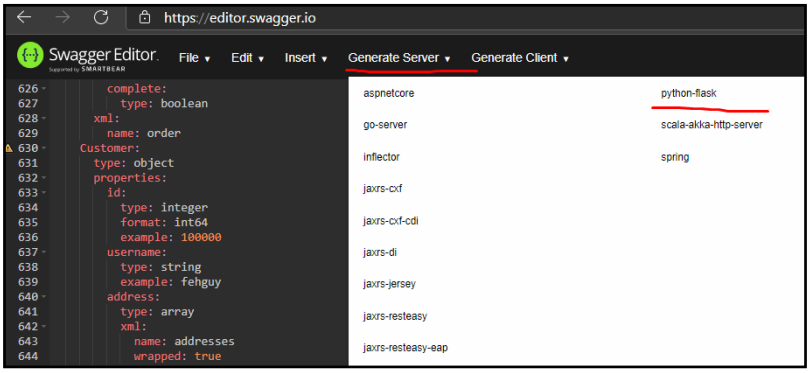
Projektowanie Bezpiecznych Aplikacji, lab4

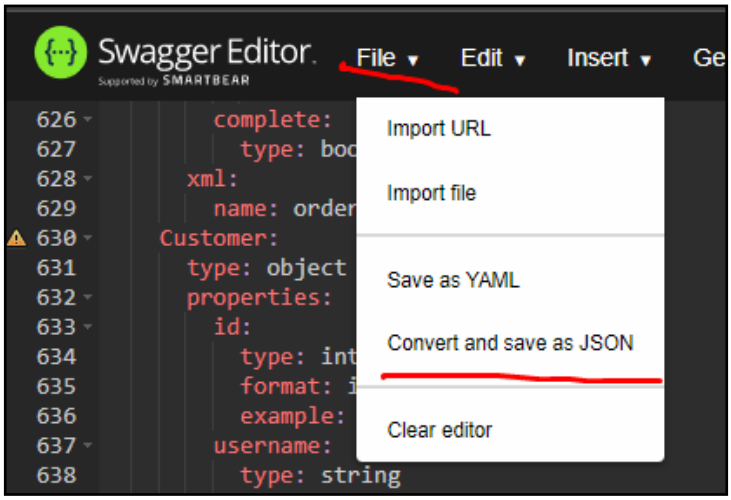
Przemysław Samsel 13.04.21 r.

Zadanie 1.

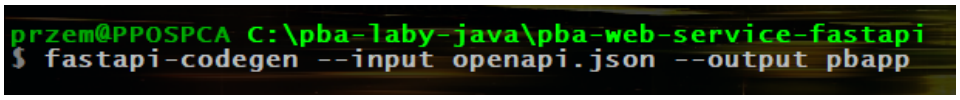
Do wykonania zadania posłużono się frameworkiem webowym FastAPI w języku Python. Pierwszy punkt polegał na wybraniu właściwego frameworka. Trzy najpopularniejsze w tym języku to django, flask oraz właśnie FastAPI. Django jest dojrzałym frameworkiem, jednak nieco skomplikowanym przy prostych projektach (ma „wysoki próg wejścia"), natomiast flask i FastAPI są bardzo do siebie podobnymi z syntaxu, nieco młodszymi, nieco mniej rozbudowanymi. Główna różnica polega na tym, że flask korzysta z interfejsu WSGI (Web Server Gateway Interface), jest nieco starszy, ma większe wsparcie i community. Kod serwera na podstawie pliku yaml można dla niego wygenerować już z samego edytora swaggera:



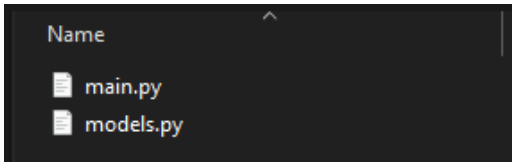
FastAPI zaś, jest dużo młodszym frameworkiem, korzysta z interfejsu ASGI (Asynchronous Server Gateway Interface), pozwalając przez to na jednoczesną obsługę wielu żądań. Jest generalnie szybszy, nieco mniej stabilny i ma odpowiednio mniejsze wsparcie. Jednak jest dynamicznie rozwijającym się projektem idącym w ciekawym kierunku. Wygenerowanie kodu z pliku yaml odbywa się z wykorzystaniem modułu **fastapi-code-generator** (pypi.org/project/fastapi-code-generator/). Najpierw należy jednak, przy użyciu np. serwisu swagger.io przekonwertować specyfikację yaml do standardu json (obowiązkowo należy także przekonwertować yaml do standardu openAPI 3.0, gdyż jest to jedyny format akceptowalny przez **fastapi-code-generator**):



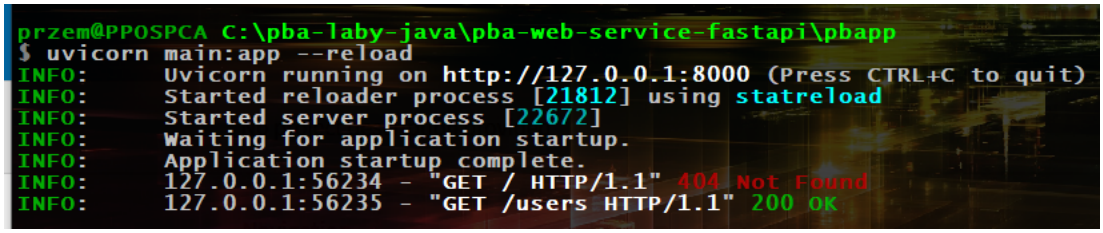
Inną czynnością, którą należało wykonać jest zmiana wszelkich ścieżek relatywnych w dokumencie (np. /user/{userid}) aby były otoczone cudzysłowem (np. „/user/{userid}”) - w przeciwnym wypadku powodowało to błąd w generatorze.
Następnie można już użyć narzędzia **fastapi-codegen** celem wygenerowania plików:



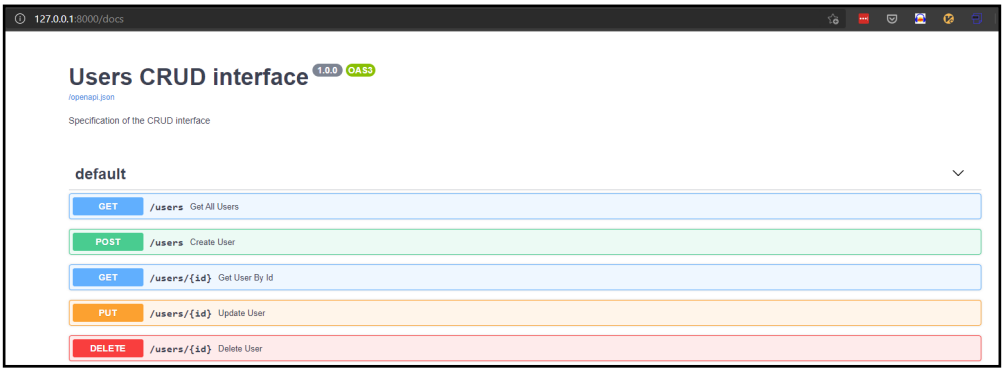
Po wygenerowaniu w bieżącej ścieżce pojawi się folder pbapp, w którym znajdować będą się dwa pliki:



Aby móc uruchomić serwer, należy wprowadzić kilka drobnych modyfikacji. Po pierwsze należy usunąć relatywny import *.models* z pliku *main.py* - w tym momencie nie jest to jeszcze strukturalnie moduł i uniemożliwi to uruchomienie serwera. Po drugie, w pliku *models.py* znajduje się kilka regexów w stylu „[\w-\.]”. Powodują one błąd przy uruchomieniu serwera, gdyż myślnik jest interpretowany przez Pythona w niniejszym regexie nie jako znak, ale jako zakres. Należy zatem zamienić wszystkie podobne wystąpienia myślnika w nawiasach kwadratowych tak, aby znajdował się na pierwszym miejscu „[-\w\.]” - wtedy wszystko powinno działać. Następnie uruchamiamy serwer komendą:

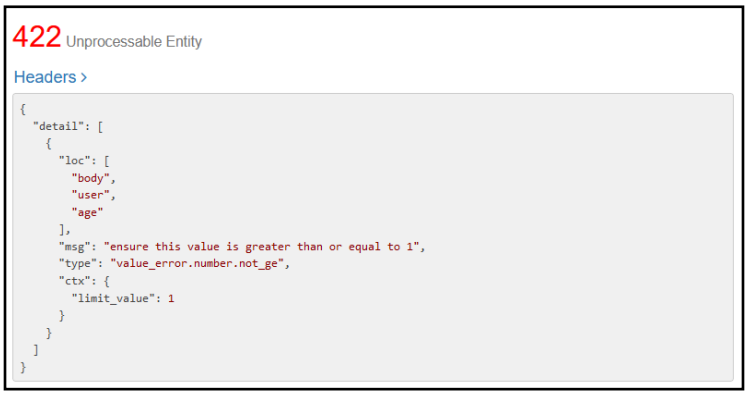


Jak widać zostały także zarejestrowane pierwsze żądania do serwera wykonane z użyciem postmana po uruchomieniu serwera. Zatem modele zostały wygenerowane poprawnie, zaś projekt jest gotowy do użycia. FastAPI udostępnia także szkic swaggera dostępny w ścieżce / docs (jest to wersja okrojona, bazująca na wygenerowanych modelach/istniejącym kodzie):



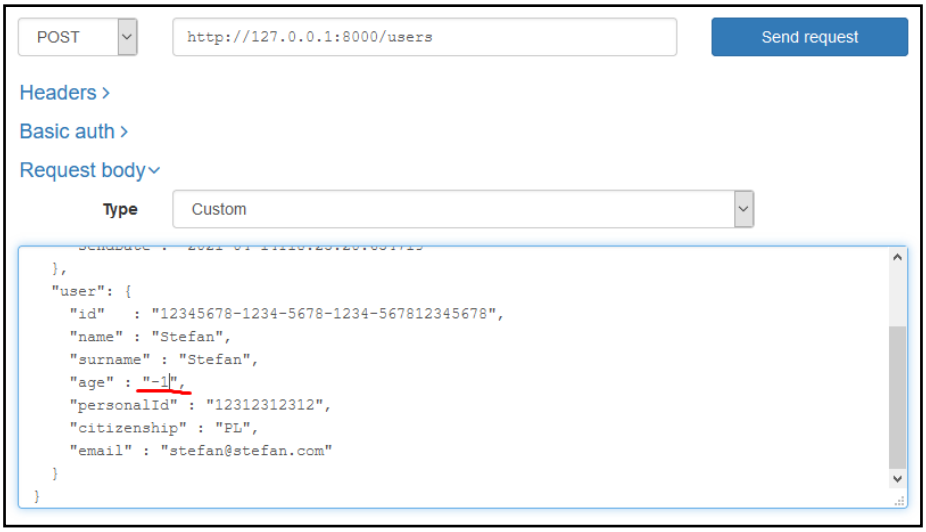
Zadanie 2.

Projekt został umieszczony w repozytorium github.com/PrzemyslawSamsel/FastAPIpba. Została utworzona podstawowa walidacja danych na dwa sposoby - po pierwsze, jako modele o określonych właściwościach utworzone przez bibliotekę Pydantic (pozwala to np. określić granice wieku użytkownika w prosty sposób), a także poprzez użycie dekoratorów (i.e. validator) z tej samej biblitoteki. Przykład błędu dla nieprawidłowego wieku użytkownika przy próbie dodania do bazy (postman):



Jak widać błędy są mocno opisowe, i mogą zawierać pewne wartości pozwalające zidentyfikować framework serwera czy nawet określić z przybliżeniem jego wersję. W środowisku produkcyjnym opisowe błędy są akceptowalne, jednak w przypadku stworzenia gotowej aplikacji należałoby jak najkrócej streścić treść błędu aby uniemożliwić potencjalne ataki.

Przykład requestu użyty do wygenerowania powyższego błędu:



Jest to defaultowy błąd zwrócony przez pydantic. Użycie wspomnianych dekoratorów pozwala na tworzenie customowych wiadomości w przypadku błędów. Przykład dla sprawdzania czy imię użytkownika nie zawiera cyfr:

```
35      @validator('name')
36      def name_cannot_contain_numbers(cls, v):
37          if any(True for e in '1234567890' if e in v):
38              raise ValueError('name cannot contain numbers')
39      #
```

Nieprawidłowe zapytanie będzie generowało w tym wypadku błąd jak poniżej:

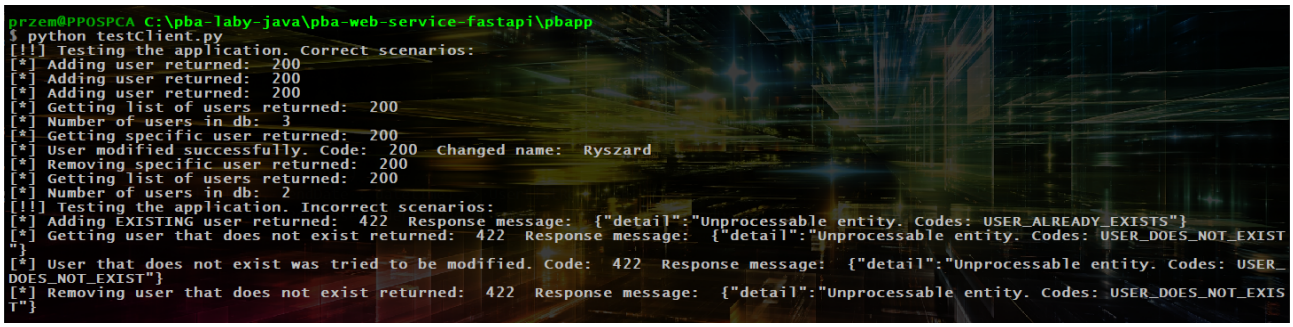
Jak widać w tym przypadku odpowiedz jest już bardziej skąpa (niż w przypadku użycia gotowych modeli Pydantic). Podsumowując, zostały utworzone wszystkie endpointy (GET, PUT, POST, DELETE), gdzie pierwszy endpoint pozwala zaciągnąć zarówno konkretnego użytkownika, jak również całą listę użytjowników (w zależności od parametrów i ścieżki w URL). Każdy z tych endpointów posiada określoną walidację danych wejściowych, oraz sprawdza ważniejsze elementy w konkretnych żądaniach np. przed usunięciem użytkownika czy użytkownik istnieje w bazie, przed modyfikacją podobnie, przed dodaniem czy użytkownik o podanym id nie jest już w bazie itd. Dodatkowo, w przypadku modyfikacji podawany jest parametr userId w ścieżce URL, natomiast samo userId występuje także w przekazywanym obiekcie user. W takim wypadku sprawdzana jest zgodność wspomnianych id, jako że jest to klucz główny w bazie danych, żeby uniknąć sytuacji gdzie user o określonym ID będzie miał w swoim rekordzie inne ID niż do niego kieruje.

Zadanie 2.

W celu weryfikacji działania interfejsu wykorzystany został moduł TestClient z biblioteki FastAPI. Całość kodu znajduje się w pliku testClient.py. Wykonane zostały testy zarówno z poprawnymi danymi, jak również niepoprawne scenariusze np. dodanie nieistniejącego użytkownika do bazy. Testy można uruchomić poprzez komendę:

>> python testClient.py

Otrzymamy wtedy zwrotkę:



Kolejne testy zawierały:

- Dodanie 3 użytkowników do bazy
- Pobranie 3 użytkowników z bazy (get user list)
- Pobranie konkretnego użytkownika z bazy

- Modyfikacja konkretnego użytkownika w bazie (zmiana imienia na Ryszard)
- Usunięcie konkretnego użytkownika
- Pobranie listy aby sprawdzić czy ilość użytkowników w bazie wyniesie 2

Następnie niepoprawne scenariusze:

- Dodanie istniejącego użytkownika do bazy (powoduje 422:USER_ALREADY_EXISTS)
- Pobranie użytkownika który nie istnieje (422:USER_DOES_NOT_EXIST)
- Modyfikacja nieistniejącego użytkownika (422:USER_DOES_NOT_EXIST)
- Usunięcie użytkownika który nie isnieje (422: USER_DOES_NOT_EXIST)