Uniwersytet
im. Adama Mickiewicza
w Poznaniu

Wydział Matematyki i Informatyki

Maciej Głowacki

Numer albumu: 434689

Filip Izydorczyk

Numer albumu: 434700

Marcin Woźniak

Numer albumu: 434812

Hubert Wrzesiński

Numer albumu: 434813

PlanNaPlan aplikacja wspomagająca proces rejestracji studentów na zajęcia

PlanNaPlan application supporting student registration for classes

Praca inżynierska na kierunku **informatyka**

napisana pod opieką

dr. Tomasza Piłki

Poznań, luty 2021

Spis treści

Streszczenie

Niniejsza praca dyplomowa opisuje technologie używane do tworzenia nowoczesnych aplikacji webowych, które zostały użyte w trakcie projektu inżynierskiego PlanNaPlan oraz spektrum zagadnień pozatechnicznych związanych z tworzeniem aplikacji.

Pracę rozpoczyna się od zapoznania z mechanizmami zapewnienia bezpieczeństwa aplikacji webowej oraz tworzonej wokół niej infrastruktury. Uzasadniony zostaje wybór Gentoo Linux w funkcji serwera proxy oraz definicja i zgłębienie tematu serwera pośredniczącego.

Następnie przybliżona zostaje definicja REST API wraz z przykładową implementacją w frameworku Spring Boot. Zostaną poruszone między innymi problemy Dependency Injection (DI), testowania endpointów tworzonego API jak i jego zabezpieczania. Na zakończenie rozdziału przytoczone zostaną narzędzia służące do przygotowywania wartościowej dokumentacji projektowej.

Trzeci rozdział opiera się na przedstawieniu definicji i zalet Aplikacji Single Page, a także wskazaniu różnic między trzema najpopularniejszymi frameworkami służącymi do budowania SPA: Angular, React oraz Vue.

Ostatni rozdział przedstawia zbiór pozatechnicznych wymogów dobrego systemu informatycznego, który poza wzorową architekturą prezentuje wartość dla jego użytkowników, co wyrażone jest przez dobre UX i UI aplikacji. Uargumentowana zostaje również wartość języka programowania Typescript w tworzeniu dobrego Developer Experience (DX).

Abstract

This diploma paper describes technologies used for building modern web application that were used during engineering project PlanNaPlan, as well as broad range of non-technical concepts related to web application development.

The paper starts with introduction to mechanisms responsible for providing security of web application and web application infrastructure. Followingly, usage of Gentoo Linux as proxy server is justified and topic of proxy server is further explored.

Afterwards REST API definition is introduced together with accompanying example implementation in Spring Boot framework. Problems of Dependency Injection, testing of created API endpoints' and security of API are also described in great detail. At the end of the chapter presented are tools for creating valuable project documentation.

Third chapter draw upon definition and benefits of Single Page Application. Differences of three most popular frameworks: Angular, React and Vue are presented and compared.

Last chapter introduces a collection of non-technical requirements for well-built information system. Apart from successful architecture, information system has to present value for every user. It is expressed through well-crafted UX and UI. There is also description of Typescript programming language and positive impact of this language in improving Developer Experience (DX).

Wstęp

Każdy semestr na uczelni wyższej rozpoczyna się od ułożenia planu zajęć. Nie da się ukryć, że jest to jeden z ważniejszych elementów poszczególnych etapów studiowania, ponieważ sam plan ma duży wpływ na to jak porządkujemy swoje życie studenckie wraz z życiem zawodowym i towarzyskim. Dla wielu studentów odgrywa to kluczową rolę, ponieważ dodatkowa praca jest często niezbędna do finansowego utrzymania się na studiach. Poza tym studenci próbują też znaleźć czas na realizację swoich zainteresowań i w tym wypadku dobrze ustalony plan okazuje się niezbędny. Ponadto pomaga wyznaczyć rytm i jest niezbędny dla sprawnego funkcjonowania uczelni, jej pracowników oraz studentów. Postać w jakiej dostaniemy nasz plan zajęć zależy w dużej mierze od uczelni, na jaką uczęszczamy. Nie musimy daleko szukać, ponieważ na sąsiednich wydziałach Uniwersytetu Adama Mickiewicza, jak na przykład wydziale Psychologii, czy wydziale Prawa, studenci mają wpływ na zajęcia na jakie uczęszczają. Jednak w procesie zapisów dużą rolę odgrywa czynnik losowy - odbywają one się przy pomocy systemu "kto pierwszy, ten lepszy". Natomiast inaczej wygląda to na Politechnice Poznańskiej, gdzie plan dostajemy odgórnie, a student nie ma żadnego wpływu na to, kiedy odbywają się jego ćwiczenia, wykłady czy laboratoria. Istnieją pojedyncze przedmioty, które rzeczywiście są obieralne, jednak nawet wtedy wybieramy go całą grupą, a nie pojedynczo oraz nie możemy wybrać godziny, w której odbywają się zajęcia. W porównaniu do innych wydziałów UAM oraz innych poznańskich uczelni, wydział Matematyki i Informatyki UAM daje nam stosunkowo więk-

szą swobodę. Czy to poprzez zmieniający się z semestru na semestr zestaw przedmiotów obieralnych, czy też poprzez możliwość wyboru terminu poszczególnych zajęć. Przykładowo mamy możliwość zaliczenia ćwiczeń poprzez uczęszczanie na zajęcia w jednym z dostępnych terminów.

Nie jest dużym zaskoczeniem więc, że przy tak szerokim zagadnieniu pojawiło się dużo możliwości usprawnienia oraz optymalizacji procesu zapisów na zajęcia. Na Wydziale Matematyki i Informatyki z pomocą przyszli studenci, którzy stworzyli aplikację umożliwiającą rejestrację na wybrane przedmioty.

Technologie informatyczne mają jednak to do siebie, że użyte rozwiązania i design jednego dnia są u szczytu swej popularności, a następnego odchodzą w zapomnienie. Stają się przestarzałe i wychodzą z użytku. Trzeba nieustannie czuwać nad rozwojem technologii i dostosowywać aplikację do zmieniających się potrzeb użytkowników, w przeciwnym wypadku aplikacja może przestać spełniać swoją pierwotną rolę a z czasem nawet irytować.

Aplikacja do zapisów na zajęcia używana przez studentów wydziału Matematyki i Informatyki liczy sobie już kilkanaście lat. Przez ten czas zebrała duży dług technologiczny. Niemożliwy stał się jej dalszy rozwój, tak by spełnić pojawiające się oczekiwania zarówno studentów jak i pracowników uczelni. Stąd też, na podstawie doświadczeń naszych oraz innych studentów powstał pomysł stworzenia jej nowej, odświeżonej wersji, która rezygnuje z przestarzałych nawyków programistycznych.

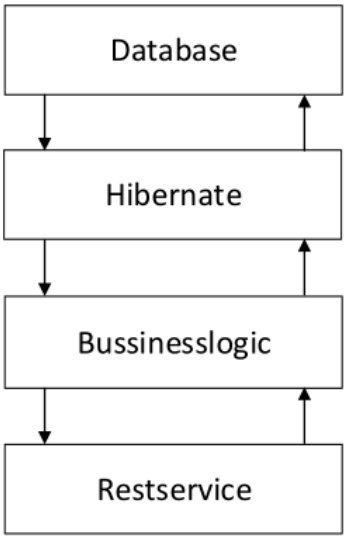
Efektem tego pomysłu jest PlanNaPlan, projekt mający na celu rozwiązanie dotychczasowych problemów związanych z procesem rejestracji, a także dodanie nowych funkcjonalności, takich jak: integracja z systemami USOS [1][1] oraz CAS [2][2] , możliwość edycji danych, przeglądanie historii i edycja planów zajęć przez dziekanat, a także transfer zajęć między studentami bezpośrednio z poziomu aplikacji.

[1] USOS - Uniwersytecki System Obsługi Studiów

[2] CAS - Central Authentication Service - Centralny System Uwierzytelniania

Warto tutaj zaznaczyć, że zadania tego podejmowały się już inne zespoły studentów, jednakże nigdy nie udało się im doprowadzić projektu do końca. Pokazuje to z jak trudnym zadaniem przyszło nam się zmierzyć.

Architektura projektu dzieli się na backend i frontend. Na backend składają się: baza danych oraz aplikacja REST udostępniająca funkcjonalności i komunikująca się z bazą danych. W celu jak rozdzielenia odpowiedzialności kodu backend został podzielony na dwa pakiety: restservice - udostępniający użytkownikowi restowe end-pointy, a nadto zajmujący się ich zabezpieczeniem, oraz businesslogic - udostępniający model danych i komunikujący się z bazą. Komunikacja ta odbywa się za pomocą Hibernate'a dostarczonego przez paczkę spring boot, która jest podstawą na której stworzony został cały backend. W naszym projekcie używamy relacyjnej bazy MariaDB [3][3].

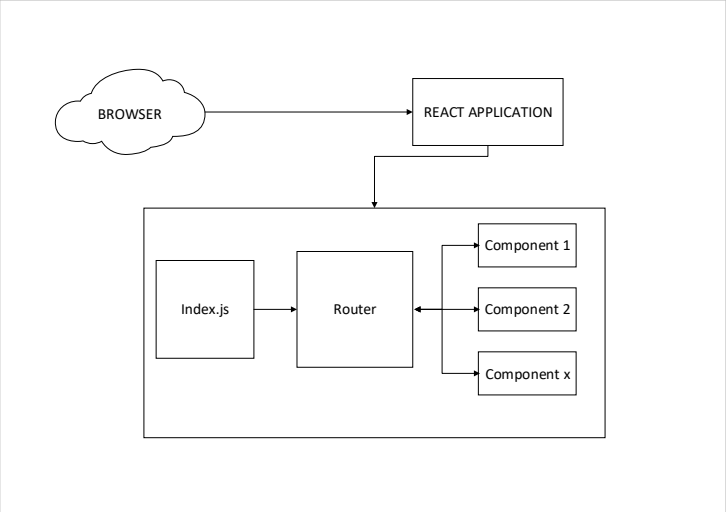


Rysunek 1: Architektura Backendu
Źródło: Opracowanie własne

Natomiast frontend został napisany w języku programowania Typescript, który zapewnia statyczne typowanie i korzysta z frameworku React. Oparty

[3] MariaDB - system zarządzania relacyjną bazą danych

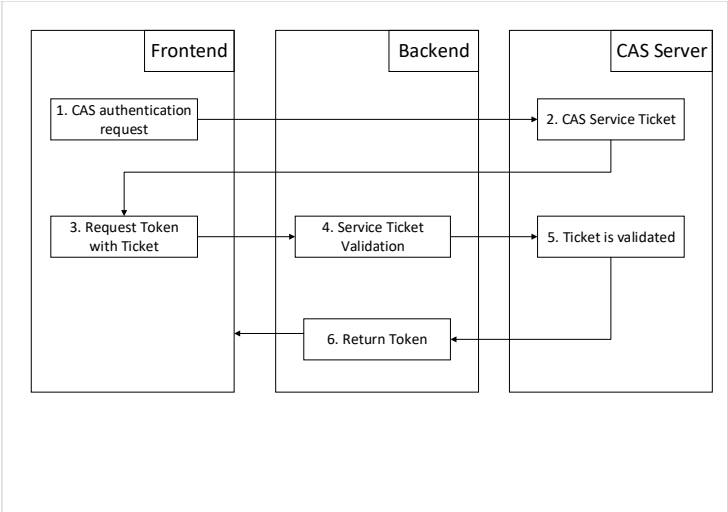
jest na architekturze re-używalnych komponentów, które komunikują się między sobą oraz backendem za pomocą providerów[4] oraz wykorzystuje paradygmat programowania deklaratywnego. Dane dotyczące autoryzacji oraz roli użytkownika przechowywane są w Local Storage przeglądarki. Do ostylowania aplikacji użyto rozwiązanie CSS-in-JS - Styled Components.



Rysunek 2: Architektura Frontendu
Źródło: Opracowanie własne

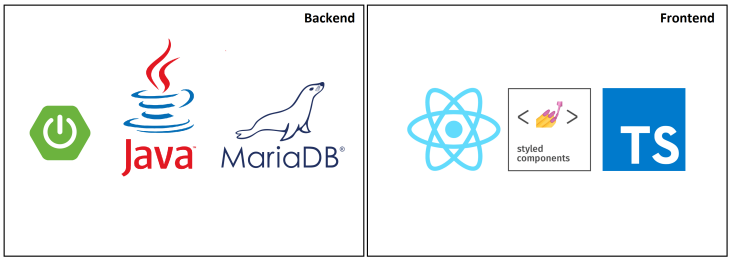
Ponadto autoryzacja jest zintegrowana z systemem CAS udostępnionym przez uczelnię. Gdy użytkownik wchodzi do aplikacji, frontend przekierowuje go do zalogowania się przez CAS. Po pomyślnej autoryzacji wraca na naszą stronę z ticketem. Tenże ticket jest następnie wysyłany na backend, który sprawdza jego poprawność i dostaje w odpowiedzi tożsamość użytkownika, który się zalogował. Po pomyślnym otrzymaniu tożsamości odsyłamy na frontend token, który jest podstawą do wykonania wrażliwych akcji wewnątrz naszego systemu takich jak np. zapis planu.
Niniejsza praca została podzielona na: wstęp, rozdziały techniczne, oraz podsumowanie.

[4] Provider - służy do zarządzania globalnym stanem aplikacji



Rysunek 3: CAS architektura

Źródło: Opracowanie własne



Rysunek 4: Użyte technologie

Źródło: Opracowanie własne

Pierwszy rozdział porusza tematykę systemów operacyjnych, serwisów, infrastruktury oraz bezpieczeństwa. Infrastruktura została podzielona na trzy części: zewnętrzną, wewnętrzną oraz logiczną. Opisana została również specyfika serwera proxy, a także inne elementy składające się na ogólnie rozumiane bezpieczeństwo całej infrastruktury. Rozdział został napisany przez Marcina Woźniaka.

W drugim rozdziale poruszona zostanie tematyka tworzenia REST API z wykorzystaniem technologii dostarczonych przez Spring. Omówione zo-

staną mapowanie obiektowo-relacyjne za pomocą Hibernate, zabezpieczanie endpointów z Spring Security oraz integracja z systemem autoryzacyjnym CAS[2]. Dodatkowo dotkniemy tematu testowania naszego API i tworzenia dokumentacji z wykorzystaniem Swaggera.

Kolejny rozdział skupia się na przedstawieniu analizy porównawczej rozwiązań używanych w celu tworzenia aplikacji. Porównana została ich popularność na przestrzeni lat, na podstawie której postaraliśmy się odpowiedzieć na pytanie jaka czeka je przyszłość. Pokazane zostały również cechy przemawiające za wyborem każdego z rozwiązań oraz mniej lub bardziej oczywiste wady, a także kwestia UX/UI aplikacji. Czyli bardzo dynamicznie rozwijających się działów, na które kładziony jest coraz większy nacisk. Rozdział został napisany przez Macieja Głowackiego.

Ostatni rozdział przedstawia różnice między trzema najpopularniejszymi frameworkami, jakie oferuje nam społeczność ludzi programujących w JavaScript, czyli React.js, Angular oraz Vue.js. Przedstawione zostały w nim wady i zalety każdego z systemów. Między innymi pod względem różnic w składni czy trudności w rozwiązywaniu potencjalnych problemów. Przedstawiono także zagadnienie jakim jest tworzenie Single Page Applications. Rozdział został napisany przez Huberta Wrzesińskiego.

Rozdział 1

Mechanizmy zapewnienia bezpieczeństwa aplikacji

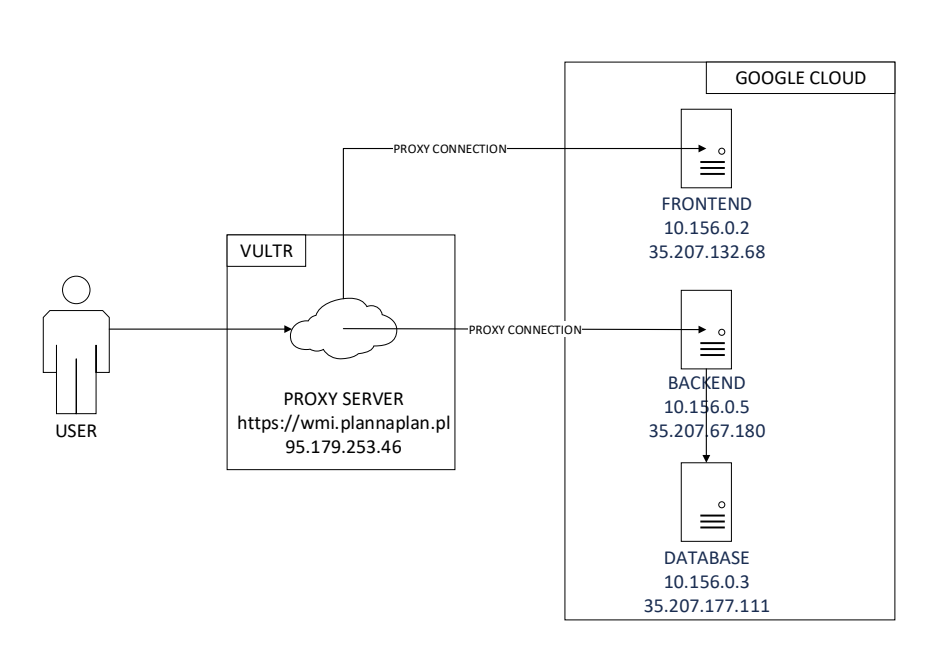
Autor rodziału: Marcin Woźniak

Bezpieczeństwo aplikacji może zostać zapewnione przez kilka dobrze znanych mechanizmów. Prawie codziennie słyszymy o niepożądany wycieku poufnych informacji takich jak hasła, dane personalne. Samo bezpieczeństwo możemy definiować jako stan dający poczucie pewności i gwarancję jego zachowania oraz szansę na doskonalenie[1]. Oczywiście nie możemy zapominać, że taki idealnie bezpieczny system nie istnieje, ale możemy starać się go wykonać jak najlepiej go potrafimy. W tym rozdziale zostaną przedstawione techniki zabezpieczające serwer, podział infrastruktury, jeden z bezpiecznych systemów operacyjnych, a także zabezpieczenia infrastruktury.

[1]Źródło definicji: `https://en.wikipedia.org/wiki/Security`

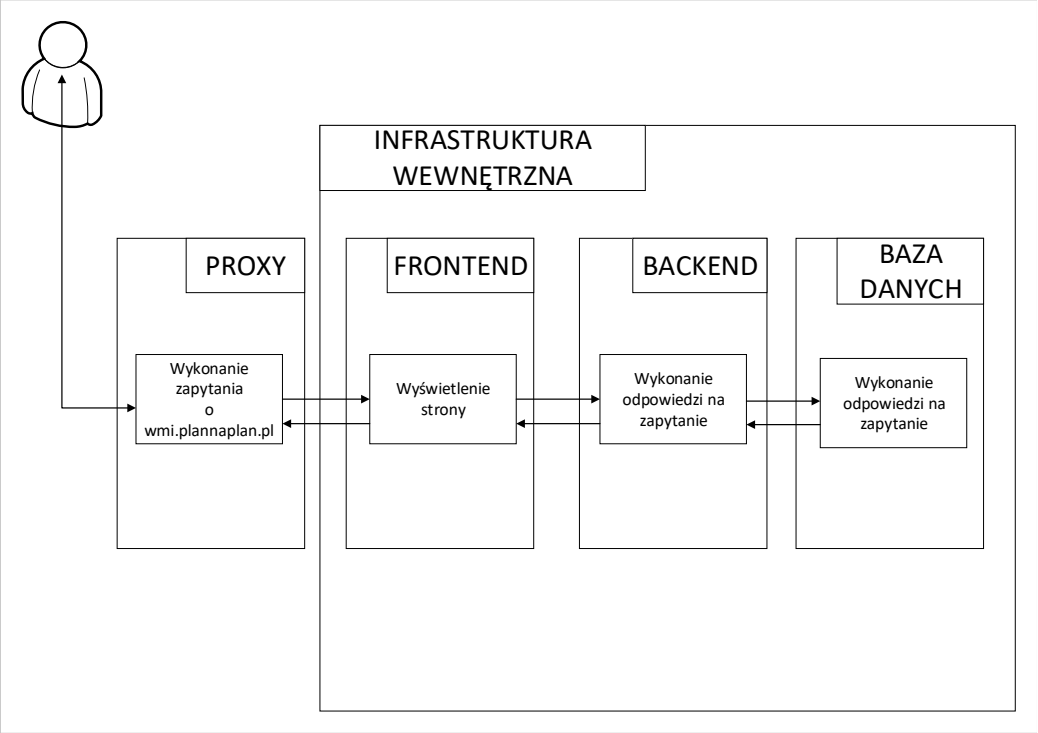
1.1 Infrastruktura wewnętrzna i logiczna

Sama infrastruktura systemu została podzielona na kilka części: infrastruktura wewnętrzna jak sama nazwa wskazuje zajmuje się wnętrzem. Zaś infrastruktura logiczna ukazuje nam samą logikę w jaki sposób ruch przechodzi przez z sieci zewnętrznej do wewnętrznej. W przedstawionym rysunku (nr. 5) ukazana jest infrastruktura w projekcie. Infrastruktura wewnętrzna znajduje się w prostokącie nazwanym "Google Cloud". Infrastruktura składa się z odpowiednich wirtualnych maszyn nazwanych kolejno: "FRONTEND", "BACKEND", "DATABASE". Każda z nich pełni określoną rolę w projekcie. Dodatkowo odseparowane środowiska ograniczają wektor ataku, także niektóre z nich są oddzielone od internetu.



Rysunek 5: Infrastruktura w projekcie
Źródło: Opracowanie własne

Infrastruktura logiczna ukazana została na rysunku (nr. 6). Przedstawia w jaki sposób kierowany jest ruch od użytkownika przez całą infrastrukturę aż ostatecznie wraca do użytkownika. Wprowadzając adres `https://wmi.plannaplan.pl` połączenie zostaje przekazywane do serwera proxy (definicja w znajduje się w podrozdziale 1.2.1) następnie infrastruktury wewnętrznej.



Rysunek 6: Infrastruktura logiczna
Źródło: Opracowanie własne

1.2 Infrastruktura zewnętrzna - Serwer proxy

1.2.1 Definicja

Serwer Proxy[2] (Serwer pośredniczący) jest to brama między użytkownikiem, a internetem. Oddziela on użytkowników końcowych od przeglądania przez nich docelowej witryny. Serwer pośredniczący zapewnia bardzo duże bezpieczeństwo jeżeli chodzi o naszą infrastrukturę.

1.2.2 Typy serwerów proxy

Wyróżnia się następujące typy serwerów proxy:

- Open proxy
  Zgodnie z definicją `Open proxy`[3] to serwer służący do częściowej anonimizacja lub przyśpieszenia komunikacji. Serwer proxy znajduje się poza kontrolą serwerów docelowych. Sama anonimizacja może obierać się na ukrywaniu adresu IP komputera na którym aktualnie pracuje użytkownik.

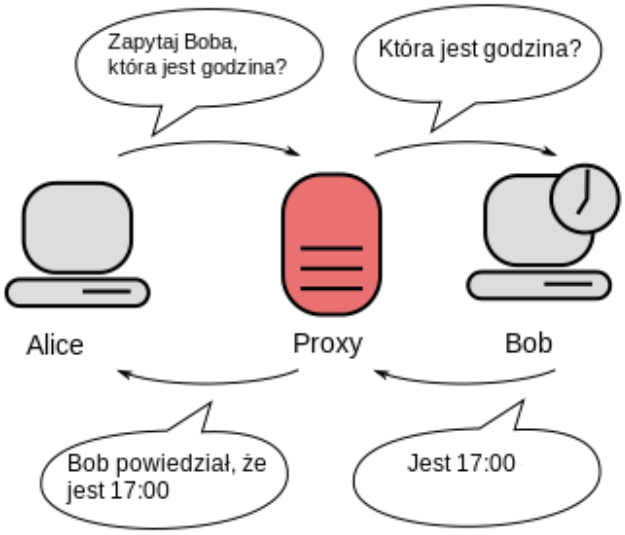
- Reverse proxy
  `Reverse proxy`[4] jest powszechnym typem serwera proxy, który jest dostępny z sieci publicznej. Duże witryny internetowe i sieci dostarczania treści wykorzystują odwrotne serwery proxy - razem z innymi technikami - w celu zrównoważenia obciążenia między serwerami wewnętrznymi. `Reverse proxy` mogą przechowywać pamięć podręczną zawartości statycznej, co dodatkowo zmniejsza obciążenie tych serwerów wewnętrznych i sieci wewnętrznej. Zwrotne proxy często dodają również funkcje, takie

[2]Źródło definicji: `https://en.wikipedia.org/wiki/Proxy_server`
[3]Źródło definicji: `https://pl.wikipedia.org/wiki/Open_proxy`
[4]Źródło definicji: `https://en.wikipedia.org/wiki/Reverse_proxy`

jak kompresja lub szyfrowanie TLS, do kanału komunikacyjnego między klientem a zwrotnym proxy.

1.2.3 Działanie serwera proxy

Jeśli korzystamy z serwera proxy to ruch do jakiejś witryny może zostać przekazany innej tak jak widzimy na rysunku 7.



Rysunek 7: Działanie serwera pośredniczący.
Źródło: `https://en.wikipedia.org/wiki/Proxy_server`

W naszym środowisku infrastruktury Alice jest użytkownikiem, który wchodzi na naszą stronę `https://wmi.plannaplan.pl` a następnie jego ruch jest przekierowany do infrastruktury w Google Cloud Platform (GCP) zgodnie z rysunkiem 5. Sam serwer proxy zapewnia bezpieczeństwo danych jeżeli chodzi o ich kontrolowanie ich oraz jak i o ich przesył. Następnie zapytanie wędruje do odpowiedniego z następnych serwisów a na koniec do użytkownika końcowego. Największą z zalet korzystając z serwera proxy istnieje prosta możliwość dodania kolejnej wirtualnej maszyny na przykład do backendu aby

dokonać zrównoważenia obciążenia (load-balance).

1.2.4 Przykładowa konfiguracja serwera proxy

W podanym kodzie (nr. 1.1) zastosowane jest wymuszone przekierowanie z `http` (port 80) na `https` (port 443). W sekcji `<VirtualHost *:443>` znajduje się `ProxyPass`[4] to główna dyrektywa konfiguracyjna proxy. W tym przypadku określa, że wszystko pod głównym adresem URL (/) jest mapowane na serwer pod podanym adresem oraz `ProxyPassReverse`[4] służy głównie do zapewniania przepisywania w locie pól nagłówka lokalizacji w odpowiedziach udzielanych przez aplikację, do której jest proxy. Kolejnymi nagłówkami są `SSLCertificateFile` oraz `SSLCertificateKeyFile`. Wskazują one na wykorzystanie certyfikatu SSL oraz klucza certyfikatu.

Kod 1.1: Przykładowy kod pochodzący z serwera proxy.

```
<VirtualHost *:80>
       ServerName wmi.plannaplan.pl
       ServerAdmin plannaplan@wmi.amu.edu.pl
       RewriteEngine on
       RewriteCond %{SERVER_NAME} =wmi.plannaplan.pl
       RewriteRule ^ https://%{SERVER_NAME}%{REQUEST_URI} [END,NE,R
           =permanent]
</VirtualHost>

<IfModule mod_ssl.c>
<VirtualHost *:443>
    SSLProxyEngine on
    ServerName wmi.plannaplan.pl
    Header set Cache-Control "max-age=0, no-cache, no-store, must-
       revalidate"
```

```
    DocumentRoot /home/website/plannaplan.pl/
    ServerAdmin plannaplan@wmi.amu.edu.pl

    ProxyPass / https://wmi-frontend.plannaplan.pl/
    ProxyPassReverse / https://wmi-frontend.plannaplan.pl/

    SSLCertificateFile /etc/letsencrypt/live/full.plannaplan.pl/
        fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/full.plannaplan.pl/
        privkey.pem
    Include /etc/letsencrypt/options-ssl-apache.conf
</VirtualHost>
</IfModule>
```

1.3 Infrastruktura wewnętrzna w chmurze

Brak możliwości uruchomienia projektu na serwerach stacjonarnych zachęcił nas do uruchomienia naszej infrastrukturze wewnętrznej w chmurze. Dodatkowo w tym celu serwery zostały rozmieszczone u różnych dostawców `Vultr` oraz `Google Cloud Platform`.

1.3.1 Vultr

Serwer Proxy został zainicjalizowany w chmurze `Vultr`. Jest to maszyna o następujących parametrach:

- 1vCPU
- 1 GB pamięci RAM
- 25 GB pojemność dysku
- System Operacyjny: Gentoo Linux (Przydatne informacje 1.4)
- Lokalizacja: Frankfurt, Niemcy

Parametry zostały dobrane z najwyższą starannością oraz jak najmniejszym kosztem miesięcznym.

1.3.2 Google Cloud Platform

Serwer backend, frontend, posiadający bazę danych zostały zainicjalizowany w chmurze `Google Cloud Platform`.

Informacje o serwerze backend:

- 2vCPU
- 2 GB pamięci RAM
- 20 GB pojemność dysku
- System Operacyjny: CentOS 8
- Lokalizacja: Frankfurt, Niemcy

Informacje o serwerze frontend:

- 1vCPU
- 600 MB pamięci RAM
- 20 GB pojemność dysku
- System Operacyjny: CentOS 8
- Lokalizacja: Frankfurt, Niemcy

Informacje o serwerze posiadający bazę danych (MariaDB):

- 2vCPU
- 1.7 GB pamięci RAM
- 20 GB pojemność dysku
- System Operacyjny: CentOS 8
- Lokalizacja: Frankfurt, Niemcy

Parametry zostały dobrane z najwyższą starannością. Serwer uruchamiający backend oraz bazę danych posiada odpowiednio większe parametry,

z powodu znaczącej ilości operacji wykonywanych na wyżej wymienionych serwerach.

1.4 System operacyjny - Gentoo Linux

Z pewnością mechanizmem, który bezpośrednio wpływa na bezpieczeństwo jest system operacyjny. On sam został wybrany pośród różnego wachlarza systemów na świecie. Wybór padł na dość niszową, ale potężną dystrybucję jaką jest Gentoo Linux[5].

Rysunek 8: Logo systemu operacyjnego Gentoo Linux
Źródło: `https://www.gentoo.org/`

Cechy, które przemawiają nad wyborem akurat tego systemu operacyjnego są następujące:

- Personalizacja od zera.
  Nie mamy żadnych instalatorów GUI[6]. Sam system instalujemy od zera. Do tego możemy użyć jakiegokolwiek bootowalnego pendrive z system Linux posiadający dostęp do internetu. Kolejne kroki instalacji opisane są w podręczniku użytkownika[6].

[5]Strona główna dystrybucji: `https://gentoo.org`
[6]GUI - Graphical user interface - Graficzny interfejs

- Wydajność

  System Gentoo Linux wspiera praktycznie każdą architekturę[5], daje nam to bardzo duże możliwości jeżeli chodzi o prędkość działania systemu jak i wydajność. Dodatkowo każda z zainstalowanych aplikacji, serwisów będzie bardziej wydajna na Gentoo Linux niż na innym systemie operacyjnym.

- Instalacja pakietów ze źródła.

  Menadżerem plików jest `Portage`, który napisany jest w języku Python. Sam sposób pobierania pakietów ze źródła opiera się na drzewie repozytorium, które jest synchronizowane za pomocą `rsync` albo `git` . Bezpośrednia kompilacja wybranego programy jest bezpieczniejsza, ponieważ nie używamy już wcześniej skompilowanej pliku binarnego, w którym nie wiem co się może znajdować. Same pakiety w repozytorium podzielone są na kategorie, a następnie na pakiety (`kategoria/pakiet`). Aby pobrać dowolny pakiet musimy skorzystać z narzędzia wiersza poleceń jakim jest `Emerge`.

- Możliwością instalowania tylko to co uważamy za potrzebne.

  Przy instalacji pakietu `Emerge`, oferuje nam użycie dowolnej oferowanej flagi (USE flags) przez pakiet. Same flagi musimy zdefiniować w pliku[7]. Przykładowe wywołanie `Portage` przy użyciu terminalowego narzędzia `Emerge`.

  Kod 1.2: Przykładowe wywołanie instacji pakietu `htop` z kategorii `sys-process`

```
yorune@Gentoo ~ $ emerge --ask sys-process/htop


These are the packages that would be merged, in order:
```

[7]Lokalizacja pliku korzystającego z USE flags `/etc/portage/package.use`

```
Calculating dependencies... done!
[ebuild N ] sys-process/htop-3.0.4-r1 USE="unicode -debug
-hwloc -lm-sensors -openvz -vserver"

Would you like to merge these packages? [Yes/No] Yes
>>> Verifying ebuild manifests
>>> Emerging (1 of 1) sys-process/htop-3.0.4-r1::gentoo
>>> Jobs: 0 of 1 complete, 1 running Load avg: 5.64, 5.89,
   6.02
```

W kodzie 1.2 dostrzegamy `USE="unicode -debug -hwloc -lm-sensors -openvz -vserver"`, widoczne tam są flagi, które podczas instalacji `Emerge` do kompiluje. Flaga oznaczana znakiem minus nie będzie brana pod uwagę.

- Dokumentacja
  Systemu Gentoo jest bardzo dobrze udokumentowany. Znajdziemy wszystkie niezbędne informacje dotyczące wybranego przez nas instalowanego pakietu, jego konfigurację (przykładowa konfiguracja pakietu `www-servers/apache`[7]), a także najnowsze wiadomości[8] ze świata Gentoo Linux.

- Trwałość
  Kolejną cechą przemawiającą za tym systemem jest jego trwałość. Same systemy operacyjne są wersjonowane (na przykład CentOS7, CentOS8[9] i tym podobne). Czyli chcąc zaktualizować system CentOS musimy go przeinstalować na nowo. W Gentoo Linux możemy posiadać niezaktualizowany system przez długi czas, gdzie w każdej chwili zaktualizować system do najnowszej wersji.

1.5 Zabezpieczenia infrastruktury

1.5.1 Automatyzacja oraz kontrolna dostępu

Z powodu dość czasochłonnej każdorazowej zmiany na aplikacji na produkcję, prostym rozwiązaniem tego problemu była automatyzacja. Gdy odbywają się prace, które mają na celu poprawę aplikacji, każda taka zmiana odbywa się na osobnej gałęzi (branchu). Następnie, osoba wykonująca zmianę prosi o zmianę (w Pull Request), a kolejna osoba sprawdza, czy wszystkie testy przechodzą na lokalnej maszynie użytkownika. W dalszym kroku akceptuje zmiany, a następnie repozytorium klonowane jest do kilku instancji (GitLab, GitHub, Gitea) w celu zapasowej oraz przeprowadzenia kompilacji i wdrożenie jej na wybrany serwis.

Podczas wyżej wymienionej automatyzacji aby zmiany zostały poprawianie zainstalowane na produkcji odpowiedni serwis musi zostać zrestartowany. Do tego celu nadane zostały prawa (zgodnie ze sztuką)[10] `sudo`, ale tylko do wykonywania odpowiednich komend do serwisu. W kodzie (nr. 1.3) użytkownik `backend` może uruchamiać, zatrzymywać, restartować serwis `plannaplan-backend` bez podawania hasła podczas używania `sudo`.

Kod 1.3: Kod `/etc/sudoers` pochodzący z serwera backend.

```
Defaults !visiblepw
Defaults always_set_home
Defaults match_group_by_gid
Defaults always_query_group_plugin
Defaults env_reset
Defaults env_keep = "COLORS DISPLAY HOSTNAME HISTSIZE KDEDIR
    LS_COLORS"
Defaults env_keep += "MAIL PS1 PS2 QTDIR USERNAME LANG LC_ADDRESS
    LC_CTYPE"
Defaults env_keep += "LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT
```

```
    LC_MESSAGES"
Defaults env_keep += "LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER
    LC_TELEPHONE"
Defaults env_keep += "LC_TIME LC_ALL LANGUAGE LINGUAS _XKB_CHARSET
    XAUTHORITY"
Defaults secure_path = /sbin:/bin:/usr/sbin:/usr/bin
root ALL=(ALL) ALL
%wheel ALL=(ALL) ALL
%backend ALL= NOPASSWD: /usr/bin/systemctl start plannaplan-backend
%backend ALL= NOPASSWD: /usr/bin/systemctl stop plannaplan-backend
%backend ALL= NOPASSWD: /usr/bin/systemctl restart plannaplan-
    backend
```

1.5.2 Iptables

`Iptables` służy do konfigurowania, utrzymywania i sprawdzania tabel reguł filtrowania pakietów IP w jądrze Linuksa. Można zdefiniować kilka różnych tabel. Każda tabela zawiera kilka wbudowanych łańcuchów i może również zawierać łańcuchy zdefiniowane przez użytkownika[8]. To zabezpieczenie głównie wykorzystywane jest serwerze proxy.

Kod 1.4: Wycinek kodu `/var/lib/iptables/rules-save` pochodzący z serwera proxy.

```
:INPUT DROP
:FORWARD DROP
:OUTPUT ACCEPT
[7362514:1236495002] -A INPUT -p tcp -m tcp --dport 22 -j ACCEPT
[4838871:875035999] -A INPUT -p tcp -m tcp --sport 22 -j ACCEPT
[19833:65179665] -A INPUT -p tcp -m tcp --sport 80 -j ACCEPT
[5622064:403244240] -A INPUT -p tcp -m tcp --dport 80 -j ACCEPT
```

[8]Źródło definicji: `https://linux.die.net/man/8/iptables`

```
[0:0] -A INPUT -s 127.0.0.1/32 -p tcp -m tcp --dport 3306 -j ACCEPT
[0:0] -A INPUT -s 127.0.0.1/32 -p tcp -m tcp --sport 3306 -j ACCEPT
```

Zgodnie z wycinkiem kodu (1.4) możemy dostrzec, że na samym początku kodu widnieje `INPUT, FORWARD` posiadające argument `DROP` oznaczające, że ruch przychodzący oraz przekierowywany jest domyślnie zablokowany, w późniejszych liniach kodu otwieramy na przykład port `22,80`. Sam port `3306` jest dostępny tylko z ramki posiadającej źródło o adresie `127.0.0.1`

1.5.3 Fail2Ban

Usługa `fail2ban`[9] skanuje pliki dziennika w poszukiwaniu wzorców określonych powtarzających się prób (na przykład nieudanych prób uwierzytelnienia SSH lub dużych ilości żądań GET / POST na serwerze internetowym), a po wykryciu automatycznie tworzy dodatkowe wpisy w `iptables`.

Kod 1.5: Kod `/etc/fail2ban/jail.d/sshd.conf` pochodzący z serwera proxy.

```
[sshd]
enabled = true
mode = aggressive
logpath = /var/log/messages
action = iptables[name=SSH, port=ssh, protocol=tcp]
bantime = 86400
maxretry = 2
```

Zgodnie z kodem o numerze 1.5 skanowanym plikiem jest `/var/log/messages` kiedy znajdzie w nim pewne anomalie dotyczące protokołu `SSH`, czyli kiedy użytkownik zaloguje się błędnie dwa razy (widoczne jest to w zmiennej `maxretry`) za pośrednictwem klucza prywatnego w przypadku serwera proxy, zostanie zablokowany na jeden dzień (widoczne jest to w zmiennej `bantime`,

[9]Źródło definicji: `https://wiki.gentoo.org/wiki/Fail2ban`

która jest równa 86400 s czyli dzień). Sam wpis, który zawiera adres, z którego pochodziło zapytanie zostaje dodany do odpowiedniego łańcucha w `iptables`.

1.5.4 Zapora w Google Cloud Platform

Rysunek 9, który przedstawia konfiguracje zapory w Google Cloud Platform (GCP). Dwa wpisy o nazwie `default-allow-api` oraz `default-allow-https` tylko wpuszczają ruch z serwera o adresie ip `95.179.253.46` (Adres serwera proxy) o podanych niżej portach. Dzięki temu wiemy, że żaden innych ruch na te wybrane porty nie będzie wpuszczany, gdy będziemy wysyłać zapytania z innego urządzenia. Sama zapora w GCP wyręcza użytkownika na poziomie wirtualnych interfejsów. Nie ma konieczności konfigurowania `iptables` w każdej maszynie wirtualnej osobno.

Name	Type	Targets	Filters	Protocols / ports	Action	Priority	Network ↑
default-allow-api	Ingress	api	IP ranges: 95.179.253.46/32	tcp:1285	Allow	1000	default
default-allow-https	Ingress	https-server	IP ranges: 95.179.253.46/32	tcp:80,443	Allow	1000	default
default-allow-internal	Ingress	Apply to all	IP ranges: 10.128.0.0/9	all	Allow	65534	default
default-allow-ssh	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:22	Allow	65534	default

Rysunek 9: Wpisy zapory w Google Cloud Platform
Źródło: Opracowanie własne

Rozdział 2

Tworzenie REST API za pomocą Spring Boot

Autor rodziału: Filip Izydorczyk

2.1 Application Programming Interface

API (ang. application programming interface) - to najprościej mówiąc zbiór reguł w jaki sposób komunikują się ze sobą programy lub podprogramy. API ma za zadanie dostarczyć nam funkcje, klasy, struktury oraz protokoły komunikacyjne z których będziemy korzystać w ramach naszego programu. Co warto zaznaczyć jego implementacja może być niezależna od definicji. Przykładowo istnieje wiele realizacji poprzez API standardowej biblioteki języka C. Można to wykorzystać jako przewagę w "sytuacji", gdy implementacja wymaga poprawek implementując nową wersję wg starego interfejsu. Zyskujemy wtedy pełną integralność ze starą wersją i nie ryzykujemy popsucia programów działających na starej implementacji. Istnieją też API niezależne

od systemów operacyjnych, których definicja jest jedna, ale implementacja różni się w zależności od systemu. Zyskuje się wówczas możliwość pisania programów multiplatformowych z wykorzystaniem jednego API zamiast kilku.

2.2 REST API

API jest zatem bardzo szerokim pojęciem obejmującym wiele jego "podkategorii" jednak pierwsze skojarzeniem wielu osób z tym pojęcie może być REST API (ang. Representational State Transfer API), które zyskało na popularności przez ostatnie kilkanaście lat[13]. Jest to obecnie najpowszechniejszy z dostępnych WebAPI. Do tego zbioru zaliczają się również SOAP (ang. Simple Object Access Protocol) i JavaScript. Ten ostatni interfejs jest przykładem API działającego po stronie klienta natomiast SOAP i REST działają po stronie serwera na zasadzie "żądanie-odpowiedź". Działa to tak, że wysyłane jest żądanie na dany endpoint (zazwyczaj dostępny przez konkretne URI) serwer wykonuje odpowiednie działania i zwraca odpowiedź w postaci danych lub informacji o powodzeniu/niepowodzeniu wykonywanej operacji. Na tym etapie można zauważyć pewne podobieństwa między SOAP i REST w związku z czym nasuwa się pytanie. Dlaczego REST tak zdominował rynek? Powodów jest kilka i mimo, że SOAP w dzisiejszych czasach nadal ma swoje zalety wygląda na to, że ważniejsze dla deweloperów jest to co oferuje REST. Używa on lżejszych formatów danych od xml (wymuszonego przez SOAP), a także nie narzuca jednego formatu danych. Najczęściej wybierany to json, możemy również stosować html, yaml, czysty text i wiele innych formatów. W parze z mniejszym formatem idzie też jego większa wydajność. Często wymienia się też przy zaletach REST jego prostotę i możliwość relatywnie szybkiej jego nauki. W kontrze do REST SOAP oferuje takie zalety jak rozszerzenie bezpieczeństwa WS-Security z wbudowaną logiką wspomagającą komunikację o błędach w integracji czy wsparcie dla komunikacji stanowej

i bezstanowej.

2.3 Spring Boot w implementacji REST API

Po zdecydowaniu się na wykorzystanie REST API jako sposobu dostarczenia naszych funkcjonalności należy wybrać narzędzia w jakich zostanie zrealizowana implementacja - język, framework, biblioteki. W języku JAVA przez bardzo długi czas dominował Spring framework. Pomimo że dostarcza on wiele udogodnień dla programistów to jak każde narzędzie miał też swoje wady jak np. konieczność posiadania zewnętrznego serwera do ładowania aplikacji, czy konieczność dbania o strukturę wprowadzanych bibliotek, które mogłyby wytworzyć problemy z kompatybilnością. Jako odpowiedź na te niedogodności w 2014 r. został wypuszczony na świat nowy framework pod nazwą Spring Boot[35]. Był on tak naprawdę rozszerzeniem Springa, który w dalszym ciągu dostarczał jego funkcjonalności, a jednocześnie rozwiązywał problemy poprzednika. Wprowadzono m.in. wbudowany serwer w celu ułatwienia procesu deploymentu, automatyczną konfigurację Springa (w tych miejscach w których było to możliwe) i startery (biblioteki wraz z konfiguracją i serwerem Tomcat) ułatwiające start i testowanie aplikacji. Projekt aplikacji w Spring Boot można stworzyć na trzy sposoby. Pierwszym jest użycie strony Spring Initializr (`https://start.spring.io/`) w której wyklikujemy co jest nam potrzebne. Drugim sposobem jest użycie wtyczki w IDE z którego korzystamy tworzącej za nas szablon projektu, który będziemy rozszerzać o kolejne moduły. Ostatnim sposobem jest użycie narzędzia do zarządzania projektem takim jak Maven lub Gradle i dodać za jego pomocą potrzebne zależności.

2.4 Dependecny injecton w Spring Boot

Przy tworzeniu większych aplikacji oraz API warto zastanowić się nad powszechnym problemem związanym z zależnościami tworzącymi się wewnątrz naszego projektu. Problem zostanie zilustrowany na prostym przykładzie.

Kod 2.1: Przykładowy kod, opracowanie własne

```
public class EmailService {
    private JavaMailSender emailSender;
    public EmailService() {
        this.emailSender = new JavaMailSender();
    }
    public void sendMail(String destination, String message) {
        SimpleMailMessage mailMessage = new SimpleMailMessage();
        mailMessage.setFrom(appEmail);
        mailMessage.setTo(destination);
        mailMessage.setSubject("[Plan na plan] INFO");
        mailMessage.setText(message);
        emailSender.send(mailMessage);
    }
}
```

W przedstawionej sytuacji klasa `EmailService` jest odpowiedzialna za utworzenie instancji klasy `JavaMailSender`. Występuje tu zatem zakodowana na stałe zależność. Na pierwszy rzut oka może to nie wydawać się aż tak groźnie jednak już na tak podstawowym przykładzie można znaleźć kilka argumentów dlaczego z tego rozwiązania zrezygnować. Po pierwsze w przypadku wspomnianej stałej zależności klasa `EmailService` uzależniona jest od `JavaMailSender` co jednocześnie oznacza, że jeżeli wystąpiłaby potrzeba zmiany tej klasy na inną to trzeba by było ingerować w kod `EmailService`. Dodatkowo jeżeli klasa `JavaMailSender` również potrzebowałaby zależności to klasa `EmailService` musi być tego świadoma i je przekazać co poniekąd

można zinterpretować jako złamanie zasady jednej odpowiedzialności klasy. Takie podejście może również utrudnić testowanie samego oprogramowania ponieważ dana klasa może być zależna od innej stworzonej przez nas w projekcie co przeszkadza w jednostkowości tych testów. Może również zaistnieć sytuacja, że w innym miejscu w kodzie również potrzebna jest identyczna instancja klasy. Niepotrzebnie wówczas jest ona tworzona dwukrotnie, gdy można by było stworzyć jedną i przekazać ją odpowiednim obiektom. Opisanych problemów pojawiło się już kilka, a z rozwojem aplikacji mogą się one mnożyć w błyskawicznym tempie. Mnogość problemów związanych z odpowiednim zarządzaniem zależnościami w obrębie naszego systemu poskutkowało wzorcem projektowym **dependency injection** (DI). Jest to dosyć elastyczny wzorzec w którym klasa wszystkie inne klasy, od których zależy, dostawała z zewnątrz. Zazwyczaj "wstrzykiwanie" tych obiektów odbywa się poprzez konstruktor, setter lub interface, ale sam pattern dopuszcza również inne rozwiązania.

Kod 2.2: Przykładowy kod, opracowanie własne

```
// Constructor injection
EmailService(JavaMailSender service) {
    this.service = service;
}

// Setter injection
public void setService(JavaMailSender service) {
    this.service = service;
}

// Interface injection
// JavaMailSender jest istniejacym obiektem w springframework,
// ale dla potrzeby przykładu uznajmy, że jest interfacem
// lub klasą abstrakcyjną z której dziedziczą inne klasy
public interface EmailSetter {
```

```
    public void setService(JavaMailSender service);
}

public class EmailService implements EmailSetter {
    private Service service;

    @Override
    public void setService(JavaMailSender service) {
        this.service = service;
    }
}
```

Springframework dostarcza swoje implementacje tego wzorca z których można skorzystać. Często wspólną cechą frameworków implementujących DI jest `IoC container`. Jak sama nazwa wskazuje jest to kontener, który w założeniu ma służyć przechowywaniu oraz montowaniu zależności w obrębie naszej aplikacji. W Spring Boot instancje takiego kontenera można stworzyć na wiele sposobów w zależności od potrzeb, a każdy taki kontener będzie implementował interfejs `ApplicationContext`. Przykładowo, aby utworzyć taki kontener z pliku `.xml` o ustalonej strukturze należy użyć `ClassPathXmlApplicationContext`, a w celu utworzenia go z klasy konfiguracyjnej oraz adnotacji (o tym więcej w dalszej części pracy) użyjem `AnnotationConfigApplicationContext`. Oczywiście klas implementujących ten interfejs jest znacznie więcej dlatego zanim zdecydujemy się na któryś ze sposobów powinniśmy zastanowić się co będzie wygodniejsze lub bardziej poprawne dla naszych założeń projektowych. Przed stworzeniem kontenera za pomocą którego będą wstrzykiwane zależności trzeba utworzyć konfigurację tzw. Bean'ów. Bean to po prostu klasa, która jest zarządzana przez `IoC container`. Taką konfigurację można stworzyć za pomocą zwykłej klasy z odpowiednią adnotacją,

Kod 2.3: Przykładowy kod, opracowanie własne

```
@Configuration
public class Config {


    @Bean
    public Item firstItem() {
        return new Item();
    }


    @Bean
    public Store store() {
        return new Store(firstItem());
    }
}
```

a także za pomocą pliku `.xml` o odpowiedniej strukturze.

Kod 2.4: Przykładowa konfiguracja za pomocą pliku xml, opracowanie własne

```
<bean id="firstItem" class="com.planaplan.Item" />
<bean id="store" class="com.plannaplan.Store">
    <constructor-arg type="Item" index="0" name="item" ref="
        firstItem" />
</bean>
```

Następnie można stworzyć kontener i wstrzyknąć zależność. W zależności od tego czy konfiguracja Bean'ów odbywała się przez klasę w Javie czy przez plik `.xml` tworzenie kontenera będzie wyglądać delikatnie inaczej.

Kod 2.5: Przykładowy kod, opracowanie własne

```
//java annotated class
ApplicationContext context = new AnnotationConfigApplicationContext
    (Config.class);

//xml file
ApplicationContext context = new ClassPathXmlApplicationContext("
    file.xml");

Item item = context.getBean(Item.class);
```

Gdy w konfiguracji zostaną utworzone `Beany` można ich używać jako zależności w następnych klasach. Wstrzyknąć je można za pomocą konstruktora:

Kod 2.6: Przykładowy kod, opracowanie własne

```
@Component
public class Market {

    @Autowired
    public Market(Store store, Item item) {
        this.store = store;
        this.item = item;
    }
}
```

Spring skanując paczkę napotka klasę `Market` i stworzy jej instancję wykorzystując konstruktor oznaczony (`@Autowired`). Obiekty `Store` i `Item` natomiast uzyska poprzez wywołanie metod opatrzonych adnotacjami `@Bean` w klasie `Config`. Gdy w kodzie potrzeba uzyskać utworzoną instancję można to zrobić za pomocą `ApplicationContex`:

Kod 2.7: Przykładowy kod, opracowanie własne

```
ApplicationContext context = new AnnotationConfigApplicationContext
    (Config.class);
Market car = context.getBean(Market.class);
```

`Autowired` można również spotkać w innych miejscach niż nad konstruktorem. Umieszczony nad seterem sprawi, że po utworzeniu instancji zostanie on wywołany z beanem typu, który seter przyjmuje jako argument. W poniższym przykładzie po utworzeniu instancji klasy `Market` zostanie wywołany seter z instancją klasy `Store` jako argument.

Kod 2.8: Przykładowy kod, opracowanie własne

```
public class Market {
    private Store store;
    @Autowired
    public void setFooFormatter(Store store) {
        this.store = store;
    }
}
```

W tym przypadku mamy do czynienia z wstrzykiwaniem zależności przez seter. Jak można przeczytać w dokumentacji Springa wstrzykiwanie zależności jest zaimplementowane głównie w dwóch wariantach: przez konstruktor oraz przez seter.

> DI exists in two major variants: Constructor-based dependency injection and Setter-based dependency injection.

Oznacza to, że nie realizuje on wspomnianego wcześniej wstrzykiwania zależności przez interface. Mimo to istnieje jeszcze inna, trzecia realizacja tego wzorca, jaką jest field injection. Jest on najprostszym sposobem na wstrzykiwanie zależności i polega on na adnotacji pola, które zawiera jakąś zależność `@Autowired`.

Kod 2.9: Przykładowy kod, opracowanie własne

```
public class UserService {


    @Autowired
    private UserRepository userRepository;
}
```

W większości przypadków należy tego rozwiązania unikać ponieważ ma ono kilka wad. Po pierwsze rozwiązanie to wiąże tworzoną klasę z `IoC container`. Oznacza to, że nie można stworzyć jej instancji bez kontenera również na etapie pisania testów jednostkowych. W konsekwencji tworzone testy mają coraz bardziej charakter testów integracyjnych niż jednostkowych. Z takim wzorcem bardzo łatwo też o dużo zależności w jednej klasie. W przypadku gdy stworzy się piętnasto argumentowy konstruktor jest to jasnym sygnałem dla programisty, że coś jest nie tak, a z wstrzykiwaną zależnością przez pole klasę użytkuje się całkiem normalnie. Nie istnieje też w api klasy żadne odzwierciedlenie jej zależności. Ani przez konstruktor, ani przez seter. Jest więc takie rozwiązanie tylko nieco lepsze od przykładu z początku podrozdziału.

2.5 Hibernate jako ORM w spring

W tworzonej aplikacji prędzej czy później prawdopodobnie pojawi się potrzeba przetrzymywania jakiegoś rodzaju danych. Spring dostarcza kilka wbudowanych możliwości uzyskania dostępu do baz danych. Są to m.in:

- Zarządzanie transakcjami,
- DAO - obiekt dostępu do danych (ang. Data Access Object),
- Dostęp do danych z JDBC,

- ORM - Mapowanie obiektowo relacyjne (ang. Object Relational Mapping)

Są to tylko wybrane z dostępnych sposobów (więcej sposobów można znaleźć w tym dokumencie oraz dokumentacji. Każdy z nich ma swoje wady i zalety i warto zanim się zdecyduje na któreś z nich to zapoznać się z nimi aby dopasować technologię do wymagań projektowych. Przykładowo chcąc utrzymać paradygmat obiektowy w całym projekcie dobrym pomysłem może być wybranie mapowania obiektowo relacyjnego. Pozwala ono zmapować daną tabele z wybranej relacyjnej bazy danych na obiekt w Javie. Takie obiekty nazywa się encjami. Technologia wykorzystywana w Springu do mapowania Hibernate. Aby użyć go w projekcie najpierw trzeba dodać niezbędne zależności do `pom.xml`

Kod 2.10: Zależności w pom.xml, opracowanie własne

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
```

Zależność `spring-boot-starter-data-jpa` to starter do Spring Data JPA z Hibernate, a druga zależność to sterownik do bazy danych (w tym przypadku `mysql`). Następnie trzeba skonfigurować połączenie z bazą. Można to zrobić w pliku `src/main/resources/application.properties`

Kod 2.11: Zależności w pom.xml, opracowanie własne

```
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.
   MySQL5Dialect
spring.datasource.url=jdbc:mysql://localhost:3306/test?useUnicode=
   yes&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=example
```

- **spring.jpa.properties.hibernate.dialect** - sterownik do połączenia z bazą
- **spring.datasource.url** - adres połączenia z bazą
- **spring.datasource.username** - login użytkownika za pomocą którego będziemy się łączyć
- **spring.datasource.password** - hasło użytkownika

Na tym etapie można już nawiązać w Springu pomyślne połączenie z bazą, ale może nasunąć się pytanie. Czy tabele w bazie powinny być już utworzone, czy `Hibernate` zrobi to automatycznie. To również można skonfigurowaćw pliku `application.properties` opcją `spring.jpa.hibernate.ddl-auto`. Dostępne są opcje:

- **validate** - `Hibernate` sprawdza czy wskazana baza ma już odpowiedni układ tabel, nie wprowadza żadnych zmian
- **update** - aktualizuje układ tabel
- **create** - tworzy wszystkie niezbędne tabele na podstawie stworzonych encji

- **create-drop** - podobnie jak `create` tworzy tabele, a gdy `SessionFactory` zostanie zamknięte są one usuwane. Zazwyczaj to się dzieje w momencie zamknięcia aplikacji, przydatna opcja w trakcie testowania wprowadzanych zmian w encjach
- **none**: nie wprowadza do bazy żadnych zmian

Istnieje więc możliwość niejako zapomnienia o modelowaniu tabel i ich relacji gdyż można ustawić `create` i modelować bazę poprzez tworzenie kolejnych obiektów-encji. Takie podejście znajdzie swoich zwolenników i przeciwników. Zarzuty wobec tej opcji[23] sprowadzają się do tego, że traci się kontrolę nad swoją bazą danych. Przykładowo istnieje szansa, że `Hibernate` zrealizuje relację `OneToMany` na trzech tabelach zamiast na dwóch. Istnieje też problem bałaganu w bazie. Gdy używana jest opcja `update` nie usuwa ona starych danych, a to w dłuższej perspektywie może się przyczynić do owego bałaganu. Dodatkowo jeżeli istnieje kilka instancji aplikacji to trzeba dbać o to by tylko jedna z nich aktualizowała bazę. Problemy mogą się dalej mnożyć jednak automatyczne tworzenie bazy może być przydatne przy testach integracyjnych lub w procesie prototypowania aplikacji. Jeżeli wiadomo, że operacje nie będą przeprowadzane na wielkich zbiorach danych oraz że nie będą się one szybko skalować, to ta opcja również może wydać się poprawną. Ponownie decyzję należy rozpatrzyć według potrzeb projektu. Gdy `Hibernate` jest już cały skonfigurowany można odwzorowywać bazę na obiekty. Zakładając, że mamy tabele `LECTURER`, która ma pola `id`, `name`, `surname` jej encja będzie wyglądać tak:

Kod 2.12: Przykładowy kod, opracowanie własne

```
@Entity
@Table(name = "LECTURER")
public class Lecturer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String surname;

    //constructors, getters, setter

}
```

Encja ta odzwierciedla rekord tabeli `LECTURER`, a przy zapisie do bazy przypisze nowemu rekordowi automatycznie zwiększone id względem ostatniego zapisanego rekordu (`@GeneratedValue(strategy = GenerationType.AUTO)`). Warto też zauważyć, że założono tutaj, że nazwy pól odpowiadają nazwą kolumn danej tabeli. Gdybyśmy chcieli pole nazwać inaczej polę niż kolumny w bazie to należy takie pole zaopatrzyć w odpowiednią adnotację `@Column(name = "db_coulmn_name")`. Ale jak taką encję w ogóle dostać? Potrzebny będzie serwis oraz repozytorium dla danej encji. Repozytorium odpowiada za wysyłanie zapytań do bazy oraz łapanie błędów w komunikacji, a serwis trzyma logikę biznesową. Repozytorium tworzy się według poniższego wzoru

Kod 2.13: Przykładowy kod, opracowanie własne

```
@Repository
public interface LecturerRepository extends JpaRepository<Lecturer,
    Long>{}
```

Stworzone repozytorium będzie odpytywało tabele `LECTURER` i będzie zakładało, że `id` w tej tabeli jest typu `Long`. Z tą implementacją dostajemy kilka podstawowych zapytań w postaci metod jak np. `SELECT * FROM LECTURER;` pod postacią `Lecturer::findAll()`, ale mamy też możliwość stworzenia własnych zapytań. Przykładowo:

Kod 2.14: Przykładowy kod, opracowanie własne

```
@Repository
public interface LecturerRepository extends JpaRepository<Lecturer,
    Long> {
    @Query("FROM Lecturer WHERE name = ?1 AND surname = ?2")
    Optional<Lecturer> find(@Param("name") String name, @Param("
       surname") String surname);
}
```

Działanie tego zapytania jest zrozumiałe dla osób zaznajomionych ze składnią jezyka `SQL`, warto tutaj zaznaczyć, że nie jest to natywny `SQL` tylko `HQL` (Hibernate Query Language)[25]. Istnieje opcja użycia native query,

Kod 2.15: Przykładowy kod, opracowanie własne

```
@Query(
  value = "SELECT * FROM Lecturer WHERE name = ‘Jan‘",
  nativeQuery = true)
```

ale `HQL` zapewnia ochronę przed atakiem sql injection. Warto też zaznaczyć, że `Lecturer` to nazwa klasy, a nie tabeli i tak samo `name` to nazwa pola w klasie, a nie kolumny w tabeli w bazie bo to może być problematyczne z początku. Implementacja serwisu dla Lecturer'a z kolei wyglądałaby następująco:

Kod 2.16: Przykładowy kod, opracowanie własne

```
@Service
public class LecturerService {
    @Autowired
    private LecturerRepository repo;

    public Optional<Lecturer> getLecturer(String name, String
        surname) {
        return repo.find(name, surname);
    }

    public Lecturer save(Lecturer lecturer) {
        repo.save(lecturer);
        return lecturer;
    }

    public void delete(Lecturer lecturer) {
        repo.delete(lecturer);
    }

    public int getLecturersAmmount() {
        return (int) this.repo.count();
    }
}
```

Stworzony serwis umożliwia zapis i usunięcie rekordu w bazie, wyświetlenie ilości rekordów w tabeli oraz znalezienie rekordu po kolumnach `name` i `surname`. Na tym etapie można w aplikacji używać i zapisywać nowe dane za pomocą `LecturerService`.

2.6 Tworzenie REST endpointów za pomocą kontrolerów

Aby dać możliwość zarządzania danymi klientom (np. aplikacji frontend lub innemu systemowi, który integruje się z naszym) trzeba dostarczyć wspomniane na początku tego rozdziału endpointy. Aby zacząć najpierw trzeba stworzyć za pomocą odpowiedniej adnotacji klasę kontroler, która będzie odpowiedzialna za trzymanie endpointów odnoszącej się do konkretnej kategorii (np. dla działań na użytkownikach danego systemu można by stworzyć `UserController`). Jest to jeden ze wspomnianych aspektów, w którym Spring Boot dostarcza pewne ułatwienia względem czystego Springa poprzez dostarczenie adnotacji `@RestController`, którą w zwykłym Springu zastępuje się dwoma adnotacjami `@Controller` oraz `@ResponseBody`.

Kod 2.17: Przykładowy kod, opracowanie własne

```
@RestController
@RequestMapping("/api/v1/users")
public class UsersController {
    @Autowired
    private UserService userService;

    @GetMapping("/students")
    public ResponseEntity<List<User>> getAllStudents() {
        final List<User> searches = this.userService.getAllStudents
            ();
        return new ResponseEntity<>(searches, HttpStatus.OK);
    }

    @GetMapping("/students/{id}")
    public ResponseEntity<User> getStudent(@PathVariable(name = "id"
        ) Long id) {
```

```
        final Optional<User> user = this.userService.getStudentById(
            id);
        return new ResponseEntity<>(user.get(), HttpStatus.OK);
    }
}
```

Powyższy przykład dobrze obrazuje całą koncepcję kontrolerów. Pojedynczy kontroler (w tym przypadku `UsersController`) jest kontenerem dla endpointów związanych z użytkownikami, a metody opatrzone odpowiednią adnotacją są już pojedynczymi endpointami. Do kontrolera poza samym `@RestController` można też dodać adnotację `@RequestMapping("wartość")`. Będzie to oznaczało to, że każdy endpoint będzie miał ustalony przedrostek. Może być to np sama kategoria (tutaj users), a jeśli istnieje potrzeba wskazania w endpoincie wersji jego implementacji to może być to dłuższa nazwa (tutaj `/api/v1/users`). Metodą w adnotacji należy wskazać typ zapytania jakie obsługuje (GET, POST , PUT itd...) oraz ścieżkę endpointu pod którym będzie dostępna. W powyższym przykładzie metoda `getAllStudents` będzie wywołana po wysłaniu zapytania `GET` pod adres `host:port/api/v1/users/students`. W tym przypadku ta informacja jest dostarczona przez adnotację `GetMapping` (dla metody `POST` byłaby to adnotacja `PostMapping` i analogicznie dla reszty), ale można to samo osiągnąć adnotacją `@RequestMapping(value = "/students", method = GET)`. W trakcie wykonywania zapytań HTTP istnieje także kilka możliwości, żeby podać jakiegoś rodzaju argumenty. Przykładem realizacji takiego przypadku w Spring Boot jest powyższa metoda `getStudent`. W tym przypadku zmienną podajemy w ścieżce (np `host:port/api/v1/users/students/5`) i z tego powodu argument `id` jest oznaczony adnotacją `@PathVariable`. Pozostałe argumenty (to jest: query parameters, form data oraz multipart requests) możemy przypisać do argumentu metody adnotacją `@RequestParam`. Adnotacja ta odszuka w requeście argument o adekwatnej nazwie i przypisze jego wartość

do zmiennej w Javie. Jeżeli takiej zmiennej nie znajdzie zostanie rzucony błąd. Jeśli dany argument ma być opcjonalny to wówczas można albo typ w javie opatrzyć w `Optional`

Kod 2.18: Przykładowy kod, opracowanie własne

```
public ResponseEntity<User> getStudent(@PathVariable(name = "id")
    Optional<Long> id)
```

lub też ustawić wartość `reqired` na `fasle` w samej adnotacji.

Kod 2.19: Przykładowy kod, opracowanie własne

```
@PathVariable(name = "id", required = false) Long offerId
```

Przy tym przykładzie warto też zauważyć, że nazwa zmiennej nie musi odpowiadać tej w zapytaniu o ile podamy tę nazwę jako atrybut `name` w adnotacji. Jeżeli istniałaby potrzeba aby przechwytywać wszystkie podane przez klienta argumenty za jednym razem można je umieścić w mapie.

Kod 2.20: Przykładowy kod, opracowanie własne

```
@RequestParam Map<String,String> allParams
```

W taki sam sposób można argument podany jako json w ciele zapytania jednak nie jest to zbyt eleganckie rozwiązanie. Zamiast tego można złapać argument jako `String` i zdeserializować go do obiektu za pomocą jakiejś biblioteki (np. Jackson). To brzmi nieco lepiej, ale ten krok również może wykonać sam Spring Boot z użyciem adnotacji `@RequestBody`.

Kod 2.21: Przykładowy kod, źródło: `https://www.baeldung.com/spring-mvc-send-json-parameters`

```
public class Product {

    private int id;
    private String name;
    private double price;

    // default constructor + getters + setters

}

@PostMapping("/create")
@ResponseBody
public Product createProduct(@RequestBody Product product) {
    // custom logic
    return product;
}
```

Podobnie sytuacja wygląda z odpowiedziami wysyłanymi do klienta. W przykładzie z użytkownikami daliśmy jako zwracany typ `ResponseEntity<User>` co oznacza, że Spring Boot sam przerobi instancję klasy `User` na json i doda do body odpowiedzi.

2.7 Zabezpieczanie endpointów z Spring security

2.7.1 Podstawowa konfiguracja

Na tym etapie każdy z tworzonych endpointów jest ogólnodostępny dla każdego kto wywoła zapytanie co w uzasadnionych przypadkach może mieć sens, ale nie mniej bardzo często będą w systemie tworzone endpointy, które powinny być zabezpieczone przed niepożądanym dostępem ze strony klienta. Najprostszym przykładem jest zabezpieczenie odpytania endpointu odpowiedzialnego za szeroko rozumianą konfigurację systemu przez zwykłego użytkownika, czy użytkownika bez uprawnień. W aplikacjach bazujących na Spring standardem uwierzytelniania i autoryzacji jest framework spring Spring Security[17]. Aby móc go używać w Spring Boot wystarczy dodanie jednej zależności `spring-boot-starter-security`, który automatycznie uwzględni wszystkie potrzebne zależności.

Kod 2.22: Zależność w pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
    <version>2.3.3.RELEASE</version>
</dependency>
```

Gdy wszystkie potrzebne zależności są już zaimportowane można przejść do konfiguracji. Podobnie jak w wielu przypadkach w Spring boot możemy zdecydować czy chcemy to zrobić przez `xml` czy klasę w Javie [36]. W tym przypadku zostanie użyta klasa w Javie. Najpierw należy ustalić sposób w jaki użytkownik będzie autoryzowany. Dostępne opcje to:

- Uwierzytelnianie JDBC,

- Uwierzytelnianie LDAP,
- Uwierzytelnianie w pamięci aplikacji (za ang. in memory authentication),
- Spersonalizowane uwierzytelnianie poprzez implementację `AuthenticationProvider`

Kod 2.23: Przykładowy kod, źródło: `https://www.baeldung.com/spring-security-authentication-provider`

```
@Configuration
@EnableWebSecurity
public class CustomWebSecurityConfigurerAdapter extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.inMemoryAuthentication()
                .withUser("admin")
                .password("{noop}admin")
                .roles("ADMIN")
        }
    @Override
    protected void configure(HttpSecurity http) throws Exception{
        http
            .authorizeRequests()
            .antMatchers("/","index","/css/*","/js/*")
            .permitAll()
            .anyRequest()
            .authenticated()
            .and()
            .httpBasic();
```

```
    }

}
```

W powyższym przykładzie zdecydowano się na uwierzytelnianie w pamięci aplikacji (`AuthenticationManagerBuilder::inMemoryAuthentication`). Oznacza to, że nie korzystamy z żadnej bazy do przechowywania danych o użytkownikach, a więc wszelkie ewentualne zmiany zostaną utracone po restarcie aplikacji. Z oczywistych przyczyn nie jest to rozwiązanie na produkcję, a jak sama dokumentacja podaje tylko w celach testowych. Do pozostałych wymienionych autoryzacji istnieją w klasie `AuthenticationManagerBuilder` odpowiednie metody, a gdyby istniała potrzeba stworzenia własnej implementacji uwierzytelniania to wymagałby to stworzenia takiej klasy.

Kod 2.24: Przykładowy kod, opracowanie własne

```
@Component
public class CustomAuthenticationProvider implements
    AuthenticationProvider {

    @Override
    public Authentication authenticate(Authentication authentication
        )
      throws AuthenticationException {

        String name = authentication.getName();
        String password = authentication.getCredentials().toString()
            ;

        if (shouldAuthenticateAgainstThirdPartySystem()) {

            // use the credentials
```

```
            // and authenticate against the third-party system
            return new UsernamePasswordAuthenticationToken(
              name, password, new ArrayList<>());
        } else {
            return null;
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(
            UsernamePasswordAuthenticationToken.class);
    }
}
```

Otrzymana w metodzie `authenticate` instancja klasy `Authentication` zawiera wszelkie informacje dostarczone przez użytkownika, które mają potwierdzać jego tożsamość, a zadaniem samej metody jest zweryfikowanie czy dostarczone informację wystarczają na uwierzytelnienie.

Następnie trzeba instancję tej klasy wstrzyknąć przez wybrane **dependency injection** do klasy konfiguracyjnej i wskazać Springowi by jej używał jako `AuthenticationProvider`.

Kod 2.25: Przykładowy kod, opracowanie własne

```
@Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.authenticationProvider(authProvider);
    }
```

Gdy jest już ustalony sposób autoryzacji użytkownika można zabrać się za wybieranie, które zasoby mają zostać zabezpieczone, a które nie. Wracając do przykładu z klasą `CustomWebSecurityConfigurerAdapter`. Jest tam metoda `configure(HttpSecurity http)`, która przekazuje modułowi bezpieczeństwa, że zasobami `"/","index","/css/*"` oraz `"/js/*"` może zarządzać każdy bez uwierzytelniania. Jest to przydatne jeżeli poza samym udostępnionym REST API udostępniać jakieś konkretne pliki `html`. W tym momencie wszystkie endpointy są niedostępne bez potwierdzenia swojej tożsamości, a jeżeli zaistniałaby potrzeba udostępnienia niektórych z nich bez autoryzacji, można to uwzględnić w klasie konfiguracyjnej za pomocą metody `antMatchers`, a następnie `permitAll`. Metoda `antMatchers` poza samą ścieżką do zasobu może też zawierać konkretną metodę http w ramach której została wywołana.

Kod 2.26: Przykładowy kod, opracowanie własne

```
.antMatchers(HttpMethod.GET, "/api/v1/users/students")
```

Jednak to nie wyczerpuje jeszcze tematu ponieważ czasem do niektórych zasob dostęp ma tylko użytkownik o konkretnej roli np. tylko admin. Istotnie po to tworząc użytkownika podano mu rolę poprzez `.roles(ĄDMIN")`. W kontekście restowych aplikacji zabezpieczamy możliwość odpytania endpointów, a robi się to podobnie jak zezwalanie każdemu na dostęp tylko, że zamiast metody `permitAll` trzeba użyć `hasRole`.

Kod 2.27: Przykładowy kod, opracowanie własne

```
http
    .authorizeRequests()
    .antMatchers("/","index","/css/*","/js/*")
    .permitAll()
    .antMatchers("/api/v1/users/students")
    .hasRole("ADMIN")
    .anyRequest()
    .authenticated()
    .and()
    .httpBasic()
```

Warto też tutaj zauważyć na to, że metody w tym bloku zachowują się podobnie jak strumienie w przypadku kolekcji, więc można przygotować łańcuch zapytań, zamiast kilka razy podawać `http` gdy chcemy wywołać kolejny raz `antMatchers`. Innym sposobem na sprawdzenie roli są odpowiednie adnotacje w kontrolerach przy odpowiednich metodach.

Kod 2.28: Przykładowy kod, opracowanie własne

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
@RestController
@RequestMapping("/api/v1/users")
public class UsersController {

    //...

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    @GetMapping("/students")
    public ResponseEntity<List<User>> getAllStudents() {
        final List<User> searches = this.userService.getAllStudents
            ();
        return new ResponseEntity<>(searches, HttpStatus.OK);
```

```
    }

    //..

}
```

Opcję te najpierw trzeba włączyć poprzez `@EnableGlobalMethodSecurity (prePostEnabled = true)`, a następnie przy konkretnym endpoincie adnotujemy jaka rola ma do niego dostęp (tutaj admin - `@PreAuthorize("hasRole ('ROLE_ADMIN')")`). String podany w adnotacji `@PreAuthorize` to **Spring Expression Language**[27]. Dostarcza on m.in możliwość łączenia warunków jak w klazuli `if` za pomocą znaków `&&` lub `||` więc jeżeli dany endpoint może wywołać użytkownik o którejś z dwóch roli można to wykorzystać.

Kod 2.29: Przykładowy kod, opracowanie własne

```
@PreAuthorize("hasRole('ADMIN') || hasRole('DEANERY')" )
```

Ostatni element, to określenie w jaki sposób użytkownik będzie dostarczał swoje dane logowania. W module bezpieczeństwa istnieje wiele gotowych implementacji takich jak:

- OAuth 2.0
- HTTP Basic authentication
- wbudowane uwierzytelnianie oparte na formularzach

W przykładowej klasie `CustomWebSecurityConfigurerAdapter` zdecydowano się na **HTTP Basic authentication** co widać po użyciu metody `HttpSecurity::httpBasic()`. OAuth 2.0 i uwierzytelnianie po formularzach również mają swoje metody.

2.7.2 Niestandardowe sposoby autoryzacji użytkownika

Czasami projekt ma szczególne przypadkach użycia, które wymuszają np. integrację logowania z zewnętrznym systemem np. CAS. CAS (ang. Central Authentication Service) to system uwierzytelniania użytkowników. Dzięki niemu można zrzucić całą odpowiedzialność trzymania danych użytkowników i ich logowania zrzucić na niego, a w następnej kolejności używać tych użytkowników w różnych systemach. Dzięki temu też użytkownik może mieć jedno konto do wielu usług. Taki system jest używany w systemach informatycznych Uniwersytetu im. Adama Mickiewicza. Studenci tej uczelni mogą się tymi samymi danymi logować się między innymi do takich aplikacji jak USOSweb, Archiwum Prac Dyplomowych, Ankieter, czy Moodle'a. Uwierzytelnianie przez CAS działa w kilku prostych krokach:

1. Odpytuje serwer CAS'a o zalogowanie użytkownika podając mu też jaki serwis prosi o uwierzytelnienie, na wmi np. link to

Kod 2.30: Przykładowy kod, opracowanie własne

```
https://cas.amu.edu.pl/cas/login?service=url-serwisu&locale=pl
```

2. Jeżeli klient nie był jeszcze zalogowany to zostanie poproszony o podanie loginu i hasła

3. Jeżeli klient został pomyślnie zalogowany wcześniej, to CAS przekierowuje klienta na podany wcześniej serwis z parametrem zapytania `ticket`.

4. Klient wysyła prośbę do serwera o walidację zwróconego `ticketa` (na wmi przykładowo link to

Kod 2.31: Przykładowy kod, opracowanie własne

```
https://cas.amu.edu.pl/cas/validate?service=url-serwisu&ticket
    =zwrócony-ticket
```

5. Jeżeli podany `ticket` był poprawny to serwer zwraca w odpowiedzi dane użytkownika, który się zalogował

Warto też zaznaczyć, że jeżeli użytkownik nie był zalogowany i podał swoje dane logując się do jakieś zintegrowanych z CAS'em usług, to gdy spróbuje zalogować się do jakiejkolwiek z innych zintegrowanych aplikacji to CAS go nie zapyta ponownie o dane autoryzacyjne.

Pojawia się pytanie: *Jak to wykorzystać w kontekście autoryzowania zapytań w restowej aplikacji?* Można np. tworzyć sesje użytkownika na podstawie zautoryzowanego `ticketa` z CAS'a. Jest dostępny również moduł do Spring Security realizujący ten scenariusz. Aby go użyć trzeba go najpierw zaimportować za pomocą mavena.

Kod 2.32: Zależność w pom.xml, opracowanie własne

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-cas</artifactId>
    <versionId>5.3.0.RELEASE</versionId>
</dependency>
```

Następnie trzeba stworzyć konfigurację podobnie jak w przykładach powyżej.

Kod 2.33: Przykładowa klasa konfiguracyjna, opracowanie własne

```
@Configuration
@EnableWebSecurity
public class CasConfig extends WebSecurityConfigurerAdapter {


    @Bean
    public CasAuthenticationFilter casAuthenticationFilter(
        AuthenticationManager authenticationManager,
            ServiceProperties serviceProperties) throws Exception {
```

```
        CasAuthenticationFilter filter = new CasAuthenticationFilter
            ();
        filter.setAuthenticationManager(authenticationManager);
        filter.setServiceProperties(serviceProperties);
        return filter;
    }

    @Bean
    public ServiceProperties serviceProperties() {
        ServiceProperties serviceProperties = new ServiceProperties
            ();
        serviceProperties.setService("http://localhost:8080/login/
            cas");
        serviceProperties.setSendRenew(false);
        return serviceProperties;
    }

    @Bean
    public TicketValidator ticketValidator() {
        return new Cas30ServiceTicketValidator("https://cas.amu.edu.
            pl/cas");
    }

    @Bean
    public CasAuthenticationProvider casAuthenticationProvider(
        TicketValidator ticketValidator,
            ServiceProperties serviceProperties) {

        CasAuthenticationProvider provider = new
            CasAuthenticationProvider();
        provider.setServiceProperties(serviceProperties);
```

```
        provider.setTicketValidator(ticketValidator);
        provider.setUserDetailsService(username -> new User("
            test@test.com", "TestUser", true, true, true, true,
                AuthorityUtils.createAuthorityList("ROLE_ADMIN")));
        provider.setKey("CAS_KEY");
        return provider;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().antMatchers("/login/cas").permitAll
            ().anyRequest().authenticated().and()
                .exceptionHandling().authenticationEntryPoint(
                    authenticationEntryPoint());
    }

    @Override
    protected void configure(final AuthenticationManagerBuilder auth
        ) {
        auth.authenticationProvider(casAuthenticationProvider(
            ticketValidator(), serviceProperties()));
    }

    @Bean
    AuthenticationEntryPoint authenticationEntryPoint() {
        CasAuthenticationEntryPoint entryPoint = new
            CasAuthenticationEntryPoint();
        entryPoint.setServiceProperties(serviceProperties());
        entryPoint.setLoginUrl("https://cas.amu.edu.pl/cas/login");
        return entryPoint;
    }
```

```
}
```

wymagane jest ustawienie wielu elementów w konfiguracji. Na początku tworzymy wszystkie potrzebne zależności jako `Beany`, żeby Spring mógł z nich korzystać. Każde z zależności ma swoje zadanie:

- **CasAuthenticationFilter** - jest potrzebny do wyciągnięcia `ticketa` po przekierowaniu,
- **ServiceProperties** - dostarcza informacji o kliencie, który będzie pytał serwer CAS,
- **TicketValidator** - będzie wykorzystywany do walidacji `ticketa`. Warto też zauważyć, że w tym przykładzie serwer korzysta z protokołu w wersji v3 stąd też nazwa `Cas30ServiceTicketValidator`. Walidator potrzebuje też adresu url gdzie może odpytać serwer,
- **CasAuthenticationProvider** - będzie używał `TicketValidator` i jeżeli walidacja zakończy się sukcesem to użyje ustawionego `UserDetailsService` żeby zwrócić instancje klasy `UserDetails` zawierające informację o użytkowniku w systemie. W tym przykładzie użytkownik jest zwracany zawsze ten sam (`TestUser`). Normalnie powinna tutaj być implementacja, która szuka na podstawie argumentu `username` użytkownika w bazie i zwraca `UserDetails` wypełnione jego danymi. Ważne jest też, że dostajemy tylko String `username`, jako argument więc musimy z CAS'a wybrać jako username taką wartość, która będzie unikalna dla każdego użytkownika (na uczelnianym CAS'ie może to być np. adres email). Wartość ta jest ustawiana przez developerów danej instancji serwera kiedy rejestrujemy domene naszego serwisu. Jeżeli nasz serwis nie jest zaufany dla danego serwera to możemy w zależności od ustawień nie dostać żadnej odpowiedzi lub odpowiedź cząstkową. `CasAuthenticationProvider` potrzebuje też dostać klucz `key`,

aby mógł zidentyfikować tokeny, które zostały wcześniej uwierzytelnione [37]

Poza tym w konfiguracji trzeba ustawić jeszcze dwa elementy:

- pozwolić na dostęp do `/login/cas` dla każdego, aby użytkownicy mogli się zalogować,
- ustawić `authenticationEntryPoint` na instancję klasy `Authentication EntryPoint` z podanym url do logowania do danej instancji CAS'a. Dzięki temu ustawieniu, jeżeli ktoś zażąda dostęp do chronionego zasobu zostanie automatycznie przekierowany do strony logowania w CAS

Podsumowując powyższą konfigurację:

1. Próba dostępu do chronionego zasobu spowoduje błąd uwierzytelniania,
2. Błąd uwierzytelniania spowoduje przekierowanie na stronę logowania CAS
3. Pomyślne zalogowanie przekieruje z powrotem do naszego serwera na `/login/cas`
4. `CasAuthenticationProvider` waliduje zwrócony `ticket`
5. Jeżeli walidacja zakończy się pomyślnie to zostanie pokazany zażądany na początku zasób

Po jednorazowym uwierzytelnieniu Spring Security powinien już pamiętać, że użytkownik przeszedł ten proces i będzie miał dostęp do chronionych zasobów. W sytuacji, w której projekt potrzebuje integracji z jakimś istniejącym systemem, którego Spring nie wspiera, istnieje opcja własnej implementacji poszczególnych elementów. Gdyby CAS nie był wspierany przez Spring to implementacja dla niego mogłaby wyglądać, że system udostępnia endpoint,

który przyjmuje `ticket` zwrócony z CAS'a (zakładając np. że frontend zajął się jego zdobyciem) i zwraca `token` w jednoznaczny sposób przypisany do uwierzytelnionego użytkownika. Następnie ten token będzie podawany w nagłówku autoryzacji. Aby obsłużyć tę sytuację w Spring security trzeba zaimplementować własny `AuthenticationFilter` oraz `AuthenticationProvider`. `AuthenticationFilter` zajmie się wówczas wyjęciem z nagłówka tokenu.

Kod 2.34: Przykładowy kod, opracowanie własne

```
public class AuthenticationFilter extends
    AbstractAuthenticationProcessingFilter {

    //...

    @Override
    public Authentication attemptAuthentication(HttpServletRequest
        request, HttpServletResponse response)
            throws AuthenticationException, IOException,
                ServletException {

        String token = request.getHeader(AUTHORIZATION);
        if (token == null) {
            token = "";
        } else {
            token = StringUtils.removeStart(token, "Bearer").trim();
        }
        Authentication requestAuthentication = new
            UsernamePasswordAuthenticationToken(token, token);
        return getAuthenticationManager().authenticate(
            requestAuthentication);

    }
```

```
}
```

Natomiast `AuthenticationProvider` znajdzie użytkownika w bazie i zwróci odpowiadającą mu instancję `UserDetails`

Kod 2.35: Przykładowy kod, opracowanie własne

```
@Component
public class CustomAuthenticationProvider extends
    AbstractUserDetailsAuthenticationProvider {

    //other fields and methods

    @Override
    protected UserDetails retrieveUser(String username,
        UsernamePasswordAuthenticationToken authentication)
            throws AuthenticationException {

        final String token = authentication.getCredentials().
            toString();
        User user = this.userService.getByToken(token)
                .orElseThrow(() -> new UsernameNotFoundException("
                    Cannot find user with given authority"));
        UserDetails response = new UserDetails() {

            private static final long serialVersionUID = 1L;

            @Override
            public Collection<? extends GrantedAuthority>
                getAuthorities() {
                final AuthorityRoles role = AuthorityRoles.
                    getAuthorityRole(user.getRole())
                        .orElseThrow(() -> new NullPointerException("
```

```
                        Failed to get user role"));
                final List<AuthorityRoles> response = Arrays.asList(
                    role);
                return response;
            }

            // other methods

            @Override
            public boolean isCredentialsNonExpired() {
                if (user.isCredentialsNonExpired()) {
                    userService.save(user);
                    return true;
                }
                return false;
            }

            @Override
            public boolean isEnabled() {
                return true;
            }

        };

        return response;
    }

}
```

Następnie należy dodać przez **dependency injection** `AuthenticationProvider` oraz utworzoną isntancję `AuthenticationFilter` w konfiguracji

Kod 2.36: Przykładowy kod, opracowanie własne

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
    {

   private static final RequestMatcher PROTECTED_URLS = new
       OrRequestMatcher(new AntPathRequestMatcher("/api/**"));

   private CustomAuthenticationProvider provider;

   public WebSecurityConfig(final CustomAuthenticationProvider
       authenticationProvider) {
       super();
       this.provider = authenticationProvider;
   }

   //...

   @Override
   protected void configure(HttpSecurity http) throws Exception {
       http./*auths*/.and().authenticationProvider(provider)
               .addFilterBefore(authenticationFilter(),
                   AnonymousAuthenticationFilter.class).
                   authorizeRequests()
               .anyRequest().authenticated();

   }

   private AuthenticationFilter authenticationFilter() throws
       Exception {
```

```
        final AuthenticationFilter filter = new AuthenticationFilter
            (PROTECTED_URLS);
        filter.setAuthenticationManager(authenticationManager());
        return filter;
    }

    //..
}
```

Opisana powyżej implementacja jest przykładowym rozwiązaniem, równoważnych można zdefiniować więcej. Wykorzystanie modułu Spring Security upraszcza nam to zadanie.Można skorzystać z gotowych rozwiązań lub w razie potrzeby przygotować własną implementację dla systemu z którym wymagana jest integracja.

2.8 Testowanie REST API

Sam proces tworzenia aplikacji wymaga szeregu testów jednostkowych i integracyjnych, same endpointy muszą również zostać przetestowane. Aby ustalić ich prawidłowe zachowanie należy rozpatrywać przypadki, takie jak zakazanie dostępu do danego zasobu przez nieautoryzowanego użytkownika, czy wysłanie zapytania bez podania wymaganych argumentów. Nawet zakładając, że programista na bieżąco wszystko testował, to w trakcie rozwoju projektu czasem nieświadomie jakąś linijką kodu możemy zmienić dostępność jakiegoś zasobu, albo usunąć pewną rolę by przetestować jej wpływ na rozpatrywany w danym momencie problem i zapomnieć dodać jej z powrotem. Z tego typu błędami testowanie jest o tyle istotne, że program się zbuduje i wstanie nawet jeśli kilka endpointów nie będzie sprawdzało roli odpytującego lub jakieś wymagane argumenty staną się opcjonalne. Programista nie będzie za każdym razem sprawdzał każdego przypadku dla wszystkich

endpointów zwłaszcza, że wraz z rozwojem projektu będzie ich przybywać. Dlatego pisanie automatycznych testów do nich jest tak istotne. Pozwala to zaoszczędzić czas oraz potrafi uchronić przed wypuszczeniem na produkcję jakiejś luki w bezpieczeństwie. Do testowania takich sytuacji przydatna jest paczka `org.springframework.test.web.servlet`. Udostępnia ona `MockMvc`, `MockMvcBuilders` których można użyć w następujący sposób.

Kod 2.37: Przykładowy kod, opracowanie własne

```
@RunWith(SpringRunner.class)
@SpringBootTest
@ContextConfiguration
public class CommisionControllerTest {

    //...

    @Test
        public void shouldAddCommisionWithSomeoneIdPrividedAsDeanary
            () throws Exception {
    final User user = this.service.checkForUser(
        TEST_COMMISIONS_DEANERY_EMAIL, null);
    final String token = this.service.login(user).getToken();

    MockMvc mockMvc = MockMvcBuilders
        .webAppContextSetup(webApplicationContext)
        .apply(springSecurity())
        .build();


    mockMvc
        .perform(post(ADD_COMMISION_ENDPOINT + "/" + this.otherUser.
            getId().toString())
```

```
        .header("Authorization", "Bearer " + token)
        .contentType(APPLICATION_JSON_UTF8)
        .content("[]"))
        .andExpect(status().isOk());
    }

    //...
}
```

Zostaje tutaj wykonane zapytanie z podanym poprawnym tokenem na wybrany endpoint z argumentami w postaci pustej tablicy w ciele zapytania oraz id użytkownika w url. Zostało tutaj podane wszystkie wymagane elementy, dlatego na końcu wywołana jest metoda `.andExpect(status().isOk())`. Następne testy powinny obejmować przypadki gdzie pewnych danych brakuje. Przykładowo, w sytuacji gdy nie zostanie podany `token` to oczekiwany rezultat można sprawdzić metodą `status().is4xxClientError()`. Dostępne są również bardziej precyzyjne metody sprawdzenia jak na przykład `status().isMethodNotAllowed()`. Dzięki adnotacją powyżej definicji klasy, w sytuacji gdy scenariusz testowy zostanie odpalony uruchomiona zostanie testowa aplikacja, a `MockMvc` będzie wiedział na jakim porcie jest ona dostępna i gdzie wysłać zapytanie.

2.9 Dokumentacja

Dokumentacja jest bardzo ważnym aspektem tworzenia i rozwoju oprogramowania. O ile na początku jest się w stanie wszystko pamiętać to aplikacje skalują się w zastraszającym tempie dlatego też chociażby zwykła referencyjna dokumentacja bywa pomocna, a jakieś dodatkowe komentarze też mogą zaoszczędzić czasu. W kontekście REST API też istotnym jest aby dokumentować wszystkie endpointy. Nawet najlepszej aplikacji nikt nie użyje jeśli nie będzie

wiedział jak. Problem z dokumentacją jest taki, że jej tworzenie równolegle do projektu byłoby czasochłonne gdyby nie fakt istniejących narzędzi w dużym stopniu automatyzujących ten proces. Standardowym narzędziem do tworzenia dokumentacji dla projektów napisanych w języku Java jest `javadoc`. W zasadzie jedyne czego potrzeba to projekt i maven. Aby wygenerować dokumentację wystarczy wpisać w terminalu `mvn javadoc:javadoc` lub kliknąć w odpowiednim miejscu w IDE. Wygeneruje to dokumentacje w postaci stron `html`. Dokumentacja ta będzie zawierała takie informację jak spis klas, paczek, metody danej klasy, listą klas oznaczonych jako przestarzałe czy hierarchiczne drzewo dziedziczenia. Widok konkretnej klasy będzie też zawierał informację o tym z czego ona dziedziczy, czy jakie interfejsy implementuje z hiperłączami do widoków ich dotyczących. W drugą stronę, interfejs będzie miał informację o tym jakie klasy go implementują. Taką dokumentację warto rozwinąć o dodatkowe komentarze.

Kod 2.38: Przykładowy komentarz javadoc, źródło: `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`

```
/**
 * Provides the classes necessary to create an
 * applet and the classes an applet uses
 * to communicate with its applet context.
 * <p>
 * The applet framework involves two entities:
 * the applet and the applet context.
 * An applet is an embeddable window (see the
 * {@link java.awt.Panel} class) with a few extra
 * methods that the applet context can use to
 * initialize, start, and stop the applet.
 *
 * @since 1.0
 * @see java.awt
```

```
 */
package java.lang.applet;
```

Dzięki temu, komentarze w dokumentacji będą zawierać takie informacje jak od jakiej wersji aplikacji dana aplikacja istnieje czy link do wybranej, istotnej klasy. Możliwe jest również korzystanie z języka `HTML` wprowadzając dodatkowe formatowanie. Jeżeli istniałaby potrzeba dodania ilustracji to można to zrobić poprzez umieszczenie jej w folderze `\src\com\example\doc-files` oraz oznaczenie w kodzie

Kod 2.39: Przykładowy komentarz javadoc, źródło: `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`

```
/**
 * This button looks like this:
 * <img src="doc-files/Button.gif">
 */
```

Przy klasie można dać również informację o tym kto ją tworzył.

Kod 2.40: Przykładowy komentarz javadoc, źródło: `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`

```
/**
 * An {@link AuthenticationProvider} implementation that integrates
     with JA-SIG Central
 * Authentication Service (CAS).
 * <p>
 * This <code>AuthenticationProvider</code> is capable of
    validating
 * {@link UsernamePasswordAuthenticationToken} requests which
    contain a
 * <code>principal</code> name equal to either
 * {@link CasAuthenticationFilter#CAS_STATEFUL_IDENTIFIER} or
```

```
* {@link CasAuthenticationFilter#CAS_STATELESS_IDENTIFIER}. It can
    also validate a
* previously created {@link CasAuthenticationToken}.
*
* @author Ben Alex
* @author Scott Battaglia
*/
```

Natomiast przy metodach można dodać informację o tym jak obsługują błędy, a jak tak to jakie oraz jakie parametry przyjmuje i co zwraca

Kod 2.41: Przykładowy komentarz javadoc, źródło: `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`

```
    * generates token for user and if user don't have name in
        database it will
    * attemp to obtain these from usos api and saves changes in
        database
    *
    * @param authority user we want to login
    * @return user with changed values after save in db
    * @throws UserNotFoundException throwed if user doesn't exist
    */
   public User login(User authority) throws UserNotFoundException {
```

To narzędzie pozwala na szybkie dokumentowanie kodu praktycznie w czasie rzeczywistym w trakcie pisania kodu. Podobne narzędzia istnieją do dokumentowania samych endpointów, a jednym z nich jest **Swagger** w połączeniu z **Swagger UI**, który dostarczy graficzną warstwę dokumentacji. Aby ich użyć trzeba je dodać do zależności aplikacji w pliku `pom.xml`

Kod 2.42: Zależności w pom.xml, opracowanie własne

```
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-
    swagger2 -->
```

```
    <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-swagger2</artifactId>
      <version>${swagger.version}</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/io.springfox/springfox-
        swagger-ui -->
    <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-swagger-ui</artifactId>
      <version>${swagger.version}</version>
    </dependency>
```

oraz utworzyć klasę konifguracyjną w której zostaną podane takie informacje jak jakie paczki **Swagger** ma skanować czy metadane, np. opis projektu.

Kod 2.43: Przykładowa klasa konfiguracyjna, opracowanie własne

```
@Configuration
@EnableSwagger2
public class Swagger2Config extends WebMvcConfigurationSupport {


    @Bean
    public Docket createRestApi() {
        Parameter authHeader = new ParameterBuilder().parameterType(
            "header").name("Authorization")
                .modelRef(new ModelRef("string")).build();
        return new Docket(DocumentationType.SWAGGER_2).apiInfo(
            apiInfo()).select()
                .apis(RequestHandlerSelectors.basePackage("com.
                    plannaplan")).paths(PathSelectors.any()).build()
```

```
                .globalOperationParameters(Collections.singletonList(
                    authHeader));
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry
        ) {
        registry.addResourceHandler("swagger-ui.html").
            addResourceLocations("classpath:/META-INF/resources/");
        registry.addResourceHandler("/webjars/**").
            addResourceLocations("classpath:/META-INF/resources/
            webjars/");
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder().title("plannaplan").description(
            "Aplikacja do zapisów na zajęcia UAM.")
                .termsOfServiceUrl("https://plannaplan.pl/")
                // .contact("")
                .version("1.0").build();
    }
}
```

Następnie **Swagger** już będzie w stanie znaleźć wszystkie endpointy i informacje o nich, a dokumentację znaleźć będzie można po włączeniu aplikacji pod adresem `${host}/swagger-ui.html#/`. Będą tam widoczne takie informacje jak chociażby url endpointu, metoda http, przyjmowane parametry, czy są wymagane czy nie oraz jak wyglądać będzie odpowiedź pomyślnego zapytania i jakie kody http może zwrócić w odpowiedzi. Ponownie to już jest dużo przydatnych informacji i prawdopodobnie z takimi informacjami developerzy już mogliby zacząć używać stworzone REST API. Jednak krótki

opis co dany endpoint robi i jakie mogą być potencjalne problemy jest zawsze mile widziana. Do tego swagger dostarcza nam kilka adnotacji.

Kod 2.44: Przykładowy kod, opracowanie własne

```
@PostMapping(value = { "/user", "/user/{id}" })
@ApiOperation(value = "Create commision with assignents to given
    groups. If group doesn't exist error will be thrown")
public ResponseEntity<String> addCommision(
    @RequestBody @ApiParam(value = "List of groups ids user want to
        assign to. If group doesnt exisit error will be thrown")
        List<Long> groups, @PathVariable(name = "id", required =
        false) Long userId) {
```

Na powyższym przykładzie widać ich zastosowanie. `@ApiOperation` służy do ogólnego opisu dla danego endpointu, a `@ApiParam` do opisu konkretnego parametru. Jeżeli odpowiedź jest czymś więcej niż zwykłym stringiem, to można ją również za pomocą odpowiednich adnotacji opisać.

Kod 2.45: Przykładowy kod, opracowanie własne

```
@ApiModel(description = "Response shows information about logged
    user.", value = "TokenResponse")
public class TokenResponse {
    @ApiModelProperty(value = "user token used to verify requests")
    private String token;
    @ApiModelProperty(value = "user token needed to refresh")
    private String refreshToken;
    @ApiModelProperty(value = "user id in database")
    private Long id;
    @ApiModelProperty(value = "user app role")
    private String authorityRole;
    @ApiModelProperty(value = "user unviersity email")
    private String email;
```

Tutaj widać już, że opisywane są konkretne pola, a nie metody jak to było w **javadoc** i ma to sens jakby się zastanowić ponieważ z punktu widzenia konsumenta REST API nie interesują go metody, a pola odpowiedzi. Tych metod w zwróconym `jsonie` nawet de facto nie będzie. Analogicznie do pierwszego przykładu i tutaj `@ApiModel` służy do opisania ogólnie odpowiedzi, a `@ApiModelProperty` do opisu konkretnych pól odpowiedzi. Podobnie jak **javadoc**, **swagger** pozwala na tworzenie wartościowej dokumentacji w czasie rzeczywistym. Oszczędza to czas na jej przygotowanie,poprzez adnotację uzupełniamy ją automatycznymi elementami oraz wprowadzamy pewien standard. Dzięki czemu staje się ona wartościowym dokumentem związanym z prowadzonym projektem.

Rozdział 3

Tworzenie aplikacji SPA w frameworkach React.js, Vue.js oraz Angular

Autor rodziału: Hubert Wrzesiński

Istnieje kilka sposobów na wykonanie aplikacji internetowej. Początkowo renderowanie aplikacji odbywało się w całości po stronie serwera, a za każdym razem, gdy użytkownik odwiedzał nową podstronę, ładowała się ona od zera. Z czasem, kiedy JavaScript zaczął się błyskawicznie rozwijać, zaczęto stosować rozwiązanie Single-Page Application (SPA), które obecnie stało się dla wielu firm swego rodzaju standardem, nowym domyślnym rozwiązaniem. Wraz z rozwojem języka JavaScript, coraz bardziej popularne stawały się także frameworki takie jak React.js, Vue.js oraz Angular, dzięki którym język ten stawał się przystępniejszy, a tworzenie z ich pomocą stron stało się znacznie łatwiejsze. Z ich pomocą można osiągnąć identyczne rezultaty,

jednak sposób w jaki to robimy różni się w zależności od tego, którego z nich użyjemy. Wszystkie trzy mają swoich zwolenników, ze względu na dość istotne różnice między nimi. W tym rozdziale zostanie położony nacisk na różnice między sposobami tworzenia Single-Page Application przy pomocy tych trzech frameworków, zostaną przedstawione ich wady i zalety, różnice w sposobie tworzenia komponentów z ich pomocą, a także zostanie pokazane dlaczego w systemie PlanNaPlan wykorzystano React.js do stworzenia Single Page Application.

3.1 Single-Page Application

Single-Page Application to podejście do tworzenia aplikacji polegające na tym, by załadować w jednym momencie cały front-end, który będzie pobierał dane, przychodzące z serwera i prezentował je w aplikacji. Współdziała ona z użytkownikiem poprzez dynamiczne ładowanie bieżącej strony przy pomocy JavaScriptu, zamiast ładowania nowych stron z serwera. Takie rozwiązanie miało swoje zastosowanie w przeszłości, kiedy możliwości przeglądarek, komputerów i języków skryptowych były mniejsze, z powodu czego używało się metodyki Multi-Page Application.

Główną zaletą korzystania z Single-Page Application jest szybkość działania naszej aplikacji, a także odciążenie serwera z ilości wykonywanych operacji. Widok aplikacji jest wczytywany tylko raz, a następnie wczytywane są jedynie elementy, które potrzebują zmiany. Dzięki temu wysyłane są wyłącznie zapytania o dane, których potrzebujemy w danej chwili, dzięki temu właśnie optymalizujemy użycie serwera. Pośród licznych zalet tego rozwiązania, istnieją jednak również wady zastosowania Single-Page Application, dla przykładu: wielu użytkowników posiada w swojej przeglądarce wyłączony JavaScript, przez co aplikacja stworzona w taki sposób nie będzie działać tak jak należy, lub nawet nie zdoła się uruchomić

3.2 Frameworki oferujące Single-Page Application

Proces tworzenia aplikacji internetowych w metodyce Single-Page Application może być niezwykle skomplikowany, jeżeli nie zdecydujemy się na użycie frameworków JavaScript. W dalszej części tego rozdziału zostaną omówione trzy najpopularniejsze frameworki do budowania Single-Page Application pod kątem ich wad i zalet, a także przedstawiona zostanie motywacja do wyboru jednego z nich w projekcie inżynierskim PlanNaPlan

3.2.1 Angular

Jednym z trzech najpopularniejszych frameworków do tworzenia aplikacji internetowych jest Angular. Został napisany przez inżynierów z Google, a ich celem było ułatwienie i zrewolucjonizowanie sposobu pisania aplikacji webowych. Angular jest w zasadzie przepisaniem AngularJS przez ten sam zespół, w nieco zmienionej formie różniącej się od siebie kilkoma znaczącymi kwestiami. Główną zmianą jest to, że Angular wykorzystuje hierarchię komponentów, jako podstawową cechę architektoniczną. Został napisany w języku TypeScript, a jego obecna stabilna wersja, 11.0.7, została wydana 7 stycznia 2021 r[1]. Angular jest bardzo popularnym frameworkiem, o czym może świadczyć to, że korzystają z niego tak wielkie firmy jak Microsoft, Google czy PayPal [40], jednakże uznawany jest on za jeden z trudniejszych frameworków, dlatego wielu programistów na samym początku swojej przygody z frontendem raczej stroni od Angulara.

[1]Źródło: `https://en.wikipedia.org/wiki/Angular_(web_framework)`

3.2.2 Vue.js

Kolejnym równie popularnym frameworkiem JavaScript jest Vue.js, podobnie jak Angular jest on oparty na modelu open source i został napisany w TypeScriptcie służąc do tworzenia aplikacji działających po stronie klienta. Podstawowym zamierzeniem Vue.js jest deklaratywne renderowanie komponentów. Główna biblioteka skupia się tylko i wyłącznie na warstwie widoku i prezentowaniu HTML. Wszelkie dodatkowe, zaawansowane funkcje wymagane do tworzenia złożonych aplikacji, takie jak routing, zarządzanie stanem i narzędzia budowania są oferowane przez oficjalne biblioteki i pakiety pomocnicze. Ponadto często są one polecane i rozwijane przez samych twórców frameworka [40]. Jego pierwsza wersja została stworzona już w roku 2014 przez Evana You, a cały framework jest na bieżąco aktualizowany, najnowsza na dzień dzisiejszy stabilna wersja pochodzi z 30 grudnia 2020 roku[2].

3.2.3 React.js

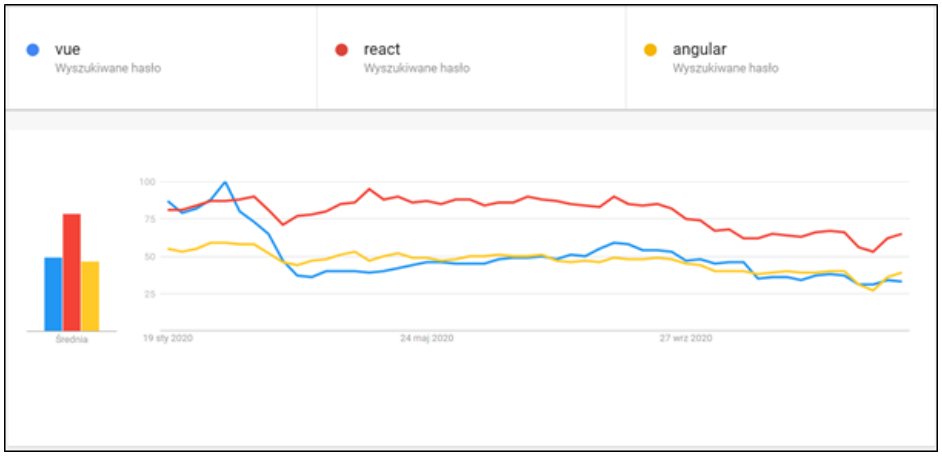
Ostatnim z przedstawianych frameworków jest biblioteka open source - React.js, napisana w całości przez programistów zajmujących się także Facebookiem, a obecnie oprócz samych twórców utrzymywany jest także przez społeczność i entuzjastów języka JavaScript i Reacta. Początkowo rozwijany był przez Jordana Walke [3], jako samodzielne narzędzie stosowane jedynie w niektórych elementach Facebooka, jednak z czasem zostało ono przekształcone w niezależną bibliotekę, przeznaczoną do użytku zewnętrznego. Służy do tworzenia jednostronicowych aplikacji internetowych opartych o komponenty wielokrotnego użytku, które są od siebie niezależne. W swoich aplikacjach tego frameworku użyły między innymi takie firmy jak Netflix, Slack, czy New York Times [40].

[2]Źródło: `https://en.wikipedia.org/wiki/Vue.js`

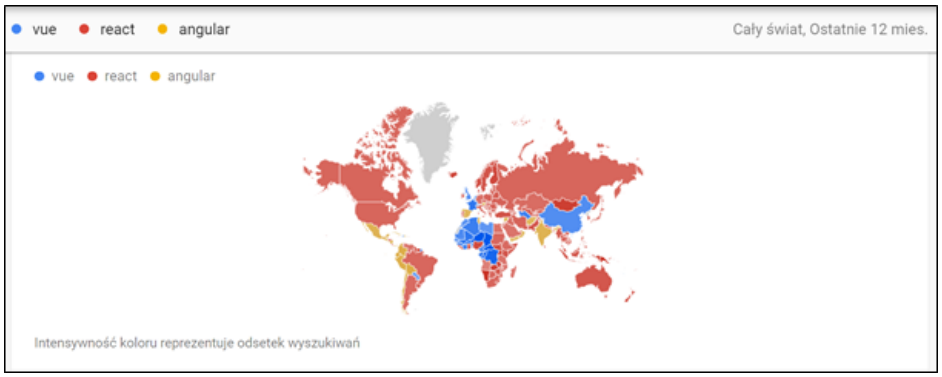
[3]Źródło: `https://en.wikipedia.org/wiki/React.js`

3.3 Popularność

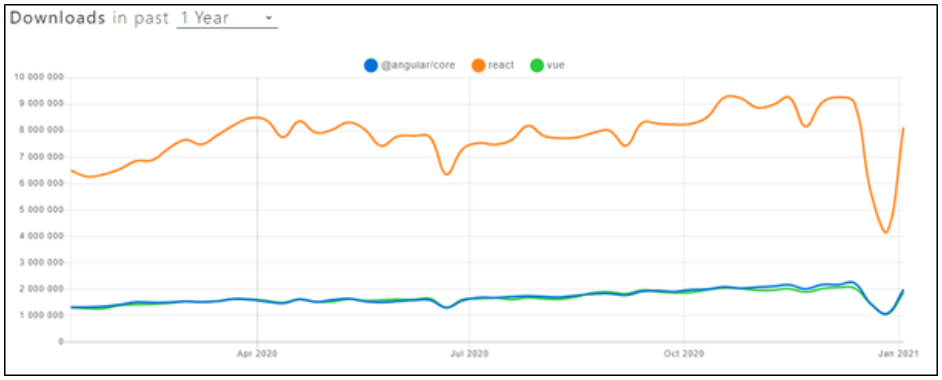
Popularność frameworka ma często dość istotne znaczenie, ponieważ jako ludzie zazwyczaj wybieramy rzeczy, które lubią też inni ludzie. Ponadto mamy duże szanse, że popularne frameworki nie przestaną być wspierane przez twórców, a także łatwiej znaleźć w Internecie pomoc, jeżeli napotkamy jakiś problem, bądź też trudność w stworzeniu czegoś.



Rysunek 10: Popularność wyszukiwania frameworków.
Źródło: Google Trends `https://trends.google.pl/`



Rysunek 11: Popularność frameworków w poszczególnych krajach.
Źródło: Google Trends `https://trends.google.pl/`



Rysunek 12: Trend liczby pobrań przy pomocy npm.
Źródło: Google Trends `https://trends.google.pl/`

Na podstawie wykresów wyszukiwani Google Trends (10, 11), a także statystyk porównawczych pobierania pakietów przy pomocy menadżera pakietów npm (12) śmiało można powiedzieć, że React jest najpopularniejszym spośród tych trzech frameworów. Jednak warto także zaznaczyć, że zdecydowana większość Francuzów i Brytyjczyków woli używać Vue, a Angular ma swoich fanów między innymi w Ameryce Środkowej, Indiach czy Kolumbii. Takie statystyki niewątpliwie działają na korzyść Reacta. Tak duża popularność skutkuje nie tylko tym, że zdecydowanie łatwiej będzie nam znaleźć rozwiązanie napotkanych problemów, ale także sprawia to, że zdecydowanie łatwiej będzie nam znaleźć miejsce pracy, z uwagi na to, że firmy zazwyczaj sięgają po rozwiązania ogólnodostępne, a także takie, których używa jak najwięcej ludzi.

3.4 Komponenty

Komponenty [38] to integralna część każdego z trzech wymienionych frameworków, nieważne czy mówimy o Vue, Reactcie czy Angularze, komponent to najważniejszy element aplikacji typu Single Page. Ich rola polega na otrzymywaniu danych wejściowych i na ich podstawie generowanie zawartości

strony internetowej. Wystarczy zadeklarować ich treść raz, a dzięki temu, że można ich używać dowolną ilość razy, w dowolnej części kodu, znacznie ułatwia tworzenie aplikacji.

3.4.1 Komponent w React.js

React używa komponentów wykorzystując tak zwany JSX - łączy standardy JavaScript ze składnią HTML składając to wszystko w jeden plik .jsx. React opiera się na koncepcji wirtualnego DOM, dzięki tej technice umożliwia tworzenie komponentów dla interfejsu użytkownika. Komponenty te mogą posiadać własne stany oraz obsługiwać złożone struktury danych, których nie można używać w standardowym DOM.

Kod 3.1: Klasowy komponent guzika, stworzony przy pomocy React.js

```
class ButtonClickCounter extends React.Component {
    constructor(props){
        super(props)
        this.state = {clicks: 0}
        this.handleClick = this.handleClick.bind(this);
    }
    handleClick(){
        this.setState({
            clicks: this.state.clicks + 1
        });
    }
    render() {
        return (
            <button onClick={this.handleClick}>
                Clicked {this.state.clicks} times!
            </button>
        );
    }
```

```
}
```

Powyżej przykład klasowego komponentu guzika (3.1), stworzonego przy pomocy React.js, który wywołuje zdarzenie onClick, wywołujące funkcję handleClick, która zwiększa stan „clicks", a następnie wartość tego stanu jest wyświetlana.

3.4.2 Komponent w Vue.js

W odróżnieniu od React.js komponent podzielony jest na dwie części:

Część HTML, w której potrzebujemy przypisać do przycisku zdarzenie click z pomocą dyrektywy Vue v-on:click.

Kod 3.2: Część HTML komponentu guzika, stworzona przy pomocy Vue.js

```
<div id="example-1">
    <button v-on:click="counter += 1">Add 1</button>
    <p>The button above has been clicked {{ counter }} times.</p>
</div>
```

Kod 3.3: Część JS, w której tworzymy obiekt Vue z danymi zmiennej „counter"

```
var example1 = new Vue({
    el: '#example-1',
    data: {
        counter: 0
    }
})
```

3.4.3 Komponent w Angular

Angular w odróżnieniu od Reacta i Vue, Angular używa Typescriptu jednak oddziela on część HTML od części TypeScriptowej, co widzimy na

przykładzie poniżej.

Kod 3.4: Część JS, w której tworzymy obiekt Vue z danymi zmiennej „counter”

```
<section ng-app="clickApp">
  <div ng-controller="clickCtrl">
        <button ng-click="addNum()">
            <p>Add Num</p>
        </button>
        <p>{{count}}</p>
        <button ng-click="reset()">
            <p>Reset Counter</p>
        </button>
    </div>
</section>

angular.module("clickApp", [])

angular.module("clickApp").controller("clickCtrl",function($scope){

$scope.count = 0;
    $scope.addNum = () => {
    $scope.count ++;
};
    $scope.reset = () => {
        $scope.count = 0;
};
});
```

3.5 Zalety i wady frameworków

Każda technologia ma swoje wady i zalety. Dlatego teraz zostaną przedstawione wady i zalety każdego z nich [39].

3.5.1 Zalety frameworka Angular

Angular posiada szereg zalet. Jedną z nich jest to, że jest on frameworkiem MVW (Model-View-Whatever), tradycyjnie używany jako MVC (Model-View-Controller). Oznacza to, że aplikacja podzielona jest na trzy połączone ze sobą części. Dzięki temu łatwiejsze jest pisanie dobrze ustrukturyzowanego kodu, co jest przydatne w sytuacji, kiedy tworzymy bardziej złożone projekty. Ponadto, komponenty korzystają z szablonów, czyniąc je bardziej czytelne, ponieważ używają standardowych znaczników HTML. W przypadku Angulara warto także pamiętać o prostym w implementacji dwukierunkowym wiązaniu danych. Jest to jedną z ważniejszych funkcji tego frameworka. Redukuje ono ilość kodu, jaki piszemy, a wszelkie zmiany w modelu wpływają na widok, i odwrotnie.

3.5.2 Wady frameworka Angular

Oprócz licznych zalet, Angular posiada też kilka wad. Po pierwsze brakuje w nim wirtualnego DOM. Wirtualny DOM jest uproszczoną kopią DOM, która umożliwia szybko zmianę dowolnego elementu bez konieczności renderowania całego DOM. W odróżnieniu od innych nowoczesnych frameworków rozwiązanie to nie jest obsługiwane przez żadną z wersji Angulara. Należy także pamiętać o tym, że Angular, mimo tego, że jest frameworkiem JavaScript został utworzony do użytku z TypeScriptem, dlatego należy poświęcić nieco czasu na naukę zmodyfikowanej składni, co może zniechęcać niektórych użytkowników. Angular posiada także stosunkowo niską prędkość renderowania. Poprzez ograniczone użycie shadow DOM i brak wirtualnego domu szybkość renderowania widoków jest wolniejsza w porównaniu z innymi frameworkami.

3.5.3 Zalety frameworka Vue.js

Podobnie jak Angular, Vue jest frameworkiem MVC (Model-View-Controller). Dzięki temu pisanie dobrze ustrukturyzowanego kodu jest znacznie prostsze, co szczególnie przydaje się w przypadku tworzenia bardziej złożonych aplikacji. Jedną z ważniejszych zalet Vue jest stosunkowo mały rozmiar frameworka. Nie posiada on wielu funkcji w momencie pobrania, ale jest również bardzo łatwo rozszerzalny. Deklaratywne szablony to kolejna z zalet tego frameworka. Dzięki temu, że szablony Vue.js zapisane są w HTML, sprawiają, że są one bardzo czytelne. W odróżnieniu od Angulara, w tym frameworku został zaimplementowany wirtualny DOM, przez co aplikacje stworzone przy pomocy Vue.js są bardzo wydajne, biorąc pod uwagę inne frameworki. Warto także wspomnieć o tym, że Vue korzysta z czystego JavaScript, dzięki temu nie ma potrzeby nauki innych języków programowania, jak to ma miejsce w przypadku chociażby Angulara, a w porównaniu do innych frameworków, jest on najłatwiejszą technologią do nauki i rozpoczęcia pracy.

3.5.4 Wady frameworka Vue.js

Do wad Vue.js można zaliczyć między innymi małą społeczność korzystającą z tego frameworka. Vue jest zdecydowanie mniej popularny w porównaniu z Reactem, co sprawia, że szukanie rozwiązań napotkanych problemów może sprawiać trudności. Z uwagi na niższą popularność, stworzono mniej rozwiązań rozszerzających funkcjonalność frameworka.

3.5.5 Zalety frameworka React.js

React.js posiada zdecydowanie więcej zalet niż wad, w konsekwencji czego React ma niewątpliwie największą społeczność, w porównaniu do innych frameworków. Możliwość komunikowania się z wieloma innymi, bardziej doświadczonymi użytkownikami znacznie ułatwia pisanie kodu. React nie używa

szablonów, a cala logika komponentów jest stworzona przy pomocy JavaScriptu, dzięki czemu oferuje on większą elastyczność, a także łatwe przesyłanie dużej ilości danych przy jednoczesnym zachowaniu stanu DOM. Podobnie jak pozostałe dwa frameworki korzysta z wirtualnego DOM, dzięki czemu zyskujemy na szybkości. Dwukierunkowe wiązanie danych było zaletą dla Angulara, a jednokierunkowe wiązanie danych we frameworku React może być również zaletą. Takie rozwiązanie, sprawia, że widok reaguje na wszystkie zmiany wprowadzone w modelu, ale nie na odwrót, co skutkuję mniejszą możliwością wystąpień jakichkolwiek błędów. Jednak w przeciwieństwie do Angulara, który wymagał od nas używania klas, interfejs w Reactcie można utworzyć przy użyciu komponentów funkcyjnych, które upraszczają kod.

3.5.6 Wady frameworka React.js

Jednak React nie jest idealny. React rekomenduje używanie JSX zamiast zwyczajnego JavaScriptu i HTMLa. Twórcy twierdzą, że JSX jest szybszy i latwiejszy niż JavaScript, jednak żeby w pełni korzystać z Reacta należy nauczyć się tej modyfikacji JavaScriptu. Sprawia to, że React nie jest najłatwiejszy do nauki, a ponadto oprócz samego Reacta należy się zapoznać z kilkoma innymi bibliotekami i modułami, które są niezbędne do stworzenia aplikacji. Kolejnym mankamentem Reacta dla niektórych może być to, że aplikacje w Reactcie nie posiadają predefiniowanej struktury. Znaczy to, że za strukturę w odpowiada programista, uzależniając ją w pełni od jego doświadczenia.

3.6 React w projekcie PlanNaPlan

W aplikacji PlanNaPlan React został użyty z kilku powodów. Po pierwsze potrzebne było narzędzie, dzięki któremu można stworzyć aplikację dynamiczną i jednostronicową. Wymagania te doskonale spełniał React. Kolejnym

czynnikiem, który wpłynął na decyzję wyboru tego frameworku było to, że React był znany, chociaż w niewielkim stopniu przez większość członków grupy, co znacznie ułatwiło sprawną eliminację napotkanych błędów. Mimo tego, że React wymaga znajomości JSX, ilość czasu, jaką poświęca się na zapoznanie z tą technologią, zwróciła się z nawiązką, jeżeli chodzi o wydajność programowania. Ostatnią rzeczą, która przesądziła o wyborze Reacta ponad innymi frameworkami było to, że dzięki funkcjonalności ContextAPI przekazywanie danych między backendem a frontendem przebiega bardzo sprawnie, a sama funkcjonalność jest prosta w użyciu.

Rozdział 4

UX/UI w tworzeniu aplikacji webowych

Autor rodziału: Maciej Głowacki

Tworzenie aplikacji, strony internetowej czy systemu informatycznego to dużo więcej niż sam pomysł na konkretną usługę lub produkt [42]. Wydawałoby się, że jest to najważniejszy element i prawdą jest, że bez dobrego pomysłu każdy z kolejnych kroków w procesie wytwarzania oprogramowania nie miałby sensu. Jednakże pomysł to dopiero początek. Istnieje szereg ogólnych praktyk przydatnych przy rozwoju oprogramowania, niezależnie od tego w jakim sektorze znajduje się finalny produkt. Wiele z tych praktyk dotyczy interfejsu aplikacji i stawia użytkownika na pierwszym miejscu. Mowa tutaj o szeroko pojętym UX i UI.

4.1 User Experience Design vs User Interface Design

Często słyszy się o tym jak świetny jest UX danej aplikacji, czy jak słaby UI ma dana strona internetowa. Te dwa określenia często używane są wymiennie, jednakże nie są ze sobą tożsame. To przecinanie się na wielu płaszczyznach przejawia się chociażby w tym, że interfejs (UI) jest sposobem w jaki dostarczamy użytkownikowi całość wrażeń takich jak wygoda, poczucie niezawodności, i użyteczności jakie doświadcza podczas korzystania z danego produktu (UX). Dlatego warto zgłębić się w temat tych zagadnień i wyróżnić cechy wspólne oraz różnice.

4.1.1 UI Design

UX design, skrót rozwijany jako user experience design to sposób tworzenia aplikacji, który stawia użytkownika na pierwszym miejscu i to na nim skupia się przede wszystkim[42]. Warto zaznaczyć, że termin ten nie dotyczy szczegółów części wizualnej, lecz ogólnego doświadczenia użytkownika z danym produktem. Dobry UX w odniesieniu do aplikacji webowej czy mobilnej równoważny jest z intuicyjnym interfejsem oraz odpowiednim zidentyfikowaniem potrzeb użytkownika, tak by stworzona aplikacja była możliwie prosta [42]. Jest to bardzo złożony proces, ale możliwe jest wyróżnienie jego najważniejszych elementów.

4.1.2 Research

Dużą część tego obszaru stanowi research czyli swoiste przeprowadzanie badań na użytkownikach. Jest to próba zrozumienia użytkownika. Wchodzi w to między innymi przeprowadzanie dogłębnych rozmów z użytkownikami oraz przeprowadzanie ankiet, tak aby zrozumieć ich wymagania wobec pro-

duktu. Bez wykonania takich badań prawidłowe zrozumienie użytkownika wydaje się być niemożliwe, a produkt, mimo dobrego pomysłu, skazany jest na porażkę [43]. Dlatego wartość takich badań jest nieoceniona, zwłaszcza w tworzeniu MVP(Minimum Viable Product), który możemy zaprezentować klientowi.

4.1.3 User personas

Na podstawie informacji zebranych od użytkowników następuje przygotowanie tak zwanych user personas, czyli portretów reprezentujących użytkowników systemu. Tworzenie oprogramowania i designów skupiających się na użytkowniku, wydaje się być niemożliwe bez przygotowania user personas, ponieważ pozwalają one zidentyfikować konkretne grupy użytkowników wraz z ich oczekiwaniami. Dzięki temu produkt nie jest oderwany od rzeczywistości [43]. Proces tworzenia user personas możemy podzielić na kilka części:

- Rozpoczyna się od przeprowadzenia rozmów z jak największą grupą docelowych odbiorców(jest to wcześniej wspomniany research). Dzięki temu możemy uzyskać realistyczny obraz użytkownika.
- Następnie musimy postarać się znaleźć cechy wspólne użytkowników i podzielić ich na grupy. Z każdej takiej grupy możemy przygotować user persona.
- Przykładowe user persona zawiera takie informacje jak status finansowy, wiek, miejsce zamieszkania, informacje o rodzinie, status cywilny czy zawód. Poza tym zawiera również zidentyfikowane schematy zachowania. Listę problemów, które należy rozwiązać w stworzonym oprogramowaniu oraz potrzeby jakie trzeba spełnić, tak aby zadowolić użytkownika.

ZDJĘCIE & DANE	SZCZEGÓŁY	CEL
 Typowy student  Pracownik dziekanatu	Student: • wiek: 20-30 • płeć: przewaga mężczyzn • dochód: aplikacja jest darmowa • cechy: jak najszybciej i najwygodniej skorzystać z usługi. • lojalność wobec marki: budowana wraz z rozwojem produktu i wdrażaniem na kolejnych uniwersytetach Pracownik dziekanatu • wiek 25-55 • płeć: przewaga kobiet • cechy: pracownicy dziekanatu nie muszą być obeznani w informatyce, z racji tego aplikacja powinna być intuicyjna i przejrzysta	Aktualna aplikacja do tworzenia planu nie działa tak, jakby chcieli tego zarówno studenci, jak i pracownicy dziekanatu. Naszym celem więc jest stworzenie aplikacji niezawodnej, działającej zgodnie z oczekiwaniami naszych studentów i pracowników placówki

Rysunek 13: User personas wykonane dla aplikacji PlanNaPlan.
Źródło: Opracowanie własne.

4.1.4 Use cases

Posiadając dotychczasowe dane możliwe jest zdefiniowanie use casów, tzw. przypadków użycia. Dzięki liście use casów można lepiej określić złożoność systemu oraz koszty jego wytworzenia i utrzymania [43]. Każdy use case opisuje kolejne kroki, które pomogą wypełnić cel, który postawił sobie użytkownik. Typowy use case zawiera:

- nazwę przypadku użycia
- aktora - rolę, zewnętrzny system lub urządzenie korzystające z use case'a. Use case świadczy usługę na rzecz aktora i dostarcza mu wartościowy rezultat
- scenariusz główny - listę kroków wykonywanych przez aktora
- scenariusz alternatywny - gdy, któryś z warunków w scenariuszu głównym nie zostanie spełniony
- warunki początkowe, które muszą być spełnione do wykonania zadania

- warunki końcowe - skutki wykonania przypadku użycia z sukcesem, dające się zaobserwować w interfejsie systemu lub w jego interakcji z otoczeniem.
- sytuacje problemowe - błędy, które wykonać może aktor w trakcie procesu

Poniżej przedstawiony jest przykładowy use case z aplikacji PlanNaPlan obrazujący aktualizację planu studenta w systemie przez pracownika dziekanatu oraz diagram przedstawiający system holistycznie.

Scenariusz główny:

1. Pracownik dziekanatu wyszukuje studenta
2. (opcjonalne) Pracownik dziekanatu wybiera grupę/przedmiot i ją usuwa
3. (opcjonalne) Pracownik dziekanatu wyszukuje grupę/przedmiot i ją dodaje

Scenariusz alternatywny - pracownik próbuje edytować plan w trakcie aktywnej tury zapisów:

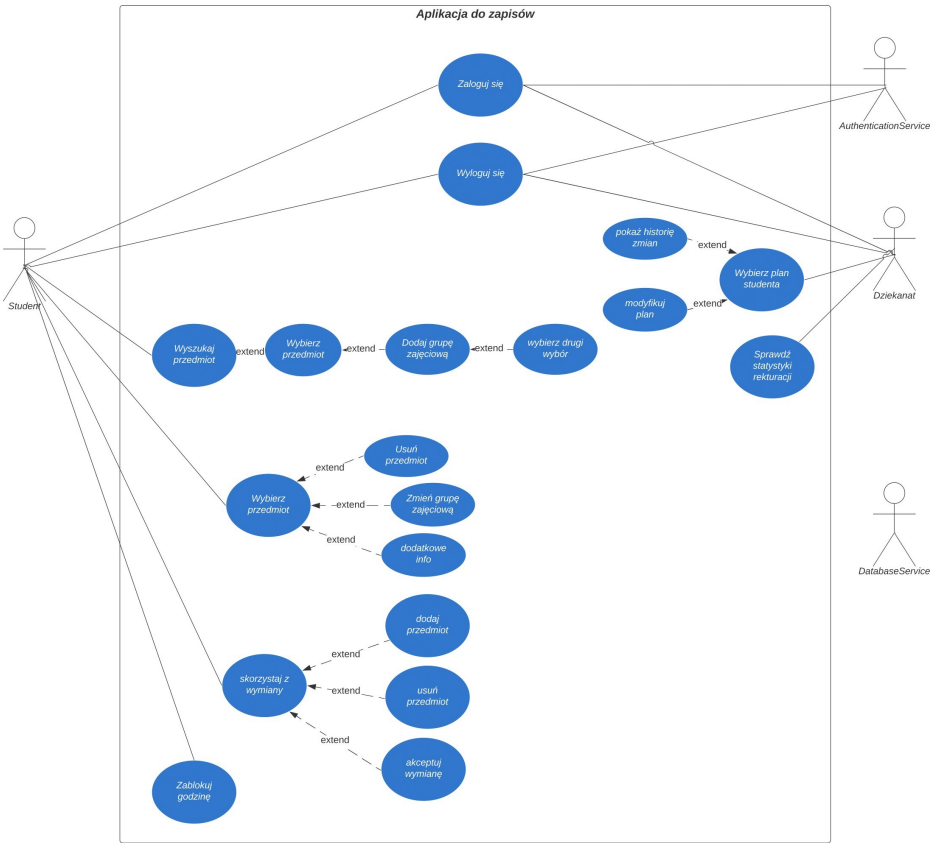
1. Pracownik próbuje dokonać zmiany
2. Zostaje zwrócony błąd

Sytuacje problemowe/exceptions:

1. Czy istnieje sytuacja, w której dziekanat powinien móc dokonać zmian w trakcie trwania zapisów?
2. Co, gdy dziekanat dokona zmiany przez przypadek i chce wrócić do poprzedniego stanu planu? Powinniśmy dostarczyć możliwość cofnięcia zmian lub przetrzymywania jakiejś historii

Warunek końcowy/postconditions:

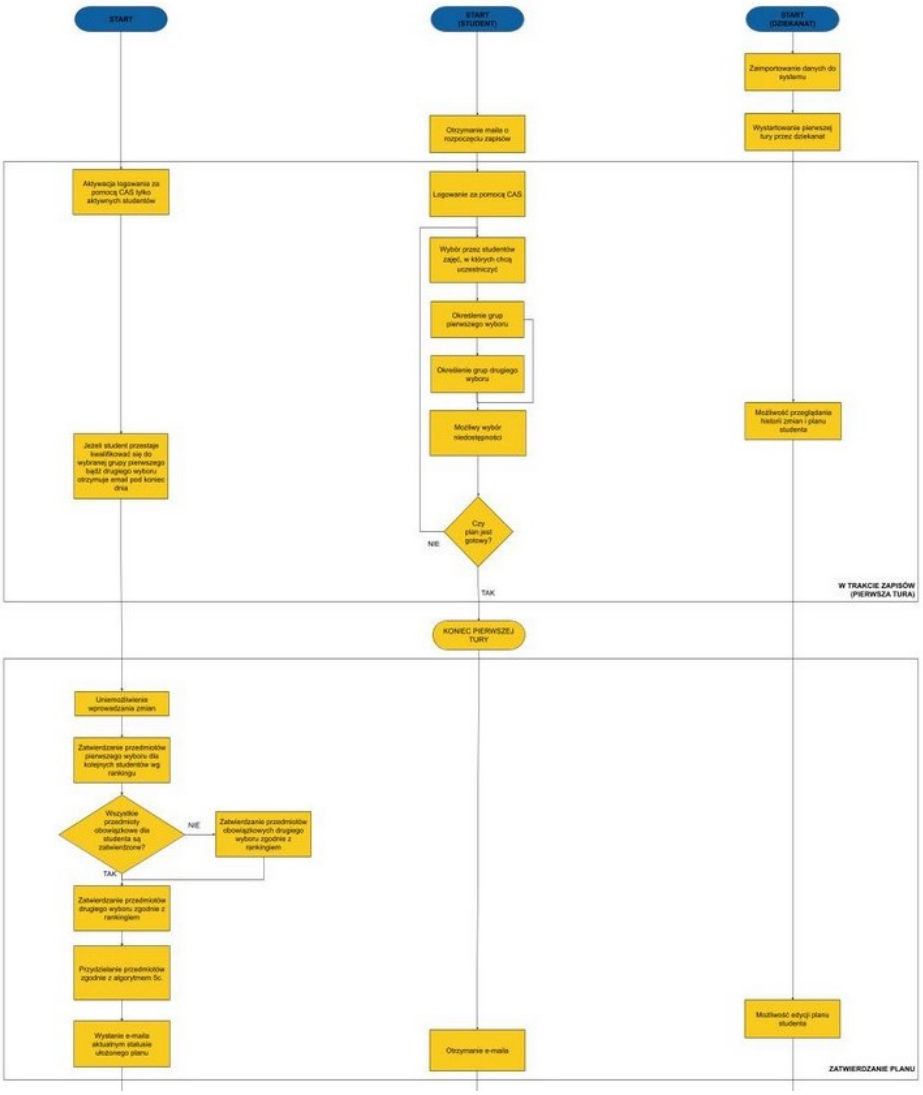
1. Plan studenta zostaje zaktualizowany w systemie.



Rysunek 14: Use cases w formie diagramu dla aplikacji PlanNaPlan.
Źródło: Opracowanie własne.

4.1.5 User flows

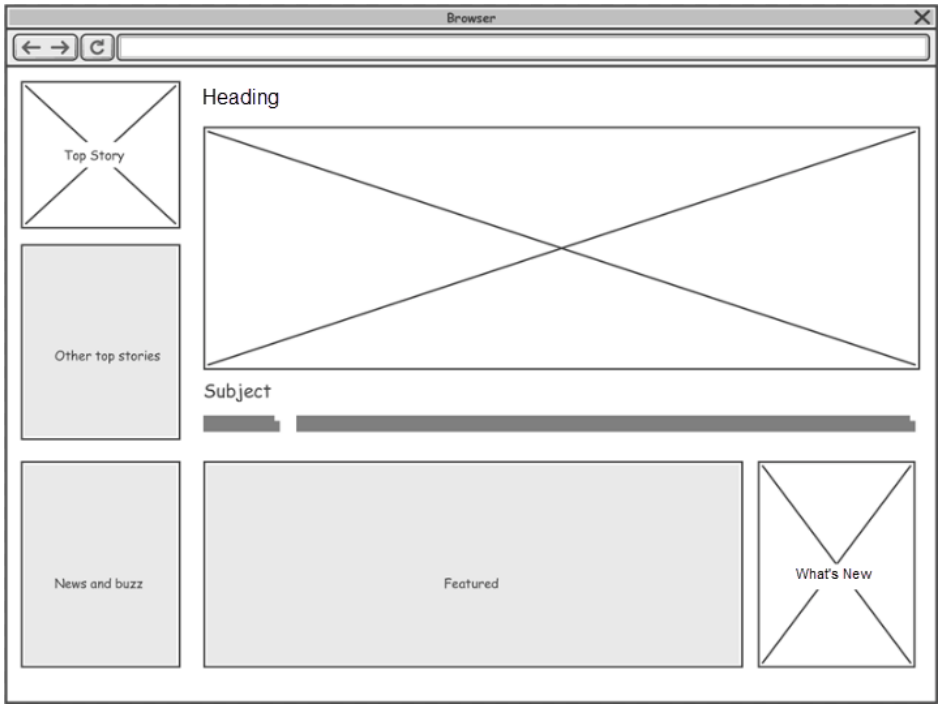
Następnym etapem jest przygotowanie user flows. Są to wizualne reprezentacje cyklu życia aplikacji oraz ścieżek, które mogą wybrać użytkownicy aplikacji. Diagram sekwencji działań zaczyna się od punktu wejścia użytkowników do systemu. W przypadku aplikacji PlanNaPlan, dla studenta jest to otrzymanie maila z informacją o rozpoczynających się zapisach, jest on przedstawiony na rysunku nr 15. Sam diagram zbudowany jest z różnych form geometrycznych, każdej posiadające swoje własne znaczenie. Prostokąty oznaczają kolejne akcje dziejące się wewnątrz systemu. Romby natomiast oznaczają punkty decyzyjne, gdzie w zależności od spełnienia określonych warunków wybierana jest inna ścieżka. Dzięki user flows tworzone są bardziej intuicyjne interfejsy oraz łatwiej jest ustalić ich kolejność pojawiania się na ekranie użytkownika. Są również świetnym sposobem prezentacji produktu potencjalnym klientom [43].



Rysunek 15: User flows dla aplikacji PlanNaPlan.
Źródło: Opracowanie własne.

4.1.6 Wire frames

Ostatnim etapem części badawczej tworzenia UX jest przygotowanie wire frames. Polega on na zdefiniowaniu przepływu informacji wewnątrz produktu i naszkicowaniu wstępnego prototypu przyszłych interfejsów. Jest to świetny sposób na zdefiniowanie sposobów interakcji użytkownika z systemem poprzez wypozycjonowanie przycisków i dostępnych menu na diagramie [43]. W tym momencie unikamy rzeczy rozpraszających uwagę takich jak dobór kolorów, czcionek czy rzeczywistego tekstu, który ma pojawić się w interfejsie, ponieważ pozwala to zidentyfikować, czy użytkownik będzie w stanie się przemieszczać swobodnie w finalnej wersji aplikacji.



Rysunek 16: User flows dla aplikacji PlanNaPlan.
Źródło: https://medium.com/mockplus/basic-ui-ux-design-concept-difference-between-wireframe-prototype-a041b95f7cce

4.2 UI Design

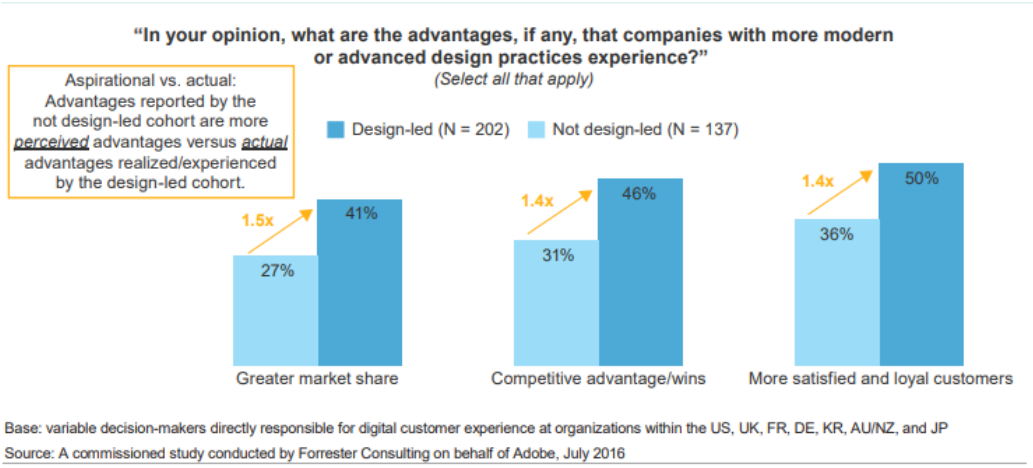
W odróżnieniu od UX design, UI design skupia się głównie na wyglądzie produktu. W przypadku aplikacji webowej jest to interfejs wyświetlany na ekranie monitora użytkownika, w przypadku aplikacji mobilnej jest to ekran smartfonu. Gdy jest wykonany dobrze, użytkownik praktycznie nie zwraca na niego uwagi. Gdy wykonany jest źle, uniemożliwia efektywne korzystanie z aplikacji. Ponadto badania pokazują, że aż 50 % użytkowników nie powraca do aplikacji z powodu nieestetycznego interfejsu[46].

4.2.1 Badania Forrester Consulting

Z badań Forrester Consulting wynika również, że firmy stawiające UI design jako jeden ze swoich głównych priorytetów, zyskują wyraźną przewagę biznesową. Odbiorcy ich produktów są bardziej lojalni i zadowoleni, co przekłada się bezpośrednio na wielkość udziału w rynku. Technikę tą zaimplementowało IBM zatrudniając pomiędzy 2012 i 2017 rokiem ponad tysiąc UI designerów i tworząc markę prawdziwie stawiającą na użytkownika.

4.2.2 Badania McKinsey

Tematu wartości biznesowej dobrego designu podjęła się również firma McKinsey, która na przestrzeni pięciu lat badała praktyki w sferze UI designu ponad 300 firm. Pod koniec badań każda z firm otrzymała tak zwany McKinsey Design Index (MDI) i na podstawie tej wartości udało się dojść do wniosku, że istnieje silna korelacja pomiędzy wysoką wartością MDI, a lepszym wynikiem biznesowym. Firmy znajdujące się w pierwszym kwartylu uzyskały wzrost przychodów o 32niż firmy posiadające niski MDI. Sprawdziło się to pośród wszystkich trzech badanych sektorów: medycznego, sprzedaży towarów i bankowości detalicznej. Zwraca to uwagę na to, że dobry UI design jest ważny bez względu na to jakiego rodzaju produkty i usługi oferuje dana



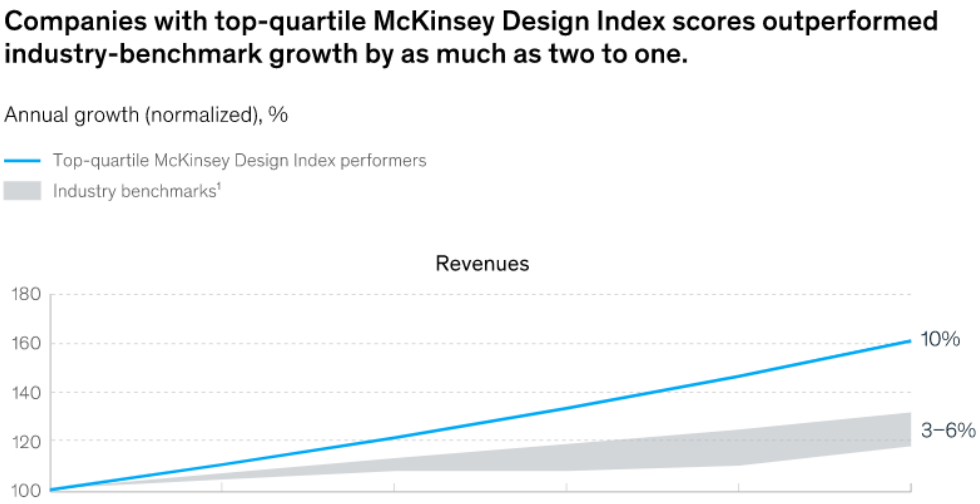
Rysunek 17: Wpływ designu na wartość biznesową.
Źródło: `https://landing.adobe.com/dam/downloads/whitepapers/305222.en.forrester.design`

firma. Warto również zaznaczyć, że różnica w przychodach pomiędzy firmami z drugiego, trzeciego i czwartego kwartyla była niewielka, co wskazuje na to, że rynek nagradza firmy wyjątkowo stawiające na UI design.

4.2.3 Inne sposoby mierzenia wartości biznesowej UI

Mierzenie wartości biznesowej dobrego UI możliwe jest także na kilka sposobów. Wśród tych metryk znajduje się:

- bounce rate, który wskazuje jaki procent odwiedzających opuszcza daną stronę internetową bądź aplikację bez przejścia do podstrony (wykonuje tylko jedno zapytanie do serwera). Wysoki bounce rate może być wynikiem błędnie prowadzonej kampanii reklamowej.
- oceny użytkowników w sklepach App Store i Play Store. Możliwa jest tutaj analiza kluczowych elementów aplikacji wymagających poprawy. Jednakże odnosi się to tylko do aplikacji mobilnych.



Rysunek 18: Firmy z pierwszego kwartyla osiągnęły lepsze wyniki biznesowe. Źródło: `https://www.mckinsey.com/business-functions/mckinsey-design/our-insights/the-business-value-of-design`

- długość trwania sesji użytkownika. Pozwala to zidentyfikować, ile czasu użytkownik poświęca na wykonanie danej akcji oraz określić czy możliwe jest wprowadzenie optymalizacji, by przyciągnąć więcej użytkowników.

4.2.4 Interfejs produktu

Interfejs, będąc graficznym layoutem składa się z wielu mniejszych elementów, które razem tworzą większą całość. Możemy tutaj wymienić przyciski, które użytkownik może kliknąć. Tekst, który użytkownik może przeczytać. Różnego rodzaju obrazki takie jak logo produktu, pola wpisywania tekstu, animacje i przejścia. Każdy z tych elementów wizualnych musi zostać odpowiednio zaprojektowany.

To właśnie UI design jest odpowiedzialny za dobór kolorystki, kształtów przycisków, szerokości linii, odpowiednich czcionek. Dobranie ikonek i zadbanie o czytelność oraz zachowanie odpowiedniego kontrastu. Wszystko to z myślą

o zachowaniu responsywności aplikacji. To w rękach osoby odpowiedzialnej za UI jest zadbanie o estetykę i atrakcyjność wizualną aplikacji, tak by komponowała się z jej planowanym przeznaczeniem. W związku z tym inaczej będzie wyglądać aplikacja rządowa służąca do wypełniania dokumentów podatkowych, w porównaniu z aplikacją służącą do pomocy w przeprowadzce. Wydaje się być to oczywiste i to właśnie dopasowanie się do potrzeb klienta jest obowiązkiem UI designera.

4.2.5 Wartości UI design

Dobry i skuteczny UI design skupia się na pewnych określonych wartościach, które mogą zostać wykorzystane zarówno przy tworzeniu tradycyjnych aplikacji webowych i mobilnych oraz przy tworzeniu aplikacji głosowych. Do tych wartości możemy zaliczyć:

- przewidywalność - użytkownik zawsze powinien być w stanie przewidzieć skutek wykonanych przez siebie akcji. Nigdy nie powinien się zadawać sobie pytań takich jak: "Do czego służy ten przycisk?", Ćo mam zrobić, żeby wykonać swoje zadanie?ćzy "Jak trafiłem na ten ekran?". Dobrym przykładem jest zmiana koloru przycisku po kliknięciu. Dzięki temu użytkownik nie musi zastanawiać się dwukrotnie czy jego akcja rzeczywiście została wykonana. Innym subtelnym wyrazem przewidywalności jest wyświetlanie paska postępu, wskazującego etap ładowania pliku.

- spójność - zgodnie z prawem Jakoba [45] użytkownik spędza zaledwie drobną część swojego czasu korzystając z tworzonego produktu. Większość czasu spędza korzystając z innych aplikacji. W związku z tym należy zachować spójność z terminologią i sposobem prezentowania treści w innych aplikacjach. Przykładowo nie warto zmieniać określenia takiego jak 'koszyk zakupów' na 'twoje idealne prezenty'. Jedynym efektem, który uda się osiągnąć będzie odrzucenie użytkownika od produktu.

Unikalność można zaprezentować na innych polach.

- Estetyka i minimalizm - według tej zasady interfejs nie powinien zawierać informacji zbędnej lub rzadko używanej. Każda dodatkowa informacja utrudnia korzystanie z systemu. Nie jest równoznaczne to z usunięciem wszelkich walorów wizualnych. Zwraca jedynie uwagę na ich odpowiedzialny i przemyślany dobór zgodnie z wnioskami, które udało się uzyskać z analizy UX. Dzięki temu użytkownik będzie mógł wykonać swoje zadania szybko i z łatwością.

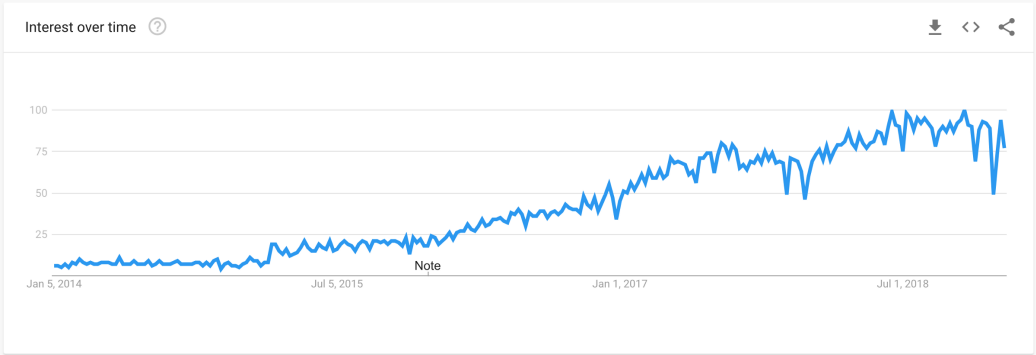
- elastyczność - jeżeli jest to możliwe, każda akcja podjęta przez użytkownika powinna być odwracalna. Zachęca to użytkownika do eksploracji systemu i korzystania z mniej znanych opcji. W przypadku aplikacji PlanNaPlan jest to możliwość usunięcia wybranego przedmiotu. Czy możliwość wielokrotnego zapisu zmienionego planu.

4.3 Typescript i jego rola w Developer Experience

W ciągu ostatnich lat odnotowano znaczny wzrost udziału języka Typescript w web developmencie. Jest to najpopularniejszy z języków kompilowanych do Javascript. Tendencja ta wciąż jest wzrostowa i u jej podstaw znajduje się kilka konkretnych powodów, które są związane z jego wpływem na polepszeniem tzw. Developer Experience czyli doświadczenia pracy programisty z danym językiem programowania.

4.3.1 Zalety języka Typescript

Typescript wprowadza do web developmentu statyczne typowanie znane z Javy czy C#. Dzięki temu poprawie ulega dokumentacja kodu. Staje się



Rysunek 19: Firmy z pierwszego kwartyla osiągnęły lepsze wyniki biznesowe. Źródło: `https://medium.com/javascript-scene/the-typescript-tax-132ff4cb175b`

on czytelniejszy dla developera bez dużego doświadczenia w danym projekcie. Zapewnia także opcjonalność typowania co sprawia, że jest bardzo elastyczny i przyjazny [41]. Poza tym wybór pluginów dla tego języka dostępnych w różnych IDE takich jak Atom, Visual Studio Code czy Webstorm jest na najwyższym poziomie i znacząco wpływa na komfort pracy. Wprowadzenie statycznego typowania wpływa także na poprawę dokumentacji API, ponieważ jest ona zawsze zgodna z najnowszą wersją kodu produkcyjnego. Ułatwiony zostaje również refactoring, poprzez korzystanie z odpowiednich skryptów. Ponadto statyczne typowanie umożliwia korzystanie z narzędzi automatycznego uzupełniania kodu takich jak Intellisense dla VS code oraz eliminuje wszelkiego rodzaju literówki, będące częstym problem w pracy z językiem Javascript [41]. Wszystkie te cechy zebrane razem przekładają się widocznie na poprawę jakości kodu oraz poprawę developer experience.

Rysunek 20: Porównanie kodu napisanego w języku Typescript z kodem napisanym w języku Javascript.

Źródło: `https://res.cloudinary.com/`

Spis rysunków

Bibliografia

[1] Dokumentacja systemu USOS,
https://www.usos.edu.pl/dokumentacja-wdrozeniowa-i-prezentacje,
dostęp marzec 2020 r.

[2] Dokumentacja systemu CAS,
https://apereo.github.io/cas/6.2.x/index.html,
dostęp maj 2020 r.

[3] Dokumentacja relacyjnej bazy danych,
https://mariadb.com/kb/en/documentation,
dostęp maj 2020 r.

[4] Dokumentacja Apache.
https://httpd.apache.org/docs-project/,
dostęp listopad 2020r.

[5] Architektury Systemu Gentoo,
https://wiki.gentoo.org/wiki/Property:Architecture,
dostęp maj 2020 r.

[6] Dokumentacja instalacji systemu Gentoo Linux,
https://wiki.gentoo.org/wiki/Handbook:Main_Page,
dostęp maj 2020 r.

[7] Dokumentacja pakietu `www-servers/apache`,
`https://wiki.gentoo.org/wiki/Apache`,
dostęp maj 2020 r.

[8] Najnowsze wiadomości i artykuły systemu Gentoo Linux,
`https://www.gentoo.org/news/`,
dostęp maj 2020 r.

[9] Uwagi do wydania CentOS,
`https://wiki.centos.org/Manuals/ReleaseNotes`,
dostęp grudzień 2020 r.

[10] Unix i Linux. Przewodnik administratora systemów.
Wydanie V
Autorzy: E.Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley, Dan Mackin
ISBN Książki drukowanej: 978-83-283-4175-3

[11] Artykuł porównujący SOAP i REST,
`https://blog.i-systems.pl/soap-vs-rest/`,
dostęp styczeń 2021 r.

[12] Artykuł porównujący SOAP i REST,
`https://global4net.com/ecommerce/soap-czy-rest-porownanie/`,
dostęp styczeń 2021 r.

[13] Artykuł o popularności REST API,
`https://blog.restcase.com/the-rise-of-rest-api/`,
dostęp styczeń 2021 r.

[14] Artykuł o tworzeniu Bean'ów w Spring Boot
`https://bykowski.pl/tworzenie-beanow-w-spring/`,
dostęp styczeń 2021 r.

[15] Dokumentacja Java Docs
https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html,
dostęp styczeń 2021 r.

[16] Dokumentacja Spring Framework oraz Spring Security
https://spring.io/,
dostęp styczeń 2021 r.

[17] Wprowadzenie do Spring Security
https://spring.io/projects/spring-security,
dostęp styczeń 2021 r.

[18] Artykuł o Spring Framework na Wikipedii
https://en.m.wikipedia.org/wiki/Spring_Framework,
dostęp styczeń 2021 r.

[19] Repozytoruim z przykładami Spring Security
https://github.com/spring-projects/spring-security/tree/5.4.2/samples,
dostęp styczeń 2021 r.

[20] Artykuł jak spring boot ulatwia tworzenie aplikacji w javie
https://global4net.com/ecommerce/jak-spring-boot-ulatwia-tworzenie-aplikacji-w-javie/,
dostęp styczeń 2021 r.

[21] Strona Hibernate
https://hibernate.org/,
dostęp styczeń 2021 r.

[22] Oficjalna strona maven
https://maven.apache.org/,
dostęp styczeń 2021 r.

[23] Artykuł o automatycznym tworzeniu bazy danych przez Hibernate
https://nullpointerexception.pl/czy-automatyczne-tworzenie-bazy-przez-hibernate-jest-dobre/,
dostęp styczeń 2021 r.

[24] Post na Stack Overflow o złym tworzeniu tabel relacji przez Hibernate
https://stackoverflow.com/questions/51700339/one-to-many-mapping-creates-3-tables,
dostęp styczeń 2021 r.

[25] Artykuł o Hibernate Query Language
https://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html,
dostęp styczeń 2021 r.

[26] Oficjalna Strona Spring
https://spring.io/,
dostęp styczeń 2021 r.

[27] Dokumentacja Spring Expression Language
https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/expressions.html,
dostęp styczeń 2021 r.

[28] Post na Stack Overflow - dlaczego należy unikać field injection
https://stackoverflow.com/questions/39890849/what-exactly-is-field-injection-and-how-to-avoid-it,
dostęp styczeń 2021 r.

[29] Oficjalna strona Swagger
https://swagger.io/,
dostęp styczeń 2021 r.

[30] Artykuł o wzorcu dependecy injection
https://typeofweb.com/wzorce-projektowe-dependency-injection/,
dostęp styczeń 2021 r.

[31] Artykuł na wikipedii o Dependency Injection
https://en.wikipedia.org/wiki/Dependency_injection,
dostęp styczeń 2021 r.

[32] Kompedium poradników na tematy związane ze Spring Boot oraz Spring Security
https://www.baeldung.com/,
dostęp styczeń 2021 r.

[33] Film instruktażowy na temat wtrzykiwania zależności przez konstruktor w Spring Boot
https://www.youtube.com/watch?v=IOZzxmJVus0,
dostęp styczeń 2021 r.

[34] Firm instrukażowy na temat Spring Security
https://www.youtube.com/watch?v=her_7pa0vrg,
dostęp styczeń 2021 r.

[35] Data wydania Spring Boot
https://www.baeldung.com/new-spring-boot-2,
dostęp styczeń 2021 r.

[36] Konfiguracja Spring Security przez xml
https://spring.io/blog/2013/07/03/spring-security-java-con

fig-preview-web-security,
dostęp styczeń 2021 r.

[37] Kod źródłowy CasAuthenticationProvider
https://github.com/spring-projects/spring-security/blob/master/cas/src/main/java/org/springframework/security/cas/authentication/CasAuthenticationProvider.java,
dostęp styczeń 2021 r.

[38] Artykuł porównujący strukturę frameworków React.js, Vue.js, Angular,
https://areknawo.com/the-world-beyond-react-vue-angular,
dostęp grudzień 2020 r.

[39] Artykuł porównujący w ogólności frameworki React.js, Vue.js, Angular,
https://clockwise.software/blog/angular-vs-react-vs-vue/,
dostęp styczeń 2021 r.

[40] Artykuł porównujący w ogólności frameworki React.js, Vue.js, Angular,
https://www.codeinwp.com/blog/angular-vs-vue-vs-react/,
dostęp styczeń 2021 r.

[41] Artykuł porównujący języki javascript i typescript,
https://infinijith.medium.com/javascript-vs-typescript-which-is-better-2020-updated-871866a3c68c,
dostęp styczeń 2021 r.

[42] Artykuł o nowoczesnym UX/UI
https://trends.uxdesign.cc/,
dostęp styczeń 2021 r.

[43] Universal UX Design: Building Multicultural User Experience
Wydanie I

Autor: Alberto Ferreira
ISBN Książki drukowanej: 978-0128024072

[44] Designing Interfaces : Patterns for Effective Interaction Design
Wydanie III
Autorzy: Jenifer Tidwell, Charles Brewer, Aynne Valencia
ISBN Książki drukowanej: 1492051969

[45] Artykuł o prawie Jakoba
https://www.oreilly.com/library/view/laws-of-ux/9781492055303/ch01.html,
dostęp styczeń 2021 r.

[46] Artykuł o wpływie UX na wyniki biznesowe
https://www.koderlabs.com/blog/5-impacts-of-great-ux-design-on-business/,
dostęp styczeń 2021 r.