

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli



Fakultet informatike u Puli

(6) Razvojni okvir FastAPI

#6

RS

FastAPI je moderni web okvir za izgradnju API-ja koji se temelji na modernom Pythonu i tipovima (*type hints*). Radi se o relativnoj novom razvojnem okviru koji je prvi put objavljen 2018. godine te je od onda u aktivnom razvoju, a bilježi sve veću popularnost među Python programerima. Glavne funkcionalnosti FastAPI-ja uključuju automatsku generaciju dokumentacije, odličnu brzinu izvođenja koja je mjerljiva sa brzinom izvođenja razvojnih okvira temeljenih na Node-u i Go-u, kao i mogućnost korištenja tipova podatka za definiranje ulaznih i izlaznih očekivanih vrijednosti, validaciju podataka temeljenu na Pydantic modelima, automatsko generiranje dokumentacije itd. Konkretno u sklopu ovog kolegija, naučit ćemo kako razvijati s FastAPI-jem u svrhu implementacije robusnijih Python mikroservisa koje možete razvijati za vaše završne projekte.

Posljednje ažurirano: 23.12.2025.

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(6\) Razvojni okvir FastAPI](#)
 - [Sadržaj](#)
- [1. Uvod u FastAPI](#)
 - [1.1 Instalacija](#)
 - [1.2 Definiranje ruta](#)
- [2. Pydantic](#)
 - [2.1 Input/Output modeli](#)
 - [2.2 Zadaci za vježbu - Osnove definicije ruta i Pydantic modela](#)
 - [2.3 Složeniji Pydantic modeli](#)
 - [2.4 Nasljeđivanje Pydantic modela](#)

- [2.5 Zadaci za vježbu: Definicija složenijih Pydantic modela](#)
- [2.6 `Field` polje Pydantic modela](#)
- [3. Obrada grešaka \(eng. Error Handling\)](#)
 - [3.1 Validacija `route` i `query` parametara](#)
 - [3.2 Zadaci za vježbu: Obrada grešaka](#)
- [4. Strukturiranje poslužitelja i organizacija kôda](#)
 - [4.1 Dependency Injection \(DI\)](#)
 - [4.2 API Router](#)
- [5. WebSockets na FastAPI poslužitelju](#)
- [Zadatak za vježbu: Razvoj FastAPI mikroservisa za dohvaćanje podataka o filmovima](#)

1. Uvod u FastAPI

FastAPI je moderni web okvir za izgradu brzih i učinkovitih API-ja. Temelji se na Python anotacije zvane [type hints](#) kako bi omogućio lakšu validaciju dolaznih HTTP zahtjeva i odgovora što smanjuje greške tijekom razvoja i egzekucije programa te povećava sigurnost i olakšava održavanje kôda. Jedna od ključnih značajki FastAPI-ja je i **automatska generacija dokumentacije** putem alata Swagger UI, ali i mogućnost korištenja Pydantic modela za validaciju složenijih podatkovnih struktura.

Po svom dizajnu, FastAPI je *non-blocking*, što znači da je sposoban obrađivati više zahtjeva istovremeno (konkurentno) bez blokiranja izvođenja glavne dretve. Kao temelj koristi [Starlette](#) web okvir koji je lagan i brz asinkroni web okvir. Pozadinska tehnologija koja omogućuje ovakvo ponašanje je [ASGI](#), odnosno *Asynchronous Server Gateway Interface*. Radi se o relativnoj novoj konvenciji za razvoj web poslužitelja u Pythonu koja je zamijenila stariju WSGI konvenciju. Glavna mana je što **WSGI nije bio dizajniran za asinkrono izvođenje**.

Primjeri razvojnih okvira koji su temeljeni i prvenstveno razvijani na WSGI konvenciji uključuju [Django](#) i [Flask](#) (iako se danas mogu učiniti asinkronim uz određene ekstenzije).

Projekt iz kolegija Raspodijeljeni sustavi moguće je napraviti koristeći FastAPI kao temeljni web okvir za izgradnju mikroservisa. U nastavku slijedi upute za instalaciju FastAPI-ja te primjere kako ga kvalitetno koristiti u praksi. Ipak, ako vam je potreban *lightweight* okvir, bez puno dokumentiranja, validacije podataka i dodatnih FastAPI značajki, ili vam je pak potrebna veća kontrola nad event loop-om, možete nastaviti koristiti i [aiohttp.web](#) poslužitelj s prethodnih vježbi.



FastAPI logotip - <https://fastapi.tiangolo.com/>

1.1 Instalacija

FastAPI je odlično dokumentiran te postoji mnoštvo resursa na internetu koji vam mogu pomoći u njegovom učenju i razvoju. Preporučuje se korištenje FastAPI dokumentacije kao primarnog izvora informacija.

Dostupno na: <https://fastapi.tiangolo.com/learn/>

Za početak, potrebno je pripremiti **virtualno okruženje**. Mi ćemo ovdje koristiti `conda` modul:

```
→ conda create --name rs_fastapi python=3.13
→ conda activate rs_fastapi
```

Isto možete napraviti i kroz [Anaconda Navigator](#) grafičko sučelje.

Nakon što smo aktivirali virtualno okruženje, instaliramo FastAPI:

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

Napravite novi direktorij, npr. `rs_fastapi` i u njemu izradite datoteku `main.py`:

Uključujemo FastAPI modul i definiramo instancu aplikacije:

```
from fastapi import FastAPI  
  
app = FastAPI()
```

FastAPI koristi [Uvicorn](#) kao ASGI server. **Uvicorn** podržava HTTP/1.1 standard te WebSockets protokole. Dolazi instaliran s FastAPI-jem (ako ste ga instalirali sa `[standard]` zastavicom kao što je prikazano iznad). U tom slučaju, možete pokrenuti FastAPI poslužitelj koristeći sljedeću naredbu:

```
→ fastapi dev main.py
```

Naredba `fastapi dev` čita datoteku `main.py` i pokreće FastAPI poslužitelj koristeći *uvicorn*. U pravilu, FastAPI poslužitelj će biti pokrenut portu `8000`, ako je slobodan.

FastAPI servis je moguće pokrenuti i direktnim pozivanjem `uvicorn` modula:

```
→ uvicorn main:app --reload
```

gdje je:

- `main` ime datoteke bez ekstenzije
- `app` instanca FastAPI aplikacije
- `--reload` zastavica označava da se poslužitelj ponovno pokrene nakon svake promjene u kôdu (*hot reload*)

Ako želimo definirati port na kojem će se poslužitelj pokrenuti, možemo to učiniti dodavanjem zastavice `--port`:

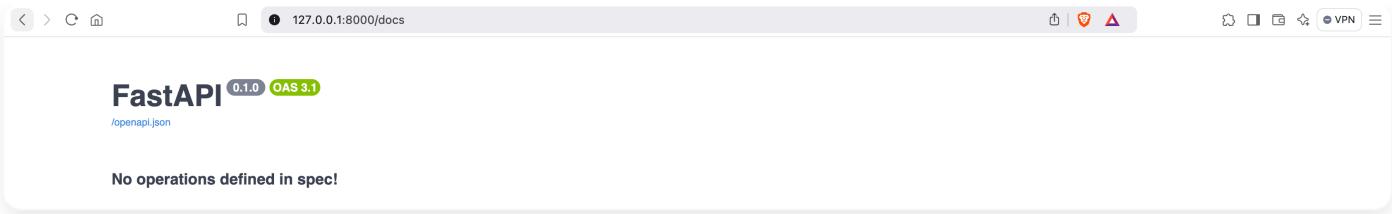
```
→ uvicorn main:app --reload --port 3000
```

Možete otvoriti web preglednik i posjetiti <http://localhost:8000> odnosno <http://localhost:8000/docs> kako biste vidjeli **generiranu dokumentaciju** ([Swagger UI](#)).

- kao alternativa, možete pristupiti i [ReDoc](#) dokumentaciji na <http://localhost:8000/redoc>.

Swagger UI i **Redoc** su alati za generiranje dokumentacije iz [OpenAPI specifikacije](#). FastAPI generira OpenAPI specifikaciju automatski na temelju definiranih ruta i Pydantic modela, a Swagger UI i ReDoc su alati koji tu specifikaciju prikazuju na korisnički prihvatljiv način - **u obliku web stranice s interaktivnim elementima**.

Ako pokušate otvoriti dokumentaciju, vidjet ćete da trenutno nema definiranih ruta.



Generirana FastAPI Swagger dokumentacija, dostupna na <http://localhost:8000/docs>

1.2 Definiranje ruta

FastAPI koristi **dekoratore** za definiranje ruta. U Pythonu, dekoratori (eng. *decorators*) su **funkcije ili klase koje proširuju funkcionalnost druge funkcije ili klase** bez promjene njene implementacije. Dekoratori omogućuju dodavanje funkcionalnosti na postojeće funkcije na čitljiviji način.

U kontekstu funkcionskog programiranja, **dekoratori su funkcije višeg reda** (eng. *higher-order functions*) koje rade sljedeće:

1. Primaju funkciju (ili klasu) kao argument
2. Mijenjaju ili proširuju njen ponašanje ili joj pridružuju dodatne metapodatke
3. Vraćaju novu (omotanu) funkciju ili klasu

Dekoratori se koriste prije definiranja funkcije kojoj želimo dodati funkcionalnost, **oznakom @ prije naziva dekoratora**.

Konkretno, FastAPI koristi dekoratore za definiranje ruta. Na primjer, sljedeći kôd definira jednostavnu GET rutu koja vraća JSON odgovor s porukom "Hello, world!"

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/") # dekorator za GET metodu na korijenskoj ruti
def read_root(): # funkcija koja se poziva kada se posjeti korijenska ruta
    return {"message": "Hello, world!"} # vraća JSON odgovor u tijelu HTTP odgovora
```

Ekvivalentan kôd koji smo pisali prilikom definiranja aiohttp rute izgledao bi ovako:

```
from aiohttp import web

def handle(request):
    return web.json_response({"message": "Hello, world!"})

app = web.Application()
app.router.add_get('/', handle)
```

Dakle, FastAPI koristi dekoratore za definiciju:

1. **Metode** HTTP za rute (GET, POST, PUT, PATCH, DELETE, itd.)
2. **Putanje** ruta (npr. /, /items/{item_id}, /users/{user_id}/items/{item_id}, itd.)

Handler funkciju koja se mora izvršiti pišemo neposredno ispod dekoratora.

U FastAPI-ju možemo koristiti sljedeće dekoratore za definiranje ruta:

- `@app.get(path)` - definira GET rutu
- `@app.post(path)` - definira POST rutu
- `@app.put(path)` - definira PUT rutu
- `@app.delete(path)` - definira DELETE rutu
- `@app.patch(path)` - definira PATCH rutu
- `@app.options(path)` - definira OPTIONS rutu
- `@app.head(path)` - definira HEAD rutu

1.2.1 Parametri ruta (eng. route parameters)

Parametre ruta definiramo na isti način kao i u `aiohttp` biblioteci, koristeći vitičaste zagrade `{}`. Na primjer, sljedeći kôd definira rutu koja očekuje `proizvod_id` kao parametar:

```
@app.get("/proizvodi/{proizvod_id}")
def get_proizvod(proizvod_id):
    return {"proizvod_id": proizvod_id}
```

HTTP zahtjev možete poslati koristeći bilo koji alat, međutim kad već radimo s FastAPI-jem, **dobra je praksa koristiti ugrađenu interaktivnu dokumentaciju** koju generira **Swagger** ili **ReDoc**.

- otvorite <http://localhost:8000/docs> u web pregledniku kako biste pristupili generiranoj dokumentaciji.

Ako je kôd ispravan, trebali biste vidjeti definiranu rutu u dokumentaciji: `GET /proizvodi/{proizvod_id}`
`Get Proizvod`

- gdje je `Get Proizvod` ustvari **naziv handler funkcije** koju smo definirali, a ruta `GET /proizvodi/{proizvod_id}` je **definirana dekoratorom**.

Odaberite rutu i kliknite na `Try it out` kako biste mogli poslati HTTP zahtjev.

- u polje `proizvod_id` unesite neku vrijednost i kliknite na `Execute`.
- ukoliko je sve ispravno, trebali biste vidjeti HTTP odgovor s definiranom vrijednosti `proizvod_id`.

The screenshot shows the FastAPI Swagger UI for the `/proizvodi/{proizvod_id}` endpoint. At the top, there's a blue header bar with the endpoint name. Below it, the "Parameters" section shows a required parameter `proizvod_id` with a type of `string`. There's also a "Try it out" button and a "Leo AI" button. The "Responses" section contains two entries: a successful response (status code 200) and a validation error response (status code 422). The 200 response details show a media type of `application/json` and an example value of `{"proizvod_id": "3"}`. The 422 response details show a media type of `application/json` and an example value of a JSON object with `"detail": [{ "loc": ["string", 0], "msg": "string", "type": "string" }]`.

Dodana ruta `GET /proizvodi/{proizvod_id}` u FastAPI Swagger dokumentaciji

Vidimo da generirana dokumentacija nudi **pregled svih podataka koje očekuje i vraća naša ruta**, odnosno sve podatke o HTTP zahtjevu koji se očekuje te o odgovoru koji će se vratiti.

This screenshot shows the detailed responses for the `/proizvodi/{proizvod_id}` endpoint. For status code 200, the response body is shown as `{"proizvod_id": "3"}` with a "Download" button. The response headers include `content-length: 19`, `content-type: application/json`, `date: Tue, 07 Jan 2025 21:19:00 GMT`, and `server: uvicorn`. For status code 422, the response body is a validation error message: `{ "detail": [{ "loc": ["string", 0], "msg": "string", "type": "string" }] }`.

U interaktivnoj dokumentaciji možemo vidjeti detaljan pregled HTTP odgovora koji vraća FastAPI poslužitelj

U Swagger interaktivnoj dokumentaciji možemo vidjeti sljedeće elemente HTTP odgovora:

- **Response body:** JSON odgovor koji je vraćen, u ovom slučaju: `{"proizvod_id": "3"}`

- **Response code:** HTTP statusni kôd koji je vraćen, u ovom slučaju: `200 OK`
- **Response headers:** zaglavla HTTP odgovora

Uz to možemo vidjeti i primjere ispravnog i neispravnog odgovora te definirane **Pydantic podatkovne modele** (`Schemas`), ako postoje. Više o tome u nastavku.

Primijetite sljedeće, FastAPI je automatski **parsirao parametar** `proizvod_id` iz URL-a i proslijedio ga kao argument funkciji `get_proizvod`.

```
@app.get("/proizvodi/{proizvod_id}")
def get_proizvod(proizvod_id):
    return {"proizvod_id": proizvod_id}
```

Ako pogledate odgovor, vidjet ćete da je vrijednost `proizvod_id` ustvari: `string: "proizvod_id": "3"`.

- **FastAPI automatski parsira parametre ruta u odgovarajući tip podatka**, ovisno o tipu koji je *hintan* u Python funkciji. Kako mi nismo definirali ništa, prepostavlja se da je tip `str`.

Python type hinting

Python *type hinting* je značajka koja omogućuje programerima da specificiraju očekivane tipove podataka za varijable, funkcionske argumente i povratne vrijednosti funkcija. Iako Python nije strogo tipiziran jezik, *type hinting* pomaže u poboljšanju čitljivosti kôda, olakšava otkrivanje grešaka tijekom razvoja te omogućuje alate za statičku analizu kôda da bolje razumiju namjere programera.

Ako bi htjeli naglasiti da je očekivani parametar `proizvod_id` tipa `int`, možemo to napraviti koristeći **Python type hinting**.

- to radimo na način da pišemo **tip podataka odvojen dvotočjem (:) nakon imena parametra**

Sintaksa:

```
@app.get("/ruta/{parametar}")
def funkcija(parametar: tip): # type hinting
    # tijelo funkcije
```

Primjer: Želimo/hintamo da je `proizvod_id` tipa `int`:

```
@app.get("/proizvodi/{proizvod_id}")
def get_proizvod(proizvod_id: int): # "hintamo" da je proizvod_id tipa int
    return {"proizvod_id": proizvod_id}
```

Pošaljite opet zahtjev u dokumentaciji i vidjet ćete da je sada vrijednost `proizvod_id` tipa `int`.

type hinting u FastAPI-ju **nije samo dekorativna značajka**, već ima i praktičnu svrhu na način da odrađuje **automatsko parsiranje i validaciju podataka**. To je zato što FastAPI direktno implementira *type-hinting*.

Međutim, ako se vratimo na dokumentaciju i poslajemo sljedeći zahtjev: `GET /proizvodi/Marko`. Vidjet ćemo da poslužitelj baca grešku jer je očekivani tip podataka `int`, a mi smo poslali `str`.

The screenshot shows a FastAPI POST request interface. In the top left, there's a field labeled "proizvod_id" with a red asterisk and the word "required" next to it. Below it is a placeholder "(path)" and a text input field containing "Marko". At the bottom of this section is a "Clear" button. To the right of the input field is a blue "Execute" button. Below these buttons is a "Responses" section. Under "Responses", there's a "Curl" block containing a curl command to make a GET request to "http://localhost:8000/proizvodi/Marko" with an "accept: application/json" header. Below that is a "Request URL" block showing "http://localhost:8000/proizvodi/Marko". Under "Server response", there are two tabs: "Code" and "Details". The "Code" tab shows the status code 422 and the error message "Error: Unprocessable Entity". The "Details" tab shows the JSON response body:

```
{ "detail": [ { "type": "int_parsing", "loc": [ "path", "proizvod_id" ], "msg": "Input should be a valid integer, unable to parse string as an integer", "input": "Marko" } ] }
```

Below the response body is a "Response headers" block showing the following information:

```
content-length: 158  
content-type: application/json  
date: Tue, 07 Jan 2025 21:39:54 GMT  
server: uvicorn
```

At the bottom of the "Responses" section are "Download" and "Copy" buttons.

FastAPI automatski baca grešku ako se očekivani tip podataka ne podudara s onim što je poslano

Dobili smo detaljnu grešku, sa statusnim kôdom `422 Unprocessable Entity` i složenim JSON objektom HTTP odgovora koji opisuje grešku:

```
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": [ "path", "proizvod_id" ],
      "msg": "Input should be a valid integer, unable to parse string as an integer",
      "input": "Marko"
    }
  ]
}
```

FastAPI poslužitelj automatski obrađuje ovu grešku za nas (**ne moramo ih obrađivati ručno kao do sada**) i sadrži sve potrebne informacije o grešci, uključujući tip greške, lokaciju greške, poruku greške i ulazne podatke koji su uzrokovali grešku.

Primitivni tipovi koji podržavaju type hinting

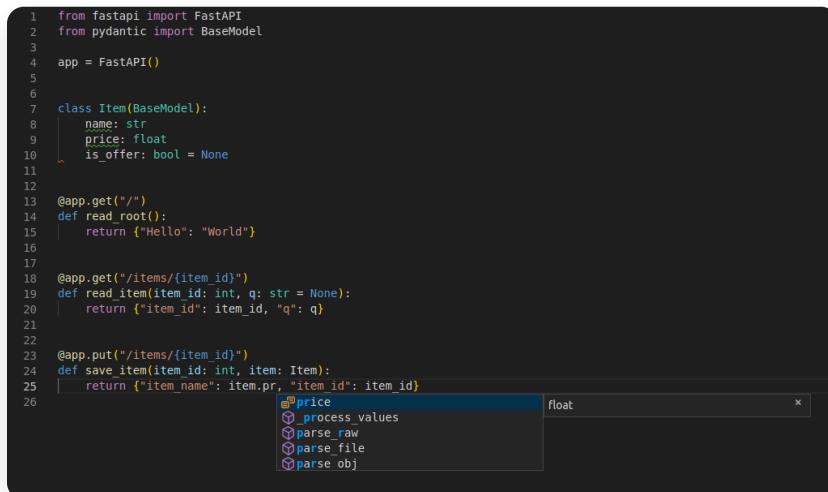
- `str` - string
- `int` - cijeli broj
- `float` - decimalni broj
- `bool` - logička vrijednost
- `bytes` - niz bajtova
- `None` - nema vrijednosti

Kolekcije koje podržavaju type hinting

- `list` - lista
- `tuple` - uređeni par
- `set` - skup
- `frozenset` - nepromjenjivi skup
- `dict` - rječnik

Više o tipovima podataka u poglavlju [2. Pydantic](#).

Zapamti: FastAPI razvojni okvir je baziran na modernom Pythonu koji koristi *type hinting* za parsiranje i validaciju podataka. Dodatna prednost kod korištenja *type hintinga* je i podrška za *autocomplete* koja je integrirana sa većinom modernih IDE-a (npr. VSCode, PyCharm, itd.), što olakšava razvoj i smanjuje mogućnost grešaka.



Python *type hinting* podrška u VSCode IDE-u omogućava brže pisanje kôda i manje grešaka zahvaljujući automatskom dovršavanju i provjeri tipova podataka tijekom pisanja kôda.

Primjer: Nadogradit ćemo postojeću aplikaciju tako da pronađe odgovarajući proizvod u *in-memory* listi proizvoda te omogućiti korisniku da ga **dohvati prema imenu**. Također, dodat ćemo rutu za **dodavanje novog proizvoda** u listu.

Definirajmo nekoliko proizvoda u listi. Svaki proizvod sadrži ključeve `id`, `naziv`, `boja` i `cijena`:

```
proizvodi = [
    {"id": 1, "naziv": "majica", "boja": "plava", "cijena": 50},
    {"id": 2, "naziv": "hlače", "boja": "crna", "cijena": 100},
    {"id": 3, "naziv": "tenisice", "boja": "bijela", "cijena": 150},
    {"id": 4, "naziv": "kapa", "boja": "smeđa", "cijena": 20}
]
```

1. Definirat ćemo prvo rutu koja će omogućiti dohvatanje svih proizvoda:

```
@app.get("/proizvodi")
def get_proizvodi(): # funkcija ne prima argumente jer nemamo parametre
    return proizvodi
```

2. **Zatim ćemo definirati rutu koja će omogućiti dohvaćanje proizvoda prema imenu**, dakle:

```
/proizvodi/{naziv}:
```

Možemo koristiti ugrađenu Python funkciju `next()` koja će nam omogućiti pronađak **prvog proizvoda koji zadovoljava uvjet**. Sintaksa nalikuje na *list comprehension*, ali s dodatnim parametrom `default` koji se vraća ako se ne pronađe nijedan element koji zadovoljava uvjet.

- nakon pronađaska prvog elementa koji zadovoljava uvjet, `next()` vraća taj element i **iteriranje se zaustavlja**

Sintaksa:

```
next((expression for iterator in iterable if condition), default)
```

- `expression` - izraz koji se evaluira
- `iterator` - iterator koji prolazi kroz elemente
- `iterable` - kolekcija elemenata (lista, rječnik, skup, tuple, itd.)
- `condition` - uvjet koji mora biti zadovoljen
- `default` - vrijednost koja se vraća ako se ne pronađe nijedan element koji zadovoljava uvjet

Definirajmo rutu za dohvaćanje proizvoda prema imenu:

```
@app.get("/proizvodi/{naziv}") # route parametar "naziv"
def get_proizvod_by_name(naziv: str): # očekujemo string kao naziv proizvoda (ako ne naglasimo se podrazumijeva da je str)
    # pronađimo proizvod gdje se njegov naziv poklapa s nazivom iz parametra rute "naziv"
    pronadjeni_proizvod = next((proizvod for proizvod in proizvodi if proizvod["naziv"] == naziv), None) # None ako se ne pronađe proizvod
    return pronadjeni_proizvod
```

Tijelo HTTP zahtjeva

3. **Dodavanje proizvoda u listu proizvoda** možemo odraditi definicijom POST zahtjeva na `/proizvodi`:

Tijelo HTTP zahtjeva možemo definirati kao argument funkcije te *hintamo* da je tijelo zahtjeva tipa `dict` (rječnik) jer očekujemo JSON objekt.

Ne navodimo tijelo zahtjeva u dekoratoru (kao što je slučaj kod parametara rute), već ga očekujemo kao argument funkcije *hintanjem* `dict` ili Pydantic modela (više u nastavku).

```

@app.post("/proizvodi") # ne definiramo tijelo zahtjeva u dekoratoru
def add_proizvod(proizvod: dict): # očekujemo JSON objekt kao proizvod u tijelu zahtjeva
    pa hintamo rječnik (dict)
    proizvod["id"] = len(proizvodi) + 1 # dodajemo novi ID (broj proizvoda + 1)
    proizvodi.append(proizvod) # dodajemo proizvod u listu
    return proizvod

```

Otvorite dokumentaciju, uočit ćete sve tri definirane rute (`GET /proizvodi`, `GET /proizvodi/{naziv}`, `POST /proizvodi`). Isprobajte svaku od definiranih ruta.

FastAPI 0.1.0 OAS 3.1

`/openapi.json`

default

- `GET /proizvodi` Get Proizvodi
- `POST /proizvodi` Add Proizvod
- `GET /proizvodi/{naziv}` Get Proizvod By Name

Schemas

- `HTTPValidationError` > Expand all `object`
- `ValidationError` > Expand all `object`

Generirana dokumentacija s tri definirane rute (`GET /proizvodi`, `GET /proizvodi/{naziv}`, `POST /proizvodi`)

Ako otvorite sučelje za rutu `POST /proizvodi`, vidjet ćete da vam se nudi opcija za unos JSON tijela zahtjeva, budući da nismo naveli parametre rute u dekoratoru:

`POST /proizvodi Add Proizvod`

Parameters

No parameters

Request body `required`

application/json

{ }

Execute

Sučelje za unos tijela zahtjeva u dokumentaciji za rutu `POST /proizvodi`

```
{ "naziv": "šal", "boja": "plava", "cijena": 30 }
```

HTTP Odgovor će biti novi proizvod s automatski dodijeljenim ID-em:

```
{
    "naziv": "šal",
    "boja": "plava",
    "cijena": 30,
    "id": 5 // automatski dodijeljen ID
}
```

1.2.2 Parametri upita (eng. *query parameters*)

query parametri su parametri koji se šalju u URL-u HTTP zahtjeva, nakon znaka `?`. Na primjer, u URL-u `/proizvodi?boja=plava` *query* parametar je `boja` s vrijednošću `plava`. Uobičajeno je koristiti *query* parametre za filtriranje podataka, sortiranje, paginaciju i slične operacije.

Na FastAPI poslužitelju, **query parametre** možemo definirati koristeći Python *type hinting* na način da ih dodamo kao argumente funkcije, **bez dodavanja u URL putanju kroz dekorator**.

- FastAPI će takve argumente automatski interpretirati kao *query* parametre.

Primjer definiranja rute koja očekuje *query* parametar `boja`:

```
@app.get("/proizvodi") # u FastAPI-ju ne navodimo _query_ parametre u URL putanji
def get_proizvodi_by_query_(boja: str): # očekujemo _query_ parametar "boja"
    pronadeni_proizvodi = [proizvod for proizvod in proizvodi if proizvod["boja"] == boja] #
    koristimo list comprehension, a ne next() jer možemo imati više proizvoda s istom bojom
    return pronadeni_proizvodi
```

Možemo definirati i više *query* parametara:

```
@app.get("/proizvodi") # u FastAPI-ju ne navodimo _query_ parametre u URL putanji
def get_proizvodi_by_query_(boja: str, max_cijena: int): # očekujemo _query_ parametre
    "boja" i "max_cijena"
    # koristimo list comprehension, a ne next() jer možemo imati više proizvoda s istom
    bojom i cijenom manjom ili jednako od max_cijena
    pronadeni_proizvodi = [proizvod for proizvod in proizvodi if proizvod["boja"] == boja
    and proizvod["cijena"] <= max_cijena]
    return pronadeni_proizvodi
```

Identični procesi primjenjuju se i za *query* parametre kao i za *route* parametre kada koristimo *type hinting*:

- automatsko parsiranje podataka
- automatska validacija podataka
- automatsko generiranje dokumentacije

query parametrima možemo dodjeljivati i **zadane (defaultne) vrijednosti**:

```

@app.get("/proizvodi") # u FastAPI-ju ne navodimo _query_ parametre u URL putanji
def get_proizvodi_by_query_(boja: str = None, max_cijena: int = 100): # očekujemo _query_
parametre "boja" i "max_cijena", ali su im zadane vrijednosti None odnosno 100
pronadeni_proizvodi = [proizvod for proizvod in proizvodi if (boja is None or
proizvod[ "boja" ] == boja) and (max_cijena is None or proizvod[ "cijena" ] <= max_cijena)]
return pronadeni_proizvodi

```

Svi navedeni *query* parametri na ovaj način postaju **opcionalni**. Ako ih ne navedemo u URL-u, poslužitelj će ih automatski postaviti na `None`.

Vidimo da se FastAPI ponaša vrlo slično kao i `aiohttp` biblioteka, ali s mnogo više **automatskih značajki** koje olakšavaju razvoj i održavanje kôda. Dodatno, tu je dokumentacija koja nam već u ovoj fazi pomaže u razvoju i testiranju API-ja. Konkretno, za primjer rute iznad možemo u dokumentaciji odmah vidjeti:

- koji se *query* parametri očekuju (`boja`, `max_cijena`)
- koji su tipovi podataka očekivani (`string`, `integer`)
- koje su defaultne vrijednosti (`None`, `100`)

Code	Description	Links
200	Successful Response Media type: application/json Example Value: string	No links
422	Validation Error Media type: application/json	No links

Dokumentacija za rutu s *query* parametrima `boja` i `max_cijena`

1.2.3 Kako razlikovati *route* i *query* parametre te tijelo zahtjeva?

U FastAPI-ju može biti zbumujuće razlikovati *route* parametre, *query* parametre i tijelo zahtjeva budući da ne navodimo eksplisitno "što je što" već se oslanjam na *type hinting*. **Evo kratkog pregleda:**

- **Route parametri - obavezno se navode u URL putanji** (dekoratoru), npr.

```
@app.get("/proizvodi/{proizvod_id}").
```

- moraju imati odgovarajući **ekvivalent u deklaraciji funkcije** i to istog naziva, npr. `def get_proizvod(proizvod_id: int):`

- sada se može poslati sljedeći zahtjev: `GET /proizvodi/3`.
 - mogu sadržavati *type hinting*, inače se podrazumijeva `str`.
 - FastAPI automatski parsira i validira podatke iz parametra rute.
- **query parametri - ne navode se u URL putanji (dekoratoru):** `@app.get("/proizvodi")`
 - deklariraju se kao argumenti funkcije, npr. `def get_proizvodi_by_query_(boja: str):`.
 - sada se može poslati sljedeći zahtjev: `GET /proizvodi?boja=plava`.
 - *query* parametri ako su navedeni bez zadanih vrijednosti postaju obavezni.
 - Zadane vrijednosti možemo postaviti dodjeljivanjem vrijednosti u deklaraciji funkcije, npr. `def get_proizvodi_by_query_(boja: str = "plava")`.
 - FastAPI automatski parsira i validira podatke iz *query* parametara.
 - **Tijelo zahtjeva - ne navode se u URL putanji (dekoratoru)**, npr. `@app.post("/proizvodi")`.
 - deklariraju se kao argumenti funkcije hintanjem `dict` ili Pydantic modela, npr. `def add_proizvod(proizvod: dict):`
 - FastAPI automatski parsira i validira podatke iz tijela zahtjeva.
 - u nastavku ćemo vidjeti kako koristiti Pydantic modele za hintanje tijela zahtjeva.

Moguće je kombinirati sva 3 pristupa.

Primjerice: Recimo da želimo definirati rutu koja će omogućiti ažuriranje podataka o proizvodu iz skladišta gdje su proizvodi podijeljeni u kategorije.

Podaci su definirani na sljedeći način:

- `id_skladiste` - cijeli broj (*route* parametar)
- `kategorija` - string (*query* parametar)
- `proizvod` - proizvod koji ažuriramo (tijelo zahtjeva)

Odabrali bi metodu PATCH budući da djelomično ažuriramo resurse (proizvode) u skladištu.

1. Definirat ćemo dekorator za PATCH metodu na `/skladiste`:

```
@app.patch("/skladiste")
```

2. Prva filtracija odnosi se na dohvat određenog skladišta prema `id_skladiste`:

- nadograđujemo dekorator
- dodajemo ekvivalentni argument funkcije

```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(id_skladiste: int):
```

3. Druga filtracija odnosi se na dohvat proizvoda u određenoj kategoriji:

- dodajemo *query* parametar u deklaraciji funkcije, **ali ne u dekoratoru**

```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(id_skladiste: int, kategorija: str):
```

4. Možemo postaviti zadalu vrijednost za *query* parametar:

- npr. `kategorija: str = "gradevinski_materijal"`

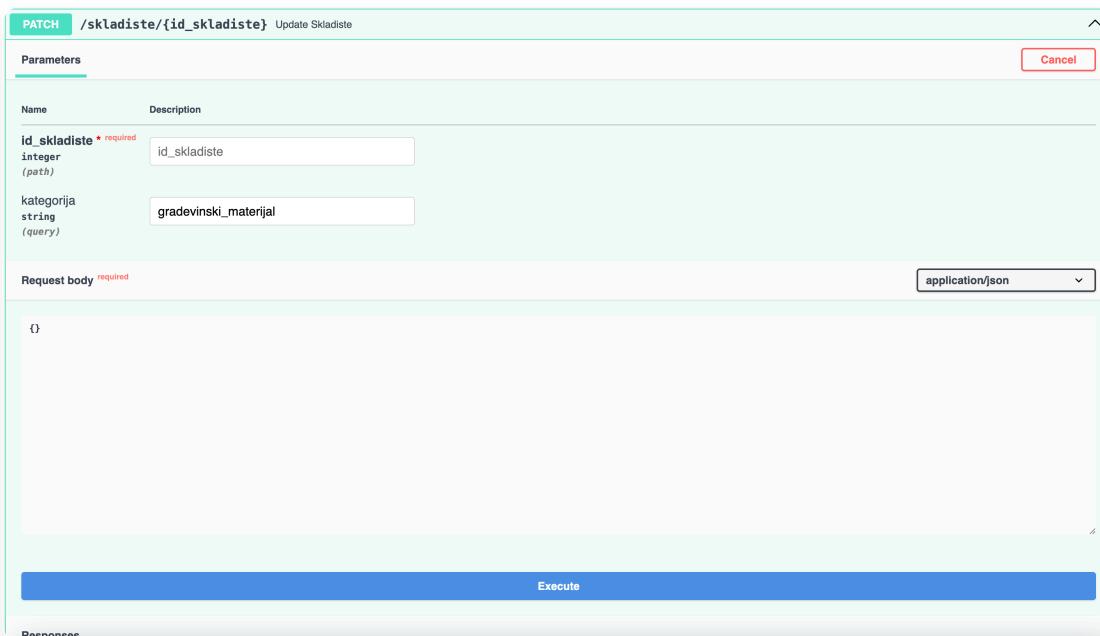
```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(id_skladiste: int, kategorija: str = "gradevinski_materijal"):
```

5. Na kraju, dodajemo tijelo zahtjeva kao argument funkcije:

- hintmo da je tijelo zahtjeva tipa `dict`
- dodajemo na početak funkcije jer vrijede ista pravila kao i za zadane argumente običnih Python funkcija (zadani argumenti dolaze na kraju)

```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(proizvod: dict, id_skladiste: int, kategorija: str =
"gradevinski_materijal"):
```

Provjerimo kako je dokumentirana definirana ruta u FastAPI dokumentaciji.



Dokumentacija za rutu `PATCH /skladiste/{id_skladiste}` s definiranim *route* parametrom, *query* parametrom i tijelom zahtjeva

U nastavku ćemo vidjeti kako validirati tijelo zahtjeva koristeći **Pydantic modele**.

2. Pydantic

Pydantic je najrasprostranjenija Python biblioteka za **validaciju podataka** koja se bazira na *type hintingu* za definiranje očekivanih tipova podataka te automatski vrši validaciju podataka prema tim definicijama. Pydantic je posebno koristan u FastAPI-ju jer se može koristiti za definiranje **modela podataka** koji se koriste za validaciju dolaznih i odlaznih podataka odnosno **tijela HTTP zahtjeva i odgovora**.

Napomena! Kada govorimo o **modelima** u kontekstu FastAPI-ja, mislimo na **Pydantic modele** koji se koriste za definiranje složenijih struktura podataka koje želimo "hintati" u različitim dijelovima aplikacije. Model u ovom kontekstu **ne predstavlja matematički model** koji se odnosi na statističke analize, model strojnog učenja ili sl. već predstavlja složenu strukturu podataka koja se koristi za validaciju, serijalizaciju te deserijalizaciju podataka te osigurava da su podaci u skladu s očekivanim tipovima. U nastavku ove skripte koristit će se termin "model" za danu definiciju.



Dokumentacija dostupna na: <https://docs.pydantic.dev/latest/>

Jedna od glavnih prednosti Pydantic-a je njegovo ponašanje u IDE razvojnim okruženjima kao što su **VS Code** ili **PyCharm**. IDE-ovi koji podržavaju Python *type hinting* automatski će prepoznati Pydantic modele i pružiti korisne informacije o očekivanim tipovima podataka, što olakšava razvoj i održavanje kôda.

Pydantic klase definiramo nasljeđivanjem `pydantic.BaseModel` klase.

Uobičajeno je Pydantic klase odvojiti o `main.py` datoteke kako bi kôd bio bolje organiziran te kako bi klase mogli koristiti u više datoteka.

- **Pydantic modele ćemo definirati u zasebnoj datoteci**, npr. `models.py` ili `schemas.py`.

Napravite novu datoteku `models.py`:

Definirajte klasu `Proizvod` koja će predstavljati model podataka za proizvod koji smo prije *hintali* kao rječnik.

- Prvo uključujemo `BaseModel` **kojeg nasljeđuju sve Pydantic klase**:

```
# models.py

from pydantic import BaseModel
```

Pišemo definiciju klase koja nasljeđuje `BaseModel`:

```
# models.py

class Proizvod(BaseModel):
    pass
```

Unutar definicije klase navodimo, koristeći *type-hinting*, atribute koje očekujemo za proizvod, to su:

- `id` - cijeli broj (`int`)
- `naziv` - string (`str`)
- `boja` - string (`str`)
- `cijena` - decimalni broj (`float`)

```
# models.py

class Proizvod(BaseModel):
    id: int
    naziv: str
    boja: str
    cijena: float
```

Uključujemo ovu klasu u `main.py` datoteku:

```
from fastapi import FastAPI

from models import Proizvod # uključujemo Pydantic model koji smo definirali
```

Međutim, kojoj je svrha ovog modela? U kojoj definiciji rute ćemo ga koristiti? **To ovdje nije jasno naglašeno.**

Primjerice: Kod POST rute za dodavanje proizvoda u listu, do sad smo koristili `dict` kao tip podataka za proizvod koristeći *type hinting*.

```
@app.post("/proizvodi")
def add_proizvod(proizvod: dict):
    proizvod["id"] = len(proizvodi) + 1
    proizvodi.append(proizvod)
    return proizvod
```

Ipak, to nije najbolji pristup budući da korisnik može poslati bilo kakav JSON objekt, odnosno objekt s proizvoljnim ključevima. Želimo ograničiti korisnika na slanje samo točno određenih ključeva u objektu, konkretno na one definirane Pydantic modelom `Proizvod`.

- jednostavno ćemo zamijeniti `dict` s `Proizvod` u definiciji rute:

```
@app.post("/proizvodi")
def add_proizvod(proizvod: Proizvod): # zamijenili smo dict s Proizvod
    proizvod["id"] = len(proizvodi) + 1
    proizvodi.append(proizvod)
    return proizvod
```

No postoji problem. Ako pokušate poslati isti zahtjev za dodavanje novog proizvoda, vidjet ćete da će FastAPI izbaciti grešku:

```
TypeError: 'Proizvod' object does not support item assignment
```

Zašto dolazi do ove greške?

► Spoiler alert! Odgovor na pitanje

Problem je što **Pydantic generira *read-only* modele**, odnosno modele koji ne podržavaju dodavanje novih ključeva (ili brisanje/ažuriranje postojećih) u objekt nakon što je objekt inicijaliziran.

Međutim, ako bolje pogledamo vidimo da je inicijalni problem što smo definirali `id` u samom modelu, a zatim *hintamo* taj tip podataka prilikom dodavanja novog proizvoda **iako znamo da se `id` automatski dodjeljuje na poslužiteljskoj strani**, odnosno vjerojatno bazi podataka u stvarnom svijetu.

Izbacit ćemo `id` iz modela `Proizvod` budući da želimo da se on automatski dodjeljuje:

```
# models.py

class Proizvod(BaseModel):
    naziv: str
    boja: str
    cijena: float
```

Ako bolje pogledate, problem i dalje postoji jer pokušavamo dodati `id` u objekt `proizvod`:

```
proizvod["id"] = len(proizvodi) + 1
```

Ulagna struktura:

```
{
    "naziv": "šal",
    "boja": "plava",
    "cijena": 30
}
```

Očekivana izlazna struktura:

```
{  
    "id": 5,  
    "naziv": "šal",  
    "boja": "plava",  
    "cijena": 30  
}
```

2.1 Input/Output modeli

Uobičajena praksa je definirati više Pydantic modela za svaku strukturu**, ovisno u kojoj fazi obrade se nalazi.

Što trebamo? Korisnik šalje podatke bez `id`-a, a poslužitelj vraća podatke s `id`-om.

Input Model koji korisnik šalje uobičajeno je nazvati s prefiksom `Create`, `Update`, `In` ovisno o kojoj se CRUD operaciji radi:

```
# models.py  
  
class CreateProizvod(BaseModel):  
    naziv: str  
    boja: str  
    cijena: float
```

Output Model koji se vraća s poslužitelja natrag korisniku uobičajeno je nazvati s prefiksom `Response` ili `Out`:

```
# models.py  
  
class Proizvod(BaseModel):  
    id: int  
    naziv: str  
    boja: str  
    cijena: float
```

Vratimo se na `main.py` datoteku i uključimo oba modela:

```
# main.py  
from fastapi import FastAPI  
  
from models import CreateProizvod, Proizvod
```

Zamjenit ćemo `dict` s `CreateProizvod` u definiciji rute:

```

@app.post("/proizvodi")
def add_proizvod(proizvod: CreateProizvod): # "ulazni proizvod" mora sadržavati naziv,
boju i cijenu
    proizvod["id"] = len(proizvodi) + 1
    proizvodi.append(proizvod)
    return proizvod

```

Međutim, **sada je potrebno napraviti novu instancu klase `Proizvod`** kako bi se mogao dodati `id`:

- izdvojiti ćemo generiranje `id`-a u samostalnu naredbu
- instancirati ćemo novi objekt `Proizvod` s dodijeljenim `id`-om te preostalim podacima iz `proizvod`
- **objekte Pydantic klase instanciramo na identičan način kao i obične Python klase**

```

@app.post("/proizvodi")
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1 # generiramo novi ID u samostalnoj naredbi
    proizvod_s_id = Proizvod(id=new_id, naziv=proizvod.naziv, boja=proizvod.boja,
    cijena=proizvod.cijena) # instanciramo novi objekt Proizvod s dodijeljenim ID-om
    return proizvod_s_id

```

Kôd radi, ali možemo skratiti posao koristeći *unpacking sintaksu* i pretvorbu Pydantic modela u rječnik.

Važno! Umjesto da navodimo svaki atribut modela `createProizvod` prilikom instanciranja `Proizvod`, možemo prvo **pretvoriti** Pydantic model u rječnik koristeći `model_dump()` metodu a potom raspakirati taj rječnik operatorom `**`

Sintaksa:

```
rjecnik = model.model_dump() # pretvaramo Pydantic model u rječnik
```

Dakle, **kôd za instanciranje objekta klase `Proizvod`** možemo skratiti na sljedeći način:

```

@app.post("/proizvodi")
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id = Proizvod(id=new_id, **proizvod.model_dump()) # koristimo ** za
    raspakiravanje rječnika "proizvod"
    return proizvod_s_id

```

Vraćamo korisniku `proizvod_s_id` koji je tipa `Proizvod`, a ne `CreateProizvod`!

Dodatno, moguće je naglasiti da je povratna vrijednost funkcije `add_proizvod` tipa `Proizvod` unutar dekoratora koristeći `response_model` argument:

Sintaksa:

```
@app.metoda("/ ruta", response_model=PydanticModel)
```

Konkretno za naš primjer:

```

@app.post("/proizvodi", response_model=Proizvod) # naglašavamo da je povratna vrijednost
tipa Proizvod
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id = Proizvod(id=new_id, **proizvod.model_dump())
    return proizvod_s_id

```

Ovo je korisno jer FastAPI automatski vrši validaciju podataka koje vraćamo korisniku, također **generira dokumentaciju na temelju ove informacije**.

The screenshot shows the 'Schemas' section of the FastAPI documentation. It displays three Pydantic models:

- CreateProizvod**:


```

CreateProizvod ^ Collapse all object
  naziv* string
  boja* string
  cijena* number
      
```
- HTTPValidationError**:


```

HTTPValidationError > Expand all object
      
```
- Proizvod**:


```

Proizvod ^ Collapse all object
  naziv* string
  boja* string
  cijena* number
  id* integer
      
```
- ValidationError**:


```

ValidationError > Expand all object
      
```

Na dnu dokumentirane rute možete vidjeti **definirane Pydantic podatkovne modele** pod `Schemas` sekcijom

The screenshot shows the API test interface for the `/proizvodi` endpoint. The method is `POST`, the path is `/proizvodi`, and the operation name is `Add Proizvod`. The 'Parameters' section shows 'No parameters'. The 'Request body' section is marked as `required` and contains the following JSON schema:

```
{
  "naziv": "string",
  "boja": "string",
  "cijena": 0
}
```

The 'Content-Type' dropdown is set to `application/json`. At the bottom is a blue 'Execute' button.

Uočite da je struktura JSON objekta koji se očekuje (prema Pydantic modelu `CreateProizvod`) odmah prikazana u dokumentaciji

Važno je još naglasiti sljedeće: Nakon što smo validirali podatke koje korisnik šalje (ulazni model `CreateProizvod`), **nije potrebno izrađivati novi objekt** `Proizvod` s dodijeljenim `id`-om budući da bi onda opet trebali pozvati metodu `model_dump()` kako bismo pohranili čisti rječnik u listu proizvoda.

```

@app.post("/proizvodi", response_model=Proizvod)
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id = Proizvod(id=new_id, **proizvod.model_dump()) # redundantno stvaranje novog objekta Proizvod
    proizvodi.append(proizvod_s_id.model_dump()) # dodajemo rječnik "čistih podataka" u listu proizvoda, a ne Pydantic model!
    return proizvod_s_id

```

Umjesto toga, ako nemamo posebnu potrebnu izrađivati novu instancu klase `Proizvod`, napraviti ćemo samo ono što je potrebno - **validirati podatke**.

U tom slučaju nećemo stvarati instancu, **već samo hintati vrijednost** `proizvod_s_id`!

- uočite da kad ne stvaramo novu instancu, moramo stvarati rječnik vitičastim zagradama `{}` i držati se pravila za definiranje rječnika, možemo i koristiti konstruktor `dict()`:

```

@app.post("/proizvodi", response_model=Proizvod)
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id : Proizvod = {"id" : new_id, **proizvod.model_dump()} # samo hintamo vrijednost, ne stvaramo novu instancu!
    proizvodi.append(proizvod_s_id) # dodajemo Pydantic model u listu proizvoda
    return proizvod_s_id

```

2.2 Zadaci za vježbu - Osnove definicije ruta i Pydantic modela

- Definirajte novu FastAPI rutu `GET /filmovi` koja će klijentu vraćati listu filmova definiranu u sljedećoj listi:

```
filmovi = [
    {"id": 1, "naziv": "Titanic", "genre": "drama", "godina": 1997},
    {"id": 2, "naziv": "Inception", "genre": "akcija", "godina": 2010},
    {"id": 3, "naziv": "The Shawshank Redemption", "genre": "drama", "godina": 1994},
    {"id": 4, "naziv": "The Dark Knight", "genre": "akcija", "godina": 2008}
]
```

- Nadogradite prethodnu rutu na način da će **output** biti validiran Pydantic modelom `Film` kojeg definirate u zasebnoj datoteci `models.py`.
- Definirajte novu FastAPI rutu `GET /filmovi/{id}` koja će omogućiti pretraživanje novog filma prema `id`-u definiranom u parametru rute `id`. Dodajte i ovdje validaciju Pydantic modelom `Film`.
- Definirajte novu rutu `POST /filmovi` koja će omogućiti dodavanje novog filma u listu filmova. Napravite novi Pydantic model `CreateFilm` koji će sadržavati atribute `naziv`, `genre` i `godina`, a kao output vraćajte validirani Pydantic model `Film` koji predstavlja novododani film s automatski dodijeljenim `id`-em.
- Dodajte *query* parametre u rutu `GET /filmovi` koji će omogućiti filtriranje filmova prema `genre` i `min_godina`. Zadane vrijednosti za *query* parametre neka budu `None` i `2000`.

2.3 Složeniji Pydantic modeli

Pydantic modeli mogu sadržavati i **složenije strukture podataka** kao što su liste, rječnici, ugniježđeni modeli i slično. U nastavku ćemo vidjeti kako definirati složenije modele i kako ih koristiti u FastAPI aplikaciji.

U Zadatku 2.2 susreli smo se s jednostavnim modelom `Film` koji sadrži samo osnovne atribute, odnosno primitivne tipove podataka. Ako želimo odraditi validaciju podataka za rutu koja vraća više filmova gdje svaki film rječnik validiran instancom klase `Film`, možemo to definirati i ugrađenom `List` klasom.

Primjerice, ako je struktura podataka sljedeća:

```
[  
    {  
        "id": 1,  
        "naziv": "Titanic",  
        "genre": "drama",  
        "godina": 1997  
    },
```

```
{  
    "id": 2,  
    "naziv": "Inception",  
    "genre": "akcija",  
    "godina": 2010  
}  
]
```

Definiramo model `FilmResponse` koji opisuje danu strukturu filma:

```
# models.py  
  
from pydantic import BaseModel  
  
class FilmResponse(BaseModel):  
    id: int  
    naziv: str  
    genre: str  
    godina: int
```

Definicija rute (bez Pydantic validacije) u `main.py` izgleda ovako:

```
@app.get("/filmovi", )  
def get_filmovi():  
    return filmovi
```

Nije potrebno svaki element rječnika eksplicitno pretvarati u instancu modela `FilmResponse`, kao što bi to radili na sljedeći način:

```
@app.get("/filmovi")  
def get_filmovi():  
    filmovi_objekti = [FilmResponse(**film) for film in filmovi] # pretvaramo svaki rječnik  
    iz filmovi u instancu modela FilmResponse  
    return filmovi_objekti
```

Iako je kôd iznad ispravan, ako bismo dodali novi film u listu `filmovi` kojemu nedostaje neki atribut, primjerice `godina`, poslužitelj će "puknuti" prilikom pokušaja pretvaranja rječnika u instancu modela.

```
filmovi = [  
    {  
        "id": 1,  
        "naziv": "Titanic",  
        "genre": "drama",  
        "godina": 1997  
    },  
    {  
        "id": 2,  
        "naziv": "Inception",  
        "genre": "akcija",  
    }]
```

```

        "godina": 2010
    },
{
    "id": 3,
    "naziv": "The Matrix",
    "genre": "sci-fi",
}
]

@app.get("/filmovi")
def get_filmovi():
    filmovi_objekti = [FilmResponse(**film) for film in filmovi] # greška prilikom
    pretvaranja rječnika u instancu modela za film s ID-em 3
    return filmovi_objekti

```

Poslužitelj vraća grešku 500, što je u redu jer je greška na strani poslužitelja.

Ono što ustvari želimo je da FastAPI automatski vrši validaciju i serijalizaciju podataka u JSON prema definiranom modelu `FilmResponse`, **bez eksplisitnog stvaranja instanci modela** za svaki film u listi te na taj način **skratiti kôd**.

Rekli smo da to postižemo koristeći parametar `response_model` koji se **dodaje u dekorator rute**:

```

@app.get("/filmovi", response_model=FilmResponse) # ali što je rezultat?
def get_filmovi():
    return filmovi

```

Kako je rezultat ove rute ustvari lista rječnika, moramo to navesti i u `response_model` kako ne bi dobili grešku. **FastAPI će automatski pretvoriti svaki rječnik u listi u instancu modela `FilmResponse`** kako bi se osigurala validacija i serijalizacija podataka, bez potrebe za eksplisitnim stvaranjem instanci modela.

U poglavlju [1.2.1 Parametri ruta \(eng. route parameters\)](#) vidjeli smo da je moguće koristiti kolekciju `list` za *type-hinting* složenijih struktura podataka.

Koristeći uglate zagrade s `list` klasom, možemo definirati da se očekuje lista rječnika, odnosno lista modela `FilmResponse`:

Sintaksa:

```
kolekcija[model]
```

Dakle, ruta sad izgleda ovako:

```

@app.get("/filmovi", response_model=list[FilmResponse]) # povratna vrijednost je lista
rječnika, sada konkretno validirana lista modela FilmResponse
def get_filmovi():
    return filmovi

```

```

Schemas
FilmResponse ^ Collapse all object
id* integer
naziv* string
genre* string
godina* integer

```

Rezultat je isti, a naš kôd je puno kraći i čišći. Dodatno, **na ovaj način FastAPI prikazuje u dokumentaciji strukturu uspješnog odgovora**, međutim nismo riješili problem obrade greške što je u redu, jer je greška nastala na strani poslužitelja, što znači da se radi o pogrešci u implementaciji koju treba ispraviti.

U nastavku ćemo vidjeti na koje sve načine možemo definirati Pydantic modele i to kombiniranjem osnovnih tipova, kolekcija, ugniježđenih modela i drugih složenijih tipova.

2.3.1 Tablica osnovnih tipova

Python Tip	Opis	<i>type-hinting</i> primjer
<code>int</code>	Cijeli brojevi	<code>starost: int = 25</code>
<code>float</code>	Decimalni brojevi	<code>cijena: float = 19.99</code>
<code>str</code>	Znakovni nizovi (tekstualni podaci)	<code>ime: str = "John"</code>
<code>bool</code>	Logičke vrijednosti	<code>je_aktivran: bool = True</code>
<code>bytes</code>	Nepromjenjivi Bajtovi	<code>nepromjenjivi_binarni_podatak: bytes = b"binary data"</code>
<code>bytearray</code>	Promjenjivi (eng. mutable) bajtovi	<code>promjenjivi_binarni_podatak: bytearray = bytearray(b"data")</code>

2.3.2 Tablica čestih kolekcija

Python Tip	Opis	Primjer
<code>list</code>	Lista elemenata bilo kojeg tipa	<code>tags: list[str] = ["tag1", "tag2"]</code>
<code>tuple</code>	Nepromjenjivi niz elemenata	<code>koordinate: tuple[float, float] = (1.0, 2.0)</code>
<code>dict</code>	Rječnik ključ-vrijednost parova	<code>config: dict[str, int] = {"key": 42}</code>
<code>set</code>	Skup jedinstvenih elemenata	<code>kategorije: set[str] = {"A", "B"}</code>
<code>frozenset</code>	Nepromjenjivi skup jedinstvenih elemenata	<code>frozen_kategorije: frozenset[str] = frozenset({"A", "B"})</code>

2.3.3 Primjeri složenijih Pydantic modela

Primjer: Želimo definirati Pydantic model `Korisnik` koji će sadržavati osnovne podatke o korisniku:

Korisnik:

- `id` - cijeli broj
- `ime` - string
- `prezime` - string
- `email` - string
- `dob` - cijeli broj
- `aktivran` - logička vrijednost

Rješenje:

```
class Korisnik(BaseModel):  
    id: int  
    ime: str  
    prezime: str  
    email: str  
    dob: int  
    aktivran: bool
```

Primjer: Želimo definirati Pydantic model `Narudžba` koji će sadržavati osnovne podatke o narudžbi i listu imena naručenih proizvoda:

Narudžba:

- `id` - cijeli broj
- `datum` - string
- `proizvodi` - lista stringova
- `ukupna_cijena` - decimalni broj
- `isporučeno` - logička vrijednost

Rješenje:

```
class Narudzba(BaseModel):  
    id: int  
    datum: str  
    proizvodi: list[str] # lista stringova  
    ukupna_cijena: float  
    isporuceno: bool
```

Osim osnovnih tipova i kolekcija, Pydantic modeli mogu sadržavati i **ugniježđene modele**, odnosno druge Pydantic modele. Ovo je korisno kada želimo definirati složenije strukture podataka koje se sastoje od više manjih dijelova.

Primjer: Želimo definirati Pydantic modele `Proizvod` i `Narudžba` gdje narudžba može sadržavati više proizvoda:

Proizvod:

- `id` - cijeli broj
- `naziv` - string
- `cijena` - decimalni broj
- `kategorija` - string
- `boja` - string

Narudžba:

- `id` - cijeli broj
- `ime_kupca` - string
- `prezime_kupca` - string
- `proizvodi` - lista Proizvoda
- `ukupna_cijena` - decimalni broj

Rješenje:

```
class Proizvod(BaseModel):  
    id: int  
    naziv: str  
    cijena: float  
    kategorija: str  
    boja: str  
  
class Narudzba(BaseModel):  
    id: int  
    ime_kupca: str  
    prezime_kupca: str  
    proizvodi: list[Proizvod] # lista proizvoda  
    ukupna_cijena: float
```

Primjer: Želimo definirati Pydantic modele `Proizvod`, `Narudžba`, `StavkaNarudžbe` gdje narudžba može sadržavati više stavki narudžbe, a svaka stavka narudžbe sadrži jedan proizvod.

Proizvod:

- `id` - cijeli broj
- `naziv` - string
- `cijena` - decimalni broj

- `kategorija` - string
- `boja` - string

StavkaNarudžbe:

- `id` - cijeli broj
- `proizvod` - Proizvod
- `narucena_kolicina` - cijeli broj
- `ukupna_cijena` - decimalni broj

Narudžba:

- `id` - cijeli broj
- `ime_kupca` - string
- `prezime_kupca` - string
- `stavke` - lista StavkaNarudžbe
- `ukupna_cijena` - decimalni broj

Rješenje:

```
class Proizvod(BaseModel):
    id: int
    naziv: str
    cijena: float
    kategorija: str
    boja: str

class StavkaNarudzbe(BaseModel):
    id: int
    proizvod: Proizvod
    narucena_kolicina: int
    ukupna_cijena: float

class Narudzba(BaseModel):
    id: int
    ime_kupca: str
    prezime_kupca: str
    stavke: list[StavkaNarudzbe]
    ukupna_cijena: float
```

Zadane vrijednosti (eng. default values)

Jednako kao kod definicije *query* parametra, moguće je koristiti **zadane vrijednosti** za atrIBUTE Pydantic modela. Zadane vrijednosti se postavljaju na isti način kao i kod običnih Python funkcija, dodavanjem `=` nakon tipa podatka.

Primjer: Definirajmo Pydantic model `Korisnik` koji će sadržavati osnovne podatke o korisničkom računu, a zadana vrijednost će biti za atribut `racun_aktiviran`.

Korisnik:

- `id` - cijeli broj
- `ime` - string
- `prezime` - string
- `email` - string
- `dob` - cijeli broj
- `racun_aktivran` - logička vrijednost, zadana vrijednost `True`

Rješenje:

```
class Korisnik(BaseModel):  
    id: int  
    ime: str  
    prezime: str  
    email: str  
    dob: int  
    racun_aktivran: bool = True
```

Rječnici, n-torke i skupovi

U tablici kolekcija vidimo da, osim lista, Pydantic modeli mogu sadržavati i rječnike, n-torke i skupove. U nastavku ćemo vidjeti kako definirati modele koji sadrže ove složenije strukture podataka.

Primjer: Definirajmo Pydantic model `Loto` koji će sadržavati rezultate loto izvlačenja, a rezultati će biti pohranjeni u rječniku gdje su ključevi cijeli brojevi, a vrijednosti broj pojavljivanja tog broja u izvlačenju.

Loto:

- `id` - cijeli broj
- `rezultati` - rječnik cijelih brojeva i njihovih pojavljivanja

Sintaksa:

```
dict[key_type, value_type]
```

Rješenje:

```
class Loto(BaseModel):  
    id: int  
    rezultati: dict[int, int]
```

Primjer: Definirat ćemo Pydantic model `GeoLokacija` koji će sadržavati informacije o geografskoj lokaciji u obliku n-torke (`latitude`, `longitude`).

GeoLokacija:

- `id` - cijeli broj
- `koordinate` - n-torka decimalnih brojeva

Sintaksa:

```
tuple[type1, type2]
```

Rješenje:

```
class GeoLokacija(BaseModel):  
    id: int  
    koordinate: tuple[float, float]
```

Primjer: Definirat ćemo Pydantic model `Inventura` koji će sadržavati naziv skladišta i rječnik proizvoda s nazivima proizvoda i njihovim količinama.

Inventura:

- `id` - cijeli broj
- `naziv_skladista` - String
- `proizvodi` - rječnik stringova i cijelih brojeva

Sintaksa:

```
dict[key_type, value_type]
```

Rješenje:

```
class Inventura(BaseModel):  
    id: int  
    naziv_skladista: str  
    proizvodi: dict[str, int]
```

Složeni tipovi iz biblioteke `typing`

U Pythonu postoji biblioteka `typing` koja sadrži dodatne tipove podataka koji se koriste za *type hinting*. Ovi tipovi su korisni kada želimo definirati složenije strukture podataka koje nisu obuhvaćene osnovnim tipovima ili kolekcijama.

Biblioteka `typing` uključena je od Pythona 3.5 te ju nije potrebno naknadno instalirati.

typing Tip	Opis	type-hinting primjer
<code>Union[T1, T2, T3, ... Tn]</code>	Unija se koristi kada vrijednost može biti jedna od više specificiranih podataka. Dakle, u primjeru <code>vrijednost</code> , ona može biti ili <code>int</code> ili <code>str</code> .	<code>vrijednost: Union[int, str] = 42</code>
<code>Optional</code>	Vrijednost može biti opcionalna, ako nije navedena moguće je definirati i zadalu vrijednost. Ekvivalentno: <code>Union[T, None]</code>	<code>ime: Optional[str] = "Nije navedeno pa se zovem Pero"</code>
<code>Any</code>	Vrijednost može biti bilo kojeg tipa podataka	<code>podatak: Any = "Može biti bilo što"</code>
<code>Callable</code>	Funkcija ili "pozivljivi" objekt (Callable). Moguće je navesti argumente funkcije te povratnu vrijednost	<code>funkcija: Callable[[int, str], str] = lambda x, y: f'{x}, {y}'</code>
<code>Literal</code>	Ograničavanje vrijednosti na unaprijed definirane opcije	<code>smjer: Literal['gore', 'dolje'] = "gore"</code>
<code>TypedDict</code>	Specijalni s definiranim tipovima ključeva i vrijednosti	<code>osoba: TypedDict('osoba', {'ime': str, ' prezime': str})</code>

Vrijednosti `typing` biblioteke ima jako puno. Ovdje su navedeni samo neki od najčešće korištenih tipova. Opsežnu dokumentaciju možete pronaći na [službenoj stranici](#).

Primjer: Definirat ćemo Pydantic model `Kolegij` koji će sadržavati informacije o kolegiju. Semestar može biti samo između vrijednosti `[1, 2, 3, 4, 5, 6]`, a vrijednost `ECTS` ne mora biti navedena, u slučaju da nije navedena, zadana vrijednost je `6`.

Kolegij:

- `id` - cijeli broj
- `naziv` - string
- `semestar` - cijeli broj, unutar `[1, 2, 3, 4, 5, 6]`
- `ECTS` - cijeli broj, opcionalan, zadana vrijednost `6`
- `opis` - string
- `profesor` - string

Rješenje:

```

from typing import Optional, Literal

class Kolegij(BaseModel):
    id: int
    naziv: str
    semestar: Literal[1, 2, 3, 4, 5, 6]
    ECTS: Optional[int] = 6
    opis: str
    profesor: str

```

Primjer: Definirat ćemo Pydantic model `Automobil` koji će sadržavati informacije o automobilu. Boja automobila može biti samo jedna od unaprijed definiranih opcija, godina proizvodnje ne mora biti navedena, snaga motora je rječnik s ključevima `kw` i `ks`, a `cijena` je rječnik s ključevima `osnovna` i `sa_pdv`.

Automobil:

- `id` - cijeli broj
- `marka` - string
- `model` - string
- `boja` - string, jedna od opcija `["crvena", "plava", "zelena", "bijela", "crna"]`
- `godina_proizvodnje` - cijeli broj, optionalan
- `snaga_motora` - rječnik s ključevima `kw` i `ks`
- `cijena` - rječnik s ključevima `osnovna` i `sa_pdv`

Rješenje:

```

from typing import Optional, Literal

class Automobil(BaseModel):
    id: int
    marka: str
    model: str
    boja: Literal["crvena", "plava", "zelena", "bijela", "crna"]
    godina_proizvodnje: Optional[int] # godina proizvodnje nije obavezna, ali ako se navede
mora biti cijeli broj
    snaga_motora: dict[str, int] # zašto je dovoljno samo [str i int]?
    cijena: dict[str, float]

```

Kako bismo instancirali ovu klasu, potrebno je navesti ključeve rječnika `snaga_motora` i `cijena`:

- svaki ključ rječnika `snaga_motora` mora biti string, a vrijednost cijeli broj, međutim **dozvoljeno je navesti neograničeno ključ-vrijednost parova**

```

automobil = Automobil(
    id=1,
    marka="Audi",
    model="A4",
    boja="crvena",
    godina_proizvodnje=2018,
    snaga_motora={"kW": 100, "KS": 136},
    cijena={"osnovna": 25000, "sa_pdv": 30000}
)

```

Kada bismo htjeli **ograničiti ključeve** atributa `snaga_motora` i `cijena`, morali bismo definirati zasebne Pydantic modele:

```

class SnagaMotora(BaseModel):
    kW: int
    KS: int

class Cijena(BaseModel):
    osnovna: float
    sa_pdv: float

class Automobil(BaseModel):
    id: int
    marka: str
    model: str
    boja: Literal["crvena", "plava", "zelena", "bijela", "crna"]
    godina_proizvodnje: Optional[int]
    snaga_motora: SnagaMotora
    cijena: Cijena

```

Ovaj automobil instancirali bi na sljedeći način:

```

automobil = Automobil(
    id=1,
    marka="Audi",
    model="A4",
    boja="crvena",
    godina_proizvodnje=2018,
    snaga_motora=SnagaMotora(kW=100, KS=136),
    cijena=Cijena(osnovna=25000, sa_pdv=30000)
)

```

Vidimo da `snaga_motora` i `cijena` više nisu rječnici, već su **ugniježđeni modeli** `SnagaMotora` i `Cijena`.

Ipak, moguće ih je definirati kao posebne rječnike tipa `TypedDict` iz modula `typing` koji omogućuje definiranje rječnika s točno određenim ključevima.

- sintaksa je ista, jedino što klase nasljeđuju `TypedDict` umjesto `BaseModel`

```
from typing import TypedDict
```

```
class SnagaMotora(TypedDict):
    kW: int
    KS: int

class Cijena(TypedDict):
    osnovna: float
    sa_pdv: float

class Automobil(BaseModel):
    id: int
    marka: str
    model: str
    boja: Literal["crvena", "plava", "zelena", "bijela", "crna"]
    godina_proizvodnje: Optional[int]
    snaga_motora: SnagaMotora
    cijena: Cijena
```

Ovaj automobil instancirali bi na sljedeći način:

```
automobil = Automobil(
    id=1,
    marka="Audi",
    model="A4",
    boja="crvena",
    godina_proizvodnje=2018,
    snaga_motora={"kW": 100, "KS": 136},
    cijena={"osnovna": 25000, "sa_pdv": 30000}
)
```

2.4 Nasljeđivanje Pydantic modela

Nasljeđivanje (*eng. inheritance*) je koncept u programiranju gdje jedan objekt (klasa) može naslijediti atribute i metode drugog objekta (klase). Već smo vidjeli na početku kolegija da je moguće nasljeđivati atribute i metode klase A na način da ju navodimo u zagradama prilikom definicije klase B.

Ista pravila vrijede za Pydantic modele. Ako želimo definirati novi Pydantic model koji će naslijediti atribute i metode nekog drugog modela, to možemo učiniti na sljedeći način:

Sintaksa:

```
# Pydantic model A
class A(BaseModel):
    atribut_a: str
    atribut_b: int

# Pydantic model B koji nasljeđuje atribute i metode modela A i dodaje vlastiti atribut

class B(A):
    atribut_c: float
```

Objekte ovakvih modela instanciramo na isti način kao i obične modele:

```
objekt_a = A(atribut_a="vrijednost_a", atribut_b=42)
objekt_b = B(atribut_a="vrijednost_a", atribut_b=42, atribut_c=3.14)
```

U kontekstu FastAPI poslužitelja i modeliranja podataka, uobičajeno je koristiti prefix `Base` za osnovne modele koji sadrže zajedničke atribute, a zatim nasljeđivati te modele u specifičnijim modelima, npr.

`Create`, `Update`, `Response`, `In`, `Out` i slično.

Ako se vratimo na model `Proizvod` koji smo imali u dosadašnjim primjerima, možemo definirati modele `BaseProizvod`, `RequestProizvod` i `ResponseProizvod`.

- kako prilikom dodavanja proizvoda ne želimo da korisnik unosi `id`, niti cijenu s PDV-om koju ćemo računati na poslužitelju (u ovom slučaju 25% PDV-a), možemo definirati `BaseProizvod` model koji sadrži osnovne atribute proizvoda
- u tom slučaju, klasa `RequestProizvod` nasljeđuje atribute iz `BaseProizvod` modela i ne dodaje ništa novo (jer to su atribute koje klijent šalje poslužitelju)
- klasa `ResponseProizvod` nasljeđuje atribute iz `BaseProizvod` modela i dodaje `id` atribut te računa cijenu s PDV-om u atributu `cijena_pdv`

```

class BaseProizvod(BaseModel):
    naziv: str
    cijena: float
    kategorija: str
    boja: str

class RequestProizvod(BaseProizvod): # nasljeđujemo attribute iz BaseProizvod modela
    pass # ne dodajemo niti jedan novi atribut

class ResponseProizvod(BaseProizvod): # nasljeđujemo attribute iz BaseProizvod modela
    id: int
    cijena_pdv: float

```

Primjer rute za dodavanje proizvoda s ukupnom validacijom podataka:

```

@app.post("/proizvodi", response_model=ResponseProizvod) # validacija i serijalizacija
HTTP odgovora prema ResponseProizvod modelu
def dodaj_proizvod(proizvod: RequestProizvod): # RequestProizvod model koristimo za
validaciju podataka koje klijent šalje
    PDV_MULTIPLIER = 1.25
    some_id = random.randrange(1, 100) # simuliramo dodjelu ID-a
    cijena_pdv = proizvod.cijena * PDV_MULTIPLIER # računamo cijenu s PDV-om
    proizvod_spreman_za_pohranu : ResponseProizvod = {**proizvod.model_dump(), "id":some_id,
"cijena_pdv":cijena_pdv} # ne instanciramo novi ResponseProizvod, već koristimo type-
hinting
    proizvodi.append(proizvod_spreman_za_pohranu)
    return proizvod_spreman_za_pohranu

```

Curl

```
curl -X 'POST' \
  'http://localhost:8000/proizvodi' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "naziv": "Sot",
    "cijena": 300,
    "kategorija": "sotovi",
    "boja": "svila"
}'
```

Request URL

<http://localhost:8000/proizvodi>

Server response

Code	Details
200	<p>Response body</p> <pre>{ "naziv": "Sot", "cijena": 300, "kategorija": "sotovi", "boja": "svila", "id": 64, "cijena_pdv": 375 }</pre> <p>Download</p> <p>Response headers</p> <pre>content-length: 93 content-type: application/json date: Sun, 12 Jan 2025 19:00:23 GMT server: uvicorn</pre>

Responses

Code	Description	Links
200	Successful Response	No links

U dokumentaciji vidimo da su poslati atributi `naziv`, `cijena`, `kategorija` i `boja`, a vraćeni atributi su `id`, `naziv`, `cijena`, `kategorija`, `boja` i `cijena_pdv`.

Primjer: Definirajmo Pydantic modele `KorisnikBase`, `KorisnikCreate` i `KorisnikResponse` koji će sadržavati osnovne podatke o korisniku, podatke koje korisnik šalje prilikom registracije i podatke koje korisnik dobiva kao odgovor prilikom registracije. Dodatno, `KorisnikResponse` model sadrži i atribut `datum_registracije` koji predstavlja trenutni datum i vrijeme registracije korisnika.

- lozinka koju korisnik šalje prilikom registracije je u tekstuallnom obliku, međutim, prilikom registracije u bazi podataka, lozinka se sprema kao heširana vrijednost
- osim heširane lozinke, povratna vrijednost nakon uspješne registracije sadrži i datum registracije koji će biti objekt tipa `datetime`

KorisnikBase:

- `ime` - string
- `prezime` - string
- `email` - string

KorisnikCreate: nasljeđuje atribute iz `KorisnikBase` modela

- `lozinka_text` - string

KorisnikResponse: nasljeđuje atribute iz `KorisnikBase` modela

- `lozinka_hash` - string
- `datum_registracije` - objekt tipa `datetime`

Rješenje:

```
from datetime import datetime

class KorisnikBase(BaseModel):
    ime: str
    prezime: str
    email: str

class KorisnikCreate(KorisnikBase):
    lozinka_text: str

class KorisnikResponse(KorisnikBase):
    lozinka_hash: str
    datum_registracije: datetime # hintamo složeni objekt tipa datetime
```

Primjer rute za registraciju korisnika:

```

@app.post("/korisnici", response_model=KorisnikResponse) # validacija i serijalizacija
HTTP odgovora prema KorisnikResponse modelu
def registracija_korisnika(korisnik: KorisnikCreate):

    lozinka_hash = str(hash(korisnik.lozinka_text)) # simuliramo heširanje lozinke
    datum_registracije = datetime.now() # trenutni datum i vrijeme registracije
    korisnik_spreman_za_pohranu : KorisnikResponse = {**korisnik.model_dump(),
"lozinka_hash" : lozinka_hash, "datum_registracije": datum_registracije} # uzimamo sve iz
KorisnikCreate + lozinka_hash i datum_registracije kako bismo zadovoljili KorisnikResponse
model

    print(f"Korisnik spremam za pohranu: {korisnik_spreman_za_pohranu}")

    korisnici.append(korisnik_spreman_za_pohranu)
    return korisnik_spreman_za_pohranu # vraćamo KorisnikResponse model

```

Validacijom podataka kroz ova tri modela postigli smo sljedeće:

- klijent šalje podatke o korisniku prilikom registracije te unosi **ime, prezime, lozinku u tekstualnom obliku i email**
- podaci koje klijent šalje se validiraju prema `KorisnikCreate` modelu
- na poslužitelju se **hešira lozinka i dodaje datum registracije** te se podaci validiraju prema `KorisnikResponse` modelu
- u bazu podataka (u ovom slučaju *in-memory* lista) sprema se heširana lozinka i datum registracije, **bez lozinke u tekstualnom obliku!**
- klijent dobiva odgovor s podacima o korisniku, **bez lozinke u tekstualnom obliku!**

```

Curl
curl -X 'POST' \
'http://localhost:8000/korisnici' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "ime": "string",
    "prezime": "string",
    "email": "string",
    "lozinka_text": "string"
}'
Request URL
http://localhost:8000/korisnici
Server response
Code Details
200 Response body
{
    "ime": "string",
    "prezime": "string",
    "email": "string",
    "lozinka_hash": "2698794031908662734",
    "datum_registracije": "2025-01-12T21:32:43.472310"
}
Download
Response headers
content-length: 129
content-type: application/json
date: Sun, 12 Jan 2025 20:32:43 GMT
server: uvicorn

```

U dokumentaciji vidimo da su poslani atributi `ime`, `prezime`, `email` i `lozinka_text`, a vraćeni atributi su `ime`, `prezime`, `email`, `lozinka_hash` i `datum_registracije`.

2.5 Zadaci za vježbu: Definicija složenijih Pydantic modela

- Definirajte Pydantic modele `Knjiga` i `Izdavač` koji će validirati podatke o knjigama i izdavačima. Svaka knjiga sastoji se od naslova, imena autora, prezimena autora, godine izdavanja, broja stranica i izdavača. Izdavač se sastoji od naziva i adrese. Ako godina izdavanja nije navedena, zadana vrijednost je trenutna godina.
- Definirajte Pydantic model `Admin` koji validira podatke o administratoru sustava. Administrator se sastoji od imena, prezimena, korisničkog imena, emaila te ovlasti. Ovlasti su lista stringova koje mogu sadržavati vrijednosti: `dodavanje`, `brisanje`, `ažuriranje`, `čitanje`. Ako ovlasti nisu navedene, zadana vrijednost je prazna lista. Za ograničavanje ovlasti koristite `Literal` tip iz modula `typing`.
- Definirajte Pydantic model `Restaurantorder` koji se sastoji od informacija o narudžbi u restoranu. Narudžba se sastoji od identifikatora, imena kupca, `stol_info`, liste jela i ukupne cijene. Definirajte još jedan model za jelo koje se sastoji od identifikatora, naziva i cijene. Za `stol_info` pohranite rječnik koji očekuje ključeve `broj` i `lokacija`. Primjerice, `stol_info` može biti `{"broj": 5, "lokacija": "terasa"}`. Za definiciju takvog rječnika koristite `TypedDict` tip iz modula `typing`.
- Definirajte Pydantic modela `cctv_frame` koji će validirati podatke o trenutnoj slici s CCTV kamere. Trenutna slika se sastoji od identifikatora, vremena snimanja, te koordinate x i y. Koordinate validirajte kao n-torku decimalnih brojeva. Ako koordinate nisu navedene, zadana vrijednost je `(0.0, 0.0)`.

2.6 Field polje Pydantic modela

U prethodnim primjerima vidjeli smo kako definirati Pydantic modele koristeći atribute i nasljeđivanje. U nekim slučajevima, možda ćemo htjeti definirati dodatne podatke o atributima, kao što su:

- zadane vrijednosti
- opisi atributa
- ograničenja
- aliasi
- ...

Za to koristimo `Field` polje koje se nalazi u modulu `pydantic`. `Field` polje koristi se za **definiranje dodatne informacije o atributima** Pydantic modela.

Sintaksa:

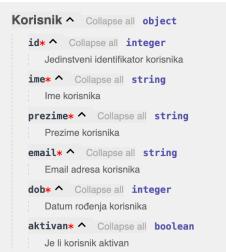
```
from pydantic import Field

class NekiModel(BaseModel):
    neki_atribut: tip = Field()
```

Primjerice, ako se vratimo na model `Korisnik` koji smo definirali ranije, možemo dodati dodatne informacije (`description`) o atributima koje bi željeli poslati korisniku u slučaju da dođe do validacijske pogreške:

```
from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika")
    ime: str = Field(description="Ime korisnika")
    prezime: str = Field(description="Prezime korisnika")
    email: str = Field(description="Email adresa korisnika")
    dob: int = Field(description="Datum rođenja korisnika")
    aktivan: bool = Field(description="Je li korisnik aktivan")
```



U dokumentaciji vidimo definirane opise atributa

Ako bismo ovdje sada htjeli dodati zadane vrijednosti, koristimo `default` parametar u `Field` polju:

```
from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika", default=1)
    ime: str = Field(description="Ime korisnika", default="John")
    prezime: str = Field(description="Prezime korisnika", default="Doe")
    email: str = Field(description="Email adresa korisnika", default="JohnDoe@gmail.com")
    dob: int = Field(description="Datum rođenja korisnika", default=1990)
    aktivan: bool = Field(description="Je li korisnik aktivan", default=True)
```

Ukoliko želimo **ograničiti vrijednosti numeričkih atributa**, koristimo `ge` i `le` parametre u `Field` polju:

- `ge` - greater than or equal to
- `gt` - greater than
- `le` - less than or equal to
- `lt` - less than

```

from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika", ge=1, le=100) # id mora biti između 1 i 100
    ime: str = Field(description="Ime korisnika")
    prezime: str = Field(description="Prezime korisnika")
    email: str = Field(description="Email adresa korisnika")
    dob: int = Field(description="Datum rođenja korisnika", ge=1900, le=2021) # datum rođenja mora biti između 1900 i 2021
    aktivavan: bool = Field(description="Je li korisnik aktivan")

```

Ukoliko želimo ograničiti duljine znakovnih nizova, koristimo `max_length` i `min_length` argumente u `Field` polju:

```

from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika", ge=1, le=100)
    ime: str = Field(description="Ime korisnika", min_length=2, max_length=50) # ime mora imati između 2 i 50 znakova
    prezime: str = Field(description="Prezime korisnika", min_length=2, max_length=50) # prezime mora imati između 2 i 50 znakova
    email: str = Field(description="Email adresa korisnika")
    dob: int = Field(description="Datum rođenja korisnika", ge=1900, le=2021)
    aktivavan: bool = Field(description="Je li korisnik aktivan")

```

U sljedećoj tablici dani su česti parametri koji se koriste u `Field` polju:

Field Parametar	Opis parametra	Primjer
<code>default</code>	Zadana vrijednost za polje.	<code>ime: str = Field("Ivan Horvat")</code>
<code>default_factory</code>	Funkcija koja dinamički generira zadanu vrijednost.	<code>kreirano: datetime = Field(default_factory=datetime.utcnow)</code>
<code>title</code>	Naslov za polje, koristi se za dokumentaciju.	<code>ime: str = Field(..., title="Puno ime")</code>
<code>description</code>	Opis polja, koristi se za dokumentaciju.	<code>dob: int = Field(..., description="Dob osobe, mora biti 18 ili više")</code>
<code>alias</code>	Alternativni naziv za polje u serijaliziranim podacima.	<code>email: str = Field(..., alias="email_adresa")</code>
<code>const</code>	Ako je <code>True</code> , vrijednost se ne može mijenjati nakon inicijalizacije.	<code>uloga: str = Field("admin", const=True)</code>
<code>gt</code>	Vrijednost mora biti veća od ove.	<code>rezultat: int = Field(..., gt=0)</code>
<code>ge</code>	Vrijednost mora biti veća ili jednaka ovoj.	<code>dob: int = Field(..., ge=18)</code>
<code>lt</code>	Vrijednost mora biti manja od ove.	<code>postotak: float = Field(..., lt=100)</code>
<code>le</code>	Vrijednost mora biti manja ili jednaka ovoj.	<code>ocjena: int = Field(..., le=10)</code>
<code>min_length</code>	Minimalna duljina stringa ili liste.	<code>korisnicko_ime: str = Field(..., min_length=3)</code>
<code>max_length</code>	Maksimalna duljina stringa ili liste.	<code>lozinka: str = Field(..., max_length=20)</code>
<code>regex</code>	Regex obrazac koji polje mora zadovoljiti.	<code>email: str = Field(..., regex=r'^\S+@\S+\.\S+\$')</code>

3. Obrada grešaka (eng. Error Handling)

Do sad smo naučili kako definirati osnovne FastAPI rute koje prihvaćaju parametre rute, *query* parametre i tijelo zahtjeva. Također smo naučili kako definirati Pydantic modele koji služe za validaciju dolaznih podataka, automatsku deserijalizaciju i serijalizaciju podataka te automatsku generaciju dokumentacije.

U ovom poglavlju ćemo se upoznati s dodatnim sigurnosnim mehanizmima koje svaki robusni poslužitelj mora imati u svojim definicijama ruta. To je naravno obrada grešaka koje mogu nastati korisničkom pogreškom (`4xx`) ili greškom na poslužitelju (`5xx`).

FastAPI ima gotovu podršku za obradu grešaka kroz `HTTPException` klasu.

```
from fastapi import HTTPException
```

Ova klasa koristi se za podizanje iznimke u slučaju greške, ustvari se radi o običnoj Python iznimci (`Exception`) koja se podiže kada dođe do greške, ali u našem slučaju sadrži dodatne informacije o statusu greške i poruci koja se vraća korisniku u kontekstu HTTP protokola.

Za **vraćanje iznimke** u Pythonu, općenito koristimo ključnu riječ `raise`:

```
raise Exception("Došlo je do greške")
```

Primjerice: ako korisnik pokuša pristupiti resursu koji ne postoji, možemo podići iznimku `HTTPException` s odgovarajućim statusom (`status_code`) i porukom (`detail`):

```
raise HTTPException(status_code=404, detail="Resurs nije pronađen")
```

Uzet ćemo sljedeći primjer: korisnik pokušava dohvatiti podatke o knjigama, međutim zatraži knjigu s naslovom koji ne postoji u bazi podataka. U tom slučaju, podižemo iznimku `HTTPException` s statusom `404` i porukom `Knjiga nije pronađena`.

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

knjige = [
    {"id": 1, "naslov": "Ana Karenjina", "autor": "Lav Nikolajević Tolstoj"},
    {"id": 2, "naslov": "Kiklop", "autor": "Ranko Marinković"},
    {"id": 3, "naslov": "Proces", "autor": "Franz Kafka"}
]

@app.get("/knjige/{naslov}", response_model=Knjiga)
def dohvati_knjigu(naslov: str):
    for knjiga in knjige:
        if knjiga["naslov"] == naslov:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=404, detail="Knjiga nije pronađena") # podižemo iznimku
    # ako knjiga nije pronađena s odgovarajućom porukom i statusnim kôdom
```

Definirat ćemo rutu za dodavanje nove knjige, međutim, ako korisnik pokuša dodati knjigu koja već postoji u bazi podataka, podići ćemo iznimku `HTTPException` s statusom `400` i porukom `Knjiga već postoji`.

Definirat ćemo prvo odgovarajuće Pydantic modele:

```
# models.py

from pydantic import BaseModel

class KnjigaRequest(BaseModel):
    naslov: str
    autor: str

class KnjigaResponse(KnjigaRequest):
    id: int
```

```
# main.py

@app.post("/knjige", response_model=KnjigaResponse)
def dodaj_knjigu(knjiga_request: KnjigaRequest):
    for pohranjena_knjiga in knjige: # prolazimo kroz sve knjige u "bazi podataka"
        if pohranjena_knjiga["naslov"] == knjiga_request.naslov:
            raise HTTPException(status_code=400, detail="Knjiga već postoji")
    new_id = knjige[-1]["id"] + 1
    nova_knjiga : KnjigaResponse = {"id": new_id, **knjiga_request.model_dump()} # ne
instanciramo novi KnjigaResponse, već koristimo type-hinting
    knjige.append(nova_knjiga) # dodajemo rječnik koji predstavlja knjigu
    return nova_knjiga
```

Općenito, klasa `HTTPException` ima sljedeće parametre:

- `status_code` - statusni kôd HTTP odgovora
- `detail` - poruka koja se vraća korisniku
- `headers` - dodatna zaglavlja HTTP odgovora

Naravno, moguće je strukturirati rutu i na način da može podići više različitih iznimki, ovisno o situaciji:

```
@app.get("/knjige/{id}", response_model=KnjigaResponse)
def dohvati_knjigu(id: int):

    if id < 1:
        raise HTTPException(status_code=400, detail="ID mora biti veći od 0")

    for knjiga in knjige:
        if knjiga["id"] == id:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=404, detail=f"Knjiga s id-em {id} nije pronađena") # podižemo iznimku ako knjiga nije pronađena s odgovarajućom porukom i statusnim kôdom
```

Osim direktnog upisa statusnih kôdova, postoji konvencija korištenja specijalnog `status` modula iz FastAPI paketa koji sadrži gotove statusne kôdove.

- na ovaj način povećavamo čitljivost kôda i smanjujemo mogućnost greške
- također, ovim principom naš IDE može bolje prepoznati statusne kôdove te ga sam editor može pronaći

```
from fastapi import status

@app.get("/knjige/{id}", response_model=KnjigaResponse)
def dohvati_knjigu(id: int):

    if id < 1:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="ID mora biti veći od 0") # koristimo status modul za statusni kôd

    for knjiga in knjige:
        if knjiga["id"] == id:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"Knjiga s id-em {id} nije pronađena") # koristimo status modul za statusni kôd
```

Sve statusne kôdove unutar ovog modula možete pronaći na sljedećoj [poveznicu](#)

Za kraj, ako radite vaš projekt koristeći WebSocket protokol, FastAPI ima podršku za podizanje iznimki kroz `WebSocket` klasu:

```
from fastapi import WebSocketException
```

Međutim, to nije predmet ovih vježbi. Za sve kojih zanima više o WebSocket protokolu, posjetite sljedeću [poveznicu](#).

3.1 Validacija *route* i *query* parametara

U primjeru iznad validirali smo tijelo zahtjeva kroz Pydantic model `KnjigaResponse`, odnosno `KnjigaRequest` za POST rutu. Međutim, ponekad želimo validirati i parametre rute i *query* parametre koje korisnik šalje u URL-u na sličan način kao što smo validirali tijelo zahtjeva.

U tu svrhu postoje `Path` i `query` polja iz modula `fastapi` koja koristimo za validaciju parametara rute i *query* parametara.

Primjer: Vidjeli smo kako možemo validirati parametre rute i *query* parametre u FastAPI ruti koristeći *type-hinting*. No, što ako moramo provjeriti kao u primjeru iznad je li ID veći od 0? Upotrijebit ćemo `Path` polje za validaciju parametara rute.

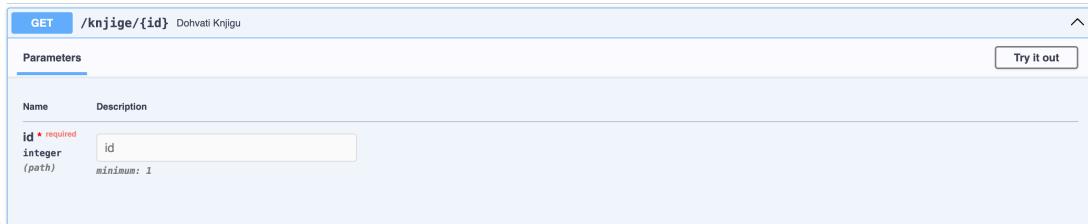
```

from fastapi import Path

@app.get("/knjige/{id}", response_model=KnjigaResponse)
def dohvati_knjigu(id: int = Path(title="ID knjige", ge=1)): # koristimo isti "ge"
    parametar kao u Field polju
    for knjiga in knjige:
        if knjiga["id"] == id:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=404, detail=f"Knjiga s id-em {id} nije pronađena") # podižemo iznimku ako knjiga nije pronađena s odgovarajućom porukom i statusnim kôdom

```

Na ovaj način, osim čišćeg kôda, dobivamo i oznaku `"minimum : 1"` u dokumentaciji koja korisniku daje informaciju o minimalnoj vrijednosti ovog parametra.



Dobivamo oznaku `"minimum : 1"` u dokumentaciji koja korisniku daje informaciju o minimalnoj vrijednosti ovog parametra.

Više u ovom obliku validacije parametra rute na [FastAPI dokumentaciji](#).

Na isti način možemo validirati i `query` parametre koristeći `query` polje. Malo ćemo proširiti podatke o našim knjigama na način da sadrže i informaciju o broju stranica i godini izdavanja.

```

knjige = [
    {"id": 1, "naslov": "Ana Karenjina", "autor": "Lav Nikolajević Tolstoj",
     "broj_stranica": 864, "godina_izdavanja": 1877},
    {"id": 2, "naslov": "Kiklop", "autor": "Ranko Marinković", "broj_stranica": 488,
     "godina_izdavanja": 1965},
    {"id": 3, "naslov": "Proces", "autor": "Franz Kafka", "broj_stranica": 208,
     "godina_izdavanja": 1925}
]

```

Nadogradit ćemo i Pydantic modele:

```
# models.py

from pydantic import BaseModel, Field

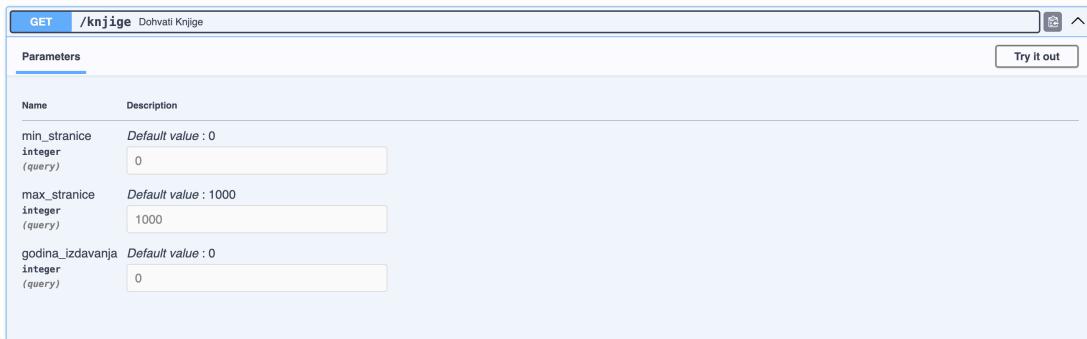
class KnjigaRequest(BaseModel):
    naslov: str
    autor: str
    broj_stranica: int = Field(ge=1) # broj stranica mora biti veći od 1
    godina_izdavanja: int = Field(ge=0, le=2024) # godina izdavanja mora biti između 0 i
    2024
```

Idemo definirati rutu za dohvaćanje svih knjiga s 3 *query* parametra: `min_stranice`, `max_stranice` i `godina_izdavanja`.

Prvo **primjer s osnovnom validacijom** *query* parametara kroz *type-hinting*:

```
@app.get("/knjige")
def dohvati_knjige(min_stranice: int = 0, max_stranice: int = 1000, godina_izdavanja: int
= 0):
    filtrirane_knjige = []
    for knjiga in knjige:
        if knjiga["broj_stranica"] >= min_stranice and knjiga["broj_stranica"] <= max_stranice
        and knjiga["godina_izdavanja"] == godina_izdavanja:
            filtrirane_knjige.append(knjiga)
    return filtrirane_knjige
```

Primjer dokumentirane rute:



U dokumentaciji vidimo da su *query* parametri `min_stranice`, `max_stranice` i `godina_izdavanja` s zadanim vrijednostima.

Međutim, možemo dodatno **proširiti validaciju *query* parametara** kroz `query` polje:

- `min_stranice` mora biti veći od 0
- `max_stranice` mora biti veći od 0
- `godina_izdavanja` mora biti između 0 i 2024
- `min_stranice` mora biti manji od `max_stranice` (ovo radimo u samoj funkciji)

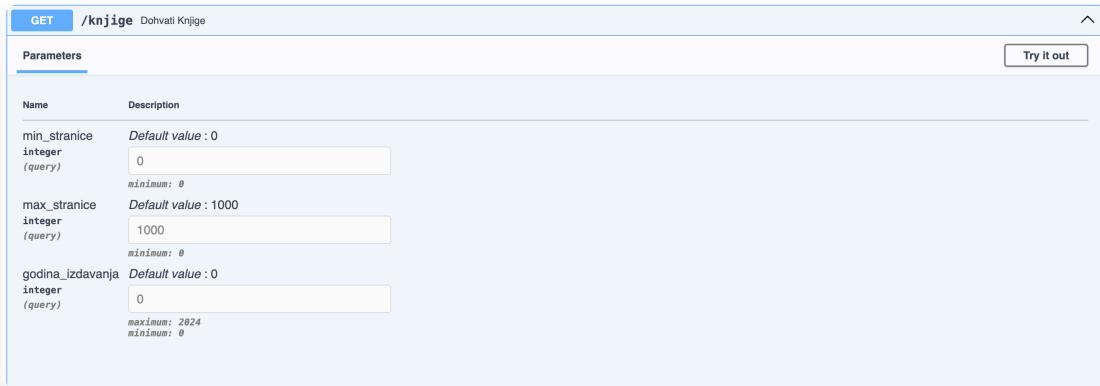
```

from fastapi import _query_

@app.get("/knjige")
def dohvati_knjige(min_stranice: int = _query_(0, ge=1), max_stranice: int = _query_(1000, ge=1), godina_izdavanja: int = _query_(0, ge=0, le=2024)):
    if min_stranice > max_stranice:
        raise HTTPException(status_code=400, detail="Minimalni broj stranica mora biti manji od maksimalnog")
    filtrirane_knjige = []
    for knjiga in knjige:
        if knjiga["broj_stranica"] >= min_stranice and knjiga["broj_stranica"] <= max_stranice and knjiga["godina_izdavanja"] == godina_izdavanja:
            filtrirane_knjige.append(knjiga)
    return filtrirane_knjige

```

Primjer dokumentirane rute s dodatnim validacijama:



3.2 Zadaci za vježbu: Obrada grešaka

- Definirajte rutu i odgovarajući Pydantic model za dohvaćanje podataka o automobilima. Svaki automobil ima sljedeće atribute: `id`, `marka`, `model`, `godina_proizvodnje`, `cijena` i `boja`. Ako korisnik pokuša dohvatiti automobil s ID-em koji ne postoji, podignite iznimku `HTTPException` s statusom `404` i porukom `Automobil nije pronađen`.
- Nadogradite prethodnu rutu s `query` parametrima `min_cijena`, `max_cijena`, `min_godina` i `max_godina`. Implementirajte validaciju `query` parametra za cijenu i godinu proizvodnje. Minimalna cijena mora biti veća od 0, a minimalna godina proizvodnje mora biti veća od 1960. Unutar funkcije obradite iznimku kada korisnik unese minimalnu cijenu veću od maksimalne cijene ili minimalnu godinu proizvodnje veću od maksimalne godine proizvodnje te vratite odgovarajući `HTTPException`.
- Definirajte rutu za dodavanje novog automobila u bazu podataka. `id` se mora dodati na poslužitelju, kao i atribut `cijena_pdv` (definirajte dodatni Pydantic model za to). Ako korisnik pokuša dodati automobil koji već postoji u bazi podataka, podignite odgovarajuću iznimku. Implementirajte ukupno 3 Pydantic modela, uključujući `BaseCar` model koji će nasljeđivati preostala 2 modela.

4. Strukturiranje poslužitelja i organizacija kôda

U ovom poglavlju ćemo se upoznati s organizacijom kôda u FastAPI poslužitelju. Kako bi naš poslužitelj bio čitljiviji i lakši za održavanje, bitno je organizirati kôd na način da bude strukturiran i pregledan.

4.1 Dependency Injection (DI)

FastAPI ima moćan **Dependency Injection** sustav koji omogućuje da se kôd poslužitelja strukturira na način da se smanji ponavljanje kôda i poveća čitljivost.

Dependency Injection (*DI*) je dizajnerski obrazac u softverskom inženjerstvu koji omogućava bolju modularnost programskog proizvoda. DI je ustvari način upravljanja ovisnostima objekta (*eng. Dependency*) u aplikaciji tako da se vanjske ovisnosti klase ili objekta "ubrizgavaju" izvana, umjesto da ih instanca klase (objekt) sam stvara ili upravlja njima.

Glavna ideja:

Umjesto da klasa A stvara klasu B unutar sebe (što stvara jaku ovisnost između A i B):

```
Class A → creates → Class B
```

Klasa A prima instancu klase B izvana (*loose coupling*) te je time manje ovisna o klasi B:

```
External code → provides Class B → Class A
```

Ovakav dizajnerski obrazac je koristan kada:

- želimo smanjiti ovisnost između klasa
- postoji logika koja se ponavlja u više klasa, odnosno koju je potrebno dijeliti
- dijeljenje konekcije na bazu podataka
- dijeljenje konfiguracijskih postavki
- dijeljenje autorizacijske logike

Kada koristimo FastAPI, DI možemo ostvariti koristeći modul `Depends` iz FastAPI paketa.

```
from fastapi import Depends
```

Dependency Injection koristimo tako da definiramo **funkciju koja vraća ovisnost**, a zatim tu funkciju koristimo kao argument u ruti.

Primjerice: Zamislimo da imamo poslužitelj koji sadrži nekoliko administratorskih ruta, ali za pristup tim rutama **korisnik mora biti autoriziran**. Simulirat ćemo funkciju koja vraća korisničko ime na temelju tokena koji pristiže s HTTP zahtjevom.

Ideja je sljedeća:

- korisnik šalje **token** s HTTP zahtjevom kojim dokazuje da je autoriziran i da je on administrator
- ako se token ne podudara s tokenom koji je potreban za pristup administratorskim rutama, korisniku se vraća greška

```
@app.get("/tajni_podaci")
def get_tajni_podaci(token: str):
    if token != "super_secret_admin_token007": # provjeravamo je li token ispravan
        # simuliramo samo naravno
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    return {"tajni_podaci": "šifra za sef je 1234"}
```

Ako dodamo još nekoliko ruta, primjerice za ažuriranje i brisanje tajnih podataka, morat ćemo ponavljati ovu provjeru u svakoj ruti.

```
@app.put("/tajni_podaci")
def update_tajni_podaci(token: str, podaci: dict):
    if token != "super_secret_admin_token007":
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    # ažuriramo podatke...
    return {"poruka": "Podaci uspješno ažurirani"}

@app.delete("/tajni_podaci")
def delete_tajni_podaci(token: str):
    if token != "super_secret_admin_token007":
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    # brišemo podatke...
    return {"poruka": "Podaci uspješno obrisani"}
```

Možemo jednostavno izdvojiti kôd za provjeru tokena u zasebnu funkciju i **koristiti je kao ovisnost u svakoj ruti**.

```
def provjeri_token(token: str):
    if token != "super_secret_admin_token007":
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    return token
```

Ili možemo simulirati vraćanje korisnika koji se nalazi u bazi podataka na temelju tokena:

```
from pydantic import BaseModel

class Admin(BaseModel):
    korisnicko_ime: str
    token: str

administratori = [
    {"korisnicko_ime": "secret_admin_007", "token": "super_secret_admin_token007"},
    {"korisnicko_ime": "secret_admin_123", "token": "admin_token123"},
    {"korisnicko_ime": "secret_admin_456", "token": "admin_token456"}
]
```

```

def provjeri_token(token: str):
    for admin in administratori:
        if admin["token"] == token:
            return Admin(**admin) # vraćamo instancu Admin klase
    raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")

```

Sada možemo koristiti ovu funkciju kao ovisnost u svakoj ruti koja zahtjeva autorizaciju.

```

@app.get("/tajni_podaci")
def get_tajni_podaci(admin: Admin = Depends(provjeri_token)): # koristimo Depends funkciju
    za "ubrizgavanje ovisnosti"
    return {"tajni_podaci": "šifra za sef je 1234"}

@app.put("/tajni_podaci")
def update_tajni_podaci(podaci: dict, admin: Admin = Depends(provjeri_token)): # "podaci"
    su tijelo HTTP zahtjeva
    # ažuriramo podatke...
    print(f"Podatke ažurirao admin {admin.korisnicko_ime}")
    return {"poruka": "Podaci uspješno ažurirani"}

@app.delete("/tajni_podaci")
def delete_tajni_podaci(admin: Admin = Depends(provjeri_token)):
    # brišemo podatke...
    print(f"Podatke izbrisao admin {admin.korisnicko_ime}")
    return {"poruka": "Podaci uspješno obrisani"}

```

Naravno, **ovo je samo simulacija**, u pravom projektu moramo koristiti stvarnu bazu podataka, sa sigurnim mehanizmima za autentifikaciju i autorizaciju zahtjeva! Primjer implementacije autentifikacijskog servisa možete pronaći u `RS5/examples/e-commerce-app/auth-service`, a za vježbu možete taj servis pokušati pretvoriti u FastAPI mikroservis.

DI se često koristi za potrebe autorizacije i autentifikacije dolaznih zahtjeva te za dijeljenje konekcije na bazu podataka, međutim ima i mnoge druge svrhe o kojima možete više pročitati u FastAPI dokumentaciji na sljedećoj [poveznici](#).

Što se tiče implementacije sigurnosnih mehanizama, FastAPI nude gotove module za autentifikaciju i autorizaciju, kao što su `OAuth2PasswordBearer` i `OAuth2PasswordRequestForm`. Više o tome također možete pronaći u dokumentaciji na sljedećoj [poveznici](#).

4.2 API Router

Osim Dependency Injection sustava, FastAPI nudi i mogućnost strukturiranja kôda kroz `APIRouter` klasu. Slično kao Express.Router u Express.js, `APIRouter` omogućuje grupiranje srodnih ruta i resursa u jednu cjelinu.

Napomena: API Router u FastAPI-u je evivalentan Express.Router objektu u Express.js poslužiteljima ili Blueprint objektu u Flask aplikacijama.

Različite rute je potrebno grupirati u odgovarajuće "podaplikacije" u zasebnim datotekama, unutar zajedničkog direktorija. Direktorij možemo nazvati `routers` ili `routes`.

```
→ mkdir routers
```

Kako bi naglasili da se radi o modulu, možemo dodati praznu `__init__.py` datoteku unutar direktorija.

```
→ touch routers/__init__.py
```

U direktoriju `routers` možemo kreirati zasebne datoteke za svaku grupu ruta. Primjerice, dodajemo rutu za korisnike:

```
# routers/korisnici.py
from fastapi import APIRouter

router = APIRouter() # router je podaplikacija koju instanciramo na isti način
```

Ili dodajemo rutu za knjige:

```
# routers/knjige.py
from fastapi import APIRouter

router = APIRouter()
```

Rute definiramo na identičan način kao i do sada, samo što ih grupiramo unutar `router` objekta.

```
# routers/korisnici.py
from fastapi import APIRouter

router = APIRouter()

@router.get("/korisnici")
def get_korisnici():
    return {"poruka": "Dohvaćeni korisnici"}

@router.post("/korisnici")
def create_korisnik():
    return {"poruka": "Korisnik uspješno kreiran"}
```

odnosno:

```
# routers/knjige.py

from fastapi import APIRouter

router = APIRouter()

@router.get("/knjige")
def get_knjige():
    return {"poruka": "Dohvaćene knjige"}

@router.post("/knjige")
def create_knjiga():
    return {"poruka": "Knjiga uspješno kreirana"}
```

Obzirom da sve rute počinju istim prefiksom (npr. `/korisnici` ili `/knjige`), možemo to naglasiti prilikom definicije `APIRouter` objekta. Tada je potrebno maknuti prefiks iz svake rute unutar datoteke.

```
# routers/korisnici.py

from fastapi import APIRouter

router = APIRouter(prefix="/korisnici")

@router.get("/") # ustvari je /korisnici/
def get_korisnici():
    return {"poruka": "Dohvaćeni korisnici"}

@router.post("/") # ustvari je /korisnici/
def create_korisnik():
    return {"poruka": "Korisnik uspješno kreiran"}

@router.get("/{id}") # ustvari je /korisnici/{id}
def get_korisnik(id: int):
    return {"poruka": f"Dohvaćen korisnik s ID-em {id}"}
```

Ove rute možemo učitati u glavnu aplikaciju koristeći `include_router` metodu.

```
# main.py

from fastapi import FastAPI
from routers.korisnici import router as korisnici_router # uključujemo router iz datoteke korisnici.py
from routers.knjige import router as knjige_router # uključujemo router iz datoteke knjige.py
app = FastAPI()

app.include_router(korisnici_router) # uključujemo rute za korisnike
app.include_router(knjige_router) # uključujemo rute za knjige
```

```
# nastavljamo dalje s definicijom rute na "main" razini
@app.get("/")
def home():
    return {"poruka": "Dobrodošli na FastAPI poslužitelj"}
```

Konačna struktura projekta sada izgleda ovako:

```
.
├── main.py
└── routers
    ├── __init__.py
    ├── korisnici.py
    └── knjige.py
└── models.py
```

Ovako organizirani poslužitelj je čitljiviji, lakši za održavanje i skalabilan. Svaka grupa ruta je odvojena u zasebnoj datoteci, a svaka ruta je odvojena u zasebnoj funkciji.

Više o organizaciji kôda u velikim aplikacijama možete pročitati u FastAPI dokumentaciji na sljedećoj [poveznici](#).

5. WebSockets na FastAPI poslužitelju

FastAPI ima ugrađenu podršku za WebSocket protokol, koji omogućuje dvosmjernu komunikaciju između klijenta i poslužitelja u stvarnom vremenu. WebSocket je koristan za aplikacije koje zahtijevaju brzu razmjenu podataka, poput chat aplikacija, igara ili aplikacija za praćenje uživo.

Na prošlim vježbama smo već vidjeli kako definirati WebSocket klijenta i poslužitelja koristeći `aiohttp` biblioteku. Sada ćemo vidjeti kako definirati **WebSocket poslužitelj** koristeći FastAPI.

Stvorite novo virtualno okruženje i instalirajte `websockets` paket koji ćemo koristiti za implementaciju **WebSocket klijenta**.

```
→ conda create -n fastapi-websockets python=3.10
```

```
pip install websockets
```

Podrška za WebSocket nalazi se unutar `fastapi` paketa, u `WebSocket` modulu:

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
```

Definiranje WebSocket rute je slično definiranju obične HTTP rute, ali koristimo `websocket` dekorator umjesto `get`, `post`, itd.

```
app = FastAPI()

@app.websocket("/ws") # uočite da koristimo .websocket dekorator
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept() # prihvaćamo WebSocket vezu
    try:
        while True:
            data = await websocket.receive_text() # primamo tekstualnu poruku od klijenta
            await websocket.send_text(f"Poruka primljena: {data}") # šaljemo odgovor klijentu
    except WebSocketDisconnect:
        print("Klijent je prekinuo vezu")
```

U ovom primjeru, definirali smo WebSocket rutu na `/ws` putanji. Kada klijent uspostavi vezu, poslužitelj prihvata vezu i ulazi u beskonačnu petlju gdje prima poruke od klijenta i šalje odgovore natrag.

Da bismo testirali naš WebSocket poslužitelj, možemo koristiti `websockets` biblioteku za kreiranje WebSocket klijenta.

```
import asyncio

import websockets

async def websocket_client():
    uri = "ws://localhost:8000/ws"
    async with websockets.connect(uri) as websocket:
        await websocket.send("Pozdrav, FastAPI WebSocket!")
        response = await websocket.recv()
        print(f"Odgovor od poslužitelja: {response}")
asyncio.run(websocket_client())
```

Pokrenite FastAPI poslužitelj:

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

Zatim pokrenite WebSocket klijenta u drugom terminalu:

```
python websocket_client.py
```

Trebali biste vidjeti odgovor od poslužitelja u terminalu klijenta.

Detalje o korištenju WebSocket protokola u FastAPI poslužitelju možete pronaći na sljedećoj [poveznici](#).

Zadatak za vježbu: Razvoj FastAPI mikroservisa za dohvaćanje podataka o filmovima

Implementirajte mikroservis za dohvaćanja podataka o filmovima koristeći FastAPI. Mikroservis treba biti organiziran u zasebnim datotekama unutar direktorija `routers` i `models`. Glavni resurs jesu filmovi, a podatke možete direktno preuzeti u JSON obliku sa sljedeće [poveznice](#).

1. Implementirajte odgovarajuće Pydantic modele za filmove prema atributima koji se nalaze u JSON datoteci.
2. Za svaki atribut filma definirajte odgovarajuće polje u Pydantic modelu.
3. Učitajte filmove iz JSON datoteke i [odradite deserijalizaciju podataka](#), a zatim ih pohranite u *in-memory* listu filmova.
4. Dodajte provjere za sljedeće attribute filma unutar Pydantic modela za film:
 - `Images` mora biti lista stringova (javnih poveznica na slike)
 - `type` mora biti odabir između "movie" i "series"
 - Obavezni atributi su: `Title`, `Year`, `Rated`, `Runtime`, `Genre`, `Language`, `Country`, `Actors`, `Plot`, `Writer`
 - Ostali atributi su neobavezni, a ako nisu navedeni, postavite im zadanu vrijednost
 - Dodajte validacije za `year` i `Runtime` atribut (godina mora biti veća od 1900, a trajanje filma mora biti veće od 0)
 - Dodajte validacije za `imdbRating` i `imdbVotes` (ocjena mora biti između 0 i 10, a broj glasova mora biti veći od 0)
5. Definirajte Pydantic model `Actor` koji će sadržavati attribute `name` i `surname`.
6. Definirajte Pydantic model `Writer` koji će sadržavati attribute `name` i `surname`.
7. Strukturirajte kôd u zasebnim datotekama unutar direktorija `routers` i `models`. U direktoriju `routers` dodajte datoteku `filmovi.py` u kojoj ćete definirati rute za dohvaćanje svih filmova i pojedinog filma po `imdbID`-u i rutu za dohvaćanje filma prema naslovu (`Title`).
8. Za rutu koja dohvata sve filmove, implementirajte mogućnost filtriranja filmova prema `query` parametrima: `min_year`, `max_year`, `min_rating`, `max_rating` te `type` (film ili serija). Implementirajte validaciju `query` parametra.
9. U glavnoj aplikaciji učitajte rute iz datoteke `filmovi.py` i uključite ih u glavnu FastAPI aplikaciju.
10. Dodajte iznimke (`HTTPException`) za slučaj kada korisnik pokuša dohvatiti film koji ne postoji u bazi podataka, po `imdbID`-u ili `Title`-u.
11. Testirajte aplikaciju koristeći generiranu interaktivnu dokumentaciju (Swagger ili ReDoc).

Rješenje učitajte na GitHub i predajte na Merlin, uz pripadajuće screenshotove dokumentacije koja se generira automatski na `/docs` ruti.

Nema univerzalnog rješenja za organizaciju kôda i implementaciju API-ja, a zadaća nosi do 2 dodatna boda ovisno o kvaliteti izrade FastAPI mikroservisa.