Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Michał Jaroń Nr albumu: 123456
Jarosław Socha Nr albumu: 123456
Piotr Zalas Nr albumu: 361374

Framework oparty o wzorzec mikrousług na przykładzie portalu dla ZUS

Praca licencjacka
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
mgra Michała Możdżonka

Maj 2017

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

Potem dopiszemy.

Słowa kluczowe

mikrousługi, SOA, trudne sprawy

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.127. Ubezpieczenia społeczne

Tytuł pracy w języku angielskim

Framework based on microservices pattern ilustrated by portal for ZUS

Spis treści

Rozdział 1

Wprowadzenie

1.1. Motywacja

Celem naszego projektu było stworzenie prototypu internetowego frameworka do komunikacji obywatela z urzędami podobnego do Platformy Usług Elektronicznych ZUS [PUE], a następnie zaimplementowanie na niej kilku wybranych przypadków użycia. Dzięki temu petent chcący coś załatwić w urzędzie nie będzie musiał wychodzić z domu. Pewną inspiracją do stworzenia takiego systemu był rządowy portal obywatel.gov.pl [MCO].

Wyzwaniem, które należało uwzględnić w fazie projektowania, był szeroki zakres użytkowników systemu: od zwykłych obywateli, poprzez urzędników, aż po przedsiębiorców. Każda z tych grup użytkowników miałaby zupełnie inne uprawnienia: byłoby co najmniej niestosowne, gdyby przedsiębiorca wnioskowałby o urlop macierzyński dla prowadzonej przez siebie firmy. Podobnie obywatel nie powinien mieć możliwości przyznania sobie emerytury lub renty (o ile nie jest urzędnikiem). Główne przypadki użycia systemu sprowadzałyby się do trzech wariantów: wypełniania wniosków, sprawdzania odpowiedzi na wypełniony wniosek i rozpatrywania wniosków. Pobocznymi przypadkami byłyby: rezerwacja terminu wizyty w urzędzie w przypadku nadzwyczaj zawiłych spraw, prezentacja różnych danych użytkownikowi (np. stanu ubezpieczenia, wysokości przyznanej renty lub wymaganego ustawowo pouczenia) oraz wymiana korenspondencji z urzędnikiem. W obecnie istniejącym systemie PUE jest jeszcze jeden przypadek użycia: czat z konsultantem. Nasza platforma powinna umożliwiać stosunkowo prostą realizację niemal wszystkich spośród wymienionych wyżej przypadków użycia.

Kiedy użytkownik zaloguje się do naszego serwisu powinien zobaczyć pulpit, na którym wyświetlone są „kafelki" odpowiadające poszczególnym usługom. Kliknięcie na wybrany kafelek powinno wyświetlić bardziej szczegółowy widok odpowiadający podjętej akcji. Szczególnym życzeniem naszego klienta było, by architektura naszej aplikacji była oparta na mikrousługach. Każda mikrousługa otrzymywałaby na wyłączność fragment pulpitu ograniczony do kafelka, z możliwością przełączenia do trybu pełnoekranowego, gdzie przejmowałaby wtedy kontrolę nad większością wyświetlanego obszaru. Aby to umożliwić, należało opracować ustandaryzowany i łatwo rozszerzalny interfejs do komunikacji między cienkim klientem a mikrousługą. Preferowane było rozwiązanie, w którym komunikacja odbywałaby się bez pośrednictwa serwera serwującego stronę internetową. Zamiast tego wykonywane byłyby asynchroniczne zapytania do mikrousługi (być może poprzez warstwę integracji).

Postawione przed nami wymagania były głównie natury niefunkcjonalnej. Dołączanie kolejnej usługi do systemu powinno być maksymalnie proste. Na tę prostotę składałby się ustandaryzowany interfejs do komunikacji pomiędzy poszczególnymi usługami oraz modułowy i łatwo rozszerzalny interfejs użytkownika. To pozwoli nam stosunkowo małym kosztem

podłączać i odłączać kolejne usługi wraz z rozwojem cyfrowej administracji. Usługi powinny tworzyć jeden ekosystem, z którego nie wychodziłby użytkownik. System powinien być w dużej mierze odporny na awarie i zachowywać spójność oraz poprawność przechowywanych danych obywateli. W szczególności powinny być spełnione normy „12 Factor App" [TFA], w tym ta mówiąca, że awaria jednego serwera lub usługi nie powinna wpływać na działanie pozostałych, niezależnych od niej udostępnianych usług. Rozwój i utrzymanie platformy powinien być możliwy także dla mniej doświadczonych programistów, których zatrudnienie generuje mniejsze koszty wytworzenia kodu. W ten sposób nasz projekt realizowałby plan zrównoważonego rozwoju, zapobiegając centalizacji ośrodków programistycznych, a z drugiej strony prowadziłby do wymiernych oszczędności na wynagrodzeniach.

1.2. Zleceniodawca - ZUS

Naszym zleceniodawcą był Zakład Ubezpieczeń Społecznych. Według „Rocznika Statystycznego Ubezpieczeń Społecznych" [RSUS] w 2011 roku w ZUS było ubezpieczonych 14 milionów obywateli. Dochód Zakładu wyniósł w owym roku 155 796 milionów złotych. Do obsługi takiej ilości petentów i zarządzania taką kwotą pieniędzy było zatrudnionych przeciętnie 44 766 urzędników.

W związku z tym, że projektowaliśmy system dla administracji państwowej z którego będą korzystać miliony obywateli, musieliśmy bardzo ostrożnie dobierać technologie, z których korzystaliśmy. Wybranie rozwiązania opartego na niekorzystnej licencji mogło się w przyszłości łączyć z problemami w postaci konieczności zapłacenia bardzo wysokiej opłaty licencyjnej, wyłączenia usługi i przebudowania jej na nowo, udostępnienia części systemu na licencji open source, a nawet konieczności nieodpłatnego przekazania przechowywanych danych obywateli licencjodawcy. Preferowane były rozwiązania, które albo były oparte na tzw. wolnych licencjach (Apache, MIT), albo takie, do których nasz klient licencje już posiadał (m. in. IBM DB2, komercyjna wersja PostgreSQL oraz korporacyjne rozwiązania firmy Microsoft). W miarę możliwości powinniśmy korzystać z nowoczesnych, przyszłościowych i rozwijanych technologii, tak, by odsunąć jak najdalej w przyszłość konieczność przebudowy systemu z powodu zmieniających się trendów i narastającego długu technologicznego.

Innym istotnym aspektem naszego projektu było zerwanie z wizerunkiem ZUSu jako instytucji przestarzałej, niewydolnej i przeciążonej biurokracją. Szata graficzna naszej platformy powinna być przyjemna dla oka, a układ graficzny elementów logiczny, prosty do zrozumienia i konsekwentny. Użytkownik nie może być bombardowany zagadkowo brzmiącymi komunikatami o „wchodzeniu na poziom bezpieczeństwa 1" i „zamiarze korzystania z usług biznesowych", a sytuacje takie miały nierzadko miejsce w obecnej, produkcyjnej wersji platformy PUE. W żadnym przypadku praca z naszym systemem nie powinna przypominać nieprzyjemnych interakcji, jakich można doświadczyć w tzw. urzędowym "okienku", a to nakłada na nas obowiązek rozwiązania problemu skalowalności aplikacji.

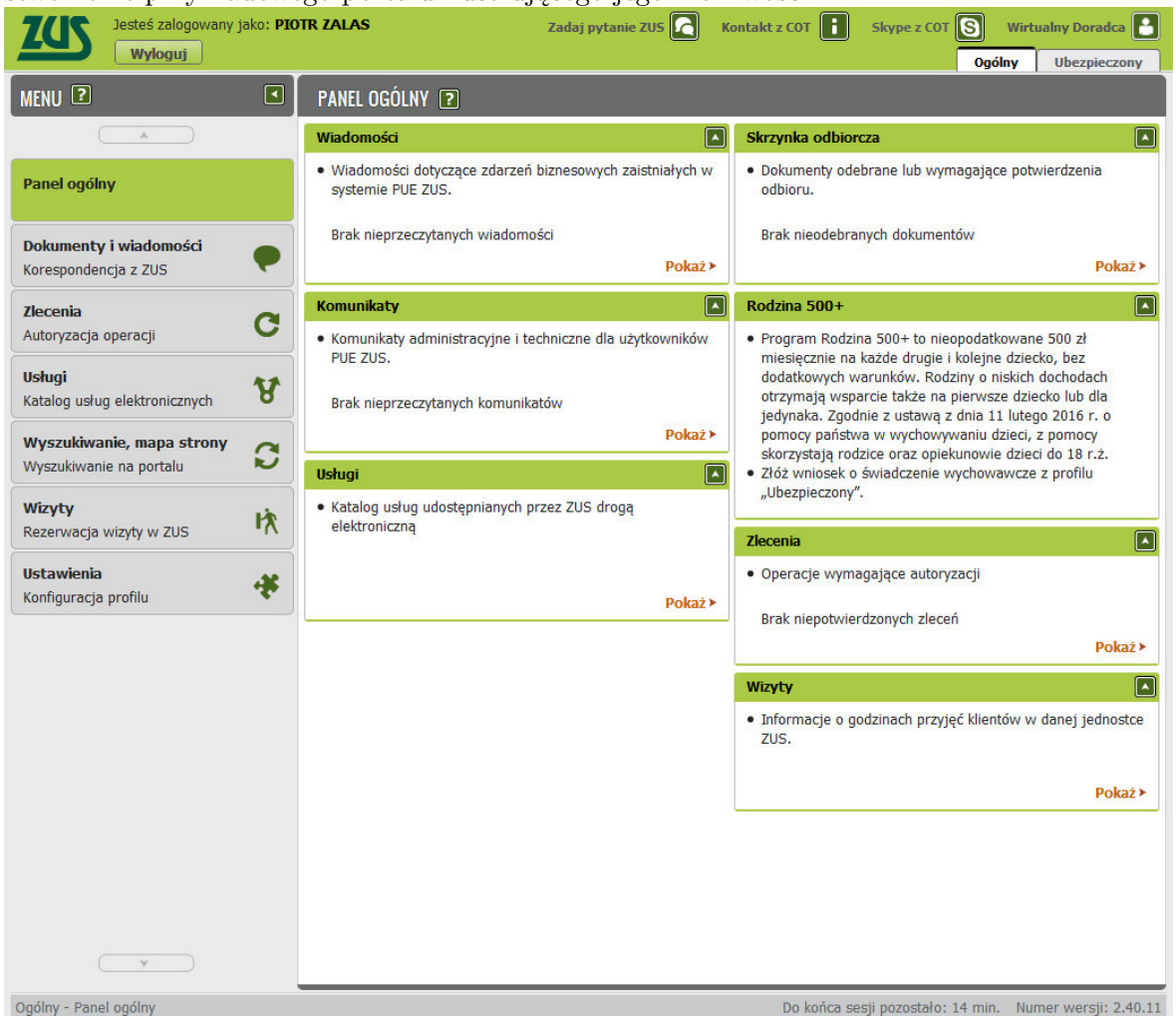
1.3. Produkt końcowy

TODO

Rozdział 2

Wymagania

Głównym wymaganiem postawionym przed naszym frameworkiem było umożliwienie reimplementacji Platformy Usług Elektronicznych ZUS (widocznej na poniższym obrazku) oraz stworzenie przykładowego portalu ilustrującego jego możliwości.



Naszemu zespołowi pozostawiono bardzo duży stopień swobody przy projektowaniu i implementacji portalu. Główny nacisk położono na innowacyjność stworzonego rozwiązania. Niezależnie od tego, w trakcie naszych rozmów z Klientem zostały wyłonione przedstawione niżej wymagania.

2.1. Wymagania funkcjonalne

Stworzony przez nas portal powinien implementować kilka rezprezentatywnych przypadków użycia celem ilustracji możliwości naszego frameworka i łatwości, z jaką się w nim tworzy aplikacje. Wybraliśmy następujące przypadki użycia: powiadomienia, skrzynkę pocztową, usługę 500 plus.

Najprostrzym w implementacji przypadkiem użycia jest sprawdzanie dostępnych powiadomień przez użytkownika. Tuż po udanym wejściu na nasz portal użytkownikowi powinien zostać wyświetlony ekran na którym zobaczyłby krótkie statystyki i informację, co zmieniło się od jego ostatniej wizyty na portalu.

Kolejna grupa przypadków użycia związana jest z mechanizmem przypominającym wewnętrzną pocztę. Użytkownik powinien być w stanie wyświetlić listę wysłanych i odebranych wiadomości. Po wybraniu odpowiedniej wiadomości z listy powinna ukazać mu się jej treść. Możliwa jest też odpowiedź na wiadomość do nadawcy. Szczególnie istotny jest fakt, że z opisanego mechanizmu mogą korzystać różne usługi. Przykładowo, usługa odpowiedzialna za obsługę wniosków 500+ wysyła wiadomości związane z różnymi zdarzeniami dotyczącymi cyklu życia wniosku.

Sztandarowym przypadkiem użycia ilustrującym pełnię możliwości naszego frameworka jest usługa odpowiedzialna za wnioski 500+. Użytkownik może przy jej pomocy wypełnić i złożyć wniosek o świadczenie. Po wysłaniu formularza powinien otrzymać stosowne powiadomienie na swoją skrzynkę pocztową. Następnie będzie mógł sprawdzić status złożonego wniosku oraz podejrzeć jego treść. W międzyczasie urzędnik ZUS otrzyma dostęp do panelu, w którym będzie mógł zdecydować o dalszych losach wniosku. Jeżeli po sprawdzeniu załączonych danych postanowi zaakceptować lub odrzucić wniosek, to będzie mógł to uczynić przy pomocy odpowiedniego guzika. Informacja o tym fakcie natychmiast zostanie doręczona do wnioskodawcy.

2.2. Wymagania niefunkcjonalne

Bardzo wiele uwagi poświęciliśmy wymaganiom niefunkcjonalnym. Nasz Klient bardzo wyraźnie życzył sobie, by były spełnione normy „12 Factor App" [TFA]. W szczególności, nasz serwis powinien cechować się wysokim współczynnikiem niezawodnośći i odporności na losowe awarie. Ma to szczególne duże znaczenie podczas komunikacji na linii obywatel – państwo, gdzie jakiekolwiek rozbieżności mogą być opłakane w skutkach. W miarę postępu cyfryzacji państwa liczba użytkowników naszego serwisu będzie zbliżać się do 14-tu milionów osób. Należy zapewnić skalowalność tworzonej aplikacji w sposób pozwalający na ich obsłużenie.

Nasz serwis powinien cechować się spójnym i jednolitym wyglądem oraz wysoką reużywalnością komponentów wykorzystywanych do tworzenia interfejsu użytkownika. Interfejsy powininny być łatwo rozszerzalne bez łamania wstecznej kompatybilności i konieczności modyfikacji już napisanych fragmentów aplikacji. W projekcie należy uwzględnić dynamicznie zmieniające się środowisko w jakim przyjdzie działać naszej aplikacji. Z powodu ciągle postępującej cyfryzacji państwa i związanych z tym znacznych zmian w prawie możemy być pewni, że po oddaniu serwisu do użytku ulegnie on znaczącym przekształceniom. Powinniśmy sprawić, by koszt owego przekształcenia był jak najmniej dotkliwy dla naszego Klienta.

Wymaganiem bardzo podobnym do powyższego jest konieczność łatwej integracji z już istniejącymi systemami informatycznymi. Przykładem takiego systemu w ZUS jest „Kompleksowy System Informatyczny". Systemy które nie znajdują się pod zarządem ZUS to między innymi „Profil Zaufany (eGO)" oraz „System Rejestrów Państwowych". Nasz serwis

powinien łatwo integrować się z wymienionym wyżej oprogramowaniem. Należy uwzględnić fakt, że możliwość modyfikacji i dostosowania zewnętrznych systemów do naszych potrzeb jest bardzo ograniczona.

Platforma powinna być prosta w użyciu nawet dla relatywnie nisko wykwalifikowanych programistów i utrudniać im popełnienie przynajmniej części błędów. Przykład problemu występującego w obecnej wersji PUE jest widoczny na poniższym obrazku. Ze względu na niewłaściwą architekturę systemu możliwe jest wyświetlenie użytkownikowi sprzecznych komunikatów dotyczących stanu zalogowania. Nasz framework nie powinien pozwalać na powstanie takiej sytuacji.

System utworzony na bazie naszego frameworka powinien cechować się znacznym stopniem automatyzacji. Dołączanie i odłączanie kolejnych serwerów powinno odbywać się w locie, bez konieczności ręcznej rekonfiguracji. Ten postulat bardzo wydatnie łączy się z wymaganiem skalowalności i niezawodności. Przeprowadzane zmiany w konfiguracji serwerów nie powinny rzutować na niezawodność serwisu.

Rozdział 3

Teoretyka

3.1. Problem

Nasz produkt powinien w sposób maksymalnie prosty umożliwiać integrację systemów informatycznych zakładu ubezpieczeń społecznych. Podstawowymi problemami jakie należało uwzględnić na etapie projektowania systemu były:

- Znaczny poziom skomplikowania istniejącego oprogramowania
- Duża ilość użytkowników naszego systemu
- Zapewnienie wysokiego poziomu odporności na awarie

Po przejrzeniu istniejących rozwiązań zaproponowaliśmy naszemu klientowi architekturę opartą o mikrousługi.

3.2. Mikrousługi

Mikrousługi to stosunkowo nowy wzorzec architektoniczny będący rozwinięciem architektury zorientowanej na usługi (ang. *service-oriented architecture*). Cechuje się ona rozbiciem poszczególnych składowych systemu na małe, luźno powiązane usługi implementujące logikę biznesową. Znacząco ułatwia to ciągłą integrację (ang. *continuous integration*) oraz wdrożenie (ang. *deployment*) dużych, bardzo złożonych aplikacji. Ponadto pozwala to na stosunkowo dużą różnorodność wykorzystanych technologii i stopniową ewolucję wykorzystywanego stosu technologicznego [MSV].

Zastosowanie mikrousług upraszcza proces skalowania całego systemu, co ma niebagatelne znaczenie dla naszego klienta. W systemach o architekturze monolitycznej skalowanie odbywa się na dwa sposoby: poprzez zastosowanie mocniejszej jednostki obliczeniowej lub przez uruchomienie kolejnej instancji aplikacji na nowym serwerze. Pierwsze podejście ma tę wadę, że stosunkowo szybko można osiągnąć górny pułap wydolności dostępnego na rynku sprzętu. W drugim przypadku niezbędne jest zapewnienie mechanizmów synchronizacji i replikacji danych, co w bardzo rozbudowanych systemach może okazać się praktycznie niewykonalne. Architektura mikrousług oferuje mitygacje wymienionych problemów, gdyż dzięki rozbiciu systemu na wiele małych usług możliwe jest wystawienie każdej usługi na osobnym serwerze, zaś luźne ich powiązanie pomaga zredukować nakłady na synchronizację pomiędzy poszczególnymi mikrousługami, dzięki czemu kod staje się prostszy w utrzymaniu i modernizacji.

Zazwyczaj uznaje się, że każda mikrousługa powinna mieć własną bazę danych [Netflix], do której pozostałe usługi nie mają bezpośredniego dostępu. Dzięki temu łatwiejsze staje

się wymuszenie luźnego powiązania usług, a ewolucja schematu bazy danych nie wymusza zmian w całym kodzie aplikacji. Ponadto zespół pracujący nad daną mikrousługą może w sposób nie wpływający na pozostałą część systemu wybrać tę bazę danych, która najbardziej odpowiada rozwiązywanemu problemowi. Niestety to podejście ma swoje wady, do których można zaliczyć konieczność utrzymywania spójności danych w niezależnych od siebie systemach bazodanowych.

3.3. Bezstanowość mikrousług

Wśród dobrych praktyk konstruowania oprogramowania opartego o mikrousługi [Netflix] jest ta, mówiąca o traktowaniu serwerów w sposób bezstanowy. Poszczególne serwery należy traktować jak wymienialnych członków grupy. Wszystkie pełnią te same role i wykonują tę samą pracę. Jeżeli jeden z nich ulegnie awarii, to obowiązki uszkodzonej maszyny mogą przejąć pozostałe instacje. Podobnie dostawienie kolejnej instancji nie powinno sprawiać większych trudności. Dzięki temu uzyskujemy teoretycznie nieograniczoną skalowalność.

3.4. Integracja mikrousług

W naszej architekturze mikrousług potrzebowaliśmy jakiejś metody komunikacji pomiędzy poszczególnymi mikrousługami. Poniżej wypisaliśmy najczęściej spotykane rozwiązania tego problemu.

3.4.1. Szyny danych

Na początku nasza uwaga została przykuta przez szyny danych jako standardowe rozwiązanie korporacyjne wykorzystywane przy integracji usług. Nasz klient jest w posiadaniu licencji na szynę danych WebMethods i to ją jako pierwszą rozpatrywaliśmy. Niestety, dostępna publicznie internetowa dokumentacja tego rozwiązania ograniczała się do kilku broszurek reklamowych i luźnych sloganów. Z czystej ostrożności zrezygnowaliśmy z tego oprogramowania, gdyż nie mieliśmy absolutnie żadnej gwarancji, że wyczytane hasła reklamowe mają jakiekolwiek pokrycie w rzeczywistości. Do kolejnej grupy sprawdzonych szyn danych należały rozwiązania takie jak: WSO2, Talend, Mule. Możliwości, które oferowały wyglądały na obiecujące, ale zupełnie niepasujące do specyfiki naszego projektu. Jednym z celów naszej platformy było opracowanie interfejsu, który pozwalałby na możliwie łatwe, automatyczne wpinanie i wypinanie mikrousług. Nijak do tak postawionego celu miała się konieczność konfigurowania szyny danych przez panel wystawiony w sieci WWW lub wręcz przy pomocy specjalnego środowiska programistycznego takiego jak Eclipse, z obowiązkową fazą kompilacji i wgrywania na serwer.

Rozwiązaniem wartym wspomnienia jest Zato, dość osobliwa szyna danych rozwijana przez osobę o polsko brzmiącym nazwisku. Jej głównymi cechami są skalowalność, możliwość rekonfiguracji w locie oraz bardzo dobra integracja z językiem python, co miało dla nas niebagatelne znaczenie na etapie wczesnych analiz. W zasadzie cała logika realizowana przez tę szynę danych mogła być zapisana w postaci skryptu pythona, co otwierało nas na zupełnie nowe możliwości integracji mikrousług. Rozważaliśmy scenariusz, w którym szyna danych odpowiadałaby za wykrywanie działających mikrousług, trzymanie ich spisu i metadanych z nimi powiązanych oraz rozgłaszanie tegoż spisu do warstwy prezentacji i innych mikrousług. Zrezygnowaliśmy z tego podejścia jako naruszającego zasadę separacji odpowiedzialności.

3.4.2. Kolejki komunikatów

Niepowodzenia przy poszukiwaniu odpowiadającej nam szyny danych skierowały nas w kierunku innych rozwiązań, takich jak kolejki komunikatów. Głównym wymaganiem jakie postawiliśmy była bardzo duża odporność na różnego rodzaju błędy i awarie. W idealnym modelu każda wiadomość w kolejce powinna mieć trzy stany: do przetworzenia, w trakcie przetwarzania, wykonany. Celem takiego modelu jest, by w przypadku awarii jednego serwera usługi jego zadanie było transparentnie przekazane serwerowi zastępczemu, bez przerywania operacji. Udało nam się znaleźć tylko jedną kolejkę spełniającą to wymaganie - Beanstalkd [Beanstlkd].

Innym sposobem osiągnięcia niezawodności może być kolejka ukryta wewnątrz mikrousługi, która w razie awarii jednego serwera pozwalałaby na odczytanie zadania przez inny serwer i dokończenie go. Niestety wtedy mogą mieć miejsce problemy z atomowością operacji. Przykładowo, w sytuacji gdy zadanie w kolejce zostało oznaczone jako wykonane, ale usługa wywołująca nie otrzymała jeszcze odpowiedzi nastąpi awaria wywoływanej mikrousługi, może dojść do duplikacji wykonanej operacji lub rozspójnienia danych.

3.4.3. Dzienniki

Jeszcze innym, ciekawym podejściem byłoby zastosowanie usługi udostępniającej niezawodny dostęp do czegoś w rodzaju dziennika. Wtedy moglibyśmy stosunkowo małym kosztem zaimplementować mechanizmy stosujące transakcyjność rodem z relacyjnych baz danych. Przykładami takich usług są DistributedLog [DL] i Kafka [Kafka]. Poszczególne mikrousługi monitorowałyby dziennik i z niego pobierały zmiany, a następnie reagowały na nie poprzez podjęcie jakiejś akcji. Można sobie wyobrazić dwa sposoby wykorzystania takiego dziennika. Pierwszy polegałby na tym, że wszystkie dokonane zmiany są atomowo publikowane w jednej dużej paczce, która musiałaby być potem rozpakowywana, a poszczególne zmiany wyłuskiwane. Drugim sposobem byłoby stworzenie dużej ilości dzienników na każdy rodzaj zdarzenia, a każda operacja byłaby wykonywana w małych krokach, w sposób potokowy.

3.4.4. Rozwiązania oparte o BPMN

W wymienionych wyżej modelach komunikacji serwery pośredniczące w wymianie komunikatów nic nie wiedzą o procesach biznesowych, w których uczestniczą. Możliwa jest jednak realizacja zgoła przeciwnego podejścia, w którym działanie wszystkich mikrousług jest koordynowane przez jedną centralną usługę.

Pozwalala ona na modelowanie procesów biznesowych przy pomocy graficznej „Notacji i Modelu Procesu Biznesowego" (ang. *Business Process Modeling Notation*), a następnie wykonanie ich na specjalnie do tego celu skonstruowanym silniku. Poszczególne mikrousługi stanowią w tym modelu „klocki", z których klient może sobie kontruować bardziej złożone procesy. Przykładami technologii, które implementują wyżej wymienioną funkcjonalność są:

- Camunda (oparty na Activiti)
- Bonita BPM
- jBPM
- Intalio BPMS

James Lewis i Martin Fowler w swojej publikacji [Fowler] stwierdzają, że powyższe podejście nie jest preferowane w środowisku mikrousług gdyż stanowi złamanie zasady „sprytnych końcówek i bezmyślnych łączy" (ang. *smart endpoints and dumb pipes*).

3.5. Problemy z współbieżnością mikrousług

Nietrudno wyobrazić sobie sytuację, w której dwie współbieżnie działające mikrousługi wprowadzają konfliktujące ze sobą zmiany. Pewną formą zabezpieczenia przed takimi sytuacjami mogłoby być rozwiązywanie konfliktów przez mikrousługę pośredniczącą w dostępie do bazy danych. Taka mikrousługa stanowiłaby wtedy coś w rodzaju sekcji krytycznej, uniemożliwiającej jednoczesne zapisy i niespójne odczyty (niezależnie od atomowości operacji na bazie danych). Niestety, takie podejście może nie wystarczać, np. w przypadku dwóch niezależnie działających usług, które wpierw odczytują jakąś informację z bazy danych, a następnie na podstawie uzyskanej informacji wybierają rodzaj akcji do podjęcia i nadpisują część bazy danych.

Jest kilka możliwych rozwiązań tego problemu. Możemy je pogrupować ze względu na następujące cechy:

- miejsce przechowywania blokad
- sposób zakładania blokad na usługi
- wielkość obiektów, które będą blokowane

Blokady mogą być przechowywane w dedykowanej, wydzielonej mikrousłudze lub w mikrousłudze, której dotyczą. W naszej opinii pierwsze podejście jest o tyle lepsze, że pozwala na wykrywanie zakleszczeń. Wadą takiego rozwiązania jest stosunkowo słabe zrównoleglanie i fakt, że taka mikrousługa stanowiłaby najsłabszy punkt systemu, którego awaria skutkowałaby zablokowaniem wszystkich pozostałych usług. Blokady mogą być zakładane pojedyńczo, w miarę postępu transakcji lub na jej samym początku (w takim przypadku mikrousługa musi dokładnie wiedzieć, z jakich zasobów ma zamiar skorzystać). Wydaje nam się, że najsensowniejszym wariantem jest wykorzystanie rozwiązań stosowanych w relacyjnych bazach danych, gdzie blokady są zakładane w miarę postępu transakcji, a w przypadku wykrycia zakleszczenia zmiany są wycofywane, sama zaś usługa wywłaszczana. Przy kolejnej próbie takiej mikrousłudze nadany zostaje większy priorytet, który zapewnia żywotność. W przypadku implementacji takiego podejścia należy pamiętać o zapewnieniu mechanizmów wywłaszczania mikrousług, księgowania zmian, śledzenia transakcji, wykrywania zakleszczeń, nadawania priorytetów oraz o możliwości zakładania możliwie małych blokad na pojedyńcze rekordy.

Innym sposobem rozwiązania wyżej przedstawionego problemu jest zakładanie blokad na początku transakcji. Ze względów wydajnościowych blokowanie przez muteks całych mikrousług zapewniających dostęp do danych nie jest możliwe. Z uwagi na opóźnienia w przesyłaniu informacji przez sieć prowadziłoby to z jednej strony do marnowania mocy obliczeniowej serwera bazy danych, a z drugiej strony znacząco ograniczałoby ilość jednoczesnych operacji. Z kolei zakładanie blokad na konkretne wiersze w poszczególnych mikrousługach jest nierealizowalne, gdyż w chwili zakładania muteksu usługa może nie wiedzieć, jakie konkretnie wiersze zmodyfikuje. Uznaliśmy, że złotym środkiem jest zakładanie blokady na właściciela danych. Wynika to z faktu, że w typowej urzędniczej praktyce wykonywane operacje dotyczą co najwyżej kilku osób.

Uzbrojeni w powyższe informacje wytypowaliśmy dwie usługi mogące działać jako zarządca rozproszonego muteksa: redisson [RDS] i ZooKeeper [AZK]. Redisson jest rozwiązaniem implementującym algorytm Redlock [RDL]. Jego najważniejszą cechą jest odporność na awarie usług poprzez automatyczne zwalnianie blokady po upływie określonego czasu. W internecie ukazała się bardzo interesująca polemika do przytoczonego algorytmu [HTL]. Głównym zarzutem czynionym wobec algorytmu Redlock jest brak mechanizmu, który zapobiegałby następującej sytuacji: mikrousługa A zakłada blokadę na zasób Z. Następnie zaczyna

operację nadpisywania danych. W trakcie operacji działanie mikrousługi z losowego powodu zostaje zawieszone (np. wystąpiło znaczne opóźnienie w połączeniu sieciowym lub włączył się odśmiecacz pamięci). W tym czasie termin ważności blokady mija, zostaje ona zdjęta, a do akcji wkracza mikrousługa B, która zakłada swoją blokadę, nadpisuje dane utworzone przez usługę A i kończy swoje działanie. Na koniec usługa A zostaje wybudzona i kontynuuje swoje działanie, prowadząc do rozspójnienia danych. Według autora polemiki rozwiązaniem tego problemu jest skorzystanie z mechanizmu współdzielonej blokady zaimplementowanej w frameworku Apache Curator [CLK] i korzystającej pod spodem z Apache ZooKeepera. Jest to bezpieczniejsze rozwiązanie które nie pozwala na wygaśnięcie blokady z powodu upływającego czasu, a w przypadku awarii mikrousługi i utraty połączenia z nią automatycznie zdejmuje blokadę. Ponadto, jest to stosunkowo nowoczesny projekt wykorzystywany i rozwijany przez duże firmy takie jak Netflix, co daje pewne nadzieje dotyczące stabilności i dojrzałości tego projektu.

3.6. Rejestracja i wykrywanie mikrousług

Zarządzanie infrastrukturą w systemie opartym o mikrousługi jest znacznie większym wyzwaniem niż w systemie o architekturze monolitycznej. Dzieje się tak z powodu dużej ilości różnych serwerów usług a także dlatego, że mikrousługi zachęcają do stosowania wielu różnorodnych, nierzadko niekompatybilnych technologii. Ręczna konfiguracja i administrowanie takim systemem pochłania znaczne zasoby, dlatego postanowiliśmy ten proces zautomatyzować. Przedstawimy kilka powszechnie stosowanych w przemyśle rozwiązań tego problemu.

Pierwszym z nich jest serwer Apache ZooKeeper [AZK]. Dane na serwerze ZooKeeper są trzymane w drzewiastej strukturze. Każdy węzeł może zawierać dowolny ciąg bajtów ograniczony do rozmiaru ok. 1 Mb. Serwer pozwala na atomowe odczyty i zapisy, ponadto może działać w klastrze. W takim przypadku gwarantowane jest zachowanie sekwencji zapisów. Dostęp do danych jest chroniony przez mechanizm ACL. Informacje o mikrousługach są przechowywane w węzłach drzewa i zazwyczaj składają się na nie identyfikator serwera, adres internetowy, port usługi, informacja o obsłudze SSL. Dzięki otwartości kodu źródłowego klient usługi dostępny jest na większości wiodących platform.

Rozwiązaniem bardzo podobnym do ZooKeepera jest Consul. Zasadniczą różnicą między tymi dwoma serwerami było zastosowanie w Consulu słownika typu klucz-wartość oraz znacznie bardziej zaawansowane raportowanie stanu mikrousługi, które obejmuje nie tylko informację o działaniu serwera usługi, ale także okresowe sprawdzanie jego stanu przy pomocy specjalnie zaprojektowanego RESTowego interfejsu. Ułatwia on integrację z platformami na których nie jest dostępny klient Consula. Consul jest projektem komercyjnym.

Podobną, chociaż nieco okrojoną funkcjonalność w stosunku do Consula zapełniają serwery etcd i Netflix Eureka. Ten pierwszy pozwalał na rozproszone składowanie danych w słowniku typu klucz-wartość. Eureka z kolei zapewniała automatyczną rejestrację, wykrywanie i odpytywanie mikrousług z zadaną częstotliwością oraz monitorowanie ich stanu. Obydwa projekty mają otwarty kod źródłowy.

3.7. Wnioski

Nasz zespół zdecydował, że opracujemy rozwiązanie wykorzystujące mikrousługi. Komunikacja z poszczególnymi serwerami powinna być bezstanowa. Konfiguracja mikrousług będzie przechowywana w formie drzewa w centralnym serwerze o dużym stopniu niezawodności.

Rozwiązania realizujące komunikację między mikrousługami uważamy za wysoce niewystarczające i zbyt mało elastyczne. Postanowiliśmy stworzyć własną usługę o funkcjonalnościach zbliżonych do szyny danych, która na dodatek pozwala na zautomatyzowaną konfigurację bez przerywania swojej pracy.

Nasz framework nie będzie rozwiązywał problemu rozproszonej synchronizacji między usługami. Wynika to z bardzo dużej różnorodności sposobów na jakie można to uczynić. Nasza platforma może integrować poprzez różne adaptery już istniejące usługi, które zupełnie nie są świadome istnienia zaproponowanego mechanizmu blokad, więc możliwe byłyby niekompatybilności między rozwiązaniem zaproponowanym przez nas a wymienionymi usługami. Przykładami problematycznych usług mogą być „System Rejestrów Państwowych" oraz „Kompleksowy System Informatyczny" do których dokumentacji z oczywistych względów nie mamy dostępu, więc nie możemy przewidzieć trudności jakie nasza platforma będzie narzucać naszym użytkownikom. To programiści korzystający z naszego systemu będą musieli rozwiązać ten problem w sposób, jaki uznają za optymalny.

Rozdział 4

Wykorzystane technologie

Nasz serwis wykorzystuje bardzo dużą ilość różnych technologii. Związane jest to z charakterem naszego projektu, który polegał na integracji usług napisanych w możliwie różnych technologiach.

4.1. Języki programowania

Do napisania części backendowej początkowo wykorzystywaliśmy język Java. Głównymi argumentami przemawiającymi za tym wyborem był bogaty zbiór bibliotek wykorzystywanych w aplikacjach korporacyjnych, wysoka wydajność maszyny wirtualnej Java oraz fakt, że obecnie istniejące systemy prawdopodobnie wykorzystują ten język. W trakcie naszych prac okazało się, że jest to zupełnie nietrafiony wybór. Narastające problemy pojawiające się podczas współpracy z wybranymi bibliotekami oraz niezadowalające tempo prac w połączeniu ze zbliżającymi się terminami skłoniły nas do przepisania już napisanego oprogramowania na język Python.

W przypadku naszego projektu radykalna decyzja o zmianie wykorzystywanego ekosystemu była o tyle prosta, że nasz serwis miał charakter prototypu ilustrującego pewne idee i niewykorzystywanego w środowisku produkcyjnym. Bardzo szybko okazało się, że była to trafiona decyzja pozwalająca na mitygacje problemów, które pojawiły się w trakcie naszych prac.

4.2. Angular 2

Aby rozwinąć swoje skrzydła, asynchroniczne działanie mikrousług wymaga również działającej asynchronicznie warstwy użytkownika. Spośród obecnie stosowanych frameworków frontendowych wyróżniały się dwa: Angular oraz React.js. Zdecydowaliśmy się na ten pierwszy z powodu kluczowej dla klienta, całkowicie otwartej licencji. Wykorzystaliśmy drugą wersję Angulara od Google, ponieważ jest oparta o nowsze rozwiązania niż pierwsza, a w szczególności wykorzystuje język TypeScript, czyli obiektowe rozszerzenie języka Javascript. W stosunku do JavaScript można stwierdzić, że wszystko co związane z obiektowością w TypeScript to tylko lukier syntaktyczny, ponieważ kod TypeScript podczas kompilacji jest tłumaczony do czystego JavaScript. TypeScript implementuje większość funkcjonalności, które język obiektowy powinien oferować - możliwe jest dziedziczenie, tworzenie interfesjów, konstruktory itp. Ciekawą opcją są dekoratory (szerzej opisane w dalszej części pracy), czyli ozdobniki klas które pozwalają na zmianę “zachowania” klasy już w trakcie kompilacji.

4.3. Bootstrap

Za wygląd strony odpowiada HTML5, połączony z Bootstrapem. Bootstrap to zbiór gotowych komponentów wizualnych, pozwalający, bez zbędnego wnikania w stronę "artystyczną", tworzyć strony estetyczne oraz wygodne w użyciu strony internetowe. Jako, że tematem naszej pracy nie jest projektowanie stron internetowych i nie chcieliśmy zbytnio wnikać w aspekty graficzne, użyliśmy gotowej skórki sbAdmin2, w której wprowadziliśmy liczne zmiany.

4.4. Django

Powszechnie przyjętym zwyczajem jest, że mikrousługi komunikują się przy pomocy RESTowych API [mammatustech]. Naszym zdaniem do kontrukcji takiego interfejsu idealnie nadają się frameworki do tworzenia aplikacji webowych. Z uwagi na swoją prostotę wybraliśmy Django. Dodatkowym argumentem za takim wyborem był tutaj fakt, że architektura mikrousług zachęca do dywersyfikacji technologii. Wybierając framework napisany w języku Python chcieliśmy pokazać, że nasze rozwiązanie dobrze integruje usługi napisane w różnych językach.

4.5. Express.js

Do serwowania części frontendowej serwisu wykorzystujemy serwer Express.js. Korzystanie z niego jest proste i stosunkowo przyjemne, znakomicie integruje się on z platformą Node.js. W przeciwieństwie do Lite–servera z którego korzystaliśmy na początku, Express.js jest dobrze udokumentowany i udostępnia bogate API, z którego skorzystaliśmy przy integracji z backendem.

4.6. gRPC

Podczas projektowania mechanizmu komunikacji pomiędzy mikrousługami nasz Klient zasugerował użycie WSDL. Nie zgodziliśmy się na to, gdyż użycie tak ciężkiego protokołu jest sprzeczne z typowymi praktykami stosowanymi w architekturze mikrousług [mammatustech]. Jednocześnie nie chcieliśmy, by nasza aplikacja była ograniczona tylko i wyłącznie do RESTowych protokołów. Ostatecznie zaproponowaliśmy użycie gRPC, który łączy zalety obu wymienionych wcześniej rozwiązań. Z jednej strony jest stosunkowo lekkim i wykorzystującym niewiele zasobów protokołem sieciowym, a z drugiej strony pozwala na ścisłe zdefiniowanie udostępnianych przez serwer usług w czytelnym dla ludzi formacie. Po napisaniu odpowiednich definicji możliwa jest ich kompilacja do gotowego kodu źródłowego włączanego bezpośrednio w pliki projektu. Dzięki temu minimalizowane jest ryzyko popełnienia błędu przy pisaniu modułu do komunikacji z innymi usługami. Co ważne, kompilator gRPC jest dostępny dla wszystkich wykorzystanych przez nas języków programowania.

4.7. Play Framework

Play Framework opiera się na podobnych ideach architektonicznych co Django. Kiedy go wybieraliśmy mieliśmy nadzieję, że będzie się on bardzo dobrze integrował z ekosystemem Javowym. Okazało się, że było to błędne założenie. Kłopoty pojawiły się już na samym początku, kiedy odkryliśmy że między frameworkiem Play i gRPC ma miejsce konflikt zależności. Zdecydowaliśmy się użyć wersji niestabilnej, w której ten konflikt był już rozwiązany.

Narastające w trakcie trwania projektu problemy ostatecznie zmusiły nas do porzucenia tego frameworka na rzecz Django.

4.8. SQLite

W naszym projekcie rodzaj użytej bazy danych nie miał większego znaczenia, gdyż dostęp do niej odbywa się poprzez wyspecjalizowaną usługę zapewniającą dostęp do danych lub w ramach jednej mikrousługi, gdzie służy jako tymczasowe składowisko informacji potrzebnych przy wykonywaniu operacji. W takim modelu rodzaj użytej bazy danych jest transparentny dla usługi wywołującej. Z uwagi na prototypowy charakter naszego projektu poprzestaliśmy na bazie SQLite, jako że znakomicie integruje się ona z Django, ponadto zastąpienie jej PostgreSQLem jest praktycznie bezkosztowe.

4.9. ZooKeeper

Jednym z elementów wizji naszego systemu był fakt, że poszczególne usługi powinny możliwie prosto „wpinać się" w system. Serwer mikrousługi w trakcie uruchamiania dopisywałby do globalnej konfiguracji wpisy związane z prowadzoną przez siebie działalnością, takie jak adresy pod którymi przyjmuje zapytania albo pozycje w menu widocznym dla użytkownika, które prowadzą do poszczególnych usług biznesowych. Doszliśmy do wniosku, że nasze oczekiwania można sprowadzić do trzech wymagań:

- przechowywania konfiguracji w drzewiastej strukturze
- automatycznego usuwania dokładnie określonych wpisów w przypadku utraty dowolnego serwera mikrousługi
- automatyczną replikację danych

Najbliższy naszym potrzebom był serwer Apache ZooKeeper [AZK] w połączeniu z biblioteką Apache Curator [ACU]. Dane na serwerze ZooKeeper są trzymane w drzewiastej strukturze. Każdy węzeł może zawierać dowolny ciąg bajtów ograniczony do rozmiaru ok. 1 Mb. Węzły są wersjonowane, a dostęp do nich jest kontrolowany przez mechanizm ACL. Na każdym węźle można ustawić obserwatora (ang. *watch*), który w przypadku jakiejkolwiek modyfikacji węzła powiadamia klienta o zmianie. Oprócz tego, każdy węzeł może mieć ustawione dwie dodatkowe flagi. Pierwsza to *ephemeral*, która oznacza, że węzeł ma istnieć tylko do czasu zakończenia sesji klienta, który utworzył dany węzeł. Ta własność jest szczególnie interesująca w kontekście automatycznego wyrejestrowywania usługi która kończy swe działanie i nie ma możliwości powiadomienia serwera o tym fakcie. Druga flaga to *sequence*. Mówi ona, że należy do nazwy tworzonego węzła dopisać rosnący numer. Ma to zastosowanie we wszelkiego rodzaju kolejkach, algorytmach synchronizujących i przy wyborze lidera [CLK].

4.9.1. Curator

Biblioteka Curator upraszcza wiele aspektów związanych z wykorzystaniem ZooKeepera, między innymi rejestrację serwerów mikrousług, zapewnia implementację najczęściej wykorzystywanych algorytmów synchronizujących oraz łatwe buforowanie drzewa węzłów wraz z automatycznym pobieraniem zmian. W trakcie prac nad integratorem okazało się, że oferowane możliwości znacznie przekraczają nasze potrzeby.

4.9.2. Kazoo

Kazoo jest nieoficjalnym klientem ZooKeepera dla języka Python. W trakcie implementacji naszej aplikacji doceniliśmy jego niezawodność i prostotę użycia.

Rozdział 5

Architektura aplikacji

Nasza aplikacja jest podzielona na trzy warstwy: frontend, mikrousługi oraz warstwę integrującą, na którą składają się serwery autoryzacji i integratora. Komunikacja pomiędzy poszczególnymi warstwami jest bezstanowa, co przynajmniej w teorii powinno pozwolić na osiągnięcie niemal nieograniczonej skalowalności.

5.1. Frontend

Główną ideą którą kierowaliśmy się przy projektowaniu frontendu było udostępnienie programiście mikrousług komponentów, z których mógłby następnie tworzyć interfejs użytkownika. Jednym z wymagań zamawiającego było uczynienie tworzenie mikrosusług prostym, stworzenie zestawu gotwoych komponentów, z których, jak z klocków, programista może składać swój projekt, niewątpliwie upraszcza proces tworzenia nowych funkcjonalności.

Rzeczony opis utrzymany jest na wysokim poziomie abstracji – jest w nim zawarta informacja **co** ma być wyświetlone, ale nie **jak**. U użytkownika wchodzącego na nasz portal poprzez przeglądarkę internetową JSON opisujący komponenty powinien zostać przekonwertowany na kod HTML dostosowany do jego przeglądarki. Co więcej, systemy służące do automatycznego zarządzania pracownikami mogą zupełnie pomijać warstwę prezentacji i operować bezpośrednio na danych dostarczanych przez warstwę integracji. Dzięki temu, że wygląd panelu użytkownika jest zapisany w formacie pośrednim możliwa jest zmiana wyglądu portalu bez konieczności ponownego wdrażania wszystkich serwerów mikrousług.

5.2. Warstwa integracji

Kluczowym elementem tej warstwy jest integrator, przez który przechodzi niemal cały ruch w systemie. Każda mikrousługa wchodząca w skład portalu musi być zarejestrowana w integratorze. Dostęp do mikrousług jest możliwy tylko poprzez integratora, który sprawdza, czy klient mikrousługi ma odpowiednie uprawnienia dostępowe, dokonuje tłumaczenia komunikatów pomiędzy różnymi interfejsami, wyszukuje adres serwera, do którego ma trafić komunikat oraz go wywołuje.

Z uwagi na to, że dane przechodzą przez niego w postaci nieszyfrowanej można go podłączyć do IDSa (ang. *Intrusion Detection System*) wykrywającego anomalie w przesyłanych danych i w ten sposób zapobiegać masowemu wyciekowi wrażliwych danych lub atakom hakerskim. Stanowi też dodatkową linię obrony, którą musi sforsować potencjalny haker. Dzięki temu system jest odporniejszy na błędy popełnione w serwerach mikrousług, a w przypadku

wykrycia jakiejś luki można wprowadzić łatkę bezpieczeństwa w serwerze pośredniczącym zamiast łatać wszystkie podatne serwery mikrousług.

Integrator współpracuje z serwerem uwierzytelniania i autoryzacji, który przechowuje listę wszystkich użytkowników systemu wraz z ich uprawnieniami. Dostęp do tego serwera jest możliwy z dowolnej części sieci. Z uwagi na konieczność zapewnienia łatwej integracji z zewnętrznymi usługami typu „Profil Zaufany (eGO)", do identyfikacji użytkowników użyliśmy jednorazowych tokenów.

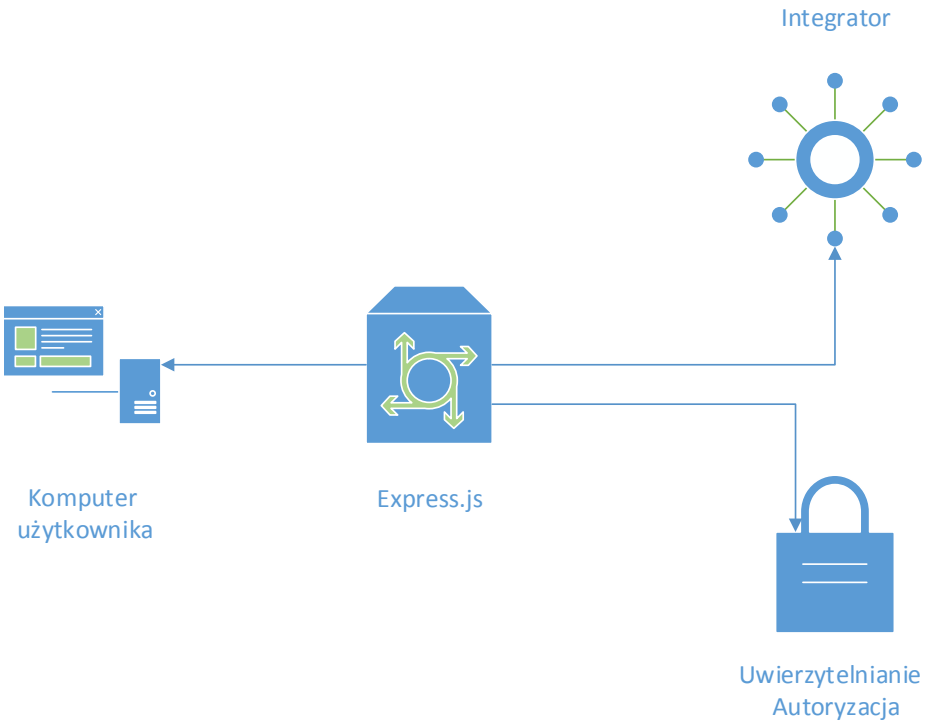
5.3. Mikrousługi

Mikrousługi realizują logikę biznesową. Typowym cyklem życia serwera usługi jest przygotowanie do obsługi nadchodzących żądań, rejestracja w integratorze i obsługa zapytań. W końcowej fazie cyklu serwer się wyrejestrowuje z integratora. Realizacja integratora oczekuje, że poszczególne serwery będą z punktu widzenia klienta nierozróżnialne.

5.4. Komunikacja między warstwami

Z uwagi na stosunkowo duży poziom złożoności architektury naszej aplikacji opis komunikacji podzieliliśmy na dwie części: backend i frontend. W obydwu dodatkowo zawarliśmy warstwę integracji.

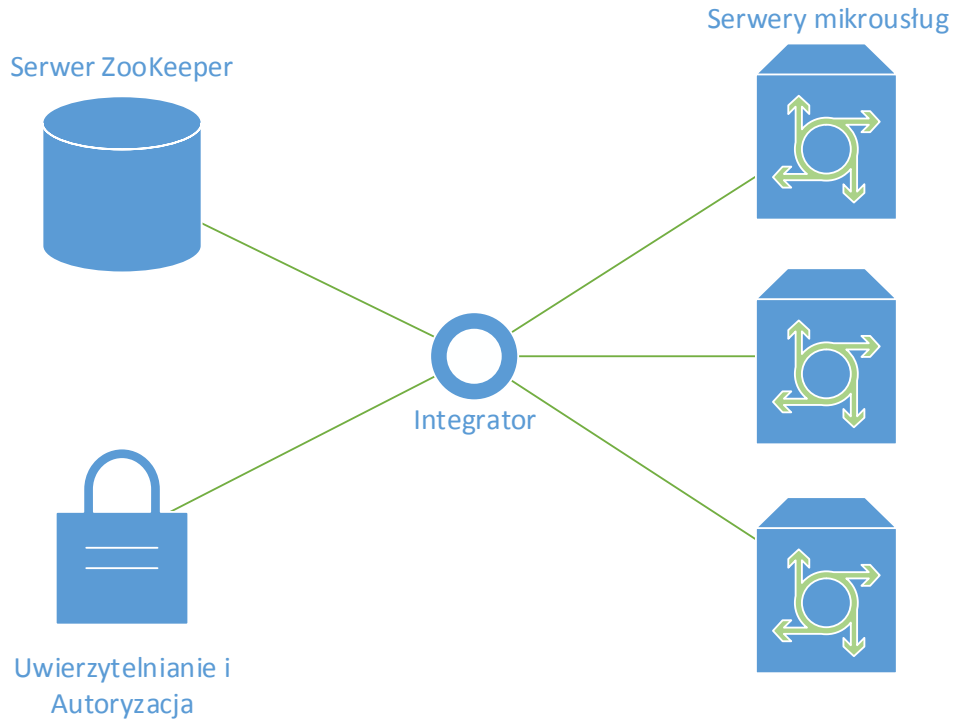
W części frontendowej (zaprezentowanej na poniższym obrazku) główną rolę odgrywa serwer express.js, który serwuje do przeglądarki na komputerze użytkownika kod aplikacji napisany w frameworku Angular. Następnie otrzymany program zostaje wykonany, a użytkownikowi zostaje zaprezentowany wygodny interfejs i możliwa staje się dalsza interakcja.



Z powodu różnych zabezpieczeń wbudowanych w przeglądarki bezpośrednie połączenie

klienta z integratorem nie jest możliwe. W związku z tym serwer express.js pełni dodatkowo rolę pośrednika pomiędzy aplikacją internetową a integratorem i serwerem autoryzacji. Pozostałe rodzaje klientów naszego portalu mogą go całkowicie pominąć.

Jak widać na poniższym schemacie, centralnym elementem w komunikacji elementów backendu jest integrator. Przechodzi przez niego praktycznie cały ruch między poszczególnymi mikrousługami i serwerami frontendu. Wyjątek stanowią tutaj serwery ZooKeepera oraz uwierzytelniania, do których dostęp jest bezpośredni. Wynika to ze specyficznych własności ZooKeepera który jest wykorzystywany do przechowywania konfiguracji integratora oraz pewnego rodzaju wyjątkowości serwera autoryzacji, która nie pozwala na zaklasyfikowanie go jako zwykłej mikrousługi.



Rozdział 6

Realizacja

6.1. Tworzenie mockupów

Jako, że naszym celem nie było stworzenie gotowej platformy (w sensie produkcyjnym), a raczej stworzenie proof of concept, intensywnie tworzyliśmy prototypy. Chcialismy pokazać jak PUE może wyglądać, niekoniecznie skupiając się na sprawach żmudnych, czasochłonnych, a niekoniecznie potrzebnych do pokazanie idei.

6.2. Strona domowa

Nastawienie społeczeństwa do ZUS, jako organizacji jest zdecydowanie negatywne, dlatego tworząc nasz projekt staraliśmy się zmienić wizerunek ZUSu. Postawiliśmy na personalizacje kontaktów z użytkownikiem odwiedzającym naszą platformę. Dlatego już po zalogowaniu użytkownik otrzymuje większość potrzebnych informacji. Strona domowa jest dopracowana pod względem UX. Panele ([1]) informują obywatela o aktualnym stanie spraw, z nim związanych. Informację te są wyraźnie wyróżnione, a kolory jakie przyjmują panele intuicyjnie wiążą się z kontekstem (np. czerwony kolor - sprawy odrzucone). Właściciel profilu nie musi marnować czasu na szukanie najbardziej potrzebnych mu informacji w innych miejschach strony. W założeniu strona domowa powinna być namiastką „Ściany" znanej z popularnego protalu społecznościowego. Panel z powiadomieniami ([2]), to po prostu dostarczony przez nas gotowy komponent o nazwie powiadomienia.

6.3. Komponenty

6.3.1. Mechanizm komponentów

Projektując naszą platformę, pamiętaliśmy o istotnym nacisku, jaki klient kładł na jej proste rozszerzanie. Postanowiliśmy zatem zaprojektować własny mechanizm, który usprawniłby wytwarzanie mikrousług gotowych do pracy w naszym systemie. Od strony backendu, dużo łatwiej jest uzgodnić wspólny interfejs, którego implementacja w nowej mikrousłudze pozwoli podłączyć ją do działającej platformy. Frontend natomiast jest wytwarzany, w zasadzie wyłącznie w języku HTML, ze wsparciem ze strony języka skryptowego Javascript oraz jego bibliotek. Docelowe mikrousługi, wykorzystywane na naszej platformie, będą komunikowały się z REST-owym backendem. Formularze z kolei posiadają zazwyczaj dosyć generyczne interfejsy graficzne i programistyczne. Stając przed tak ocenioną sytuacją, postanowiliśmy nie oddawać w ręce programistów mikrousług możliwości umieszczenia na stronie platformy

dowolnego kodu HTML. Przede wszystkim, ze względów bezpieczeństwa - usługa mogłaby uruchamiać złośliwy bądź posiadający krytyczne luki kod javascript. Ważne dla nas były jednak także kwestie stylistyczne - dużo wygodniej, niż publikować przewodniki styli (tzw. "Style Guide"), byłoby nam udostępnić już gotowe, wystylizowane przez nas kawałki kodu HTML.

Osadzanie wewnątrz naszej aplikacji przekazanego w jakiś sposób szablonu HTML nie jest trudne, ale pozostaje pytanie o to, na kim będzie spoczywała odpowiedzialność jego napisania. Nie chcieliśmy żeby była do tego potrzebna wiedza frontendowa, dlatego postanowiliśmy zwiększyć poziom abstrakcji w naszych szablonach, a co za tym idzie - zbudować mały framework (szkielet), który uprości ich tworzenie, przekazywanie i interpretację, z poziomu osoby lub programu czytającego ich opis.

6.3.2. Opis komponentów

Nazwa: Powiadomienie
Opis: Rodzaj krótkiej wiadomości, składający się z tytyłu i treści powiadomienia.
Końcówki: Końcówka pobierająca powiadomienie
JSON: title: msg: time:
Opis działania: Powiadomienie zostaje jednorazowo pobrane z podanej końcówki.
Wygląd:

Nazwa: Lista powiadomień
Opis: Pojemnik na powiadomienia, koordynujący pobieranie powiadomień
Końcówki: Końcówka z powiadomieniami Końcówka z liczbą powiadomień
JSON: title: msg: time:
Opis działania: Cyklicznie (w ustalonych odstępach czasu, obecnie co 5 sekund), komponent wysyła zapytanie o liczbę dostę
Wygląd:

Nazwa: Panel
Opis: Kontener z paskiem tytułowym i zawartością.
Końcówki: końcówka pobierająca tytuł

JSON: title: msg: time:
Opis działania: Komponent, w zależnosci od konfiguracji, cyklicznie lub jednorazowo pobiera treść górenego paska. W cie
Wygląd:

6.3.3. System tworzenia automatycznej dokumentacji

Każdy komponent można opisać w podobny sposób(Co udowadnia powyższy podrodział). Z uwagi na tę regularność stworzyliśmy mechanizm tworzenia automatycznej dokumentacji. Jako, że platforma składa się z wielu komponentów, w założeniu tworzonych przez wielu programistów, istnieje potrzeba usystematyzowania opisu komponentów. Nasz mechanizm przyczynia się do zwiększenia spójności platformy, pozwala zachować porządek wśród komponentów, oraz niejako wymusza na programiście większą staranność. Na podstawie naszych doświadczeń stwierdzialiśmy, że bardzo wygodne jest opisywanie kodu w trakcie i miejscu jego tworzenia. Dlatego używając dekoratorów, stworzyliśmy mechanizm pozwalający tworzyć komponenty niejako w sposób opisowy. Zapewne dużym nadużyciem będzie nazwanie użytego podejście terminem programowanie w paradygmacie opisowym, jednak na dobrą sprawę sam sposób kreacji komponentów został, faktycznie, sprowadzony do automatycznego generowanie kodu na podstawie opisy składowych. Dekoratory pozwalają na zmianę zachowanie dekorowanego przez siebie elementu już w momencie kompilacji, lub w momencie wywływania danego elementu(dla metod). Stworzyliśmy następujące dekoratory: Dekorator register - dodaje komponent do bibliotki komonentów. Wymaga podania nazwy komponentu, jego opisu ogólnego, słów kluczowych(charakteryzacja komponentu), oraz opisu działania. Dekorator dekoruje klasę, tzn. modyfikuje zawartość klasy w momencie kompilacji.

Dekorator Attr - dodaje do opisu komponentu, w którym się znajduję, opis samego siebie. Komponent to nic innego jak zbiór wielu parametrów, które określają jego wygląd i zachowanie. Oczywiście niewszystkie atrybuty danego komponentu są widocznego na zewnątrz, niektóre wykorzystywane są tylko w wewnętrznych działaniach (obliczeniach klasy), dlatego też nie wprowadziliśmy mechanizmu, który automatycznie listowałby wszystkie atrybuty danego komponentu. To od programisty zależy, które parametry będą "wyprowadzone" na zewnątrz, czyli które komponentu bedą edywalne z zewnątrz i będą miały wpływ na zachowanie komponentu, a które będą ukryte na potrzeby wewnętrznych działan komponentu.

6.3.4. Edytor

TODO

6.3.5. Platforma developera

Sposób dodawania mikrosusług, style guide.

6.4. Serwer uwierzytelniania i autoryzacji

Z uwagi na prototypowy charakter naszej pracy nasz Klient nie przywiązywał większej wagi do rozwiązania problemu uwierzytelniania i autoryzacji użytkowników. Zdecydowaliśmy się

jednak zaimplementować przykładowy serwer celem ukazania kilku ciekawych możliwości integratora. Serwer autoryzacji jest napisany w języku Python i wykorzystuje framework gRPC do udostępniania usług, którymi są: zalogowanie i pobranie tokena dostępowego, unieważnienie tokena, pobranie identyfikatora i uprawnień użytkownika. Informacją, która identyfikuje użytkownika jest jednorazowy sesyjny token, który z punktu widzenia pozostałych usług nie zawiera żadnych użytecznych informacji. Dzięki temu możliwa jest reimplementacja serwera autoryzacji bez utraty kompatybilności z pozostałymi częściami systemu. Uprawnienia użytkowników są składowane jako maska bitowa, w której zapalony bit oznacza posiadanie danego uprawnienia.

6.5. Integrator

Integrator został zrealizowany jako standardowa aplikacja internetowa obsługująca klientów poprzez protokół HTTP(S). Każda mikrousługa rejestrująca się w integratorze może poprosić o wystawienie dowolnej liczby tzw. końcówek (ang. *endpoint*). Końcówka jest jednoznacznie określonym adresem, pod którym serwer usługi udostępnia swoje zasoby. Wyróżniamy trzy typy końcówek:

- View - służy do udostępniania statycznych danych bez bezpośredniego udziału usługi, która ją utworzyła. Głównym zastosowaniem tej końcówki jest przechowywanie JSONa opisującego interfejs użytkownika po stronie frontendu. Integrator umożliwia dynamiczną zmianę zawartości końcówek tego typu, co znacząco ułatwia prototypowanie interfejsu użytkownika, gdyż projektant nie musi pisać własnej mikrousługi która serwowałaby stosowną treść.
- Read - służy do pobierania danych z serwera mikrousługi bez powodowania skutków ubocznych. Format przesyłanych informacji zależy od typu komponentu proszącego o nie w warstwie frontendu.
- Write - służy do wykonywania akcji mających efekt uboczny, np. złożenia wniosku.

Do jednej końcówki typu Read i Write może być przypisanych dowolnie wiele serwerów usługi. Integrator dla każdego zapytania losowo wybiera jeden z nich i do niego kieruje wchodzące zapytanie, a następnie przekazuje odpowiedź z powrotem.

Dzięki współpracy integratora z serwerem uwierzytelniania dostęp do poszczególnych końcówek może być ograniczony tylko do autoryzowanych użytkowników. Identyfikacja uprawnionych klientów odbywa się poprzez przekazanie tokenu sesji w parametrze zapytania.

W przypadku końcówek o typie Read i Write możliwe jest przechwytywanie i modyfikacja przesyłanych danych przy pomocy tzw. haków (ang. *hook*). Haki mogą być osobno zakładane na każdą końcówkę. Na jednej końcówce może być założonych dowolnie wiele haków, których kolejność wykonywania zależy od nadanego im priorytetu. Wyróżniamy dwa typy haków:

- In - modyfikują zapytania wchodzące do mikrousługi.
- Out - modyfikują wychodzące z mikrousługi odpowiedzi na zapytania.

Integrator oferuje dwie metody pobierania listy aktywnych końcówek. Dostępne są one pod adresami:

- /menu - zawiera informację o pierwszej końcówce, która powinna zostać wyświetlona po załadowaniu frontendu oraz spis zawartości menu. Menu jest personalizowane dla każdego użytkownika osobno i zawiera wpisy tylko o tych usługach, do których ma dostęp

użytkownik. Na wpis menu składają się stopień zagnieżdżenia w strukturze drzewa, wyświetlany tytuł i ikonka oraz końcówka, która powinna zostać wyświetlona po kliknięciu danego elementu.

- /list - wyświetla dane o wszystkich końcówkach i serwerach mikrousług do których dostęp ma obecnie zalogowany użytkownik. Otrzymana lista odzwierciedla informacje składowane w bazie danych, co czyni ją przydatną przy poszukiwaniu błędów podczas fazy rejestracji mikrousług.

W każdym z tych przypadków integrator przysyła odpowiedź w formacie JSON, dzięki czemu możliwe jest maszynowe jej przetwarzanie (na przykład w edytorze mikrousług).

Konfiguracja integratora przechowywana jest w bazie danych ZooKeeper. Każda mikrousługa musi podczas uruchamiania dodać do niej swoje wpisy. Takie rozwiązanie jest świadomym złamaniem zasady mówiącej, że baza danych z której korzysta usługa nie powinna być bezpośrednio dostępna dla innych mikrousług [MSV]. Wynikało ono z unikalnych własności wpisów w ZooKeeperze, które mogą być automatycznie usuwane po ustaniu pracy serwera mikrousługi, np. na skutek awarii lub utraty połączenia internetowego. Ponadto poszczególne węzły które składają się na drzewo przechowujące konfigurację integratora mogą być chronione poprzez mechanizm ACL. Uznaliśmy, że w naszym prototypie nie ma potrzeby reimplementowania funkcjonalności zapewnianej przez gotowe komponenty.

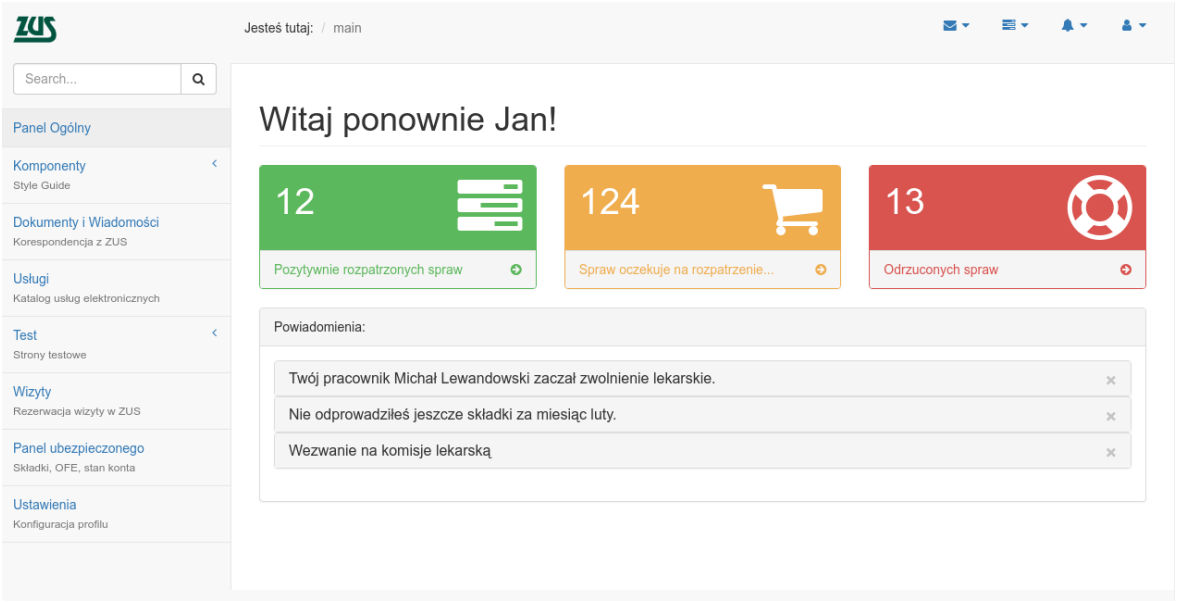
Początkowo integrator był implementowany w technologii Play Framework. Niestety w miarę postępu prac ujawnialiśmy kolejne problemy związane z tym frameworkiem, począwszy od konfliktu zależności pomiędzy pakietami, poprzez niedziałający tryb SSL, aż po zawieszanie się serwera. Być może było to spowodowane koniecznością użycia wersji niestabilnej frameworka. Wymienione kłopoty w połączeniu z niezadowalającym postępem prac skłoniły nas do przepisania integratora w języku Python. Nowa wersja nazwana przez nas integratorem 2 nie posiada wymienionych wyżej wad i została dogłębnie przetestowana.

6.6. Mikrousługi

W trakcie prac nad naszym portalem zaimplementowaliśmy niżej wymienione rodzaje mikrousług. Ich głównym zadaniem było zilustrowanie interfejsu programisty wykorzystywanego przy współpracy z integratorem, niekoniecznie zaś implementacja produkcyjnej funkcjonalności. Naszemu Klientowi zależało bowiem, by integracja poszczególnych warstw aplikacji była możliwie prosta i osiągalna nawet dla nisko wykształconych i niedoświadczonych programistów.

6.6.1. Powiadomienia

Powiadomienia są pierwszą zaimplementowaną mikrousługą. Jej zadaniem jest wyświetlenie liczby rozpatrzonych spraw. Aby dokładniej odzwierciedlić realia panujące w administracji przy każdym zapytaniu odpowiedź jest losowana. Otrzymane dane zostają następnie wyświetlone na poniższym ekranie:



Rejestracja usługi odbywa się podczas startu serwera przy użyciu kawałka kodu analogicznego do poniższego:

```
import notify.installer as inst

addr = "http://localhost:8000"
inst.zk.start()

inst.add_read_endpoint("notify-pending", "server1", addr + "/notify/pending/")
inst.add_write_endpoint("notify-set_variable", "server1", addr +
        "/notify/set_variable/")
```

W przedstawionym przykładzie następuje zaimportowanie biblioteki dostarczanej wraz z integratorem służącej do komunikacji z nim. Następnie zostaje nawiązane połączenie z serwerem integratora i tworzone są dwie końcówki typu Read i Write o nazwach „notify-pending” i „notify-set_variable” przypisane do serwera o identyfikatorze „server1”. Kiedy integrator zostanie poproszony o dane z końcówki „notify-pending”, wyśle on zapytanie pod adres `http://localhost:8000/notify/pending/` i przekaże odpowiedź z powrotem. Wyrejestrowanie usługi z integratora nastąpi automatycznie po wyłączeniu serwera.

6.6.2. Logger

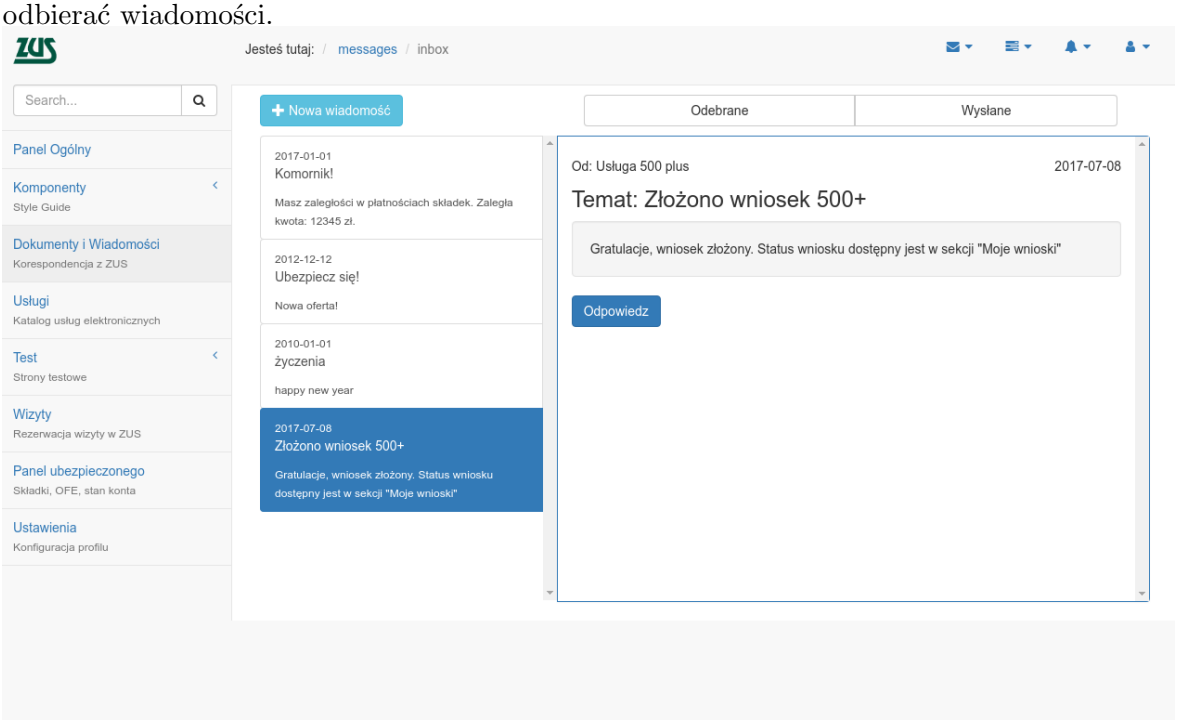
Logger jest mikrousługą ilustrującą działanie mechanizmu haków. Na początku rejestrowana jest końcówka typu Read o nazwie „google” wraz z odpowiadającym jej serwerem. Następnie zakładane są haki przy pomocy następujących poleceń:

```
inst.add_node('/hook/in/read/google', '', temp=False)
inst.add_node('/hook/out/read/google', '', temp=False)
inst.add_node('/hook/in/read/google/h1', addr + "/in/", temp=True)
inst.add_node('/hook/out/read/google/h1', addr + "/out/", temp=True)
```

Od tej pory cała komunikacja do i z wymienionej końcówki przechodzi przez serwer mikrousługi, która wykorzystuje ten fakt do wypisywania całego ruchu na konsolę. Kolejność aplikowania haków zależy od nazwy haka (w tym przypadku „h1”), tj. haki o mniejszych leksykograficznie nazwach są wykonywane wcześniej.

6.6.3. Poczta

Mikrousługa poczty była pierwszą zaimplementowaną usługą, z którą użytkownik naszej platformy mógł podjąć interakcję. Jak sama nazwa wskazuje, można przy jej pomocy wysyłać i odbierać wiadomości.

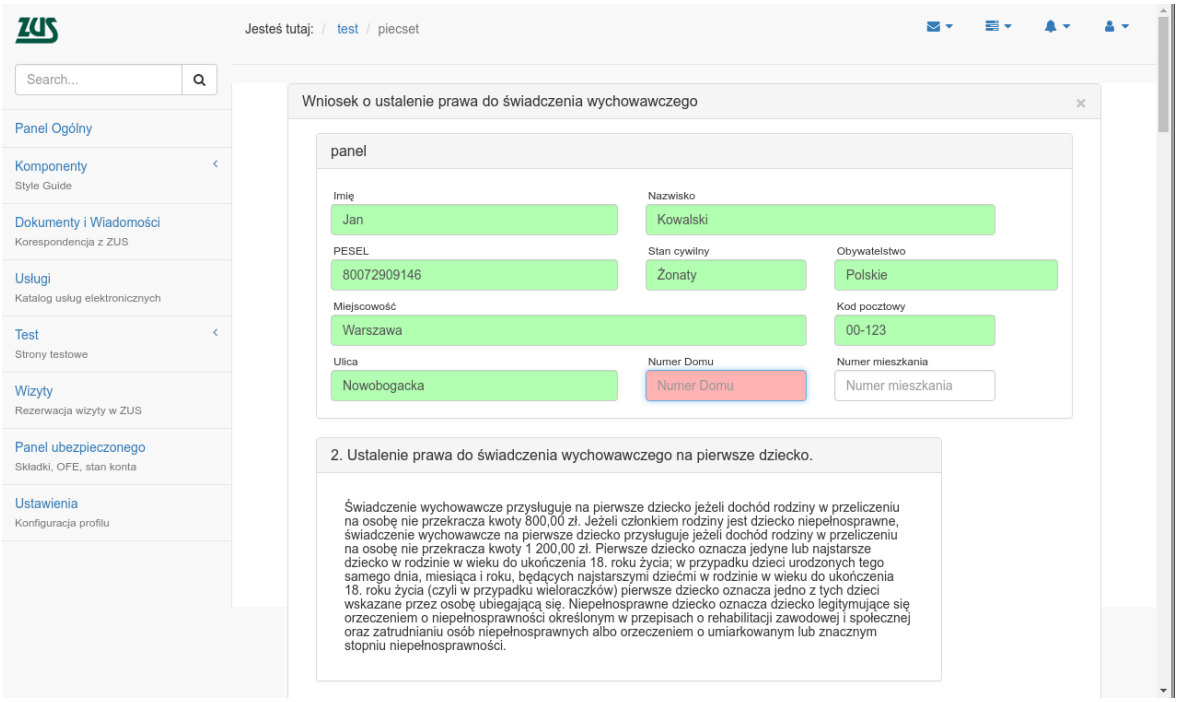


Tutaj ujawniły się pierwsze mocne strony przyjętej przez nas architektury całego portalu. Dzięki jasnemu odseparowaniu warstwy prezentacji od warstwy danych możliwe staje się korzystanie przez mikrousługi z dokładnie tego samego interfejsu, z którego korzysta warstwa frontendowa portalu do prezentacji danych użytkownikowi. Przykładowo, kiedy osoba siedząca przed komputerem wysyła wiadomość, to w tej samej chwili z tej samej końcówki może korzystać aplikacja kliencka na smartfona, system do zarządzania kadrami, a nawet inna mikrousługa znajdująca się w serwerowni ZUS. Wykorzystaliśmy ten fakt przy tworzeniu usługi 500+ do wysyłania powiadomień o udanym złożeniu wniosku. Unifikacja interfejsów sprawia, że utworzenie, utrzymanie i rozwój mikrousługi jest tańsze i prostsze. Eliminujemy w ten sposób konieczność utrzymywania kilku redundantnych kanałów do komunikacji z serwisem oraz w jego obrębie.

6.6.4. Plus500

Z powodu ograniczeń nałożonych na nazwy pakietów w języku Python usługę odpowiedzialną za obsługę wniosków o świadczenie 500+ nazwaliśmy „Plus500”. Nasza mikrousługa umożliwia wyświetlenie złożonych wniosków, rozpatrzenie ich przez urzędnika oraz wysłanie wypełnionego formularza.

Szablon formularza wniosku o świadczenie 500+ plus znajduje się w pliku „500plus.form” i jest opisany w formacie JSON. Po połączeniu z serwerem mikrousługi szablon jest wysyłany do klienta i wyświetlany. Po naciśnięciu odpowiedniego guzika zawartość wniosku jest serializowana i wysyłana do mikrousługi.



Rejestracja usługi w integratorze zajmuje tylko 7 linijek kodu:

```
inst.add_view_endpoint_from_file("plus500-wniosek", '500plus.form')
inst.add_read_endpoint("plus500-lista", "server1", addr + "/lista/")
inst.add_read_endpoint("plus500-pokaz", "server1", addr + "/pokaz/")
inst.add_write_endpoint("plus500-send", "server1", addr + "/wyslij/")
inst.add_write_endpoint("plus500-ustaw", "server1", addr + "/ustaw/")
inst.add_menu_element("Moje wnioski 500+", "read/plus500-lista", "/plus500lista")
inst.add_menu_element("Złóż wniosek 500+", "view/plus500-wniosek", "/plus500nowy")
```

W powyższym fragmencie widać większość możliwości integratora. Na początku instalujemy w integratorze plik opisujący formularz o świadczenie 500+. Następnie dodajemy końcówki które pozwalają na złożenie wniosku i wyświetlenie ich stanu. Na samym końcu rejestrujemy wpisy w menu. Od tej chwili zmiany są widoczne w interfejsie użytkownika i możliwa jest interakcja z mikrousługą.

Rozdział 7

Wkład poszczególnych członków zespołu w projekt

Członek zespołu	**Wykonane zadania**
Michał Jaroń	TODO
Jarosław Socha	TODO
Piotr Zalas	Zaproponowanie wstępnej architektury aplikacji. Implementacja integratora i integratora 2. Implementacja mikrousług powiadomień, loggera, poczty, 500+. Implementacja usługi autoryzacji. Implementacja serwera frontendu w Express.js. Przygotowanie diagramów ilustrujących architekturę. Spisanie niniejszej pracy (z wyjątkiem rozdziałów ...)

Rozdział 8

Podsumowanie

TODO

Dodatek A

Spis zawartości dołączonej płyty CD

Dokładny spis zawartości towarzyszącej płytki (p. dalej). To bardzo ważne, proszę zapisać jako osobny rozdział (czyli np. nie podrozdział). Płytka CD/DVD/Blu-ray/...

Zawiera:
Pełną dokumentacją projektu w łatwo dającym się odczytać formacie (najlepiej pdf + źródło). Program (w postaci źródłowej i potencjalnie umożliwiającej uruchomienie, to może oznaczać np. dostarczenie stosownych plików makefile, pomocniczych plików z danymi, opisu instalacji itp.). Wszelkie inne dokumenty powstałe podczas zajęć (np. teksty prezentacji, teksty pracy z poprzedniego dużego punktu, itp.).

Płyta jest częścią pracy - trzeba tyle płyt co drukowanych egzemplarzy pracy. Płytkę trzeba przymocować do pracy, tak by a) nie wypadała b) dało się ją wyjąć i odczytać w komputerze :).

Bibliografia

[TFA] Autor nieznany, *The Twelve-Factor App*, https://12factor.net/

[SDM] Autor nieznany, *Government Service Design Manual*, https://www.gov.uk/service-manual/index.html

[MSV] Chris Richardson, *Microservice architecture patterns and best practices*, http://microservices.io/

[MCO] Ministerstwo Cyfryzacji, *Portal Rzeczypospolitej Polskiej - Opis projektu*, https://mc.gov.pl/projekty/portal-rzeczypospolitej-polskiej/opis-projektu

[PUE] ZUS, *Platforma Usług Elektronicznych*, http://pue.zus.pl/

[RDL] Redis, *Distributed locks with Redis*, https://redis.io/topics/distlock

[RDS] *Redisson*, http://redisson.org/

[AZK] Apache, *Apache ZooKeeper*, https://zookeeper.apache.org/

[ACU] Apache, *Apache Curator*, http://curator.apache.org/

[HTL] Martin Kleppmann, *How to do distributed locking*, http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

[CLK] Apache, *Apache Curator - Shared Lock*, http://curator.apache.org/curator-recipes/shared-lock.html

[Fowler] James Lewis, Martin Fowler, *Microservices*, https://martinfowler.com/articles/microservices.html

[RSUS] ZUS, *Rocznik Statystyczny Ubezpieczeń Społecznych*, http://www.zus.pl/baza-wiedzy/statystyka/rocznik-statystyczny-ubezpieczen-spolecznych

[Beanstlkd] Keith Rarick, *Beanstalkd*, http://kr.github.io//beanstalkd/

[Kafka] Apache, *Kafka*, https://kafka.apache.org/

[DL] Apache, *DistributedLog*, http://distributedlog.incubator.apache.org/

[Netflix] Tony Mauro, *Adopting Microservices at Netflix: Lessons for Architectural Design*, https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[mammatustech] Rick Hightower, *Java Microservices Architecture*, http://www.mammatustech.com/java-microservices-architecture