Contents

[FASTAPI 2](#_Toc186193799)

[KEY FEATURE 2](#_Toc186193800)

[SETTING UP FASTAPI 2](#_Toc186193801)

[CREATE A VIRTUAL ENVIRONMENT 2](#_Toc186193802)

[VIRTUAL ENVIRONMENT 2](#_Toc186193803)

[SETTING UP VENV IN WINDOWS (USING PIP) 3](#_Toc186193804)

[INSTALLING PACKAGES 3](#_Toc186193805)

[CREATING HELLO WORLD APP 4](#_Toc186193806)

[SWAGGER UI 4](#_Toc186193807)

[CRUD OPERATION 4](#_Toc186193808)

[GET 5](#_Toc186193809)

[POST 7](#_Toc186193810)

[PUT – (UPDATE) 7](#_Toc186193811)

[DELETE 7](#_Toc186193812)

[REQUEST DATA VALIDATION – PYDANTIC 7](#_Toc186193813)

[HOW TO USE PYDANTIC 7](#_Toc186193814)

[EXAMPLE 8](#_Toc186193815)

[VALIDATION ON FIELD LEVEL 9](#_Toc186193816)

[PYDANTIC CONFIGURATION FOR SWAGGER UI 11](#_Toc186193817)

[CRUD APPLICATION 13](#_Toc186193818)

[PATH VARIABLE – DATA VALIDATION 14](#_Toc186193819)

[QUERY PARAMETER – DATA VALIDATION 15](#_Toc186193820)

[STATUS CODE - HTTP EXCEPTION 15](#_Toc186193821)

[EXPLICTLY SETTING STATUS CODE 17](#_Toc186193822)

[FASTAPI WITH DATABASE 19](#_Toc186193823)

[SETUP THE PROJECT 19](#_Toc186193824)

[SETTING UP ORM – SQLALCHEMY 19](#_Toc186193825)

[MODEL 20](#_Toc186193826)

# 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

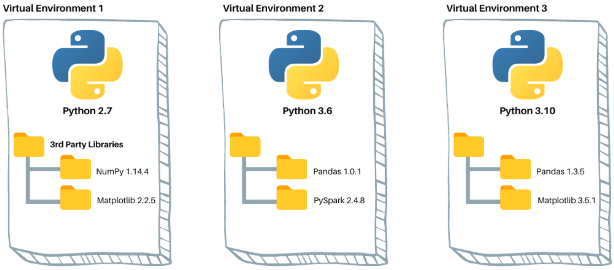
* **High performance**: FastAPI is built on Starlette, an asynchronous web framework, which allows for high-performance and scalability.
* **Easy to use**: FastAPI is designed to be easy to use and reduces the amount of boilerplate code required.
* **Automatic validation and serialization**: FastAPI 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

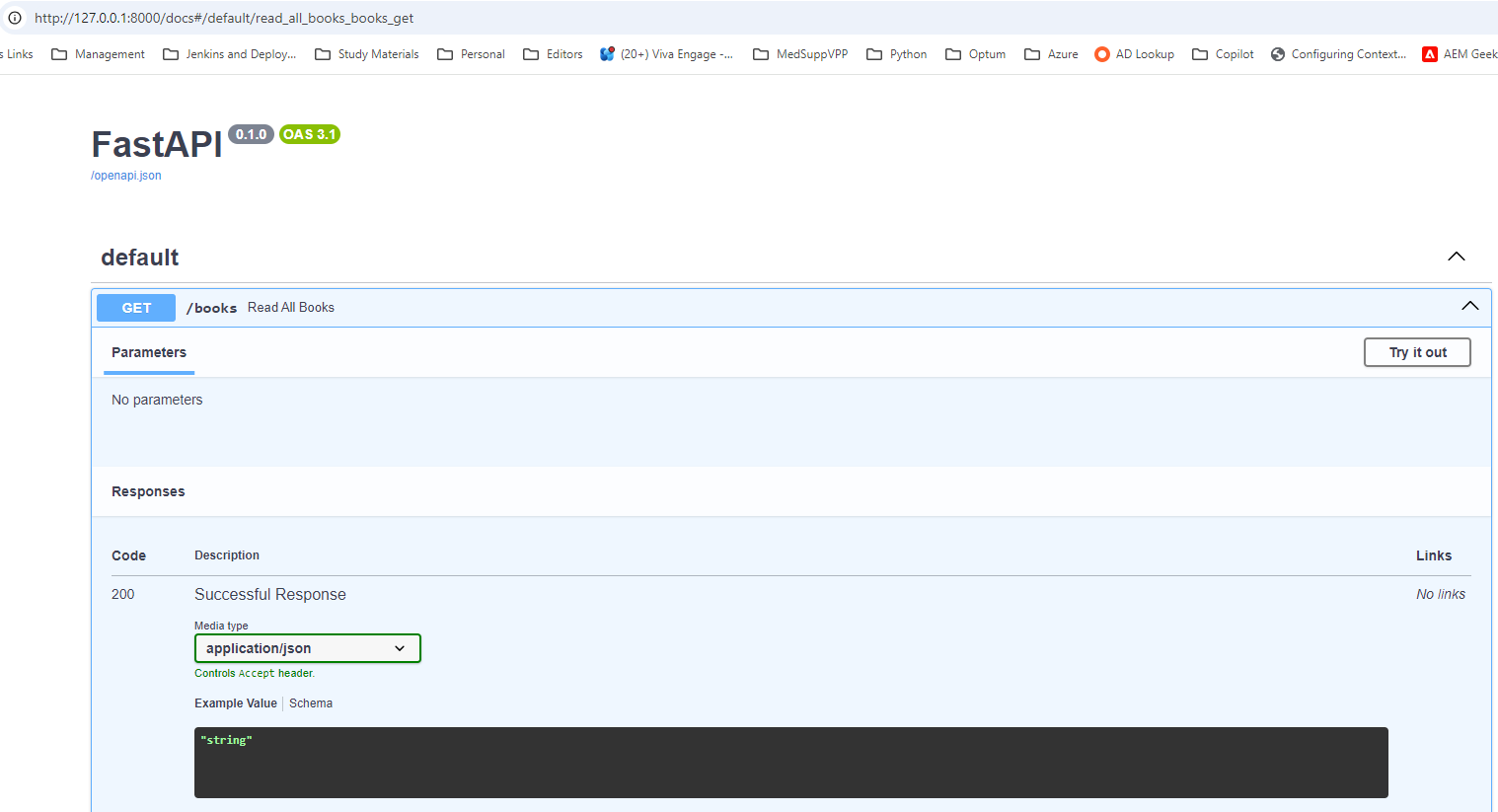
|  |  |
| --- | --- |
| INSTALL FASTAPI | **pip install fastapi** |
| INSTALL uvicorn :  **pip install "uvicorn[standard]"**   * UVicorn is a high-performance ASGI (Asynchronous Server Gateway Interface) server that is commonly used to run Python web applications. * It is specifically designed for running 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** :   * fastapi run <filename> * Example – fastapi **run** books.py   **To run in DEV mode** :   * 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

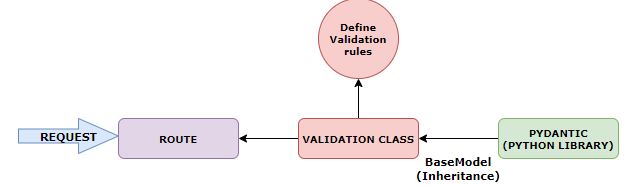
|  |
| --- |
| **@app.delete("/books/delete/{book\_title}")** def delete\_book(book\_title: str):  for index,book in enumerate(BOOKS):  if book['title'].lower() == book\_title.lower():  return BOOKS.pop(index)  return "Book not found" |

## 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.
* Pydantic comes preinstalled with FastAPI installation

### HOW TO USE PYDANTIC

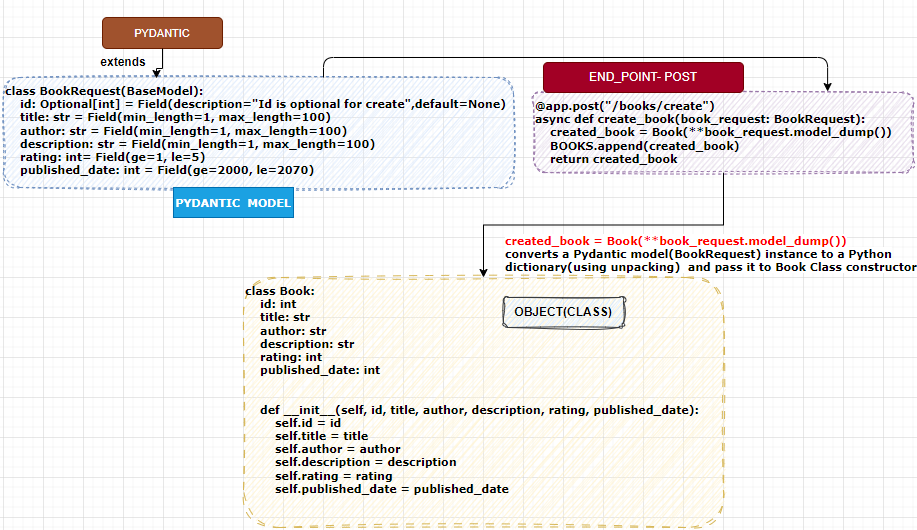
|  |  |
| --- | --- |
| 1. Step1: Create a request model for data validation 2. Step 2: Add field validation on each variable/element | A screen shot of a computer program |



### 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
* 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



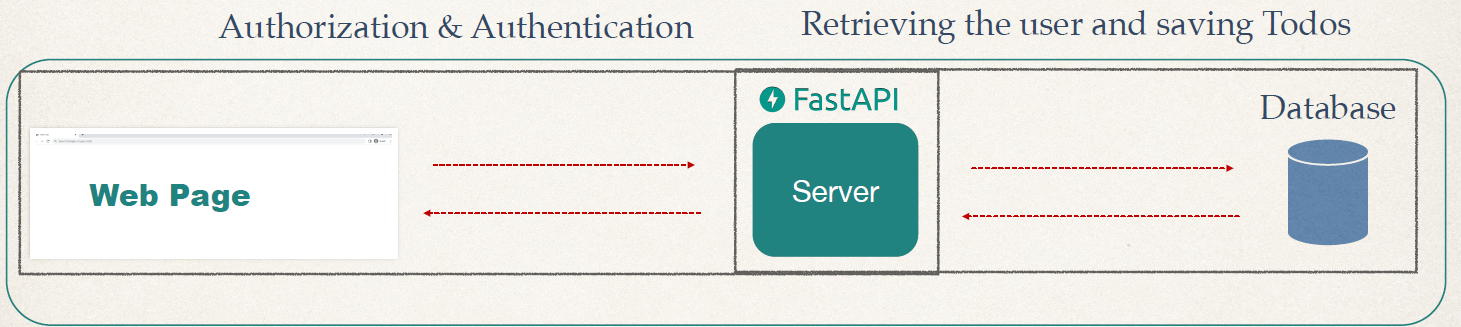
* HTTP exception can be to raised within our method, which will cancel the functionality of the method

|  |
| --- |
| 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 |

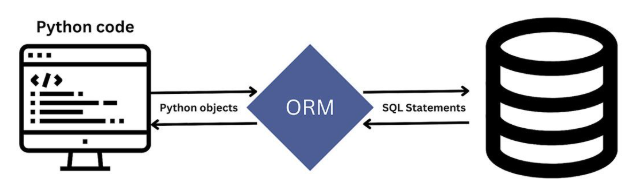
# FASTAPI WITH DATABASE



## 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 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.

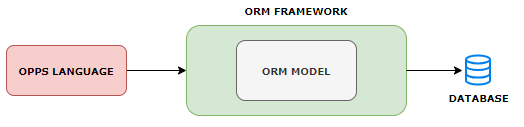
**CREATE A PYTHON FILE FOR DB CONNECTION CONFIGURATIONS - database.py**

|  |  |  |
| --- | --- | --- |
| 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. | | |

**TABLE STRUCTURE (todos)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id (pk) | title | description | priority | complete |
| integer | string | string | integer | boolean |

## 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