Systemy bezpieczne i FTC

Wiarygodność systemów w pracy zespołowej - jak czynnik ludzki wpływa na różnice w implementacji specyfikacji

Szymon Bagiński
Artur Walasz

Prowadzący: Mgr inż. Tomasz Serafin

Styczeń 2018

Spis treści

1 Wstęp

1.1 Cel projektu

Wykorzystywanie nawet najlepszego oprogramowania, w którym zaimplementowano najbardziej zaawansowane technologie i najbezpieczniejsze algorytmy, nie jest w stanie zapewnić systemowi 100-procentowego bezpieczeństwa. Dzieje się tak dlatego, że w rozwoju i implementacji oprogramowania uczestniczą ludzie, którzy z natury mają skłonność do popełniania błędów. W rezultacie, ludzie, którzy są częścią systemu, zawsze będą najsłabszym ogniwem bezpieczeństwa systemu. Czynnik ludzki jest głównym powodem, dla którego udaje się tak wiele ataków na komputery i systemy.

Celem tego projektu było zbadanie wpływu czynnika ludzkiego na różne implementacje tej samej specyfikacji. W jego ramach stworzono dwie niezależne implementacje oprogramowania z jednolitym, ustandaryzowanym interfejsem, sposobem testowania oraz wymaganiami funkcjonalnymi. Na ich podstawie wyciągnięto wnioski na temat tego, jak indywidualne podejście, interpretacja dostarczonych wymagań oraz wybór technologii wpływają na finalną wersję wytworzonego produktu.

1.2 Realizacja

Punktem wyjściowym do projektu było stworzenie wspólnej specyfikacji wymagań projektowych oraz zbioru testów akceptacyjnych [2] prostego serwisu aukcyjnego w architekturze klient-serwer.

Na potrzeby projektu zdefiniowano specyfikację interfejsu serwisu uwzględniającą następujące funkcje:

- Rejestracja użytkowników
- Logowanie użytkowników
- Tworzenie ofert typu:
  - aukcja: “auction”
  - teraz: “buynow”
- Kupowanie / licytacja towarów (ofert) przez użytkowników
- Przeglądanie ofert z filtracją
  - opis zawiera tekst zadany tekst
  - cena minimalna
  - cena maksymalna
  - ilość dostępnych przedmiotów
- Przeglądanie własnych ofert
- Obsługa własnych ofert:
  - modyfikacja ofert
  - usuwanie ofert

Specyfikacja zawiera opis wszystkich wymaganych endpointów (dodatek A), w tym:

- adres url
- dozwolone metody HTTP
- wymagany format danych, które powinny zostać przesłane w zapytaniach POST
- oczekiwane statusy odpowiedzi w zależności od warunków

Dokładne wymagania dla konkretnych funkcji serwisu zostaną opisane w dalszej części dokumentacji. Porównane zostaną obydwie implementacje pod kątem zgodności z dokumentacją oraz ze sobą wzajemnie.

2 Porównanie implementacji

2.1 Wybór technologii

Podstawową różnicą między dwiema powstałymi implementacjami jest użyta technologia. Każde z rozwiązań posiada oczywiście inny zestaw cech, niektóre rzeczy wykonuje się łatwiej kosztem innych. Każda z nich ma także inne ograniczenia, czy zachowania domyślne. Wybór ten zatem mocno rzutuje na różnice, które będą widoczne przy konkretnych funkcjach systemu, bądź na nakład pracy jaki trzeba włożyć przy spełnianiu konkretnych wymagań.

2.1.1 Implementacja 1

Do pierwszej implementacji postanowiono użyć języka programowania Rust [4]. Jest to kompilowany język stworzony z myślą o tworzeniu stabilnych, bezpiecznych, i przede wszystkim wydajnych aplikacji.

Do stworzenia aplikacji, korzystającej z protokołu HTTP, skorzystano z biblioteki Rocket [3]. Jest to bardzo wygodne rozwiązanie. W połączeniu z menedżerem pakietów Cargo, pierwszą najprostszą aplikację można stworzyć w kilka minut. Rozszerzanie jej o kolejne funkcjonalności również nie przysparza żadnych problemów. Jednocześnie zapewnia ona bezpieczeństwo danej aplikacji, elastyczność a przed wszystkim stabilność (m. in. poprzez "type safety"). W połączeniu z ogólnymi cechami języka Rust daje nam to duże prawdopodobieństwo, że skompilowany program, będzie działał bez problemów przez długi czas.

Kod źródłowy można znaleźć pod poniższym linkiem:

`https://github.com/sbag13/FTC/tree/master/baginski`

2.1.2 Implementacja 2

W drugiej implementacji postanowiono wykorzystać język Python. W języku tym dostępnych jest kilka frame-worków pozwalających na stworzenie aplikacji webowej, są to między innymi:

- Tornado
- Flask
- Django

Na potrzeby projektu zdecydowano się wykorzystać framework Django, który opiera się o wzorzec architektoniczny model-template-view. Zakres implementacji określał jedynie zbudowanie interfejsu w protokole HTTP (Web API), dlatego pełne możliwości fremeworku nie zostały wykorzystane. Niemniej jednak odseparowanie logiki biznesowej od modeli w bazie danych oraz łatwość zarządzenia tymi modelami przeważyła nad wyborem właśnie tego frameworku. W implementacji wykorzystano ponadto narzędzie Django REST framework [5], które w znacznym stopniu ułatwia tworzenie Web API. Django REST framework dostarcza wielu gotowych generycznych widoków, które zapewniają na przykład wzorzec [C]RUD dla danego modelu, czy też pozwalają na wyświetlanie całych list obiektów w formie zdefiniowanej przez dostarczoną klasę Serializera.

2.2 Rejestracja

Mogłoby się wydawać, że tak prosty *endpoint* jak rejestracja nie powinien spowodować dużych rozbieżności. Niemniej jednak niedociągnięcia specyfikacji pozostawiają pole do różnych interpretacji.
Specyfikacja określała, że do rejestracji użytkownika potrzebne jest wysłanie zapytania POST które w swoim ciele zawiera strukturę JSON z dwoma polami: "mail" oraz "password". Wartość pola "mail" musi być oczywiście niepowtarzalna. Istniej zatem możliwość aby użyć jej jako identyfikatora danego użytkownika, zarówno w bazie danych jak i w aplikacji. Postanowiono tak zrobić w implementacji 1., gdzie mail jest kluczem pierwotnym dla użytkownika. W implementacji 2. identyfikatorem użytkownika jest domyślnie klucz <id: int>, na adres email narzucono jedynie warunek jego unikalności.
W implementacji 2. natknięto się na istotny problem. Framework Django dostarcza domyślną obsługę modelu użytkownika. Problem polega na tym, że zdefiniowany specyfikacji sposób reprezentacji użytkownika wymagał wprowadzenia bardzo wielu czasochłonnych zmian. Samo wprowadzenie niestandardowego modelu nie jest skomplikowane, ale dostosowanie pozostałych gotowych wtyczek, które tego modelu używają już tak. Problem zostanie opisany dalej w sekcji 2.3. We frameworku Django dostępna jest klasa AbstractUser którą można wykorzystać w celu zdefiniowania

niestandardowego użytkownika. Podczas rejestracji należało też nadpisać zachowanie się serwera. Domyślnie rejestracja wymagała wysłania zapytania POST ze strukturą JSON z następującymi polami: "username", "email", "password1" oraz "password2". Wynika stąd wniosek, że specyfikacja zniosła domyślne zabezpieczenie obecne w wielu serwisach, które polega na dwukrotnym podaniu jednakowego hasła w celu uniknięcia błędów podczas jego wpisywania. W naszych implementacjach wysłanie zapytania z niezamierzenie błędnym hasłem spowoduje, że użytkownik nie zostanie o tym fakcie poinformowany i nie będzie w stanie zalogować się na swoje konto. Specyfikacja nie zakładała wprowadzenia mechanizmu zmiany haseł użytkowników.
Następnie należy rozpatrzyć przypadek, którego specyfikacja nie określa jednoznacznie. Gdy nie uda się poprawnie zinterpretować przesłanych danych jako JSON, bibliotek Rocket obsłuży taki przypadek wysyłając status 422 (Unprocessable Entity). W implementacji 2, serwer Django zwraca kod 415 wraz z informacją: *'Unsupported media type "text/plain" in request.'* jeśli podano zły nagłówek Content-Type. Jeśli nagłówek jest poprawny, czyli ma wartość "application/json", to kod odpowiedzi jest zgodny z wymienionym w specyfikacji kodem 400 (Bad Request).
Wszystkie wymienione wyżej kody odpowiedzi dla zaistniałej sytuacji są prawidłowe, ponieważ ich interpretacja jest, jak w przypadku wielu innych kodów odpowiedz HTTP, kwestią subiektywną i silnie zależy od indywidualnej interpretacji. Z tego powodu projektując system należy zwrócić szczególną uwagę na zdefiniowanie zachowania Web API dla wszystkich potencjalnych sytuacji, zwłaszcza, jeśli bazując na kodzie odpowiedzi aplikacja wykorzystująca dane API będzie podejmowała konkretne działania.
Specyfikacja określa co powinno się stać gdy zapytanie zostanie wysłane poprawnie za pomocą innej niż POST metody HTTP. W takim przypadku powinien zostać zwrócony status 405 (Method Not Allowed). W przypadku implementacji 1. skutkuje to bardzo dużym nakładem mało istotnej pracy. Rocket, gdy nie uda mu się dopasować żadnego istniejącego endpointu, wysyła domyślnie status 404 (Not Found). Dzieje się tak również w przypadku gdy nie znajdzie odpowiedniej metody HTTP, nawet jeśli reszta parametrów się zgadza. Aby otrzymać zgodność ze specyfikacją należałoby więc utworzyć endpointy dla wszystkich możliwych metod protokołu HTTP, których jedyną odpowiedzialnością byłoby wysłanie błędu o statusie 405. W implementacji 1. zdecydowano się nadpisać tylko metody get, put oraz delete. W implementacji 2. w przypadku wykorzystania generycznych widoków problem jest rozwiązany automatycznie. Wysłanie zapytania z niezdefiniowaną metodą HTTP dla zarejestrowanego adresu url zawsze skutkuje zwróceniem statusu 405, chyba, że zachowanie to zastanie świadomie nadpisane.

2.3 Logowanie

Przy implementacji logowania napotkano problemy opisane w sekcji 2.2. Podczas tworzenia specyfikacji postanowiono, że do uwierzytelniania będą wykorzystywane tokeny JWT (JSON Web Token) [1]. Jest to jednak elastyczna struktura i jej elementy nie zostały jednoznacznie określone. W przypadku implementacji 1. nagłówek JWT został wygenerowany w sposób domyślny, który używa algorytmu HS256, a ładunkiem tokena zostało tylko pole mail danego użytkownika. W żaden sposób nie został określony czas ważności sesji. Rażącym niedopatrzeniem podczas tworzenia dokumentacji jest także fakt, że nie pomyślano o funkcji wylogowywania. Aby usunąć sesję należy więc zalogować się na konto innego użytkownika albo usunąć pliki “cookies” z przeglądarki.
W przypadku implementacji 2. skorzystano z zestawu gotowych aplikacji, m. in.:

- rest_framework_jwt.authentication.JSONWebTokenAuthentication
- allauth.account

Użyto także innych aplikacji, od których powyższe są zależne. Gotowe rozwiązanie dostarcza mechanizmy do generacji tokenów (logowania) jak i też wylogowywania. Tutaj pojawił się wspomniany w poprzedniej sekcji 2.2 problem z modelem użytkownika. Gotowe aplikacje posiadają wiele ustawień, które należało odpowiednio sparametryzować, by autentyfikacja odbywała się za pomocą pola "mail". Domyślnie wykorzystywanym polem jest pole "username", które pierwotnie zostało całkowicie usunięte. Aplikacje o których mowa, pomimo że potrafią za pomocą odpowiednich ustawień zmienić metodę autentyfikacji na pole "mail", do prawidłowego działania wciąż potrzebowały pola "username". Znalezienie problemu i jego rozwiązania było najcięższym problemem w implementacji 2. Projekt był przepisywany trzykrotnie, aby sprostać postawionym w specyfikacji wymaganiom. Finalnie pole username jest sztucznie dodawane do modelu użytkownika, ale zawsze jest ono puste i nigdzie nie jest wykorzystywane.
W implementacji 2. nie przewidziano umieszczania tokena JWT w plikach “cookies” przeglądarki, ponieważ wszystkie testy wykonywane były z poziomu frameworku pytest ustawiając nagłówek "Authorization".

2.4 Tworzenie ofert

W specyfikacji zdefiniowane zostały dwa typy ofert: aukcja ("auction") oraz kup teraz ("buynow"). Wspólny jest dla nich adres endpointu "/offers". Według specyfikacji zapytanie wysłane pod ten adres metodą GET powinno zwrócić filtrowaną listę ofert (dokładny opis w sekcji 2.6.1). Zapytanie metodą POST z odpowiednim dla danego typu oferty ciałem powinno skutkować utworzeniem nowej oferty i zwrócenie kodu odpowiedzi 201.

Przykładowe dane do utworzenia oferty typu "kup teraz":

```
{
    "type": "buynow"
    "description" : "some description",
    "price" : 25.99,
    "amount": 15
}
```

Przykładowe dane potrzebne do utworzenia oferty typu "aukcja":

```
{
    "type": "auction"
    "description" : "some description",
    "price" : 25.99,
    "date": 1547307780
}
```

W implementacji 1. parametr price został uznany za obligatoryjny. Pole to oznacza cenę wyjściową licytacji lub cenę przedmiotu w przypadku oferty typu "buynow". W implementacji 2. pole "price" podczas tworzenia oferty jest opcjonalne. Jego podanie skutkuje ustawieniem pola "last_bid". Przechowywane i zwracane dane szczegółowe konkretnych ofert zostały opisane w sekcji 2.6.2. W specyfikacji nie napisano wprost, ale wynika to z listy testów akceptacyjnych, że zarówno pole "price" jaki i "amount" nie może być ujemne. W implementacji 2. zdroworozsądkowo założono też, że "amount" - czyli dostępna liczba sztuk towaru, musi być liczbą naturalną (typu integer). W implementacji 1. takich założeń nie poczyniono.

Specyfikacja określa ponadto, że próba utworzenia nowej oferty przez nie zalogowanego użytkownika powinna się nie powieść, a odpowiedź powinna zwracać status 401 - Unauthorized.
W implementacji 2. została ustawiona domyślna klasa uprawniań *rest_framework.permissions.IsAuthenticated*. Dzięki temu automatycznie wszystkie operacje wymagają uwierzytelnienia, a wyjątkowe uprawnienia zostały nadane osobno dla poszczególnych operacji, które tego wymagały. Przykładem są zapytania o listę lub szczegóły ofert. Dla tych operacji zostały ustawione uprawnienia *IsAuthenticatedOrReadOnly*, które pozwalają na swobodny odczyt danych, ale zabraniają ich modyfikacji nie zalogowanym użytkownikom. W implementacji 1. uprawnienia są sprawdzane wyłącznie w przypadkach opisywanych przez specyfikację i nie ma ustawionych żadnych domyślnych zachowań.

W specyfikacji wymieniono przypadek zwrócenia kodu 403 - Forbidden - kiedy użytkownik jest zalogowany na nieuprawnione konto. W obu implementacjach taki status nie jest nigdzie zwracany, ponieważ specyfikacja nie określiła jednoznacznie co ten fakt oznacza. W systemie nie przewidziano żadnych ról z niestandardowymi uprawnieniami dla poszczególnych użytkowników (grup użytkowników).

Specyfikacja określa kod odpowiedzi 400 (Bad Request) i podaje przykłady:

- niepoprawny JSON
- brakujące lub nadmiarowe pola

W obu implementacjach niepoprawny JSON jest obsługiwany automatycznie, jak zostało to opisane w sekcji 2.2. Brakujące pola są wyłapywane przez obiekty serializujące, które jednoznacznie określają, jak powinny się nazywać wymagane oraz opcjonalne pola. Nie został jednak rozwiązany problem z polami "nadmiarowymi", ponieważ te są przez serializer odrzucane na etapie weryfikacji i nie są dalej przetwarzane. Takie zachowanie wymagałoby nadpisywania domyślnych generycznych widoków, dlatego zostało zaniechane, ponieważ mechanizm weryfikacji i czyszczenia danych przez

obiekty Serializerów we frameworku Django był w tym przypadku satysfakcjonujący i dodatkowe pola nie wpływały w żaden sposób na bezpieczeństwo danych w bazie.

2.5 Kupowanie / licytacja

W obydwu implementacjach nie stwierdzono żadnych rozbieżności w stosunku do specyfikacji dla endpointu `/offers/id/buy`.
W przypadku implementacji 1. informacje o dokonanych zakupach oraz o licytacjach są przechowywane w jednej tabeli bazy danych `transactions`. Każda transakcja posiada pola: `id`, `offer_id`(klucz obcy do oferty), `buyer`(klucz obcy do użytkownika), `amount`(opcjonalne pole dla offert typu "buy-now"), `bid`(opcjonalne pole dla ofert typu "auction").
W implementacji 2. pominięto mechanizm przechowywania informacji o dokonanych transakcjach - potraktowano go jako wykraczający poza wymagania specyfikacji. W implementacji zadbano o zabezpieczenia wynikające o założonych testów akceptacyjnych. Zabronione jest kupowanie przedmiotów w ofertach, których zalogowany użytkownik jest właścicielem. Nie jest też możliwe ponowne licytowanie aukcji, którą dany użytkownik aktualnie wygrywa.

2.6 Przeglądanie ofert

2.6.1 Lista ofert wraz z filtrowaniem

W odpowiedzi na zapytanie GET `/offers` serwer powinien zwracać listę ofert dla każdego użytkownika - nie zależnie od tego, czy jest on zalogowany czy nie. Takie wymagania zostały zdefiniowane w testach akceptacyjnych, ze specyfikacji nie wynika to wprost. Specyfikacja nie określa też w żaden sposób formy w jakiej ma być zwracana lista ofert. Jedyna klarowna informacja na ten temat to założenie, że jeśli użyte zostaną filtry i żadna z ofert nie będzie spełniała wymagań, to zwrócona zostanie pusta lista `"[]"`.

W implementacji 2. zastosowano dziedziczenie klas ofert "aukcji" i "kup teraz" po wspólnej klasie oferty. Klasa ta zawiera wspólne pola takie jak id. operacji, opis "description" oraz pole zawierające informację o id właściciela. Do zwracania listy wykorzystano serializator, który zwraca jedynie id oferty, jej opis oraz typ. Założono, że informacje szczegółowe będą dostępne za pośrednictwem endopintu "/offers/<offer_id>", opisanego w sekcji 2.6.2. Przykładowa odpowiedź wygląda więc następująco:

```
[
    {
        "id": 1,
        "type": "auction",
        "description": "Awesome item"
    },
    {
        "id": 2,
        "type": "buynow",
        "description": "best items"
    },
    ...
]
```

W implementacji 1. każda oferta w odpowiedzi zawiera jeszcze pole "amount" bądź "date", zależnie od typu oferty.

Specyfikacja definiuje możliwe do wykorzystania w adresie url filtry:

- "contains" - wyrażenie które będzie szukane w opisie oferty
- "price_min" - cena minimalna (w przypadku aukcji ostatnio licytowana kwota)
- "price_max" - cena maksymalna (w przypadku aukcji ostatnio licytowana kwota)
- "type" - typ oferty, "auction" lub "buynow"
- "created_by_me" - jeśli ustawiony, zapytanie powinno zwracać tylko oferty których zalogowany użytkownik jest właścicielem

W implementacji 1. napotkano nieprzewidziane komplikacje spowodowane faktem, że słowo "type" jest słowem kluczowym języka Rust. Niemożliwym było więc skorzystanie z automatycznego

przypisywania parametrów do zmiennych, jako że nie da się zadeklarować zmiennej `type`. Rozwiązanie w tym przypadku wymagało obudowania wartości w formularz. Opracowanie rozwiązania pochłonęło znaczną ilość czasu.

W implementacji 2. wykorzystano obiekty Django Q (querry) [6], które pozwalają na wygodne budowanie zapytania wykorzystanego do filtracji listy zwracanej przez bazę danych. Obiekty te pozwalają na elastyczne łączenie warunków, dzięki czemu złożony filtr jest aplikowany tylko raz.

Specyfikacja nie podaje dosłownie jaki warunek powinien być sprawdzany dla filtrów "price_min" i "price_max", to znaczy czy ceny mają być większe (mniejsze), czy większe (mniejsze) lub równe. Biorąc pod uwagę, że nazwy filtrów zawierają w sobie "max" w implementacji 2. zdecydowano się na warunek większy lub równy (mniejszy lub równy). Inaczej jest w przypadku implementacji 1., gdzie warunki nie zawierają w sobie równości.

2.6.2 Szczegóły oferty

Dla ednpointu /offers/<offer_id> specyfikacja definiuje model RUD (Retrive-Update-Delete). Wymienione są zachowania (w tym oczekiwane statusy odpowiedzi) dla poszczególnych metod HTTP: GET, PATCH oraz DELETE.

Metoda GET powinna zwracać szczegóły na temat konkretnej oferty. W przypadku oferty typu "kup teraz" specyfikacja podaje wymagane pola:

- "type" - powinien być równy "buynow"
- "description" - opis oferty
- "price" - cena za sztukę
- "amount" - liczba dostępnych sztuk

Dla wygody w implementacji 2. odpowiedź jest dodatkowo uzupełniana o "id" konkretnej oferty. W przypadku implementacji 1. zwracane są tylko pola wymienione w specyfikacji. W implementacji 2. pole zawierające informacje o właścicielu oferty pozostaje ukryte, ponieważ nie wymagała tego specyfikacja:

```
{
    "id": 186,
    "type": "buynow",
    "description": "Actually piece apply.",
    "amount": 10,
    "price": "566.00"
}
```

Dla oferty typu "aukcja" specyfikacja podaje wymagane pola:

- "type" - powinien być równy "auction"
- "description" - opis oferty
- "status" - określa, czy oferta jest aktualna: "active" lub "expired"
- "last_bid" - ostatnio licytowana kwota
- "customer_id" - Id użytkownika, który aktualnie wykrywa daną licytację
- "expiration_ts" - timestamp, który określa termin zakończenia aukcji

W implementacji 2. odpowiedź tak samo jak dla oferty typu "kup teraz" jest uzupełniana o "id" konkretnej oferty, co również nie ma miejsca w przypadku implementacji 1. W związku z niejednoznacznie określoną operacją tworzenia oferty (specyfikacja podaje pole "price" które odpowiada cenie wyjściowej aukcji") w implementacji 2. przyjęto, że jeśli podana zostanie cena wyjściowa, to zostanie ona zwracana jako pole "last_bid". Pole "customer_id" pozostaje wtedy puste (null). W implementacji 1. uznano pole “price” za obligatoryjne, z racji na następujący znak ’*’ po nazwie, więc pole “last_bid” zawsze zwróci jakąś wartość jeśli oferta istnieje. Natomiast pole “customer_id” będzie pustym ciągiem znaków(""), jeśli nikt jeszcze nie licytował danego przedmiotu. Jest to następstwem tego, że w implementacji 1. identyfikatorem użytkownika jest jego mail, także w bazie danych.

Implementacja 1:

```
{
    "type":"auction",
    "description":"dsc",
    "status":"expired",
    "last_bid":32.5,
    "customer_id":"",
    "expiration_ts":64
}
```

Implementacja 2:

```
{
    "id": 184,
    "type": "auction",
    "description": "Executive feel course them low memory.",
    "date": 1547552089,
    "status": "active",
    "last_bid": "942.00",
    "customer_id": null
}
```

Na powyższej przykładowej odpowiedzi widać ponadto, że popełniony został błąd w nazwie pola "expiration_ts". Błąd ten wynika z przeoczenia, to samo pole w momencie tworzenia aukcji nazywało się "date". W implementacji 2. pole to pełni tą samą zakładaną funkcję, ale jest nie zgodne ze specyfikacją.
Na tym błędzie można wysnuć kolejny istotny wniosek. Tworząc specyfikację należy zwrócić uwagę na konflikty w nazwach pól. Przemienne nazywanie tych samych danych w różny sposób może prowadzić (i na ogół prowadzą) do niespójności w bazie danych oraz w wysyłanych informacjach. Ma to szczególne znaczenie w rozbudowanych systemach, gdzie w firmie wydzielone są zespoły zajmujące się różnymi odseparowanymi częściami systemu, które komunikują się ze sobą za pomocą interfejsu http (lub innego). Bardzo często dochodzi do sytuacji, że to samo określenie w innym dziale oznacza zupełnie inne operacje / dane. Takie sytuacje prowadzą do tego, że błędy są wychwytywane dopiero w późnym etapie projektu, w momencie integracji poszczególnych części systemu.

Metody PATCH i DELETE dla `/offers/{id}` nie przysporzyły żadnych problemów i działają w sposób intuicyjny, zgodny z dokumentacją i przypadkami testowymi. Niemniej jednak specyfikacja nie precyzuje, że tylko właściciel oferty może ją zmieniać, ale można to wywnioskować z opisu testów systemu.

3 Podsumowanie

Celem projektu było pokazanie jak bardzo różne mogą powstać implementacje systemu opartego o tę samą specyfikację. Jak widać duże rozbieżności mogą powstać nawet w przypadku prostego systemu, składającego się z kilku funkcji. Główną przyczyną jest oczywiście czynnik ludzki. Na przykład preferencje programistów oraz wybór technologi prowadzi do sytuacji, w których korzysta się z gotowych rozwiązań dostarczanych przez frameworki. Może to skutkować między innymi różnymi zachowaniami w sytuacjach nie przewidzianych przez specyfikację, bądź nawet zmianą funkcji z uwagi na alternatywny nakład pracy. Opisany efekt jest dodatkowo potęgowany niską jakością wykonanej specyfikacji systemu. Na jej stworzenie poświęcono zbyt mało czasu. Nie przewidziano wielu problematycznych przypadków, a nawet zabrakło określenia tak podstawowych spraw jak na przykład wartości graniczne podczas filtrowania. Błędy koncepcyjne i niedociągnięcia, które powstały na pierwszym etapie mają tym większe odzwierciedlenie w powstałych różnicach między produktami końcowymi, które są zaskakująco wydatne.

Odwołania

[1] *JSON Web Tokens* [online], Data dostępu: 12.01.2019.
`https://jwt.io/`

[2] *Opis testów funkcji systemu* [online], Data dostępu: 12.01.2019.
`https://drive.google.com/drive/folders/194qvJLmUYSD427WfbAkm365Q3SCY1ZaY`.

[3] *Rocket - Simple, Fast, Type-Safe Web Framework for Rust* [online], Data dostępu: 12.01.2019.
`https://rocket.rs/`.

[4] *Rust* [online], Data dostępu: 12.01.2019.
`https://www.rust-lang.org/`.

[5] *Django REST framework* [online], Data dostępu: 12.01.2019.
`https://www.django-rest-framework.org/`.

[6] *Django Q* [online], Data dostępu: 12.01.2019.
`https://docs.djangoproject.com/en/2.1/topics/db/queries/`.

A Interfejs serwisu aukcyjnego

```
 /registration
POST
{
    mail:     string,
    password: string
}

201 - Created (pomyślne utworzenie użytkownika)
400 - Bad Request (np. zły email)
409 - Conflict (konto istnieje)
500 - Internal Server Error
503 - Server Unavaiable (powód podany w opisie)

not POST
405 - Method Not Allowed
-----------------------------------------------------------------
/login
POST
{
    mail:     string,
    password: string
}

200 - OK, (JWT token in response)
401 - Unauthorized
404 - Not Found
500 - Internal Server Error
503 - Server Unavaiable (powód podany w opisie)

not POST
405 - Method Not Allowed
-----------------------------------------------------------------
/offers
POST
{
    type*:        [auction|buynow],
    description*: string,
    price*:       float, (cena min. - akcja; cena za sztukę - kup teraz),
    date*:        timestamp, (sekundy, tylko dla aukcji),
    amount*:      int (tylko dla kup teraz)
}

201 - added
{
    offer_id: int
}
400 - Bad Request (niepoprawny JSON, brakujące lub nadmiarowe pola)
401 - Unauthorized (niezalogowany użytkownik)
403 - Forbidden (zalogowany na nieuprawnione konto)
500 - Internal Server Error
503 - Server Unavaiable (powód podany w opisie)

GET
Dozwolone filtry w url:
- contains - pole description zawiera ciąg znaków
- price_min - minimalna cena towaru (i aukcji)
- price_max - maksymalna cena towaru (i aukcji)
- type - [auction/buynow] - typ oferty
- created_by_me: boolean

200 - (oferty w odpowiedzi - może być pusta "[]")
400 - Bad Request (niepoprawny filtr w url)
500 - Internal Server Error
503 - Server Unavaiable (powód podany w opisie)
-----------------------------------------------------------------
/offers/{id}
PATCH
{
    **fields_to_modify...
}

użytkownik może modyfikować:
```

```
- price, amount, description - dla ofert typu "buynow"
- price, description - dla ofert typu "auction"

202 - Accpepted
400 - Bad Request
403 - Unauthorized
404 - Not Found (nie znaleziono ofert)

DELETE
(no payload)
202 - Accepted
403 - Unauthorized
404 - Not Found (nie znaleziono oferty)

GET
(no payload)
200 - Ok (w odpowiedzi szczegóły aukcji)
{
     type:          "auction",
     description:   "opis",
     status:        [active / expired],
     last_bid:      float,
     customer_id:   int,
     expiration_ts: <timestamp>
}
200 - Ok (szczegóły oferty "buynow")
{
     type:        "buynow",
     description: string,
     price:       int,
     amount:      int
}
404 - Not Found
-----------------------------------------------------------------
/offers/{id}/buy
POST
{
     bid:    int (auction)
     amount: int (buynow)
}

202 - Accepted
400 - Bad Request (niepoprawny JSON, brakujące lub nadmiarowe pola)
401 - Unauthorized (niezalogowany użytkownik)
409 - Conflict:
     409 - {"minimal_bid": <minimalny bid>}
     409 - {"max_amout": <ilość dostępnych produktów>}
     409 - {"status": "expired"}
     409 - {"conflict": "unable to order own items"}
     409 - {"conflict": "you can not bid on the auction you win"}
500 - internal server error
503 - Server Unavaiable (powód podany w opisie)
```