

Merkmale

FastAPI Merkmale

FastAPI ermöglicht Ihnen folgendes:

Basiert auf offenen Standards

- OpenAPI für API-Erstellung, zusammen mit Deklarationen von Pfad Operationen, Parameter, Nachrichtenrumpf-Anfragen (englisch: body request), Sicherheit, etc.
- Automatische Dokumentation der Datenentitäten mit dem JSON Schema (OpenAPI basiert selber auf dem JSON Schema).
- Entworfen auf Grundlage dieser Standards nach einer sorgfältigen Studie, statt einer nachträglichen Schicht über diesen Standards.
- Dies ermöglicht automatische **Quellcode-Generierung auf Benutzerebene** in vielen Sprachen.

Automatische Dokumentation

Mit einer interaktiven API-Dokumentation und explorativen webbasierten Benutzerschnittstellen. Da FastAPI auf OpenAPI basiert, gibt es hierzu mehrere Optionen, wobei zwei standardmäßig vorhanden sind.

- Swagger UI, bietet interaktive Exploration: testen und rufen Sie ihre API direkt vom Webbrowser auf.
- Alternative API-Dokumentation mit ReDoc.

Nur modernes Python

Alles basiert auf **Python 3.6 Typ**-Deklarationen (dank Pydantic). Es muss keine neue Syntax gelernt werden, nur standardisiertes modernes Python.

Wenn Sie eine kurze, zweiminütige, Auffrischung in der Benutzung von Python Typ-Deklarationen benötigen (auch wenn Sie FastAPI nicht nutzen), schauen Sie sich diese kurze Einführung an (Englisch): [Python Types{.internal-link target=_blank}](#).

Sie schreiben Standard-Python mit Typ-Deklarationen:

```
from typing import List, Dict
from datetime import date
```

```
from pydantic import BaseModel
```

```
# Deklariere eine Variable als str
# und bekomme Editor-Unterstützung innerhalb der Funktion
def main(user_id: str):
```

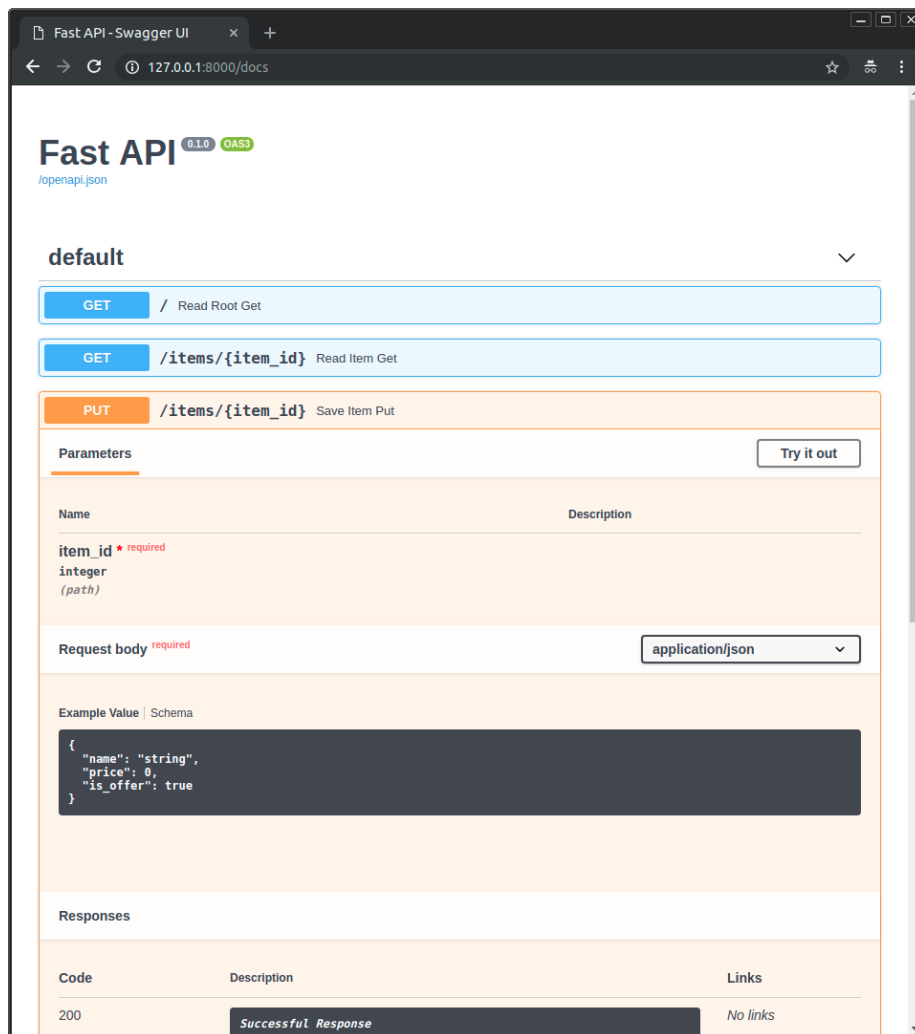


Figure 1: Swagger UI Interaktion

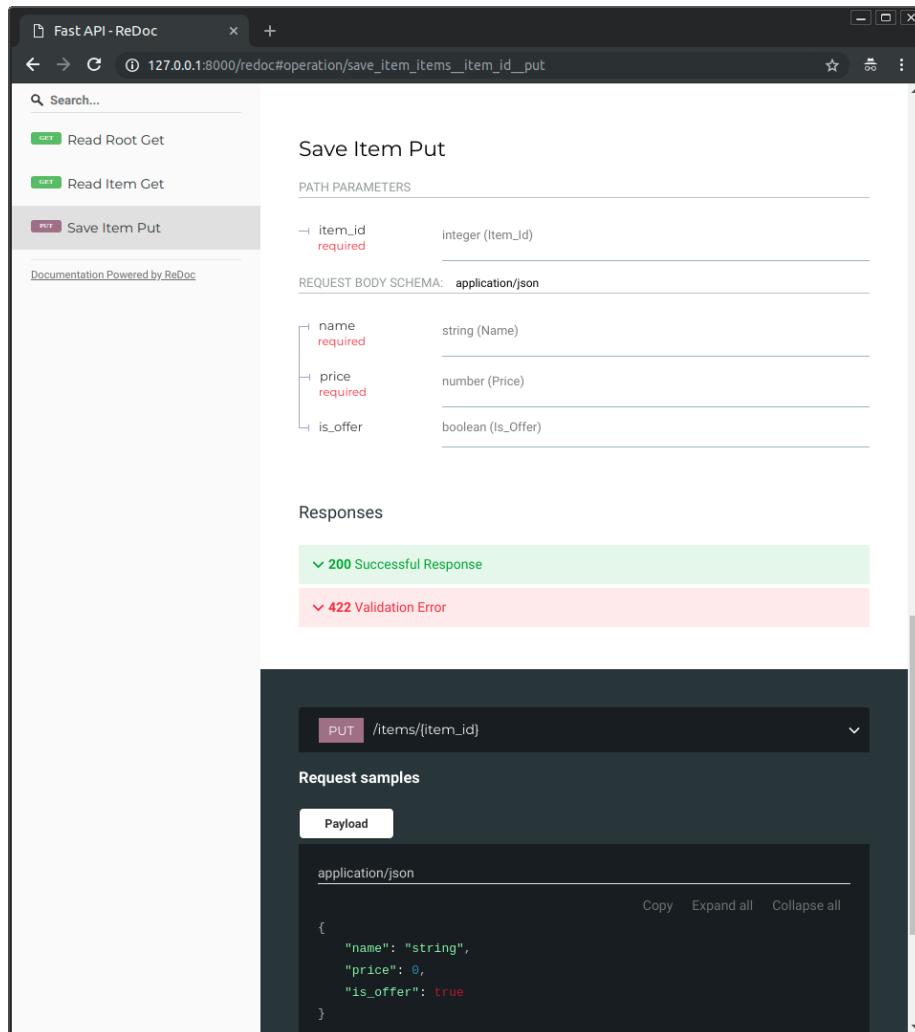


Figure 2: ReDoc

```
    return user_id
```

```
# Ein Pydantic model
class User(BaseModel):
    id: int
    name: str
    joined: date
```

Dies kann nun wie folgt benutzt werden:

```
my_user: User = User(id=3, name="John Doe", joined="2018-07-19")
```

```
second_user_data = {
    "id": 4,
    "name": "Mary",
    "joined": "2018-11-30",
}
```

```
my_second_user: User = User(**second_user_data)
```

!!! info ****second_user_data** bedeutet:

Übergebe die Schlüssel und die zugehörigen Werte des `second_user_data` Datenwörterbuches

Editor Unterstützung

FastAPI wurde so entworfen, dass es einfach und intuitiv zu benutzen ist; alle Entscheidungen wurden auf mehreren Editoren getestet (sogar vor der eigentlichen Implementierung), um so eine best mögliche Entwicklererfahrung zu gewährleisten.

In der letzten Python Entwickler Umfrage stellte sich heraus, dass die meist genutzte Funktion die "Autovervollständigung" ist.

Die gesamte Struktur von **FastAPI** soll dem gerecht werden. Autovervollständigung funktioniert überall.

Sie müssen selten in die Dokumentation schauen.

So kann ihr Editor Sie unterstützen:

- in Visual Studio Code:
- in PyCharm:

Sie bekommen Autovervollständigung an Stellen, an denen Sie dies vorher nicht für möglich gehalten hätten. Zum Beispiel der `price` Schlüssel aus einem JSON Datensatz (dieser könnte auch verschachtelt sein) aus einer Anfrage.

Hierdurch werden Sie nie wieder einen falschen Schlüsselnamen benutzen und sparen sich lästiges Suchen in der Dokumentation, um beispielsweise

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6
7 class Item(BaseModel):
8     name: str
9     price: float
10     is_offer: bool = None
11
12
13 @app.get("/")
14 def read_root():
15     return {"Hello": "World"}
16
17
18 @app.get("/items/{item_id}")
19 def read_item(item_id: int, q: str = None):
20     return {"item_id": item_id, "q": q}
21
22
23 @app.put("/items/{item_id}")
24 def save_item(item_id: int, item: Item):
25     return {"item_name": item.pr, "item_id": item_id}
26
```

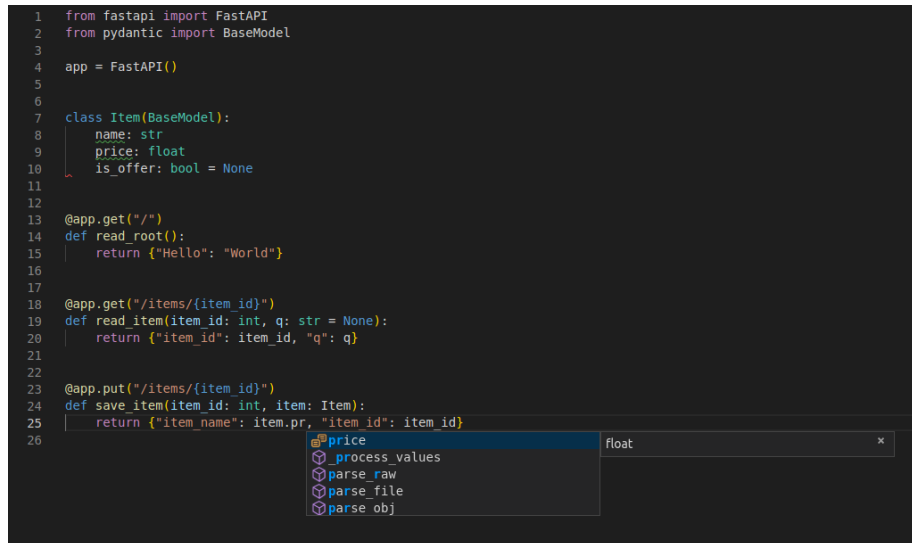


Figure 3: editor support

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6
7 class Item(BaseModel):
8     name: str
9     price: float
10     is_offer: bool = None
11
12
13 @app.get("/")
14 def read_root():
15     return {"Hello": "World"}
16
17
18 @app.get("/items/{item_id}")
19 def read_item(item_id: int, q: str = None):
20     return {"item_id": item_id, "q": q}
21
22
23 @app.put("/items/{item_id}")
24 def save_item(item_id: int, item: Item):
25     return {"item_name": item.pr, "item_id": item_id}
26
```

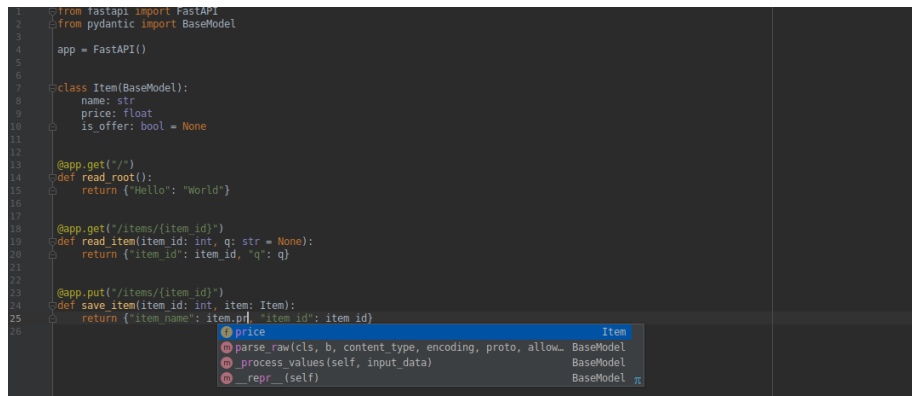


Figure 4: editor support

herauszufinden ob Sie `username` oder `user_name` als Schlüssel verwenden.

Kompakt

FastAPI nutzt für alles sensible **Standard-Einstellungen**, welche optional überall konfiguriert werden können. Alle Parameter können ganz genau an Ihre Bedürfnisse angepasst werden, sodass sie genau die API definieren können, die sie brachen.

Aber standardmäßig, “**funktioniert einfach**” alles.

Validierung

- Validierung für die meisten (oder alle?) Python **Datentypen**, hierzu gehören:
 - JSON Objekte (`dict`).
 - JSON Listen (`list`), die den Typ ihrer Elemente definieren.
 - Zeichenketten (`str`), mit definierter minimaler und maximaler Länge.
 - Zahlen (`int`, `float`) mit minimaler und maximaler Größe, usw.
- Validierung für ungewöhnliche Typen, wie:
 - URL.
 - Email.
 - UUID.
 - ... und andere.

Die gesamte Validierung übernimmt das etablierte und robuste **Pydantic**.

Sicherheit und Authentifizierung

Sicherheit und Authentifizierung integriert. Ohne einen Kompromiss aufgrund einer Datenbank oder den Datenentitäten.

Unterstützt alle von OpenAPI definierten Sicherheitsschemata, hierzu gehören:

- HTTP Basis Authentifizierung.
- **OAuth2** (auch mit **JWT Zugriffstokens**). Schauen Sie sich hierzu dieses Tutorial an: OAuth2 mit JWT.
- API Schlüssel in:
 - Kopfzeile (HTTP Header).
 - Anfrageparametern.
 - Cookies, etc.

Zusätzlich gibt es alle Sicherheitsfunktionen von Starlette (auch **session cookies**).

Alles wurde als wiederverwendbare Werkzeuge und Komponenten geschaffen, die einfach in ihre Systeme, Datenablagen, relationale und nicht-relationale Datenbanken, ..., integriert werden können.

Einbringen von Abhängigkeiten (meist: Dependency Injection)

FastAPI enthält ein extrem einfaches, aber extrem mächtiges Dependency Injection System.

- Selbst Abhängigkeiten können Abhängigkeiten haben, woraus eine Hierarchie oder ein **“Graph” von Abhängigkeiten** entsteht.
- **Automatische Umsetzung** durch FastAPI.
- Alle abhängigen Komponenten könnten Daten von Anfragen, **Erweiterungen der Pfadoperations**-Einschränkungen und der automatisierten Dokumentation benötigen.
- **Automatische Validierung** selbst für *Pfadoperationen*-Parameter, die in den Abhängigkeiten definiert wurden.
- Unterstützt komplexe Benutzerauthentifizierungssysteme, mit **Datenbankverbindungen**, usw.
- **Keine Kompromisse** bei Datenbanken, Eingabemasken, usw. Sondern einfache Integration von allen.

Unbegrenzte Erweiterungen

Oder mit anderen Worten, sie werden nicht benötigt. Importieren und nutzen Sie Quellcode nach Bedarf.

Jede Integration wurde so entworfen, dass sie einfach zu nutzen ist (mit Abhängigkeiten), sodass Sie eine Erweiterung für Ihre Anwendung mit nur zwei Zeilen an Quellcode implementieren können. Hierbei nutzen Sie die selbe Struktur und Syntax, wie bei Pfadoperationen.

Getestet

- 100% Testabdeckung.
- 100% Typen annotiert.
- Verwendet in Produktionsanwendungen.

Starlette’s Merkmale

FastAPI ist vollkommen kompatibel (und basiert auf) Starlette. Das bedeutet, auch ihr eigener Starlette Quellcode funktioniert.

FastAPI ist eigentlich eine Unterklasse von **Starlette**. Wenn sie also bereits Starlette kennen oder benutzen, können Sie das meiste Ihres Wissen direkt anwenden.

Mit **FastAPI** bekommen Sie viele von **Starlette**’s Funktionen (da FastAPI nur Starlette auf Steroiden ist):

- Stark beeindruckende Performanz. Es ist eines der schnellsten Python frameworks, auf Augenhöhe mit **NodeJS** und **Go**.
- **WebSocket**-Unterstützung.

- Hintergrundaufgaben im selben Prozess.
- Ereignisse für das Starten und Herunterfahren.
- Testclient basierend auf **requests**.
- **CORS**, GZip, statische Dateien, Antwortfluss.
- **Sitzungs und Cookie** Unterstützung.
- 100% Testabdeckung.
- 100% Typen annotiert.

Pydantic's Merkmale

FastAPI ist vollkommen kompatibel (und basiert auf) Pydantic. Das bedeutet, auch jeder zusätzliche Pydantic Quellcode funktioniert.

Verfügbar sind ebenso externe auf Pydantic basierende Bibliotheken, wie ORMs, ODMs für Datenbanken.

Daher können Sie in vielen Fällen das Objekt einer Anfrage **direkt zur Datenbank** schicken, weil alles automatisch validiert wird.

Das selbe gilt auch für die andere Richtung: Sie können jedes Objekt aus der Datenbank **direkt zum Klienten** schicken.

Mit **FastAPI** bekommen Sie alle Funktionen von **Pydantic** (da FastAPI für die gesamte Datenverarbeitung Pydantic nutzt):

- **Kein Kopfzerbrechen:**
 - Sie müssen keine neue Schemadefinitionssprache lernen.
 - Wenn Sie mit Python's Typisierung arbeiten können, können Sie auch mit Pydantic arbeiten.
- Gutes Zusammenspiel mit Ihrer/Ihrem **IDE/linter/Gehirn:**
 - Weil Datenstrukturen von Pydantic einfach nur Instanzen ihrer definierten Klassen sind, sollten Autovervollständigung, Linting, mypy und ihre Intuition einwandfrei funktionieren.
- **Schnell:**
 - In Vergleichen ist Pydantic schneller als jede andere getestete Bibliothek.
- Validierung von **komplexen Strukturen:**
 - Benutzung von hierarchischen Pydantic Schemata, Python `typing`'s `List` und `Dict`, etc.
 - Validierungen erlauben klare und einfache Datenschemadefinition, überprüft und dokumentiert als JSON Schema.
 - Sie können stark **verschachtelte JSON** Objekte haben und diese sind trotzdem validiert und annotiert.
- **Erweiterbar:**
 - Pydantic erlaubt die Definition von eigenen Datentypen oder sie können die Validierung mit einer `validator` dekorierten Methode erweitern..
- 100% Testabdeckung.