How to Use FastAPI [Detailed Python Guide]

Discover how to use FastAPI to build robust and high-performance web applications.

12 min. read · View original

FastAPI Python combines modern Python features with high-performance web development capabilities. This framework stands out for its speed, ease of use, and built-in support for asynchronous programming. Whether you're building APIs, microservices, or full-stack applications, FastAPI offers tools to streamline your development process. This guide will walk you through using FastAPI to create efficient, scalable, and well-documented web applications, highlighting key features that set it apart from other Python web frameworks.

What is FastAPI?

FastAPI Python leverages standard type hints to create high-performance APIs. Built for Python 3.6 and above, this framework accelerates development with its intuitive design and robust features. FastAPI excels in production environments, efficiently handling heavy loads while maintaining code readability and reducing bugs. Its automatic API documentation and validation make it an ideal choice for developers prioritizing both speed and quality in their projects.

Why choose FastAPI for your projects?

1. **Speed**: FastAPI is one of the fastest Python frameworks available, rivaling NodeJS and Go.
2. **Easy to learn**: If you know Python, you're already halfway there.
3. **Fast to code**: Increase the speed to develop features by about 200% to 300%.
4. **Fewer bugs**: Reduce about 40% of human (developer) induced errors.
5. **Intuitive**: Great editor support. Completion everywhere. Less time debugging.
6. **Easy**: Designed to be easy to use and learn. Less time reading docs.
7. **Short**: Minimize code duplication. Multiple features from each parameter declaration.
8. **Robust**: Get production-ready code. With automatic interactive documentation.
9. **Standards-based**: Based on (and fully compatible with) the open standards for APIs: OpenAPI and JSON Schema.

Getting Started with FastAPI

Installing FastAPI and its dependencies

To start using FastAPI, you first need to install it along with its dependencies. Here's how you can do it using pip:

```
pip install fastapi
pip install "uvicorn[standard]"
```

This will install FastAPI along with Uvicorn, an ASGI server that we'll use to run our FastAPI application.

Setting up your development environment

For the best development experience, it's recommended to use a virtual environment. Here's how you can set one up:

```
python -m venv fastapi-env
source fastapi-env/bin/activate  # On Windows, use `fastapi-env\Scripts\activate`
```

Creating your first FastAPI application

Let's create a simple "Hello World" example to get started with FastAPI:

1. Create a new file named `main.py`:

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
async def root():
    return {"message": "Hello World"}
```

2. Run the server:

```
uvicorn main:app --reload
```

3. Open your browser and navigate to `http://127.0.0.1:8000`. You should see a JSON response: `{"message": "Hello World"}`.

FastAPI Fundamentals

Understanding Python type hints and how FastAPI uses them

FastAPI leverages Python's type hints to provide automatic data validation, serialization, and documentation. Here's an example:

```
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = None
@app.post("/items/")
async def create_item(item: Item):
    return {"item_name": item.name, "item_price": item.price}
```

Request handling and routing in FastAPI

FastAPI makes it easy to define routes and handle different HTTP methods:

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

This route handles GET requests to `/items/{item_id}`, where `item_id` is a path parameter and `q` is an optional query parameter.

Path parameters and query parameters

FastAPI allows you to easily work with path and query parameters:

- Path parameters: `@app.get("/users/{user_id}")`
- Query parameters: `@app.get("/items/")`

Example:

```
@app.get("/users/{user_id}/items/")
async def read_user_item(
    user_id: int, item_id: str, q: str = None, short: bool = False
):
    item = {"item_id": item_id, "owner_id": user_id}
    if q:
        item.update({"q": q})
    if not short:
        item.update(
            {"description": "This is an amazing item that has a long description"}
        )
    return item
```

Request body and data validation

FastAPI uses Pydantic models to define the structure of request and response bodies:

```
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None
@app.post("/items/")
async def create_item(item: Item):
    return item
```

Response models and status codes

You can specify response models and status codes in FastAPI:

```
from fastapi import FastAPI, status
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
    name: str
    price: float
@app.post("/items/", response_model=Item, status_code=status.HTTP_201_CREATED)
async def create_item(item: Item):
    return item
```

Advanced FastAPI Features

Dependency Injection in FastAPI

FastAPI's dependency injection system allows you to easily manage shared logic across multiple route handlers:

```
from fastapi import Depends, FastAPI
app = FastAPI()
async def common_parameters(q: str = None, skip: int = 0, limit: int = 100):
    return {"q": q, "skip": skip, "limit": limit}
@app.get("/items/")
async def read_items(commons: dict = Depends(common_parameters)):
    return {"message": "Read items", "params": commons}
@app.get("/users/")
async def read_users(commons: dict = Depends(common_parameters)):
    return {"message": "Read users", "params": commons}
```

Security and authentication

FastAPI provides built-in support for various security and authentication methods. Here's an example using OAuth2 with JWT tokens:

```
from fastapi import Depends, FastAPI, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel
# ... (omitted for brevity: secret key, token URL, algorithm definitions)
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
app = FastAPI()
@app.post("/token")
async def login(form_data: OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=400, detail="Incorrect username or password")
    access_token = create_access_token(data={"sub": user.username})
    return {"access_token": access_token, "token_type": "bearer"}
@app.get("/users/me")
async def read_users_me(current_user: User = Depends(get_current_user)):
    return current_user
```

Background tasks and asynchronous operations

FastAPI supports background tasks and asynchronous operations out of the box:

```
from fastapi import BackgroundTasks, FastAPI
app = FastAPI()
def write_notification(email: str, message=""):
    with open("log.txt", mode="w") as email_file:
        content = f"notification for {email}: {message}"
        email_file.write(content)
@app.post("/send-notification/{email}")
async def send_notification(email: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(write_notification, email, message="some notification")
    return {"message": "Notification sent in the background"}
```

WebSocket support in FastAPI

FastAPI also provides support for WebSockets:

```
from fastapi import FastAPI, WebSocket
from fastapi.responses import HTMLResponse
app = FastAPI()
html = """
<!DOCTYPE html>
<html>
    <head>
```

```
        <title>Chat</title>
    </head>
    <body>
        <h1>WebSocket Chat</h1>
        <form action="" onsubmit="sendMessage(event)">
            <input type="text" id="messageText" autocomplete="off"/>
            <button>Send</button>
        </form>
        <ul id='messages'>
        </ul>
        <script>
            var ws = new WebSocket("ws://localhost:8000/ws");
            ws.onmessage = function(event) {
                var messages = document.getElementById('messages')
                var message = document.createElement('li')
                var content = document.createTextNode(event.data)
                message.appendChild(content)
                messages.appendChild(message)
            };
            function sendMessage(event) {
                var input = document.getElementById("messageText")
                ws.send(input.value)
                input.value = ''
                event.preventDefault()
            }
        </script>
    </body>
</html>
"""
@app.get("/")
async def get():
    return HTMLResponse(html)
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message text was: {data}")
```

Monitoring and Tracing with OpenTelemetry

FastAPI can be easily integrated with OpenTelemetry for comprehensive monitoring and tracing. This can significantly enhance your ability to understand and optimize your application's performance.

To get started with OpenTelemetry in your FastAPI application:

1. Install the necessary packages:

```
pip install opentelemetry-api opentelemetry-sdk opentelemetry-instrumentation-fastapi
```

2. Instrument your FastAPI application:

```
from fastapi import FastAPI
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
app = FastAPI()
FastAPIInstrumentor.instrument_app(app)
@app.get("/")
async def root():
    return {"message": "Hello World"}
```

For more detailed information on setting up OpenTelemetry with FastAPI, check out this guide on OpenTelemetry FastAPI monitoring.

If you're interested in deeper insights into tracing with OpenTelemetry in Python, you can refer to the OpenTelemetry Tracing API for Python.

Database Integration with FastAPI

Using SQLAlchemy with FastAPI

FastAPI works well with SQLAlchemy for database operations. Here's a basic example:

```
from fastapi import Depends, FastAPI, HTTPException
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, Session
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)
Base.metadata.create_all(bind=engine)
app = FastAPI()
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = crud.get_user_by_email(db, email=user.email)
    if db_user:
        raise HTTPException(status_code=400, detail="Email already registered")
    return crud.create_user(db=db, user=user)
```

Testing FastAPI Applications

Unit testing with pytest

FastAPI works seamlessly with pytest for testing. Here's an example of a test case:

```
from fastapi.testclient import TestClient
from main import app
client = TestClient(app)
```

```
def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello World"}
def test_create_item():
    response = client.post(
        "/items/",
        json={"name": "Foo", "price": 42.0},
    )
    assert response.status_code == 200
    assert response.json() == {
        "name": "Foo",
        "price": 42.0,
        "is_offer": None,
    }
```

To run these tests, simply use the `pytest` command in your terminal.

FastAPI Performance Optimization

Async and await for improved concurrency

FastAPI is built on top of Starlette, which allows for asynchronous request handling:

```
@app.get("/items/{item_id}")
async def read_item(item_id: str):
    result = await some_async_operation(item_id)
    return result
```

This allows your application to handle many concurrent requests efficiently.

Documenting Your FastAPI Application

Automatic API documentation with Swagger UI

FastAPI Python revolutionizes API documentation with its built-in Swagger UI. Accessible at the '/docs' endpoint, this interactive interface automatically generates comprehensive documentation for your API. Developers can explore endpoints, test requests with different parameters, and view response models in real-time. This feature not only accelerates development and testing but also serves as living documentation, always in sync with your code. The Swagger UI in FastAPI simplifies API consumption for both internal teams and external developers, enhancing collaboration and reducing integration time.

Deploying FastAPI Applications

Containerizing your FastAPI app with Docker

Here's a simple Dockerfile for a FastAPI application:

```
FROM python:3.9
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
COPY ./app /code/app
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

To build and run the Docker container:

```
docker build -t myapp .
docker run -d --name mycontainer -p 80:80 myapp
```

FastAPI Best Practices and Design Patterns

Project structure for scalable FastAPI applications

A recommended project structure for a FastAPI application:

```
├── app
│   ├── __init__.py
│   ├── main.py
│   ├── dependencies.py
│   ├── routers
│   │   ├── __init__.py
│   │   ├── items.py
│   │   └── users.py
│   ├── internal
│   │   ├── __init__.py
│   │   └── admin.py
│   ├── schemas
│   │   ├── __init__.py
│   │   ├── item.py
│   │   └── user.py
│   └── crud
│       ├── __init__.py
│       ├── base.py
│       ├── item.py
│       └── user.py
├── tests
│   ├── __init__.py
│   ├── test_main.py
│   ├── test_items.py
│   └── test_users.py
```

Error handling and custom exception classes

FastAPI allows you to create custom exception handlers for more granular control over error responses:

```
from fastapi import FastAPI, Request, status
from fastapi.responses import JSONResponse
app = FastAPI()
class UnicornException(Exception):
    def __init__(self, name: str):
        self.name = name
@app.exception_handler(UnicornException)
async def unicorn_exception_handler(request: Request, exc: UnicornException):
    return JSONResponse(
        status_code=418,
        content={"message": f"Oops! {exc.name} did something. There goes a rainbow..."},
    )
@app.get("/unicorns/{name}")
```

```
async def read_unicorn(name: str):
    if name == "yolo":
        raise UnicornException(name=name)
    return {"unicorn_name": name}
```

Middleware implementation in FastAPI

Middleware allows you to add custom functionality to your FastAPI application. Here's an example of a simple middleware that adds a custom header to every response:

```
from fastapi import FastAPI
from starlette.middleware.base import BaseHTTPMiddleware
from starlette.responses import Response
app = FastAPI()
class AddCustomHeaderMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        response = await call_next(request)
        response.headers["X-Custom-Header"] = "Some Value"
        return response
app.add_middleware(AddCustomHeaderMiddleware)
@app.get("/")
async def root():
    return {"message": "Hello World"}
```

Logging and monitoring FastAPI applications

Implementing proper logging is crucial for monitoring and debugging your FastAPI application. Here's a basic setup:

```
import logging
from fastapi import FastAPI, Request
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
app = FastAPI()
@app.middleware("http")
async def log_requests(request: Request, call_next):
    logger.info(f"Request: {request.method} {request.url}")
    response = await call_next(request)
    logger.info(f"Response status: {response.status_code}")
    return response
@app.get("/")
async def root():
    logger.info("Processing root request")
    return {"message": "Hello World"}
```

FastAPI Ecosystem and Extensions

Popular FastAPI extensions and libraries

FastAPI has a growing ecosystem of extensions and libraries. Here are some popular ones:

1. **FastAPI Users**: Provides ready-to-use and customizable users management for FastAPI.
2. **FastAPI SQL Database**: Simplifies database operations with SQLAlchemy.
3. **FastAPI Cache**: Adds caching support to FastAPI endpoints.
4. **FastAPI CORS**: Handles Cross-Origin Resource Sharing (CORS) for FastAPI applications.
5. **FastAPI Limiter**: Adds rate limiting to FastAPI routes.

Integrating FastAPI with other Python frameworks

FastAPI can be integrated with other Python frameworks and libraries. For example, you can use FastAPI with Celery for task queues:

```
from fastapi import FastAPI
from celery import Celery
app = FastAPI()
celery = Celery("tasks", broker="redis://localhost:6379")
@celery.task
def add_numbers(x, y):
    return x + y
@app.get("/add/{x}/{y}")
async def add(x: int, y: int):
    result = add_numbers.delay(x, y)
    return {"task_id": result.id}
```

Real-world FastAPI Use Cases

Building a RESTful API with FastAPI

FastAPI is excellent for building RESTful APIs. Here's a simple example of a CRUD (Create, Read, Update, Delete) API for a todo list:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List, Optional
app = FastAPI()
class Todo(BaseModel):
    id: Optional[int] = None
    item: str
    completed: bool = False
todos = []
@app.post("/todos/", response_model=Todo)
async def create_todo(todo: Todo):
    todo.id = len(todos) + 1
    todos.append(todo)
    return todo
@app.get("/todos/", response_model=List[Todo])
async def read_todos():
    return todos
@app.get("/todos/{todo_id}", response_model=Todo)
async def read_todo(todo_id: int):
    try:
        return next(todo for todo in todos if todo.id == todo_id)
    except StopIteration:
        raise HTTPException(status_code=404, detail="Todo not found")
@app.put("/todos/{todo_id}", response_model=Todo)
async def update_todo(todo_id: int, todo: Todo):
    for i, t in enumerate(todos):
        if t.id == todo_id:
            todo.id = todo_id
            todos[i] = todo
            return todo
```

```
    raise HTTPException(status_code=404, detail="Todo not found")
@app.delete("/todos/{todo_id}")
async def delete_todo(todo_id: int):
    for i, todo in enumerate(todos):
        if todo.id == todo_id:
            del todos[i]
            return {"message": "Todo deleted successfully"}
    raise HTTPException(status_code=404, detail="Todo not found")
```

Creating a real-time chat application

FastAPI's WebSocket support makes it easy to create real-time applications. Here's a simple chat application:

```
from fastapi import FastAPI, WebSocket
from fastapi.responses import HTMLResponse
app = FastAPI()
html = """
<!DOCTYPE html>
<html>
    <head>
        <title>Chat</title>
    </head>
    <body>
        <h1>WebSocket Chat</h1>
        <form action="" onsubmit="sendMessage(event)">
            <input type="text" id="messageText" autocomplete="off"/>
            <button>Send</button>
        </form>
        <ul id='messages'>
        </ul>
        <script>
            var ws = new WebSocket("ws://localhost:8000/ws");
            ws.onmessage = function(event) {
                var messages = document.getElementById('messages')
                var message = document.createElement('li')
                var content = document.createTextNode(event.data)
                message.appendChild(content)
                messages.appendChild(message)
            };
            function sendMessage(event) {
                var input = document.getElementById("messageText")
                ws.send(input.value)
                input.value = ''
                event.preventDefault()
            }
        </script>
    </body>
</html>
"""
@app.get("/")
async def get():
    return HTMLResponse(html)
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message text was: {data}")
```

Comparing FastAPI to Other Web Frameworks

FastAPI vs Flask

Feature	FastAPI	Flask
Performance	Very fast due to Starlette	Fast, but slightly slower than FastAPI
Type checking	Built-in with Pydantic	Requires additional libraries
Async support	Native	Requires extensions
Documentation	Automatic Swagger UI	Manual or with extensions
Learning curve	Steeper, but rewarding	Gentle, very beginner-friendly

FastAPI vs Django

Feature	FastAPI	Django
Focus	API development	Full-stack web development
Performance	Very fast	Slower compared to FastAPI
Database ORM	Requires SQLAlchemy or other ORMs	Built-in ORM
Admin interface	Not included	Built-in admin interface
Async support	Native	Limited, improving in recent versions
API documentation	Automatic Swagger UI	Requires additional packages

When to choose FastAPI for your project

Choose FastAPI when:

1. You're building high-performance APIs
2. You want automatic API documentation
3. You need strong type checking
4. You're working on a microservices architecture
5. You want to leverage async programming in Python
6. You need WebSocket support

Conclusion

Recap of key points on how to use FastAPI

1. FastAPI provides a modern, fast, and easy-to-use framework for building APIs.
2. It leverages Python type hints for automatic validation and documentation.
3. FastAPI supports both synchronous and asynchronous programming.
4. It offers built-in support for OAuth2, JWT tokens, and other security features.
5. FastAPI integrates well with various databases and ORMs.
6. It provides automatic interactive API documentation with Swagger UI.
7. FastAPI is highly extensible and has a growing ecosystem of libraries and tools.

Future of FastAPI and web development

FastAPI is positioned to play a significant role in the future of web development, especially in the realm of high-performance, type-safe API development. As Python continues to evolve with better support for asynchronous programming and type hinting, FastAPI is well-positioned to leverage these advancements.

Next steps for mastering FastAPI

1. Dive deeper into advanced FastAPI features like dependency injection and background tasks.
2. Explore FastAPI's ecosystem and try out popular extensions.
3. Practice building real-world applications, from simple CRUD APIs to complex microservices.
4. Contribute to the FastAPI community by creating extensions or submitting pull requests to the main project.
5. Stay updated with the latest FastAPI releases and best practices.

Additional Resources

Official FastAPI documentation

The official FastAPI documentation is an excellent resource for learning and reference.

OpenTelemetry and Monitoring

For those interested in implementing robust monitoring and tracing in their FastAPI applications:

2. Get started with OpenTelemetry Python
3. Python Tracing API

These resources will help you set up comprehensive monitoring for your FastAPI applications, allowing you to gain deeper insights into your application's performance and behavior.

FastAPI community forums and support channels

1. FastAPI GitHub Discussions
2. Stack Overflow tag: fastapi

By mastering FastAPI, you'll be well-equipped to build high-performance, modern web applications and APIs. Happy coding!

Frequently Asked Questions

1. **Is FastAPI suitable for beginners?** While FastAPI has a steeper learning curve compared to some frameworks, it's still accessible to beginners who have a good grasp of Python. The extensive documentation and intuitive design make it a great choice for those looking to build modern APIs.
2. **How does FastAPI handle database operations?** FastAPI doesn't include a built-in ORM, but it works well with SQLAlchemy and other database libraries. You can easily integrate these tools to handle database operations in your FastAPI applications.
3. **Can FastAPI be used for full-stack web development?** While FastAPI excels at building APIs, it can be used for full-stack development when combined with a front-end framework like React or Vue.js. However, for complex full-stack applications, you might want to consider a framework like Django.
4. **How does FastAPI perform in production environments?** FastAPI is designed to be production-ready out of the box. Its high performance, combined with features like automatic API documentation and robust error handling, make it an excellent choice for production environments.
5. **Is it easy to deploy FastAPI applications?** Yes, FastAPI applications are relatively easy to deploy. They can be containerized using Docker and deployed to various cloud platforms or on-premises servers. The framework's compatibility with ASGI servers like Uvicorn makes deployment straightforward.

You may also be interested in: