



Express, FastAPI e OpenAPI

Scaletta

Modulo 1: Express.js - Creazione di API in JavaScript

Setup ed Endpoints Base

- **Esercizio Guidato:** Configurazione di un progetto Express.js
 - Installazione delle dipendenze necessarie
 - Struttura base di un'applicazione Express
 - Creazione del primo endpoint GET



CRUD e Middleware

- **Esercizio Guidato:** Implementazione di un'API CRUD completa
 - Creazione di endpoint GET, POST, PUT, DELETE
 - Utilizzo dei middleware per la validazione e l'autenticazione
 - Gestione degli errori e risposte appropriate



Scaletta

Modulo 2: FastAPI - API in Python

Iniziare con FastAPI

- **Esercizio Guidato:** Setup di un progetto FastAPI
 - Installazione e configurazione iniziale
 - Creazione dei primi endpoint con type hints
 - Utilizzo della documentazione automatica

Modelli di Dati e Validazione

- **Esercizio Guidato:** Sviluppo di un'API con Pydantic
 - Definizione di modelli di dati con Pydantic
 - Implementazione di una API CRUD completa
 - Gestione di path e query parameters



Scaletta

Modulo 3: OpenAPI e Swagger

Documentazione Automatica

- **Esercizio Guidato:** Documentazione delle API con Swagger
 - Configurazione di Swagger UI in Express e FastAPI
 - Personalizzazione della documentazione
 - Test delle API attraverso l'interfaccia Swagger

Progetto Pratico Integrato

- **Esercizio Guidato:** Creazione di una API completa con documentazione
 - Sviluppo di un'API con endpoint multipli
 - Implementazione di autenticazione base
 - Documentazione completa con Swagger/OpenAPI



Modulo 1

Express.js

Express.js

Introduzione a Express.js

- Framework web minimale e flessibile per Node.js
- Ideale per la creazione di API RESTful e applicazioni web
- Vasto ecosistema di middleware e plugin
- Alta performance e basso overhead

Vantaggi di Express.js

- Sintassi intuitiva e concisa
- Middleware system potente e versatile
- Eccellente per creare API RESTful
- Grande comunità e documentazione robusta
- Compatibile con numerose librerie Node.js



Express.js

Prerequisiti

- Node.js e npm installati
- Editor di codice (VS Code consigliato)
- Postman o strumento simile per testare le API

Installazione e Setup

```
1 // Creare una directory per il progetto  
2 mkdir express-api-demo  
3 cd express-api-demo  
4  
5 // Inizializzare un progetto npm  
6 npm init -y  
7  
8 // Installare Express  
9 npm install express
```

Express.js

Struttura del Progetto Base

```
1 express-api-demo/
2   └── node_modules/
3   └── package.json
4   └── package-lock.json
5   └── server.js
```



Express.js

Prima Applicazione Express

```
1 // server.js
2 const express = require('express');
3 const app = express();
4 const PORT = 3000;
5
6 // Middleware per il parsing del JSON
7 app.use(express.json());
8
9 // Rotte base
10 app.get('/', (req, res) => {
11   res.json({ message: 'Benvenuto alla nostra API!' });
12 });
13
14 // Avvio del server
15 app.listen(PORT, () => {
16   console.log(`Server in esecuzione su http://localhost:${PORT}`);
17 });
```



Express.js

Esecuzione dell'Applicazione

```
1 # Avviare il server  
2 node server.js  
3  
4 # Output atteso:  
5 # Server in esecuzione su http://localhost:3000
```



— www.Duccio.ME —

Express.js

Anatomia di una Richiesta Express

```
1 app.get('/users/:id', (req, res) => {
2   // req.params - Parametri URL (es: id)
3   console.log(req.params.id);
4
5   // req.query - Query string (es: ?sort=asc)
6   console.log(req.query.sort);
7
8   // req.body - Corpo della richiesta (per POST/PUT)
9   console.log(req.body);
10
11  // Invio di una risposta
12  res.status(200).json({ user: 'Mario Rossi' });
13});
```



Express.js

Setup dei Dati per l'API CRUD

```
1 // Array di esempio come "database"
2 let users = [
3   { id: 1, name: 'Mario Rossi', email: 'mario@example.com' },
4   { id: 2, name: 'Luigi Verdi', email: 'luigi@example.com' },
5   { id: 3, name: 'Anna Bianchi', email: 'anna@example.com' }
6 ];
```



Express.js

Implementazione READ (GET)

```
1 // Ottenere tutti gli utenti
2 app.get('/api/users', (req, res) => {
3   res.json(users);
4 });
5
6 // Ottenere un utente specifico
7 app.get('/api/users/:id', (req, res) => {
8   const id = parseInt(req.params.id);
9   const user = users.find(user => user.id === id);
10
11  if (!user) {
12    return res.status(404).json({ error: 'Utente non trovato' });
13  }
14
15  res.json(user);
16});
```

Express.js

Implementazione CREATE (POST)

```
● ● ●

1 app.post('/api/users', (req, res) => {
2   const { name, email } = req.body;
3
4   // Validazione base
5   if (!name || !email) {
6     return res.status(400).json({ error: 'Nome ed email sono richiesti' });
7   }
8
9   // Creare nuovo utente
10  const newUser = {
11    id: users.length + 1,
12    name,
13    email
14  };
15
16  users.push(newUser);
17  res.status(201).json(newUser);
18 });


```

Express.js

Implementazione UPDATE (PUT)

```
 1 app.put('/api/users/:id', (req, res) => {
 2   const id = parseInt(req.params.id);
 3   const { name, email } = req.body;
 4
 5   // Trovare l'indice dell'utente
 6   const index = users.findIndex(user => user.id === id);
 7
 8   if (index === -1) {
 9     return res.status(404).json({ error: 'Utente non trovato' });
10   }
11
12   // Aggiornare l'utente
13   const updatedUser = {
14     id: users[index].id,
15     name: name || users[index].name,
16     email: email || users[index].email
17   };
18
19   users[index] = updatedUser;
20   res.json(updatedUser);
21 });
```



Express.js

Implementazione DELETE

```
1 app.delete('/api/users/:id', (req, res) => {
2   const id = parseInt(req.params.id);
3
4   // Trovare l'indice dell'utente
5   const index = users.findIndex(user => user.id === id);
6
7   if (index === -1) {
8     return res.status(404).json({ error: 'Utente non trovato' });
9   }
10
11  // Rimuovere l'utente
12  const deletedUser = users[index];
13  users = users.filter(user => user.id !== id);
14
15  res.json(deletedUser);
16});
```



Express.js

Middleware in Express

- Funzioni che hanno accesso a `req`, `res` e `next`
- Eseguite in ordine sequenziale
- Possono modificare oggetti `req/res`
- Possono terminare il ciclo request-response
- Possono chiamare il prossimo middleware con `next()`



Express.js

Middleware Personalizzato

```
1 // Middleware di logging
2 const logger = (req, res, next) => {
3   console.log(`${new Date().toISOString()} - ${req.method}
4   ${req.url}`);
5   next(); // Passa al prossimo middleware/route handler
6 }
7 // Applicazione globale del middleware
8 app.use(logger);
9
10 // Applicazione a route specifiche
11 app.use('/api/users', logger);
```



Express.js

Middleware di Autenticazione

```
1 // Middleware di autenticazione semplice
2 const authenticate = (req, res, next) => {
3   const apiKey = req.headers['x-api-key'];
4
5   if (!apiKey || apiKey !== 'chiave-segreta') {
6     return res.status(401).json({ error: 'Non autorizzato' });
7   }
8
9   next();
10 };
11
12 // Proteggi le route che richiedono autenticazione
13 app.use('/api/users', authenticate);
```



Express.js

Gestione degli Errori

```
1 // Middleware di gestione errori (4 parametri)
2 app.use((err, req, res, next) => {
3   console.error(err.stack);
4   res.status(500).json({
5     error: 'Errore interno del server',
6     message: process.env.NODE_ENV === 'development' ? err.message : undefined
7   });
8 });
9
10 // Lanciare un errore da una route
11 app.get('/error-demo', (req, res, next) => {
12   try {
13     // Operazione che può fallire
14     throw new Error('Qualcosa è andato storto');
15   } catch (error) {
16     next(error); // Passa l'errore al middleware di gestione errori
17   }
18 });
```



Express.js

Organizzazione delle Route



```
1 // routes/users.js
2 const express = require('express');
3 const router = express.Router();
4
5 router.get('/', (req, res) => {
6   // Restituisce tutti gli utenti
7 });
8
9 router.post('/', (req, res) => {
10  // Crea un nuovo utente
11 });
12
13 module.exports = router;
14
15 // server.js
16 const userRoutes = require('./routes/users');
17 app.use('/api/users', userRoutes);
```

Express.js

Controller Pattern

```
1 // controllers/userController.js
2 const getAllUsers = (req, res) => {
3   // Logica per ottenere tutti gli utenti
4 };
5
6 const getUserById = (req, res) => {
7   // Logica per ottenere un utente specifico
8 };
9
10 module.exports = { getAllUsers, getUserById };
11
12 // routes/users.js
13 const { getAllUsers, getUserById } = require('../controllers/userController');
14 router.get('/', getAllUsers);
15 router.get('/:id', getUserById);
```



Express.js

Test delle API

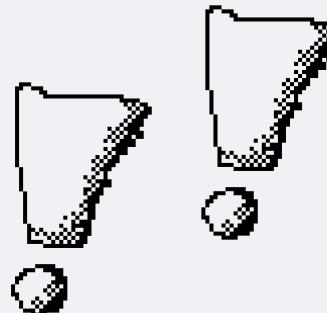
- Utilizzare Postman o strumenti simili
- Testare ogni endpoint CRUD
- Verificare codici di stato HTTP appropriati
- Testare validazioni e gestione errori
- Verificare l'autenticazione



Express.js

Risorse Aggiuntive

- Documentazione ufficiale Express: <https://expressjs.com/>
- Middleware popolari: cors, helmet, morgan
- Best practices per API REST
- Connessione a database (MongoDB, PostgreSQL)



— www.Duccio.ME —

Express.js

Best Practices per API Express

- Utilizzare il versioning delle API (/api/v1/users)
- Implementare rate limiting per la sicurezza
- Validare sempre gli input
- Implementare CORS correttamente
- Utilizzare variabili d'ambiente per configurazioni
- Fornire feedback utile con messaggi di errore appropriati

Prossimi Passi:

- Autenticazione avanzata (JWT)
- Integrazione con database
- Validazione avanzata con librerie come Joi o Yup
- Testing automatizzato
- Documentazione API (come vedremo nel Modulo 3)



Modulo 2

FastAPI

FastAPI

Introduzione a FastAPI

- Framework moderno e ad alte prestazioni per API in Python
- Basato su standard aperti come OpenAPI e JSON Schema
- Supporto nativo per la convalida dei dati tramite Pydantic
- Documentazione API automatica integrata
- Performance eccezionali grazie a Starlette e Uvicorn

Vantaggi di FastAPI

- Type hints di Python per validazione automatica dei dati
- Riduzione degli errori e del codice boilerplate
- Documentazione interattiva automatica
- Prestazioni vicine a quelle di Node.js e Go
- Supporto nativo per operazioni asincrone



— www.Duccio.ME —

FastAPI

Prerequisiti

- Python 3.7+ installato
- Conoscenza base di Python
- Familiarità con HTTP e principi REST
- Editor di codice (VS Code o PyCharm consigliati)
- Postman o strumento simile per testare le API

Installazione e Setup

```
1 # Creazione di un ambiente virtuale
2 python -m venv venv
3 source venv/bin/activate # Linux/Mac
4 venv\Scripts\activate # Windows
5
6 # Installazione di FastAPI e Uvicorn
7 pip install fastapi uvicorn
8
9 # Creare la struttura del progetto
10 mkdir fastapi-demo
11 cd fastapi-demo
```

— www.Duccio.ME —

FastAPI

Struttura del Progetto Base

```
1 fastapi-demo/
2   └── venv/
3   └── app/
4     ├── __init__.py
5     └── main.py
6   requirements.txt
```

FastAPI

Prima Applicazione FastAPI

```
1 # app/main.py
2 from fastapi import FastAPI
3
4 app = FastAPI(
5     title="Demo API",
6     description="API di esempio creata con FastAPI",
7     version="0.1.0"
8 )
9
10 @app.get("/")
11 def read_root():
12     return {"message": "Benvenuto alla nostra API FastAPI!"}
```



— www.Duccio.ME —

FastAPI

Avvio dell'Applicazione

```
1 # Avviare il server con uvicorn
2 uvicorn app.main:app --reload
3
4 # Output atteso:
5 # INFO: Uvicorn running on http://127.0.0.1:8000
```



— www.Duccio.ME —

FastAPI

Type Hints in Python

```
1 def add(a: int, b: int) -> int:  
2     return a + b  
3  
4 # Con type hint FastAPI può:  
5 # - Validare automaticamente i tipi  
6 # - Convertire i dati di input  
7 # - Generare documentazione OpenAPI  
8 # - Fornire auto-completamento nell'IDE
```



FastAPI

Endpoint con Type Hints

```
● ● ●  
1 from fastapi import FastAPI, Path, Query  
2  
3 app = FastAPI()  
4  
5 @app.get("/items/{item_id}")  
6 def read_item(  
7     item_id: int = Path(..., title="ID dell'oggetto", ge=1),  
8     q: str = Query(None, min_length=3, max_length=50)  
9 ):  
10    return {"item_id": item_id, "q": q}
```



FastAPI

Documentazione Automatica

- FastAPI genera automaticamente:
 - Specifiche OpenAPI (precedentemente Swagger)
 - Documentazione interattiva con Swagger UI
 - Documentazione alternativa con ReDoc
- Accessibile a `/docs` e `/redoc`
- Aggiornata in tempo reale con il codice



— www.Ducke.it —

FastAPI

Introduzione a Pydantic

- Libreria per la validazione dei dati basata su type hints
- Conversione automatica tra JSON e oggetti Python
- Validazione automatica di tipi e vincoli
- Integrazione nativa con FastAPI
- Schema JSON generato automaticamente



FastAPI

Modelli Pydantic di Base

```
1 from pydantic import BaseModel, Field, EmailStr
2 from typing import Optional
3
4 class User(BaseModel):
5     id: Optional[int] = None
6     name: str = Field(..., min_length=2, max_length=50)
7     email: EmailStr
8     is_active: bool = True
9
10    class Config:
11        schema_extra = {
12            "example": {
13                "name": "Mario Rossi",
14                "email": "mario@example.com",
15                "is_active": True
16            }
17        }
```



FastAPI

Utilizzare Modelli Pydantic con FastAPI

```
 1 from fastapi import FastAPI, HTTPException
 2 from pydantic import BaseModel
 3
 4 app = FastAPI()
 5
 6 class User(BaseModel):
 7     name: str
 8     email: str
 9
10 # Creazione con validazione automatica
11 @app.post("/users/")
12 def create_user(user: User):
13     return user
```



FastAPI

Setup dei Dati per l'API CRUD

```
 1 from typing import List, Dict
 2 from fastapi import FastAPI, HTTPException
 3 from pydantic import BaseModel, EmailStr
 4
 5 app = FastAPI()
 6
 7 # Modello Pydantic
 8 class User(BaseModel):
 9     id: int
10     name: str
11     email: EmailStr
12
13 class UserCreate(BaseModel):
14     name: str
15     email: EmailStr
16
17 # Database "finto"
18 users_db: Dict[int, User] = {
19     1: User(id=1, name="Mario Rossi", email="mario@example.com"),
20     2: User(id=2, name="Luigi Verdi", email="luigi@example.com"),
21     3: User(id=3, name="Anna Bianchi", email="anna@example.com")
22 }
```



FastAPI

Implementazione READ (GET)

```
1 @app.get("/users/", response_model=List[User])
2 def read_users(skip: int = 0, limit: int = 10):
3     return list(users_db.values())[skip : skip + limit]
4
5 @app.get("/users/{user_id}", response_model=User)
6 def read_user(user_id: int):
7     if user_id not in users_db:
8         raise HTTPException(status_code=404, detail="Utente non trovato")
9     return users_db[user_id]
```



FastAPI

Implementazione CREATE (POST)

```
1 @app.post("/users/", response_model=User, status_code=201)
2 def create_user(user: UserCreate):
3     # Genera un nuovo ID
4     user_id = max(users_db.keys(), default=0) + 1
5
6     # Crea l'utente nel DB
7     new_user = User(id=user_id, **user.dict())
8     users_db[user_id] = new_user
9
10    return new_user
```



FastAPI

Implementazione UPDATE (PUT)

```
1 class UserUpdate(BaseModel):
2     name: Optional[str] = None
3     email: Optional[EmailStr] = None
4
5 @app.put("/users/{user_id}", response_model=User)
6 def update_user(user_id: int, user: UserUpdate):
7     if user_id not in users_db:
8         raise HTTPException(status_code=404, detail="Utente non trovato")
9
10    # Ottiene l'utente corrente
11    current_user = users_db[user_id]
12
13    # Aggiorna i campi non nulli
14    user_data = user.dict(exclude_unset=True)
15    updated_user = current_user.copy(update=user_data)
16
17    # Salva l'utente aggiornato
18    users_db[user_id] = updated_user
19    return updated_user
```

FastAPI

Implementazione DELETE

```
1 @app.delete("/users/{user_id}", response_model=User)
2 def delete_user(user_id: int):
3     if user_id not in users_db:
4         raise HTTPException(status_code=404, detail="Utente non trovato")
5
6     # Rimuove e restituisce l'utente cancellato
7     deleted_user = users_db.pop(user_id)
8     return deleted_user
```



FastAPI

Dipendenze in FastAPI

```
1 from fastapi import Depends, HTTPException, status
2 from fastapi.security import APIKeyHeader
3
4 api_key_header = APIKeyHeader(name="X-API-Key")
5
6 def verify_api_key(api_key: str = Depends(api_key_header)):
7     if api_key != "chiave-segreta":
8         raise HTTPException(
9             status_code=status.HTTP_401_UNAUTHORIZED,
10            detail="API Key non valida"
11        )
12    return api_key
13
14 @app.get("/secure-endpoint/", dependencies=[Depends(verify_api_key)])
15 def secure_endpoint():
16     return {"message": "Accesso consentito all'endpoint protetto"}
```

FastAPI

Gestione Avanzata dei Parametri

```
 1 from enum import Enum
 2 from typing import List, Optional
 3 from fastapi import Query, Path
 4
 5 class SortOrder(str, Enum):
 6     asc = "asc"
 7     desc = "desc"
 8
 9 @app.get("/products/")
10 def list_products(
11     category: Optional[str] = Query(None, description="Filtra per categoria"),
12     tags: List[str] = Query([], description="Filtra per tag"),
13     sort: SortOrder = Query(SortOrder.asc, description="Ordine di ordinamento"),
14     page: int = Query(1, ge=1, description="Numero di pagina"),
15     size: int = Query(10, ge=1, le=100, description="Elementi per pagina")
16 ):
17     return {
18         "category": category,
19         "tags": tags,
20         "sort": sort,
21         "pagination": {"page": page, "size": size}
22     }
```



FastAPI

Router di FastAPI

```
1 # app/routes/users.py
2 from fastapi import APIRouter, HTTPException
3 from app.models.user import User, UserCreate
4
5 router = APIRouter(
6     prefix="/users",
7     tags=["users"],
8     responses={404: {"description": "Non trovato"}}
9 )
10
11 @router.get("/", response_model=list[User])
12 def read_users():
13     # ...
14
15 @router.post("/", response_model=User)
16 def create_user(user: UserCreate):
17     # ...
```



FastAPI

Background Tasks



```
1 from fastapi import BackgroundTasks
2
3 def notify_user(email: str, message: str):
4     # Simulazione invio email
5     print(f"Invio email a {email}: {message}")
6
7 @app.post("/orders/")
8 async def create_order(order: Order, background_tasks: BackgroundTasks):
9     # Crea l'ordine
10    order_id = save_order(order)
11
12    # Aggiunge un task in background
13    background_tasks.add_task(
14        notify_user,
15        email=order.user.email,
16        message=f"Il tuo ordine #{order_id} è stato creato"
17    )
18
19    return {"order_id": order_id}
```

FastAPI

API Asincrona

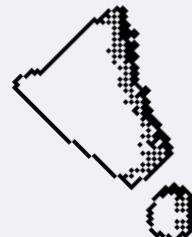
```
 1 import asyncio
 2 from fastapi import FastAPI
 3
 4 app = FastAPI()
 5
 6 async def get_user_data(user_id: int):
 7     # Simulazione di una operazione asincrona
 8     await asyncio.sleep(1)
 9     return {"id": user_id, "name": f"User {user_id}"}
10
11 @app.get("/users/{user_id}")
12 async def read_user(user_id: int):
13     user = await get_user_data(user_id)
14     return user
```



FastAPI

Esercizio Pratico

1. Creare un'API FastAPI completa con CRUD per un nuovo tipo di risorsa
2. Implementare validazione avanzata con Pydantic
3. Aggiungere autenticazione tramite dipendenze
4. Organizzare il codice usando router
5. Esplorare e personalizzare la documentazione automatica



FastAPI

Best Practices per FastAPI

- Organizzare il codice in moduli
- Utilizzare dipendenze per funzionalità riutilizzabili
- Sfruttare `response_model` per filtrare dati sensibili
- Validare sempre gli input con Pydantic
- Aggiungere documentazione descrittiva
- Utilizzare modelli separati per Create/Update/Response

Prossimi Passi

- Integrazione con database (SQLAlchemy, MongoDB)
- Autenticazione OAuth2 con JWT
- Testing automatizzato
- Integrazione con sistemi di cache
- Deployment in produzione



Modulo 3

OpenAPI e Swagger

OpenAPI e Swagger

Introduzione a OpenAPI e Swagger

- OpenAPI: Standard aperto per la definizione di API RESTful
- Swagger: Suite di strumenti per implementare lo standard OpenAPI
- Vantaggi:
 - Documentazione interattiva e aggiornata automaticamente
 - Generazione di client/server in diversi linguaggi
 - Test delle API direttamente dall'interfaccia
 - Contratto formale tra front-end e back-end

OpenAPI vs Swagger

- OpenAPI: Lo standard di specifica (attualmente v3.1)
- Swagger: Toolset che include:
 - Swagger UI: Interfaccia per visualizzare e testare API
 - Swagger Editor: Editor per scrivere specifiche OpenAPI
 - Swagger Codegen: Generazione di codice client/server
- Storia: Swagger è diventato OpenAPI nel 2016 (donato alla Linux Foundation)



OpenAPI e Swagger

OpenAPI - Struttura Base

```
● ● ●  
1 openapi: 3.0.0  
2 info:  
3   title: API di Esempio  
4   version: 1.0.0  
5   description: Una semplice API di esempio  
6 paths:  
7   /users:  
8     get:  
9       summary: Ottiene la lista degli utenti  
10      responses:  
11        '200':  
12          description: Operazione riuscita  
13          content:  
14            application/json:  
15              schema:  
16                type: array  
17                items:  
18                  $ref: '#/components/schemas/User'  
19 components:  
20   schemas:  
21     User:  
22       type: object  
23       properties:  
24         id:  
25           type: integer  
26         name:  
27           type: string
```



OpenAPI e Swagger

Swagger UI in Express.js

```
1 // Installazione
2 // npm install swagger-ui-express swagger-jsdoc
3
4 const express = require('express');
5 const swaggerJSDoc = require('swagger-jsdoc');
6 const swaggerUi = require('swagger-ui-express');
7
8 const app = express();
9
10 // Opzioni Swagger
11 const swaggerOptions = {
12   swaggerDefinition: {
13     openapi: '3.0.0',
14     info: {
15       title: 'API Express',
16       version: '1.0.0',
17       description: 'API Express documentata con Swagger'
18     },
19     servers: [
20       {
21         url: 'http://localhost:3000'
22       }
23     ],
24   },
25   apis: ['./routes/*.js'] // Percorso ai file con
26   // annotazioni
27 };
28 const swaggerDocs = swaggerJSDoc(swaggerOptions);
29 app.use('/api-docs', swaggerUi.serve,
  swaggerUi.setup(swaggerDocs));
```



OpenAPI e Swagger

Documentare Endpoint POST in Express (JSDoc)

```
1 /**
2  * @swagger
3  * /api/users:
4  *   post:
5  *     summary: Crea un nuovo utente
6  *     tags: [Users]
7  *     requestBody:
8  *       required: true
9  *       content:
10 *         application/json:
11 *           schema:
12 *             $ref: '#/components/schemas/CreateUser'
13 *     responses:
14 *       201:
15 *         description: Utente creato con successo
16 *         content:
17 *           application/json:
18 *             schema:
19 *               $ref: '#/components/schemas/User'
20 *       400:
21 *         description: Dati di input non validi
22 */
23 app.post('/api/users', (req, res) => {
24   // Implementazione
25 });
```



— www.Duccio.ME —

OpenAPI e Swagger

Swagger UI in FastAPI

```
1 # FastAPI integra già OpenAPI e Swagger UI
2 # Non è necessaria alcuna configurazione aggiuntiva!
3
4 from fastapi import FastAPI
5
6 app = FastAPI(
7     title="API di Esempio",
8     description="Una API di esempio con documentazione automatica",
9     version="1.0.0",
10    openapi_tags=[
11        {"name": "users", "description": "Operazioni sugli utenti"},
12        {"name": "items", "description": "Operazioni sugli articoli"}
13    ]
14)
15
16 # Accesso alla documentazione:
17 # - Swagger UI: http://localhost:8000/docs
18 # - ReDoc: http://localhost:8000/redoc
```



OpenAPI e Swagger

Personalizzazione in FastAPI

```
 1 from fastapi import FastAPI
 2
 3 app = FastAPI(
 4     title="API di Gestione Utenti",
 5     description="""
 6         API per la gestione degli utenti.
 7
 8         ## Funzionalità
 9
10         * Creazione utenti
11         * Lettura utenti
12         * Aggiornamento utenti
13         * Cancellazione utenti
14     """,
15     version="1.0.0",
16     terms_of_service="http://example.com/terms/",
17     contact={
18         "name": "Admin",
19         "url": "http://example.com/contact/",
20         "email": "admin@example.com",
21     },
22     license_info={
23         "name": "Apache 2.0",
24         "url": "https://www.apache.org/licenses/LICENSE-2.0.html",
25     },
26 )
```

OpenAPI e Swagger

Documentazione Modelli in FastAPI

```
 1 from typing import Optional, List
 2 from pydantic import BaseModel, Field
 3 from fastapi import FastAPI
 4
 5 app = FastAPI()
 6
 7 class User(BaseModel):
 8     id: Optional[int] = Field(None, description="ID univoco dell'utente")
 9     name: str = Field(...,
10                     description="Nome completo dell'utente",
11                     min_length=2,
12                     max_length=50,
13                     example="Mario Rossi")
14     email: str = Field(...,
15                     description="Indirizzo email valido",
16                     example="mario@example.com")
17
18     class Config:
19         schema_extra = {
20             "example": {
21                 "id": 1,
22                 "name": "Mario Rossi",
23                 "email": "mario@example.com"
24             }
25         }
```

— www.Duccio.ME —

OpenAPI e Swagger

Documentazione Endpoint in FastAPI

```
1 @app.get(
2     "/users/",
3     response_model=List[User],
4     summary="Ottiene la lista degli utenti",
5     description="Restituisce tutti gli utenti registrati nel sistema.",
6     response_description="Lista di utenti",
7     tags=["users"]
8 )
9 def read_users(
10     skip: int = Query(0, description="Numero di utenti da saltare"),
11     limit: int = Query(100, description="Numero massimo di utenti da restituire")
12 ):
13     return users_db[skip : skip + limit]
14
15 @app.get(
16     "/users/{user_id}",
17     response_model=User,
18     responses={
19         200: {"description": "Utente trovato"},
20         404: {"description": "Utente non trovato"}
21     }
22 )
23 def read_user(user_id: int):
24     # Implementazione
```



OpenAPI e Swagger

Raggruppamento con Tag

```
1 # In Express (con JSDoc)
2 /**
3  * @swagger
4  * tags:
5  *   - name: Users
6  *     description: Operazioni sugli utenti
7  *   - name: Products
8  *     description: Operazioni sui prodotti
9 */
10
11 # In FastAPI
12 @app.get("/users/", tags=["users"])
13 def read_users():
14     # ...
15
16 @app.get("/products/", tags=["products"])
17 def read_products():
18     # ...
```



OpenAPI e Swagger

Autenticazione in OpenAPI

```
 1 from fastapi import Depends, FastAPI, HTTPException, status
 2 from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
 3
 4 app = FastAPI()
 5
 6 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
 7
 8 @app.post("/token")
 9 async def login(form_data: OAuth2PasswordRequestForm = Depends()):
10     # Autenticazione...
11     return {"access_token": "token", "token_type": "bearer"}
12
13 @app.get(
14     "/users/me",
15     summary="Ottiene l'utente corrente",
16     responses={
17         200: {"model": User},
18         401: {"description": "Non autorizzato"}
19     }
20 )
21 async def read_users_me(token: str = Depends(oauth2_scheme)):
22     # Implementazione
```

OpenAPI e Swagger

Documentazione API in Express con File YAML

```
1 const express = require('express');
2 const YAML = require('yamljs');
3 const swaggerUi = require('swagger-ui-express');
4
5 const app = express();
6
7 // Carica il file di specifica OpenAPI
8 const swaggerDocument = YAML.load('./openapi.yaml');
9
10 // Serve Swagger UI
11 app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
```



```
1 # openapi.yaml
2 openapi: 3.0.0
3 info:
4   title: API Esempio
5   version: 1.0.0
6 paths:
7   /users:
8     get:
9       summary: Lista utenti
10      # ...
```

OpenAPI e Swagger

Best Practices

- Aggiornare sempre la documentazione con il codice
- Utilizzare descrizioni chiare e concise
- Fornire esempi realistici
- Documentare tutti i possibili errori
- Raggruppare gli endpoint con tag significativi
- Versione le API (v1, v2, ecc.)
- Implementare correttamente gli status HTTP
- Validare gli input in modo rigoroso

Risorse Aggiuntive

- [Documentazione OpenAPI](#)
- [Swagger UI](#)
- [FastAPI Docs](#)
- [Express + Swagger JSDoc](#)
- [OpenAPI Generator](#)
- [Swagger Editor Online](#)



— WWW.DUCKTALE.ME —