |
| **<!doctype html> <html lang="en">  <head>  <meta charset="utf-8">  <meta name="viewport" content="width=device-width, initial-scale=1">  <title>{% block title %}{% endblock %}</title>  <link rel="stylesheet" href="{{url\_for('static', path='css/base.css')}}">  <link rel="stylesheet" href="{{url\_for('static', path='css/bootstrap.css')}}">  </head>  <body>  <div class="container">  {%block content%}{% endblock %}  </div>  <script src="{{url\_for('static', path='js/jquery-slim.js')}}"></script>  <script src="{{url\_for('static', path='js/popper.js')}}"></script>  <script src="{{url\_for('static', path='js/bootstrap.js')}}"></script>  <script src="{{url\_for('static', path='js/base.js')}}"></script>  </body> </html>** |
| **home.html** | **{% extends 'base.html' %}** **{% block title %}Home Page{% endblock %}** **{% block content %}  <h1>Home Page</h1> {% endblock %}** |