POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Michał Sadowski

Nr albumu 17983

System wspierający tworzenie nowych projektów informatycznych

Praca Magisterska napisana pod kierunkiem:

Dr. Inż. Mariusz Trzaska

Warszawa, Luty, 2020

Streszczenie

Poniższa praca prezentuje praktyczną propozycje systemu upraszczającego proces tworzenia oprogramowania. Z mojego dotychczasowego doświadczenia, większość nowych projektów, szczególnie projektów prototypów (*Proof-of-Concept*), posiada wiele wspólnych elementów. Kod, ale także przygotowanie środowisk i narzędzi deweloperskich są często współdzielone pomiędzy wieloma projektami. Przygotowanie wspomnianych elementów w przypadku tworzenia nowych projektów jest czasochłonne i kosztowne. Wykorzystanie platform typu *Low-Code*, czyli platform tworzenia oprogramowania z małą ilością kodowania, jest coraz popularniejszą propozycją optymalizacji. Zastosowanie takich rozwiązań, od początku procesu tworzenia do pierwszej wersji deweloperskiej, może być znaczną oszczędnością zasobów. Jest to szczególnie pożądane w przypadku tworzenia prototypów. Wykorzystanie narzędzi tworzenia i procesowania szablonów, połączone z serwisem umożliwiającym interakcje z platformą poprzez przystępny interfejs graficzny, pozwoliłoby sprostać tym wymaganiom a także obniżyć próg wejścia w tworzeniu oprogramowania osobom nietechnicznym. W pracy przedstawiono propozycje takiego prototypu w formie platformy webowej opartej o języki programowania *Java* i *Dart*, wykorzystującej różne technologię szablonowania. Platforma wspiera kolaboracje pomiędzy użytkownikami, umożliwiając dzielenie się szablonami poprzez zewnętrzne repozytorium kodu GitHub. Prototyp został zaprezentowany architektowi systemów informatycznych w dużej spółce o profilu techniczno-farmaceutycznym.

Słowa kluczowe:

Rapid Prototyping, Low-Code Development, Java, Web Application

Podziękowania

Autor pracy wyraża podziękowanie promotorowi dr. Mariuszowi Trzasce za wsparcie merytoryczne, cierpliwość i motywację, a także panu Mateuszowi Filipowiczowi, architektowi systemów informatycznych w firmie Roche Polska S.A za pomoc w identyfikacji procesów biznesów, które mogą zostać zoptymalizowane poprzez zastosowanie tematyki pracy.

Spis treści

1. Wstęp

Przewaga rynkowa firmy, w dzisiejszych czasach, jest blisko skorelowana z przewagą technologiczną firmy. Umiejętność szybkiego zaadaptowania i możliwość przetestowania nowych pomysłów i technologii może być kluczowe, żeby zdobyć przewagę nad konkurencją. W skrajnych przypadkach, takich jak firmy farmaceutyczne, przewaga może zapewnić wieloletnią wyłączność na sprzedaż swoich produktów. Żeby zapewnić sobie konkurencyjność, firmy kładą duży nacisk na skrócenie czasu tworzenia nowych systemów i redukcję okresu testowania nowatorskich rozwiązań technologicznych (*Rapid Prototyping* [1]). W tym celu, powstają platformy typu *Low-code/No-code development platform* [2], które pozwalają na szybkie i zwinne tworzenie aplikacji zapewniających spełnienie podstawowych wymagań (*Minimal Viable Product* - MVP) [3] i wygenerowanie minimalnej wartości biznesowej spełniającej założenia projektu.

Przedmiotem pracy jest praktyczne podejście do problemu, prezentując projekt platformy wspomagającej początkowe etapy projektu. Przykładami takich etapów są: przygotowanie narzędzi deweloperskich, środowisk pracy i automatyzacja implementacji powtarzającego się kodu.

1.1 Aplikacja webowa do tworzenia projektu aplikacji i definiowania środowiska pracy

W dzisiejszych czasach, bardzo popularnym medium dystrybucji usług jest Internet. Aplikacje i inne systemy oferujące usługi tworzone są w sposób umożliwiający wykorzystanie ich funkcjonalności z poziomu przeglądarki. Aplikacje wykorzystujące powyższy paradygmat, nazywane są aplikacjami internetowymi (webowymi) [4] . Model biznesowy, którym się posługują jest modelem software-jako-usługa. Wartym zwrócenia uwagi jest poprawne zdefiniowanie tych dwóch pojęć. Aplikacja internetowa, zwana także webową, jest programem komputerowym o architekturze klient-serwer, gdzie warstwa logiki biznesowej znajduje się po stronie serwera a warstwa prezentacyjna jest wyświetlana przez klienta, najczęściej przeglądarkę internetową. SaaS (*Software-as-a-Service*) [5] jest modelem serwowania usług dostarczanych przez aplikację internetową, poprzez wykorzystanie Internetu jako medium a przeglądarki internetowej klienta jako odbiorcy. W ostatniej dekadzie, SaaS stał się bardzo popularnym modelem dostarczania i licencjonowania oprogramowania do wirtualizacji, zarządzania klientami CRM, zarządzania zasobami ERP, systemami rekrutacyjnymi czy zarządzania wiedzą CM. Jest także jednym z konceptów pojawiających się w technologiach chmury, wraz z IaaS (*Infrastructure-as-a-Service*) i PaaS (*Platform-as-a-Service*).

Rozwiązania bazujące na modelu SaaS, są proste z perspektywy klienta, gdyż jest on wyłącznie konsumentem danych serwowanych przez taką aplikację. Rozwój i utrzymanie konkurencyjności leżą w interesie twórcy oprogramowania. Jest to rozwiązanie łatwo skalowalne, nie wymagające instalacji oprogramowania po stronie klienta. Umożliwia to użytkownikom dostęp to usług z dowolnej lokacji. Ponadto, procesy wsparcia i aktualizacji systemu dokonywane są w sposób automatyczny i niewymagający ingerencji konsumenta.

Ogólny trend w kierunku rozwiązań internetowych, bazujących na rozwiązaniach chmurowych, spowodował, że wiele aplikacji desktopowych doczekało się swoich odpowiedników webowych. Przykładem tego trendu jest pakiet Microsoft Office i jego internetowy odpowiednik *Office 365* [6]. Przy tworzeniu nowego systemu, warto jest wziąć pod uwagę architekturę SaaS jako model udostępniania funkcjonalności. Usługi aplikacji mogłby być konsumowane poprzez interfejs graficzny dostępny z poziomu przeglądarki. Dodatkowo, komunikacja z aplikacją powinna być możliwa w sposób programistyczny, poprzez API w standardzie REST [7]. Nawigacja wykorzystująca interfejs graficzny obniża próg wejścia dla potencjalnego użytkownika, a także umożliwia skuteczniejsze zarządzanie zasobami. Interfejs programistyczny pozostawia natomiast możliwość automatyzacji procesów z wykorzystaniem zewnętrznych aplikacji oraz tworzenie rozszerzeń funkcjonalności.

Bazując na zdobytym doświadczeniu w projektowaniu i wdrażaniu rozwiązań informatycznych autora, praca jest propozycją przyspieszenia i ułatwienia procesu tworzenia nowych aplikacji.

1.2 Platformy Low-/No-Code

Wspomniane we wstępie zagadnienia: *Low-Code* i *No-Code Development* są prezentacją nowatorskich metod kreowania oprogramowania. Te terminy są bardzo blisko powiązane ze sobą. Oba określają metodyki mające na celu przyśpieszanie procesu kreowania oprogramowania oraz jego uproszczenie, poprzez redukcję ilości kodu napisanego przez zespół programistyczny. Założeniem *Low-Code Development* jest zastąpienie tradycyjnego programowania edytorem wizualnym. Platformy umożliwiające wizualny proces nazywają się *LCDP - Low Code Development Platform*. Dają one możliwość użytkownikowi zdefiniowanie podstawowych fragmentów, procesów i aspektów aplikacji wykorzystując mechanizmy *drag-and-drop* i *point-and-click*. Programista wybiera odpowiednie funkcjonalności z menu platformy *LCDP* i bezpośrednio dodaje do finalnego rozwiązania. Taki sposób kreowania oprogramowania pomaga skupić się na implementacji danego procesu biznesowego, zdefiniowania baz danych czy interfejsów użytkownika dla aplikacji webowych. Aplikacje stworzone

z wykorzystaniem tego typu platform mogą zostać wdrożone nawet o 70% szybciej, w stosunku do tradycyjnego podejścia [8].

No-Code Development i *No-Code Development Platform (NCDP)* [9] są dalszym rozwinięciem ideologii LCDP, gdyż z założenia, proces tworzenia produktu nie wymaga wsparcia ze strony zespołów deweloperskich. W takim modelu, klient mógłby sam, przy użyciu tego typu platformy, zdefiniować swoją aplikację za pomocą GUI, a decyzje dotyczące architektury i implementacji rozwiązania zostałyby podjęte przez platformę.

2. Cel pracy

Celem pracy jest zdefiniowanie wymagań platformy przyspieszającej proces wytwarzania oprogramowania, zaprojektowanie technicznego aspektu systemu oraz implementacja prototypu. W trakcie przeprowadzania analizy, zostały zidentyfikowane główne procesy, których optymalizacja zredukowałaby czas potrzebny na wytworzenie biznesowej wartości dodanej. Szczegółowy opis tych procesów wraz z objaśnieniem znajduje się w następnych rozdziałach.

2.1 Istota problemu

Podczas tworzenia pracy, problematyka platform LCDP została dogłębnie przeanalizowana. Zidentyfikowane zostały procesy w tworzeniu oprogramowania, które mogłyby zostać zastąpione przez zautomatyzowany system, skonfigurowany poprzez graficzny interfejs użytkownika.

Poniższe trzy aspekty zostały wytypowane jako wymagające usprawnień przy tworzeniu oprogramowania na potrzeby wewnętrzne firm:

1. Redukcja czasu pomiędzy oficjalnym startem projektu, a posiadaniem pierwszej, bardzo podstawowej wersji systemu działającej na zewnętrznej infrastrukturze. Jest to powiązane z długim procesem zamawiania i dostosowania narzędzi wokół deweloperskich, takich jak systemy kontroli wersji, systemy zarządzania incydentami czy strumienie *Continuous Integration, Continuous Deployment,*
2. Brak jasnego procesu lub narzędzia, wspierającego współdzielenie fragmentów funkcjonalności, konfiguracji i dokumentacji dla standardowych problemów i wymagań. Przykładem mogą być funkcjonalności logowania w aplikacjach korporacyjnych lub definicje obrazów kontenerów służących do wdrażania rozwiązań,
3. Problem między-projektowej powtarzalności kodu.

Poprzez analizę powyższych problemów, stworzone zostały wymagania dla projektu, których pochodną jest specyfikacja techniczna rozwiązania i implementacja prototypu.

2.2 Analiza optymalizowanych procesów biznesowych

W perspektywie kontekstu tej pracy, wartym jest zrozumienie standardowego procesu tworzenie oprogramowania [10]. Pozwoli to zrozumieć specyfikę usprawnień wprowadzanych przez opisywane rozwiązanie.

W zdecydowanej większości przypadków, proces rozpoczyna się od zgłoszenia przez klienta zapotrzebowania na nowy system. Ma on za zadanie zoptymalizować, najczęściej poprzez częściową automatyzację, pewien proces biznesowy lub być implementacją nowego procesu. Ten proces jest poddany wnikliwej analizie, podczas której są spisane i skatalogowane wymagania, które nowy system musi spełniać, aby dostarczyć oczekiwaną wartość biznesową dla klienta. Na podstawie zebranych wymagań, dokonuje się estymacji czasu i wymaganych zasobów potrzebnych, aby ukończyć projekt.

Jeżeli estymacja jest zgodna z oczekiwaniami klienta, można rozpocząć prace nad tworzeniem specyfikacji technicznej oraz architektury systemu, definiując budowę i złożoność jego implementacji a także odpowiadający wymaganiom stos technologiczny.

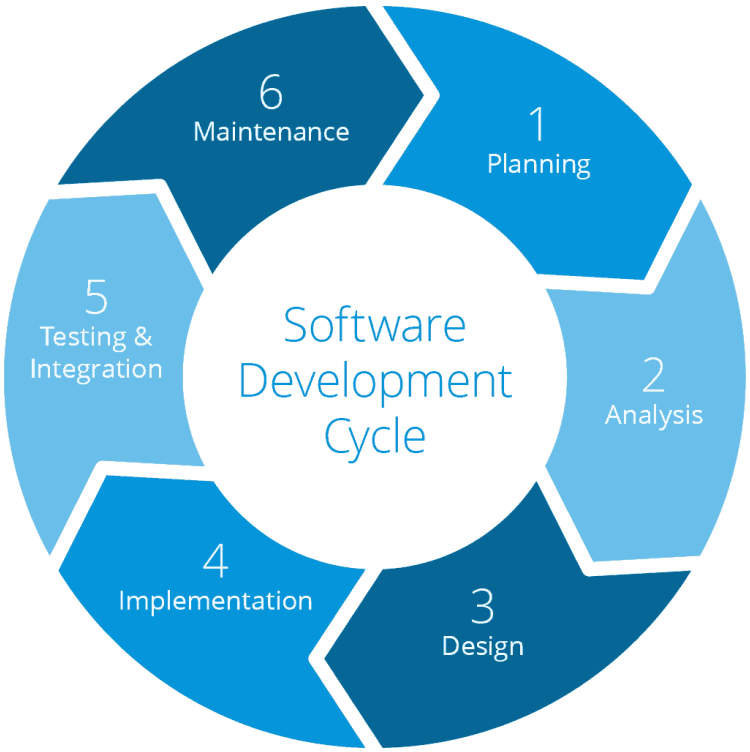
W tym momencie rozpoczyna się faza development'u, czyli implementacji procesów biznesowych, poczynając od dostosowania środowisk programistycznych do wymogów pracy a także przygotowania odpowiednich systemów wspierających, takich jak systemy zarządzania incydentami, planowania pracy czy systemy kontroli wersji. Sposób ich konfiguracji, a także skład zespołu deweloperskiego może być zależny od przyjętej metodologii wytwarzania oprogramowania.

Najbardziej popularną metodyką zarządzania projektami informatycznymi jest metoda "Agile" [11]. Jest to metodologia zakładająca wysoką zdolność projektu do adaptacji do zmian i iteracyjnemu podejściu do dostarczania dodanej wartości biznesowej. Zespół w tej metodologii zorganizowany jest multidyscyplinarnie, wraz z przedstawicielem strony klienta aktywnie uczestniczącym w procesie. Ponadto jest samoorganizujący się, planując prace nad dostarczeniem funkcjonalności biznesowej według priorytetów wyznaczonych przez właściciela systemu. Produkt dostarczany jest w procesie iteracyjnym, gdzie w każdej następnej iteracji, zwanej też *sprintem*, dodawany jest zestaw nowych funkcjonalności. Ten proces pozwala zespołowi wyciągnąć wnioski i bazować na doświadczeniach z poprzedniej iteracji, szerząc zrozumienie specyfiki projektu i jego wymagań. Każda iteracja przeważnie zakończona jest demonstracją przed interesariuszami.

Każdy sprint zawiera wszystkie fazy tradycyjnego podejścia do tworzenia oprogramowania:

1. Zbiór wymagań danego sprintu
2. Projekt systemu
3. Proces Implementacji rozwiązań
4. Weryfikacja implementacji poprzez testowanie
5. Projekt planu utrzymania systemu

Wizualizacja opisanego wyżej procesu przedstawiona jest na rysunku 1.



Rysunek 1 Cykl tworzenia oprogramowania w metodologii Agile; Źródło: [12]

Proces iteracyjny trwa, póki wszystkie zdefiniowane wymagania nie zostały obsłużone przez system lub do momentu wyczerpania zasobów. Zakończenie development'u wiąże się z przekazaniem w ręce klienta pełnej wersji oprogramowania i materiałów produkcyjnych. Dokumentacja techniczną rozwiązania oddawana jest w ręce zespołu operacyjnego, którego zadaniem są prace utrzymaniowe i modernizacyjne systemu.

2.3 Rezultaty pracy

Rezultatem pracy jest wynik analizy wymagań do stworzenia nowej platformy i problematyki zagadnień związanym z jej projektem. Na jej podstawie została przeprowadzona implementacja systemu informatycznego. Prototyp został podzielony na następujące elementy, zgodnie z domenowym rozdziałem obowiązków:

- Moduł rejestracji i zarządzania danymi użytkownika,
- Moduł graficznego panelu użytkownika,
- Moduł generowania pustych projektów programistycznych,
- Moduł generowania klas z szablonów,
- Moduł przechowywania i zarządzania szablonami,
- Moduł definiowania strumieni zadań umożliwiających dopasowanie i rozszerzenie funkcjonalności do indywidualnych potrzeb projektowych,
- Moduł integracji z systemem wersji Git,

Projekt zyskał także nazwę: *Zoran.io*. Nawiązania w tekście do tej nazwy odwołują się do opisywanego projektu.

Dzięki zastosowaniu platformy generowania kodu z szablonów w swoim projekcie, klient oszczędza dużo czasu, a zatem i funduszy, automatyzując początkowe fazy projektu. Usprawnione zostałyby procesy połączone z pierwszą fazą tworzenia oprogramowania: z ustawieniem projektu, przygotowaniem narzędzi deweloperskich (systemy wersjonowania, śledzenia zadań i systemu budowania), definiowanie zależności i pakietów i stosu technologicznego. System pozwoliłby użytkownikowi wybrać z poziomu graficznego interfejsu, potrzebne zależności, narzędzia (GitHub itp.) i automatycznie wygenerować szkielet nowego projektu, gotowego do deployment'u.

Zbudowana platforma kierowana jest głównie do zespołów deweloperskich operujących w języku Java, Groovy lub Kotlin. Platforma została zaprojektowana z naciskiem na generyczność i wsparcie dodatkowych języków można osiągnąć przy niskim nakładzie pracy. Wspomniana generyczność dotyczy również silnika do procesowania szablonów, pozwalając na dodanie innych technologii szablonowych rozszerzając abstrakcyjna fasadę systemu.

2.4 Organizacja pracy

Praca została podzielona na osiem rozdziałów. Pierwsze dwa rozdziały zawierają wstęp do pracy oraz opis założeń i celu pracy. Rozdział trzeci zawiera opis rozwiązań istniejących w trakcie analizy projektu.

W następnych rozdziałach jest zaprezentowany projekt systemu wraz z opisem wykorzystanych technologii oraz implementacji systemu. Praca jest zwieńczona wizją rozwoju aplikacji oraz wnioskami końcowymi.

3. Istniejące rozwiązania

Low Code Development nie jest nowym zagadnieniem na rynku. Pierwsze systemy spełniające funkcje platform umożliwiających generowanie aplikacji spotkały się jednak z krytyką ze strony architektów, przez co cała idea została zakwestionowana. Argumentem opozycji była analiza produktów końcowych stworzonych z wykorzystaniem systemów *Low Code*. Jej wynikiem była teza, że takie systemy tworzyły luki w wytwarzanym oprogramowaniu, czyniąc je podatnymi na ataki z zewnątrz. Ponadto, analizy wykazały ze posiadały one nieefektywne i nieprawidłowe architektury [13]. Niemniej jednak, w ostatnim okresie wzrosło zainteresowanie tymi platformami, głównie poprzez ich możliwość znacznego przyspieszenia procesu prototypowania i tworzenia mniej kluczowych aplikacji dla użytku wewnętrznego firmy. Rynek, naturalnie, odpowiada na zapotrzebowanie i popyt. W momencie pisania pracy dostępnych jest kilka platform, które zostały scharakteryzowane przez agencje Gartner'a [14] w ich raporcie. Rysunek 2 zawiera wykres przedstawiający wynik raportu, klasyfikując dostępne na rynku, komercyjne platformy *Low-/No-Code*.

Rysunek 2 Klasyfikacja dostępnych platform Low-/No-Code; Źródło: [15]

Jak pokazuje rysunek 2, na rynku istnieje już kilkanaście platform *LCD*. Ponadto, dostępne są również rozwiązania kierowane do zespołów deweloperskich. Celem tych serwisów jest umożliwienie sprawniejszego tworzenia oprogramowania poprzez, na przykład, automatyczne generowanie kodu. W poniższych podrozdziałach znajduje się opis oraz analiza wybranych rozwiązań dostępnych na rynku w trakcie pisania pracy.

3.1 Platforma *Yeoman*

Platforma *Yeoman* lub *yeoman.io* [16], jest nowoczesnym narzędziem umożliwiającym generowanie nowych projektów. Aplikacja tworzona jest z wykorzystaniem generatorów, które zawierają opis i konfigurację projektu. W trakcie generowania produktu końcowego, stosowane są dobre praktyki programistyczne i wzorce projektowe, w celu poprawienia jakości stworzonej aplikacji. *Yeoman* jest rozwiązaniem pracującym w formie aplikacji dostępnej poprzez interfejs terminala (*Command Line Interface - CLI*), działając na maszynie klienta końcowego. Dostępnych jest kilka tysięcy generatorów, tworzących fragmenty aplikacji w wielu różnych technologiach. Celem platformy jest usprawnienie procesu tworzenia oprogramowania poprzez automatyczne generowanie kodu. W tym celu wykorzystywane są następujące narzędzia, Yeoman jako platforma generująca kod, npm jako system zarządzania zależnościami i Gulp, jako system budujący aplikację. Platforma działa w oparciu o zasady Open Source, umożliwiając wykorzystanie swoich funkcjonalności bez opłat.

3.2 jHipster.com

JHipster [17] jest przykładem platformy *LCDP* kierowanej do deweloperów umożliwiając im automatyczne generowanie projektów programistycznych. Użytkownicy końcowi mają możliwość konfiguracji definicji projektu końcowego z wykorzystaniem CLI. Jest to najlepiej rozbudowany i najbardziej obszerny projekt wspomagający proces produkcyjny, umożliwiający tworzenie podstawowych aplikacji wraz z zdefiniowanym modelem danych bez potrzeby pisania kodu. *JHipster* oferuje także wiele możliwości ułatwienia kreowania systemów do wdrażania aplikacji na środowiska deweloperskie. Znaczącą różnicą pomiędzy *JHipster* a aplikacją, której dotyczy ta praca, jest fakt, że *Zoran.io* pozwala na personalizacje łańcuchów wykonawczych, wykonywanych podczas procesu wdrożeniowego. Umożliwia to łatwiejszą integrację rozwiązań w środowisku typu *enterprise* [18]. Środowiska *enterprise* są to środowiska skonfigurowane na zabezpieczonej infrastrukturze wewnętrznej firmy, w której działają i komunikują się systemy typu CRM, CS, ERP. Często są to środowiska ściśle przestrzegające wyznaczonych standardów bądź regulowane przez dodatkowe jednostki administracyjne. Przykładem są firmy farmaceutyczne, których sieci informatyczne i aplikacje podlegają kontroli ze strony zewnętrznych organizacji rządowych (*Computer System Validation - CSV)* [19].

JHipster jest systemem, którego funkcjonalność jest dostępna z poziomu terminala, ale także za pośrednictwem przeglądarki internetowej - interfejsu graficznego.

3.3 start.spring.io

start.spring.io jest platformą oferującą możliwość wygenerowania szkieletu aplikacji bazujących na stosie technologicznym *Spring* i języka programowania *Java, Groovy* czy *Kotlin*. Konsumowanie serwowanych funkcjonalności jest możliwe poprzez interfejs graficzny w postaci serwisu internetowego. Jest to rozwiązanie ograniczone do projektów wykorzystujących stos technologiczny *Spring*, a także nie posiada możliwości tworzenia własnych procesów wdrażania aplikacji na środowiska produkcyjne. Platforma *start.spring.io* oferuję możliwość integracji modułu generacji kodu w zewnętrzną aplikację, udostępniając bibliotekę generacyjną na bazie licencji otwartej. Biblioteka ta jest jedną z bibliotek używanych przez *Zoran.io*.

3.4 Wybrane LCDP/NCDP

Istnieje wiele platform, których celem jest budowanie rozwiązań realizujących pewne procesy biznesowe, w sposób ograniczający, lub wręcz eliminujący, potrzebę implementacji logiki w kodzie. Przykładami tego typu platform są *Salesforce, OutSystems* czy *Appian*. Umożliwiają one tworzenie gotowych rozwiązań w sposób deklaratywny, implementując logikę aplikacji poprzez interfejs graficzny. Obniża to próg wejścia w proces tworzenia oprogramowania, a także skraca czas potrzebny do wdrożenia aplikacji. Są to bardzo potężne narzędzia pomagające w automatyzacji procesu tworzenia oprogramowania. Platformy *LCDP/NCDP* dają użytkownikowi do dyspozycji szeroki wachlarz modułów ("*connectors*"), implementujących fragment algorytmu lub nawet pełnych procesów biznesowych. Moduły umożliwiają też komunikację i integrację z zewnętrznymi aplikacjami w ekosystemie. Istnieją także specjalne moduły domenowe, takie jak nauczanie maszynowe czy duże zbiory danych.

Analiza porównawcza tych platform została przeprowadzona w oparciu o dane zamieszczone w dwóch niezależnych raportach zleconych przez firmy konsultingowe *Forrester* [20] i *Gartner* [12]. Do ewaluacji zostały wybrane rozwiązania wiodące, o znacznej obecności rynkowej i występującej w przynajmniej jednym, z wyżej wymienionych, raportów.

3.4.1 The *Appian Platform*

The Appian [21] jest wiodącą firmą oferującą produkt typu *LCDP*. Ich rozwiązanie, The *Appian Platform*, została doceniona jako kompletne rozwiązanie dla BPM (*Business Process Modelling*), wspierające kolaboracje biznes-programista w każdym stadium tworzenia oprogramowania. Platforma posiada szeroką paletę dodatków poszerzających jej możliwości. Przykładami dodatkowych modułów

są na przykład nowatorskie rozszerzenia wspierające sztuczną inteligencję czy zarządzanie danymi *Big Data*. *The Appian* posiada dwie wady, degradujące doświadczenia użytkownika końcowego. Po pierwsze, dodatkowe elementy kreują mocne zależności z zewnętrznymi dostawcami usług, przykładowo komponent ML/AI z *Azure, GCP* i *Amazon*. Po drugie, nie istnieje możliwość modyfikacji i dodawania procesów wizualizacji danych, poza tymi, które są wbudowane. Raport *Forrester'a* [19] wskazuję na brak elastyczności w cenie subskrypcji, sprawiając, że platforma może być niedostępna dla mniejszych klientów.

3.4.2 *OutSystems*

OutSystems [22] jest bardzo dojrzałym rozwiązaniem realizującym strategie zwiększenia pokrycia przypadków użycia nowoczesnych aplikacji biznesowych. Platforma posiada szeroką społeczność deweloperską oraz dedykowany sklep, w którym istnieje możliwość dokupienia dodatkowych rozszerzeń. Jako wiodąca platforma *LCDP*, jest bardzo dynamicznie rozwijana jednak jej głównym problemem są ubogie specjalistyczne narzędzia domenowe. Powoduje to, że próg wejścia w platformę jest stosunkowo wysoki i ograniczony z perspektywy dostępnych akcji.

3.4.3 *Salesforce*

Salesforce [23] jest największą platformą *Low-Code* z trzech wymienionych, pod względem ilości klientów. Jej częsty wybór może być efektem wysokiego stopnia satysfakcji z usług deweloperskich, zintegrowanego środowiska *cloud*-owego, dojrzałości rozwiązania i jej skalowalności. *Salesforce* jest na rynku od ponad 15 lat i wspomniana wcześniej dojrzałość oznacza także pewien dług technologiczny. Oznacza to, że platforma potrzebuje czasu i znacznych nakładów pracy, żeby móc konkurować w dynamicznym świecie innowacji i mobilności.

3.5 Wnioski po analizie istniejących rozwiązań

Przytoczone przykłady istniejących rozwiązań pokazują, że temat *Low-Code Development*, jest odpowiedzią na rosnące zapotrzebowanie rynku na systemy umożliwiające szybkie prototypowanie. Obecnie, dostępne rozwiązania są coraz popularniejsze, bardziej elastyczne i umożliwiają kompleksową obsługę procesu tworzenia oprogramowania. Podejścia do zaspokojenia potrzeb rynkowych są skrajne. Od platform, których celem jest całkowite odejście od tradycyjnego programowania na rzecz implementacji procesów wykorzystując narzędzia wizualne, do systemów umożliwiających szybki start w procesie *development*'u.

Te podejścia prezentują różne wymagania, które nowe platformy muszą spełniać, jednak można wyodrębnić zbiór wspólnych elementów:

- Wszystkie platformy umożliwiają konsumpcje funkcjonalności poprzez graficzny interfejs użytkownika. Łatwość użycia i niski próg wejścia są priorytetem.
- Systemy typu *open-source* są zwykle systemami, dla których deweloper jest klientem końcowym, więc oferują możliwość konsumpcji funkcjonalności poprzez interfejs programistyczny.
- Głównym problemem systemów był brak możliwości personalizacji i rozszerzalności funkcjonalności oferowanych przez platformę.
- Bardzo ważnym punktem spójnym jest nie tylko możliwość szybkiego tworzenia aplikacji, ale także zapewnienie prostego wdrożenia w ekosystem korporacyjny.

4. Propozycja nowego systemu LCDP

W poprzednich rozdziałach omówiona została problematyka pracy, a także przedstawione zostały różne podejścia stosowane w już istniejących systemów typu *Low-Code*. Konsolidacja nabytej wiedzy, pozwoliła odkryć słabe strony funkcjonujących rozwiązań i jednocześnie zaproponować nowe narzędzie pozbawione niedogodności.

4.1 Wizja systemu

Analiza procesu tworzenia oprogramowania przedstawiła ich nieefektywność. W celu przypomnienia poniżej znajduję się lista zidentyfikowanych problemów:

1. Długi okres oczekiwania pomiędzy oficjalnym startem projektu, a pierwszą działającą wersją systemu,
2. Brak procesów lub narzędzi, wspomagających wymianę wiedzy i doświadczeń pomiędzy zespołami projektowymi,
3. Nadmierna powtarzalność kodu między źródłami różnych systemów.

W klasycznym podejściu, zanim programista będzie w stanie kontrybuować swój kod do repozytorium, należy przygotować środowisko projektowe. W zależności od ustawienia projektu, ilość zewnętrznych systemów wymagających konfiguracji jest różna. Poczynając od zdefiniowania zewnętrznego repozytorium kodu, a kończąc na przygotowaniu systemu zarządzającego incydentami, ukształtowanie struktury projektu jest skomplikowane i czasochłonne. Co gorsza, jest to praca najczęściej wykonywana manualnie poprzez wystawienie odpowiedniego zapytania w wewnętrznym systemie wsparcia IT. Proces ten można znacznie ukrócić wykorzystując automatyzację. Jeżeli istniałby system umożliwiający złożenie wszystkich próśb dostępu z poziomu pojedynczego formularza, znacznie uprościłoby to pracę deweloperów. Niestety, taki system wymagałby standaryzacji ekosystemów informatycznych we wszystkich firmach korzystających z narzędzia. Przykładowo, jeżeli jedna firma korzysta z serwera Atlassian BitBucket jako zewnętrze repozytorium kodu, proces zakładania nowego projektu będzie inny niż przypadku firmy działającej na AWS CodeCommit. Bardzo ważne jest więc, żeby nowy system miał możliwość personalizacji wspieranych procesów oraz dodania własnych rozwiązań spełniających indywidualne przypadki użycia.

Kolejnym problemem jest brak narzędzi wspomagających wymianę wiedzy i doświadczeń. W firmowych systemach informatycznych rozwiązania na standardowe wymagania są tworzone na nowo. Przykładowo, logika umożliwiająca logowanie w oparciu od protokół LDAP jest tworzona od początku w różnych systemach, podczas gdy mogłaby być spójna. Różnicą, w tym przypadku, pomiędzy implementacjami są filtry, po których protokół wyszukiwałby użytkowników. Ponowne wykorzystanie

istniejącego już fragmentu rozwiązania znacznie zredukowałoby ogólny czas produkcji oprogramowania, a także wprowadziłoby standaryzacje. Stworzenie narzędzia, które by to umożliwiło jest niemałym wyzwaniem, gdyż różne zespoły korzystają z różnych technologii i mają różne problemy wyzwania. Oznacza to, że rozwiązanie musi być agnostyczne względem wykorzystanej technologii, oraz być niezależne od tego, czy zespoły współdzielą całe projekty, pojedyncze klasy czy tylko pliki konfiguracyjne.

Obecne systemy nie posiadają funkcjonalności, które adresowałyby powyższe problemy. Istnieje więc popyt, szczególnie wśród środowisk korporacyjnych, na narzędzie, które umożliwiłoby optymalizację pracy wokół wymienionych niedogodności. Głównymi założeniami prototypu zaprezentowanego w poniższej jest automatyczna generacja kodu i generyczność.

Idea szablonów jest bezpośrednią odpowiedzią na problem powtarzalności kodu. Jeżeli implementacje logiki różnią się tylko nieznacznie, tak jak w przypadku wspomnianego przykładu, gdzie różnicą pomiędzy implementacjami są tylko filtry, można by całą klasę zastąpić szablonem. Operując dalej na tym przykładzie, szablon przyjmowałby parametr, będący definicją filtra. Umożliwia to spersonalizowanie logiki oraz obsłużenie wymagania funkcjonalnego logowania dla nowej aplikacji. Wygenerowanie klasy wykorzystując szablon znacznie uprościłoby prace i skróciłoby czas produkcji oprogramowania. W podobny sposób, zespoły mogłyby współdzielić nie tylko fragmenty logiki biznesowej, ale także konfiguracje środowisk, infrastrukturę jako kod oraz definicje ciągów CI/CD.

Drugim założeniem jest generyczność. Tak jak zostało to wspomniane powyżej, środowiska firmowe są bardzo różne. Rozbieżności nie leżą tylko w wykorzystanym języku programowania, ale także w środowiskach wdrożeniowych oraz specyfikacji procesu biznesowego. Na przykład, niektóre systemy muszą podlegać dodatkowym obostrzeniom, takim jak walidacja, zanim trafią na środowisko produkcyjne. W takich okolicznościach, kluczowym jest możliwość dostosowania procesów oferowanych przez narzędzie, do specyficznych przypadków użycia klienta końcowego. Dotyczy to również możliwości dodawania własnej, niestandardowej logiki. Prototyp będący podmiotem tej pracy, swoją logikę opiera na koncepcie ciągów wykonawczych. Są kolejki zadań wykonujących pewne zadania, których wynikiem jest artefakt. Projekt zakłada opcję dodawania zewnętrznych zadań, poszerzając zakres możliwości oferowanych przez narzędzie. Wspomniana generyczność dotyczy również mechanizmu operacji na szablonach. W podstawowej implementacji, system wspiera modele oparte na technologii Mustache (Rozdział 5.1.4 Obsługa Szablonów). Jednak architektura systemu pozwala na dodanie wsparcia dodatkowych technologii szablonowych.

4.2 Specyfikacja wymagań systemowych

Analiza problematyki pracy pozwoliła na stworzenie listy wymagań stawianych nowej platformie. Poniżej została przedstawiona lista wymogów wyszczególnionych dla platformy automatyzującej początkowe fragmenty procesu tworzenia oprogramowania.

4.2.1 System musi wspierać logowanie, autoryzacje i autentykację użytkowników

Żeby obronić aplikację przed nadmiernym ruchem i wysokim wykorzystaniem zasobów, wymagana powinna być autentykacja. Ponadto, logowanie umożliwia monitorowanie wykorzystania funkcji biznesowych. To wymaganie jest także niezbędne, żeby spełnić wymaganie nr 2 i umożliwi identyfikację zasobów przynależnych do użytkownika.

4.2.2 System musi pozwalać na identyfikację zasobów po użytkownikach

Użytkownicy z korporacji lub firm posiadających dane tajne lub poufne mogą nie mieć przyzwolenia do dzielenia się danymi o generowanych zasobach nawet pomiędzy członkami tej samej jednostki organizacyjnej. W tym przypadku, logowanie rozwiązuje ten problem poprzez powiązanie zasobu z danym użytkownikiem. Niektóre zasoby powinny być dzielone pomiędzy użytkownikami, przez co wymagane jest także usługa współdzielenia zasobów.

4.2.3 System musi umożliwiać kreowanie nowych zasobów

W tym punkcie wymogiem także staje się umożliwienie użytkownikowi tworzenie zasobów w sposób prosty i przejrzysty. Przykładem takiej implementacji może być interfejs graficzny widziany w *start.spring.io* czy *jHipster*. Graficzne interfejsy oferują zdecydowanie niższy próg wejścia w porównaniu z rozwiązaniami opartych na terminalu. Interfejs graficzny powinien być realizowany poprzez wspomnianą wcześniej aplikację webową z dedykowaną aplikacją *front-end'ową*.

4.2.4 System musi mieć możliwość generowania kodu na podstawie szablonów

Jednym z problemów wskazywanych w istniejących systemach *LCDP* i podobnych był brak możliwości personalizacji procesu. Generycznym rozwiązaniem tego problemu jest umożliwienie użytkownikowi tworzenia własnych szablonów. System powinien posiadać możliwość procesowania tych szablonów tworząc z nich fragmenty aplikacji.

4.2.5 System musi umożliwiać przechowywanie szablonów

System powinien mieć możliwość przechowywania szablonów w repozytorium szablonów. Repozytorium powinno być dostępne do przeglądania dla użytkowników oraz pozwalać na wersjonowanie szablonów.

4.2.6 System musi umożliwiać dzielenie się zasobami

System powinien wspierać między projektową wymianę zasobów. Współdzielone szablony, definicje projektów i inne materiały muszą być w prosty sposób dostępne dla uprawnionych użytkowników. Dzięki mechanizmom promującym kolaborację projektową, wiele elementów wspólnych nie będzie musiało być tworzonych od zera. Przyśpieszy to ogólny proces tworzenia oprogramowania, zmniejszy ilość potrzebnych zasobów oraz wesprze standaryzację rozwiązań.

4.2.7 System musi być rozszerzalny

Nie jest możliwym, aby na etapie projektowania systemu wspierającego proces wytwarzania oprogramowania, móc przewidzieć wszystkie potrzeby klientów końcowych. W takim razie system musi być zaprojektowany w taki sposób, aby można było do niego dodać własne moduły. Elementy podlegające personalizacji to na przykład: standard wykorzystywany przy interpretacji szablonów lub zadania wewnątrz ciągów wykonawczych.

4.2.8 System musi posiadać środowisko ułatwiające wdrożenie aplikacji

Większość ewaluowanych systemów i platform posiada ekosystem ułatwiający proces wdrożeń. Ponieważ istnieje wiele sposobów wdrażania aplikacji oraz równie dużo celów wdrożeniowych (Cloud Deployment, Containers, GitLab, GitHub Triggers, Travis-CI etc..) istnieje potrzeba stworzenia systemu generycznego, łatwo rozszerzalnego o nowe funkcjonalności.

4.2.9 Integracja systemu w ekosystem firmowy powinien być bardzo prosty

Budowanie i instalacja systemu powinna być bardzo prosta i nie wymagać żadnej dodatkowej konfiguracji środowisk. Dotyczy to przede wszystkim konfiguracji wymagających integracji z systemami zewnętrznymi oraz zależności do innych systemów.

5. Zastosowane technologie

Poniższy rozdział zawiera opis wymagań stawianych przed nową platformą typu *Low-Code*, oraz opis projektu sugerowanego rozwiązania. Na podstawie zdefiniowanych wymagań zostały wybrane technologie realizujące postawione wyzwania. W dzisiejszym świecie technologicznym istnieje bardzo szeroka gama narzędzi i technologii, które umożliwiają skuteczną i stabilną implementacje systemu.

5.1 Aplikacja serwerowa

Z wymagań wiadomym jest, że Z*oran.io* musi być aplikacją webową i w tym celu został wykorzystany język generalnego przeznaczenia - *Java*. Java [24] jest bardzo popularnym językiem współbieżnym o charakterystyce obiektowej. Według rankingu StackOverflow Survey 2019, [25] Java uzyskała wynik 39.2%, będąc piątą najbardziej popularną technologią wśród profesjonalnych programistów. Swoją popularność zawdzięcza bardzo bogatemu API, szerokiemu wachlarzowi potężnych narzędzi deweloperskich a także swojej prostocie. Dzisiaj, aplikacje Java'owowe napędzają ponad 15 miliardów urządzeń co jednoznacznie podkreśla popularność tego języka. Rozwiązania bazujące na Java, dzięki możliwości kompilacji do kodu bajtowego ("*write once, run everywhere*"), można znaleźć w każdym ekosystemie korporacyjnym czy nawet jako język popularnego systemu operacyjnego Android.

W kontekście projektu, Java jest bardzo dobrym kandydatem do bycia językiem implementującym aplikacje serwerową (*backend*). Dzięki swojej popularności, posiada bardzo bogatą i różnorodną społeczność deweloperską, tworzącą niewiarygodną ilość bibliotek licencjonowanych w trybie '*Open Source*' [15]. Dostępność rozszerzeń umożliwia szybką implementację popularnych problemów i mechanizmów, jak na przykład implementację systemów szablonowych. Łatwość w użytkowaniu, bogate API i szeroka społeczność są czynnikami motywującymi wykorzystanie języka Java w projekcie, w stabilnej wersji Java 8 (1.8.152).

Funkcjonalność Java'y komplementuje struktura Pivotal Spring, bazująca na wzorcach projektowych [26]: odwrócenie sterowania (*Inversion of Control IOC*) i jej implementacji: wstrzykiwania zależności (*Dependency Injection*). Są to ważne wzorce mające na celu poradzenie sobie z narastającą ilością zależności pomiędzy komponentami w aplikacji. Jest to poważny problem, zwiększający ryzyko błędów regresyjnych poprzez potrzebę wprowadzania zmian we wszystkich komponentach zależnych. Ponadto znacznie utrudnia testowanie aplikacji. Wraz ze zwiększającym się rozmiarem aplikacji, problem powiązanych zależności się pogłębia. Powoduje to wzrost kosztów

wprowadzenia zmian, oraz wydłużenie czasu TTM (Time-to-Market), czyli czasu pomiędzy rozpoczęciem prac a ich efektem na środowisku produkcyjnym. Obie sytuacje są równoznaczne z utratą środków, czasu i zasobów, co jest wielce niepożądane w systemach typu '*enterprise'*. Poniżej, znajdują się definicje wspomnianych wcześniej wzorców projektowych.

Inversion of Control (IOC) jest wzorcem mającym na celu redukcję zależności pomiędzy komponentami poprzez zmianę kierunku kontroli. W tradycyjnym rozwiązaniu to deweloper ma całkowitą kontrolę nad systemem oraz nad tym, kiedy jego kod zostanie wywołany. W paradygmacie odwrócenia sterowania, to kontener IOC jest odpowiedzialny za inicjalizację komponentów i wywołanie kodu dewelopera.

Dependency Injection (DI) jest wzorcem zakładającym istnienie serwisu zajmującym się uzupełnianiem zależności w instancjonowanych obiektach. W momencie zdefiniowania potrzebnych zależności serwis automatycznie wstrzykuje potrzebne zasoby. Są one identyfikowane po typie lub identyfikatorze i następnie przekazane odpowiednim obiektom. W zależności od języka programowania wspierającego ten wzorzec, istnieje kilka metod wstrzykiwania zależności. W Java'ie, rozszerzenie Spring oferuje wstrzykiwanie zależności poprzez konstruktor, *setter* lub zewnętrzną konfigurację w pliku XML.

Poprawne wykorzystanie powyższych wzorców projektowych pomaga poprawić modularyzację projektu, ułatwiając jego testowanie, rozszerzalność, testowanie funkcjonalności oraz zarządzanie kodem.

Spring to nie tylko implementacja wspomnianych wyżej wzorców projektowych, ale także szkielet aplikacyjny umożliwiający zespołom deweloperskim skupienie się na implementacji logiki biznesowej. Spring oferuje abstrakcje nad typowymi aspektami aplikacji opartych na *Java Enterprise Edition (JEE)*, dzięki czemu, logika biznesowa pozostaje bez powiązania do konkretnej implementacji wdrożeniowej. Spring Boot jest dodatkowym zbiorem funkcjonalności, usprawniającym proces tworzenia aplikacji Spring. Głównymi dodatkami są: funkcjonalność auto konfiguracji komponentów aplikacyjnych oraz zintegrowany serwer aplikacyjny *Apache Tomcat*, dzięki któremu treść aplikacji może być serwowana przez Internet.

5.1.1 Apache Maven

Apache Maven jest systemem umożliwiającym łatwe zarządzanie oprogramowaniem i zasobami w nim zawartymi. Oparty o model POM (Project Object Model), Apache Maven umożliwia trywialne zarządzanie zewnętrznymi bibliotekami, systemem budowania aplikacji oraz dokumentowania jakości systemu z centralnego pliku z właściwościami.

Wprowadzenie Apache Maven w ekosystem aplikacji, zapewnia warstwę abstrakcji nad skomplikowane procesy kompilacji i wdrożenia systemu a także gwarantuje jednolitą metodę zarządzania produkcją oprogramowania. Jest to rozwiązanie bardzo popularne, oferujące pełną gammę rozszerzeń i wtyczek, wzbogacających procesy o spersonalizowanie zachowań jak na przykład wtłaczanie danych *git* jako zmienne w projekcie.

5.1.2 YML

YML lub YAML, jest uniwersalnym standardem zapisywania treści i informacji w formalny i ustrukturyzowany sposób. Głównym celem tego standardu jest jasny i czytelny dla człowieka opis różnych źródeł danych. Trywialność, a także niezależność od języków programowania, czynią go bardzo popularnym wyborem do przechowywania konfiguracji. W Spring Boot, o który oparty jest omawiany system, YAML jest wykorzystywany do opisywania zmiennych środowiskowych i konfiguracji systemu.

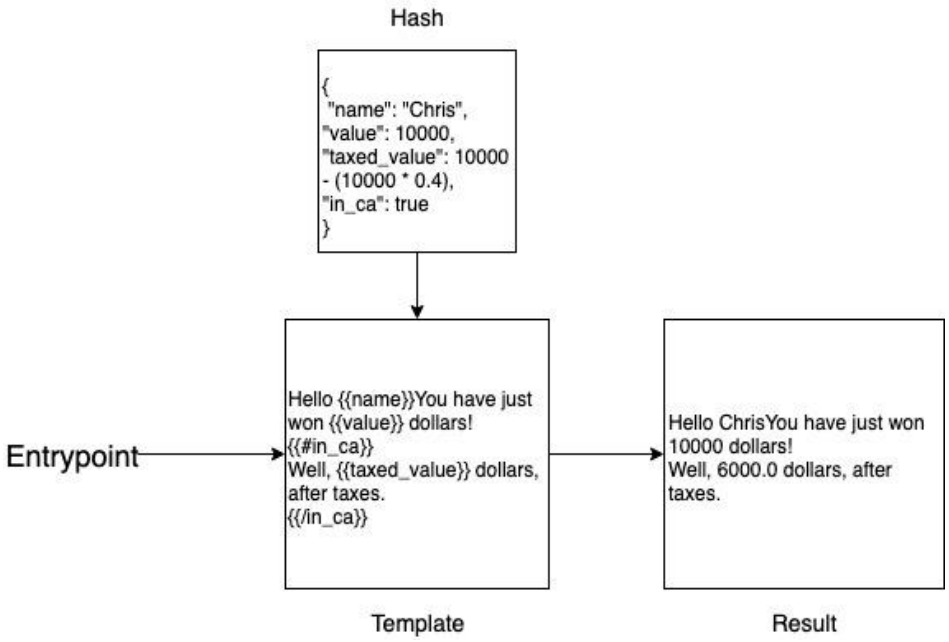
Łatwość w odczytywaniu informacji przez człowieka oraz wsparcie w programistycznym interpretowaniu plików YML, umożliwiły wykorzystanie tej technologii w aplikacji Zoran.io jako język opisujący obiekty typu Manifest. Koncept Manifestu, wytłumaczony jest w rozdziale poświęconym projektowi oraz implementacji systemu „5.2.2 Dane w serwisie GitHub”.

5.1.3 Git

Git jest rozproszonym systemem kontroli wersji, umożliwiającym śledzenie zmian w trakcie produkcji oprogramowania. Każda iteracja modyfikacji źródeł traktowana jest jako rewizja, będąca obrazem całego projektu. Umożliwia to rozproszoną i jednoczesną pracę wielu programistów, będących lub nie będących, podłączonymi do sieci, w której hostowane jest zdalne repozytorium. System będący tematem tej pracy został zbudowany wykorzystując system kontroli wersji git. Ponadto, wraz z aplikacją serwerową pełniącą rolę git serwera - Github.com, git został wykorzystany do wersjonowania i przetrzymywania szablonów stworzonych przez użytkowników systemu. Opis tej funkcjonalności znajduję się w rozdziale „5.3.3 Silnik Szablonowy”.

5.1.4 Obsługa szablonów

Serwis generowania szablonów powinien oferować abstrakcję nad procesem generowania kodu, umożliwiając implementacje własnego rozwiązania do procesowania szablonów. W wersji prototypu, system implementuje szablony w standardzie *Mustache* [27]. *Mustache* jest rozwiązaniem oferującym prosty sposób na generowanie plików - konfiguracyjnych, HTML, klas, bezpośrednio z plików szablonów.



Rysunek 3 Proces tworzenia dokumentu z szablonu z wykorzystaniem Hash; Opracowanie własne

Rysunek 3 przedstawia schemat działania szablonów opartych o Mustache. Każdy plik szablonowy zawiera pola, które muszą zostać wypełnione podczas instancjonowania pliku. Konfiguracja takiego pliku odbywa się przy użyciu dodatkowego dokumentu zawierającego listę klucz-wartość, zwanego „*hash*”. Silnik czytając klucze z *hash* dopasowuje je do odpowiednich pól w szablonie, wstawiając wartość odpowiadającą kluczowi. Tym sposobem, z szablonu tworzona jest nowa instancja dokumentu skonfigurowana poprzez zewnętrzny plik - *hash*.

5.1.5 Identity Access Management

Identity Access Management jest informatycznym systemem zaufanym, zapewniającym metodę autentykacji i autoryzacji użytkowników. Istnieje wiele zewnętrznych dostawców oferujących taką usługę. W kontekście projektu, bardzo ważne jest, żeby był to dostawca ogólnodostępny, czyli taki, który oferuje możliwość rejestracji każdemu użytkownikowi. Metodą wykorzystaną w projekcie jest standard OAuth 2.0 (RFC 6749) [28]. OAuth 2.0 jest szeroko stosowanym wzorcem wywodzącym się z mediów społecznościowych, umożliwiającym użytkownikom uzyskanie autoryzacji w zewnętrznym serwisie przez HTTP. Specyfikacja adresuje problem przetrzymywania loginu i hasła użytkownika przez zewnętrzne aplikacje w postaci czystego tekstu oraz niwelują słabość związaną z autoryzacją z wykorzystaniem haseł. Dostęp do zewnętrznych zasobów jest gwarantowany przez system autoryzacyjny bez wymiany hasła użytkownika pomiędzy serwerem a klientem. Poziom dostępu jest ograniczany poprzez zastosowanie zakresów wbudowanych w *token*, który służy do autoryzacji oraz wymiany informacji pomiędzy klientem a serwerem.

W projekcie, IAM jest realizowany za pośrednictwem OAuth 2.0 z wykorzystaniem GitHub.com jako usługodawcy. GitHub.com jest webowym repozytorium kodu dla programistów i bardzo popularnym miejscem kolaboracji wśród społeczności informatycznej. Wybór tego portalu jest motywowany także faktem, iż system został wybrany jako wersjonowane repozytorium szablonów (Rozdział 5.1.6 Warstwa danych).

5.1.6 Warstwa danych

W projekcie rozróżniane są dwa rodzaje typów danych. Pierwszym są typowe dane aplikacyjne, takie jak definicje zasobów czy dane użytkowników. Są to dane, które mogą zostać przedstawione w postaci dokumentów i są bezpośrednio wykorzystywane przez system, poprzez poziom abstrakcji oferowany przez rozszerzenie Spring Boot Data. Dane te są przechowywane w bazie *noSQL* [29] - MongoDB.

MongoDB jest nierelacyjną bazą danych, opartej na koncepcji dokumentu. W MongoDB każda encja przedstawiona jest jako dokument JSON, posiadająca dynamiczną strukturę, która wraz z czasem i rozwojem systemu, może się zmieniać. Brak określonego początkowego schematu, powoduje, że jest do często wybierany silnik bazodanowy dla nowych projektów, gdzie ostateczny schemat nie jest jeszcze określony. Ponadto, jest to darmowe rozwiązanie, a przez to łatwo dostępne, z szeroką rzeszą fanów i kontrybutorów. Bogata i aktywna społeczność deweloperska, pozwala na szybkie znalezienie rozwiązań potencjalnych problemów na forach programistycznych.

Drugim typem danych są dane zawierające pakiety kodu szablonów. Różnica pomiędzy tymi dwoma typami danych polega na tym, że dane szablonowe powinny wspierać wersjonowanie a także umożliwić użytkownikom kontrybuowanie własnych szablonów. Dodatkowo różni się typ przechowywanych danych. Szablon z perspektywy projektu, jest odpowiednikiem folderu, zawierającym plik szablonu oraz plik konfiguracyjny - *hash*. Żeby spełnić powyższe wymagania, zastosowany został system wersjonowania git, wraz ze zdalnym repozytorium Github.com. Wraz z możliwością logowania oraz dodawania kodu do personalnych kont użytkowników, Github.com jest bardzo ważnym elementem integracyjnym.

5.2 Aplikacja prezentacyjna (front end)

Aplikacje webowe tworzone w języku Java, tradycyjnie wykorzystują technologie JSF/JSP (*Java Server Facelets/Java Server Pages*) do budowania widoków. Są to dość przestarzałe, skomplikowane i trudne w wykorzystaniu technologie odradzane przez społeczeństwo informatyczne [30]. Przykładem dojrzałej technologii, wykorzystywanej w budowie warstwy prezentacyjnej jest Angular. Powołując się na ankiety przeprowadzone przez portal *stateofjs.com* [31] prezentującej trendy technologiczne wokół języka skryptowego JavaScript, Angular plasuje się w czołówce najbardziej popularnych technologii frontend'owych.

Angular to platforma zaprojektowana przez Google jako implementacja wzorca *Model-Widok-Kontroler* (MVC). Jest bardzo powszechnie wykorzystywana jako fundament do budowy aplikacji webowych i mobilnych. Bazuje natywnie na języku JavaScript, jednak istnieje wiele pochodnych tej platformy, wykorzystującej inne języki programowania, na przykład TypeScript. W projekcie została wykorzystana wersja bazująca na języku Dart - AngularDart.

Dart jest językiem wywodzącym się z rodziny ALGOL, wraz z Java, C, C# i wieloma innymi. Jest to język obiektowy, ogólnego przeznaczenia, posiadający mechanizm czyszczenia pamięci (*Garbage Collector*), z opcjonalną możliwością kompilacji do JavaScript. Dart został stworzony przez firmę Google jako wieloplatformowy język wykorzystywany w programowaniu zorientowanym na budowaniu klienta, a także jako język do tworzenia aplikacji mobilnych wykorzystując AOT-Compiler (*Ahead-of-time*) oraz DartNative, kompilując kod Dart do natywnego maszynowego.

5.3 Systemy wdrożeniowe rozwiązania

Żeby zrozumieć dokładnie typy i mechanizmy wdrożeniowe, warto najpierw zaznajomić się z możliwą infrastrukturą aplikacyjną klienta. Aplikacje webowe funkcjonują w ramach infrastruktury: serwerów aplikacyjnych, baz danych, sieci i systemów wirtualizacyjnych, oraz integrują się z wieloma innymi systemami korporacyjnymi. W większości rozwiązań korporacyjnych, serwery aplikacyjne znajdują się w wewnętrznej sieci firmowej, działających na fizycznych serwerach należących i obsługiwanych przez daną firmę. Takie rozwiązanie zapewnia pracownikom fizyczny dostęp do maszyn hostujących oprogramowanie, a także powoduje, że osoby i systemy trzecie nie mają dostępu do, często poufnych lub krytycznych, danych. Takie rozwiązanie wymaga jednak dedykowanego zespołu techników zajmujących się utrzymaniem i konserwacją serwerów, oraz konfiguracją maszyn wirtualnych na potrzeby deweloperskie. W przypadku dużych korporacji, proces zamawiania i utrzymywania wielu środowisk aplikacyjnych, jest bardzo długi i potrafi spowolnić prace deweloperskie, a przez co, tracić zasoby. W związku z tym, powstała konkurencyjna metoda wdrażania aplikacji na infrastrukturę w chmurze.

Infrastruktura w chmurze (*cloud-based infrastructure*), jest pojęciem określającym korzystanie z zasobów infrastrukturalnych w formie serwisu. Poszczególne komponenty, które są wykorzystywane przez użytkownika, znajdują się w fizycznie innym miejscu a odpowiedzialność za jej utrzymanie, konserwację i zarządzanie spoczywa na usługodawcy. Z perspektywy zespołu deweloperskiego jest to bardzo wygodne rozwiązanie, redukujące czas oczekiwania na nową infrastrukturę, a także umożliwia dynamiczne skalowanie horyzontalne rozwiązania. W klasycznym podejściu, usługodawca takich rozwiązań tworzy poziom abstrakcji nad maszynami fizycznymi, powodując, że dla dewelopera nie jest widoczny fizyczny aspekt maszyny. Dostęp do środowisk jest możliwy poprzez interfejs programistyczny API, dostępny z poziomu terminala CLI lub poprzez interfejs graficzny. Rozwiązania bazujące na chmurze stają się bardzo popularne wśród rozwiązań o skali firmowej a także projektów

indywidualnych, ze względu na ich zerowy koszt początkowy i brak potrzeby manualnego zarządzania maszynami. Na rynku istnieje wiele usługodawców oferujących infrastrukturę jako serwis. Powołując się na raport Gartnera z 2018 [32], można wskazać liderów rozwiązań w chmurze należą następujący gracze: Amazon Web Services, Microsoft Azure i Google Cloud Platform.

Łatwo zauważyć, iż podejście wdrożenia w infrastrukturze chmurowej i na infrastrukturę lokalną mają zastosowanie w różnych sytuacjach i mogą być wykorzystywane jednocześnie w ramach jednego ekosystemu. Jednak taka sytuacja, powoduje problem z brakiem standardu wdrożeń. Inny będzie proces wdrażania aplikacji na lokalny maszyny wirtualne, a inny na wdrażanie aplikacji w chmurę. To ograniczenie dotyka również systemu, którego dotyczy poniższa praca. Rozwiązaniem tego problemu są kontenery aplikacyjne.

Kontener [33] jest wyizolowaną jednostką, posiadająca zapakowany kod aplikacyjny wraz z niezbędnymi zasobami i zależnościami potrzebnymi do poprawnego funkcjonowania. Powoduje to, że każdy kontener może być traktowany jako niezależny i w pełni samodzielny proces. Wyizolowany czas życia programu, powoduje, że aplikacja będzie działać identycznie, niezależnie od infrastruktury, na której działa kontener. Ta właściwość powoduje, że konteneryzacja aplikacji staje się standardem w mieszanym środowisku infrastrukturalnym. Wiodącą technologią konteneryzacyjną jest platforma Docker. Pozwala ona tworzyć obrazy, czyli definicje kontenerów, z których są tworzone instancje uruchomione wewnątrz Docker Deamon. Definicje obrazów tworzy się poprzez pliki Dockerfile, w których określa się zasoby potrzebne danemu kontenerowi do działania, jego przynależność w sieci, a także wolumeny danych. W implementacji projektu został wykorzystany komponent Docker Compose, będącym rozszerzeniem samej platformy Docker, umożliwiającej definiowanie systemów wielokontenerowych. Dzięki temu użytkownicy mogą korzystać z systemu wykorzystując tylko i wyłącznie obraz Docker, bez potrzeby specjalnej konfiguracji środowiska pracy. Rozwiązanie jest na tyle proste, że niezależnie czy jest to infrastruktura typu *on-premise* czy infrastruktura w chmurze, jedyne wymaganie jest, żeby dane środowisko wspierało Docker Engine, czyli platformę kontenerową.

5.3.1 Travis-CI

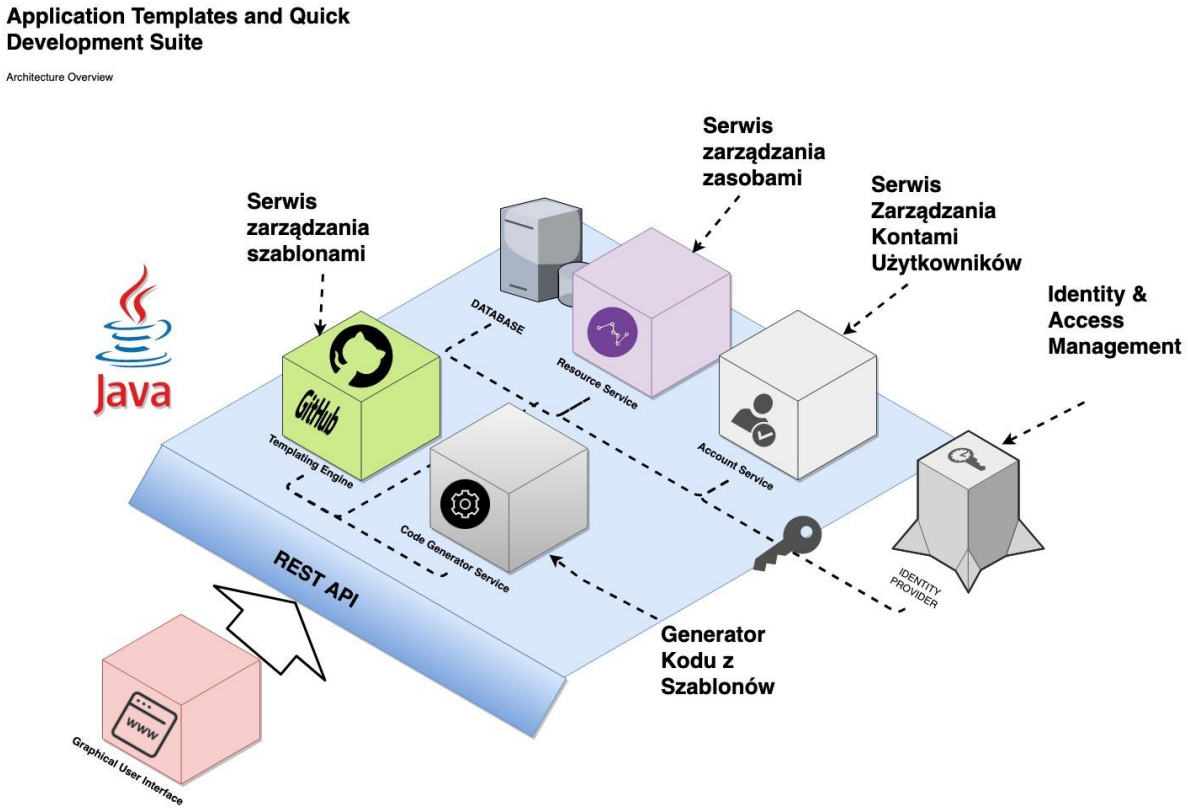
Travis-CI [34] jest projektem umożliwiającym klientom, prosty w tworzeniu i utrzymaniu, system zarządzania procesami wdrożeniowymi aplikacji. Projekty mogą być zdalnie budowane i testowanew sposób automatyczny, przenosząc odpowiedzialność z programisty na zewnętrzną usługę. Dzięki temu, developerzy mogą się skupić na tworzeniu i dostarczaniu funkcjonalności biznesowej, pozostawiając procesy operacyjne systemowi Travis-CI. W opisywanym projekcie, Travis-CI został zastosowany jako domyślna platforma wspierająca proces wdrożeniowy, budując i testując aplikację, a raport z wykonania tych działań, wysyłany jest poprzez e-mail do programisty.

6. Projekt i implementacja rozwiązania

W poniższym rozdziale przedstawiona zostanie propozycja implementacji prototypu spełniającego wymagania przedstawione w rozdziale 4.2. Poniższa sekcja jest podzielona na podrozdziały, prezentujące wysokopoziomową architekturę rozwiązania, projekt warstwy danych, implementacje aplikacji serwerowej, a także aplikacji webowej.

6.1 Wysokopoziomowa architektura rozwiązania

Wynikiem analizy wstępnych wymagań jest wysokopoziomowa architektura przedstawiona na rysunku 4. Aplikacja opiera się o dwupoziomową architekturę aplikacji webowej, składającej się z dwóch oddzielnych aplikacji: frontend'owej, realizującej wymagania dotyczące interfejsów graficznych i warstwy prezentacyjnej oraz backend'owej (serwerowej), implementującej warstwy logiki biznesowej, integracji i połączeń z bazą danych. Wymóg dotyczący autoryzacji i bezpieczeństwa aplikacji realizowany jest przy pomocy zewnętrznej usługi autoryzacyjnej. Poniższy rozdział zawiera krótkie podsumowanie specyfikacji każdego z wyodrębnionych elementów.



Rysunek 4 Wysokopoziomowa architektura aplikacji Zoran.io; Opracowanie własne

6.1.1 Serwis zarządzania szablonami

Serwis zarządzania szablonami jest interfejsem komunikacyjnym pomiędzy logiką generowania kodu a repozytoriami szablonów. Jego rolą jest poprawne zarządzanie dostępem do plików, poprawne indeksowanie danych a także realizowanie operacji pobierania szablonów.

6.1.2 Serwis zarządzania zasobami

Zasobem w kontekście omawianego systemu, nazywamy definicje nowych projektów generowanych przy pomocy Zoran.io. Każdy zasób zawiera pełny schemat konfiguracji potrzebnej do stworzenia projektu a także jest przypisany do danego użytkownika i udostępniony zdefiniowanej wcześniej grupie. Serwis zarządzania zasobami, umożliwia tworzenie, usuwanie i modyfikację zasobów a także zarządza dostępem do nich.

6.1.3 Serwis zarządzania użytkownikami

Serwis zarządzania użytkownikami realizuje wymagania dotyczące bezpieczeństwa aplikacji i przechowywanych przez nią zasobów poprzez autoryzowanie użytkowników z zewnętrznym Identity & Access Manager (IAM), czyli systemem oferującym zewnętrzną autoryzację.

6.1.4 Identity & Access Manager

Wspomniana wcześniej zewnętrzna usługa autoryzacji.

6.1.5 Generator Kodu z Szablonów

Serwis generujący gotowe szkielety aplikacji wykorzystując definicje zasobów pobranych z Serwisu Zarządzania Zasobami. Przeprocesowane zasoby są następnie przygotowywane do wdrożenia.

6.2 Przygotowanie warstwy danych

W poniższym rozdziale przedstawiony zostanie proces instalacji i dodawania bazy danych MongoDB oraz integracja repozytoriów wersjonowanych git w projekcie.

6.2.1 MongoDB

Warstwa danych biznesowych w projekcie realizowana jest na podstawie, opisanej wcześniej, nierelacyjnej bazy danych MongoDB. Sama baza działa niezależnie od aplikacji i nie jest to wersja osadzona w pamięci podręcznej aplikacji (*Main Memory Database* - MMDB). Nie jest zalecane, aby takie bazy wykorzystywane były w wersjach produkcyjnych aplikacji, gdyż ich zawartość jest ulotna. Wraz z restartem aplikacji lub maszyny, fizyczna pamięć podręczna jest usuwana a wraz z nią, osadzona

w niej zawartość bazy danych. Jest to świetne rozwiązanie w celach testów automatycznych, gdyż jest bardzo wydajna i lekka. W opisywanym projekcie, wersja działająca w pamięci podręcznej została wykorzystana właśnie w testach integracyjnych.

6.2.1.1 Instalacja i konfiguracja MongoDB

Jak zostało to wspomniane w poprzednich rozdziałach, pakiet instalacyjny bazy danych MongoDB jest całkowicie darmowy. Oznacza to, że wystarczy pobrać źródła i baza danych jest gotowa do działania. Pakiet MongoDB jest dystrybuowany wraz z serwerem MongoDB, umożliwiającym zdalną komunikację z bazą.

W projekcie wykorzystana została wersja w postaci obrazu Docker'owego 'mongo' pobranym z ogólnodostępnego repozytorium hub.docker.com. Baza danych uruchamia się w postaci kontenera, przy użyciu następującej komendy w terminalu:

```
$ docker run -p 27017:27017 mongo:latest
```

Polecenie „*docker"* wywołuje serwis Docker Deamon, „*run"* oznacza uruchomienie kontenera z obrazu zwany „*mongo"*, oznaczonego tagiem „*latest"*. Ponieważ kontenery uruchamiane są w wyizolowanym środowisku, wymagane jest wyeksponowanie, czyli zmapowanie niezbędnych portów wewnątrz kontenera na porty hosta. W tym celu wykorzystujemy flagę „*-p*", i jako parametr przekazujemy lokalny port 27017, który będzie się mapował na port 27017 wewnątrz kontenera. Port 27017 jest domyślnym portem, pod którym działa baza danych MongoDB.

Taka konfiguracja portów, umożliwia odpytywanie bazy danych z zewnątrz kontenera, przez aplikacje, a także zewnętrznego klienta graficznego, na przykład Robo 3T.

6.2.1.2 Integracja bazy danych z aplikacja

Integracja z bazą danych odbywa się poprzez warstwę abstrakcji dostarczanej przez platformę programistyczną Spring Boot Data MongoDB. Ponieważ projekt korzysta z Apache Maven, wsytarczy zadeklarować zależność dla pakietów Spring Data w pliku budującym `pom.xml`:

```
53          <dependency>
54              <groupId>org.springframework.boot</groupId>
55              <artifactId>spring-boot-starter-data-mongodb</artifactId>
56          </dependency>
```

Rysunek 5 Definicja zależności na SpringBoot Data w pliku budującym Maven; Opracowanie własne

Głównym punktem wejścia abstrakcji bazodanowej z poziomu kodu aplikacji jest interfejs `Repository`, umożliwiający tworzenie własnych kolekcji dokumentów. Interfejs przechwytuje typ

obiektu, z którym użytkownik ma zamiar pracować oraz pozwala na implementację funkcjonalności CRUD dla danej kolekcji poprzez wykorzystanie interfejsów dziedziczących po interfejsie `Repository`. Ponieważ projekt bazuje na bazie nierelacyjnej MongoDB wykorzystany został interfejs dedykowany architekturze bazy danych systemów Mongo, `MongoRepository`.

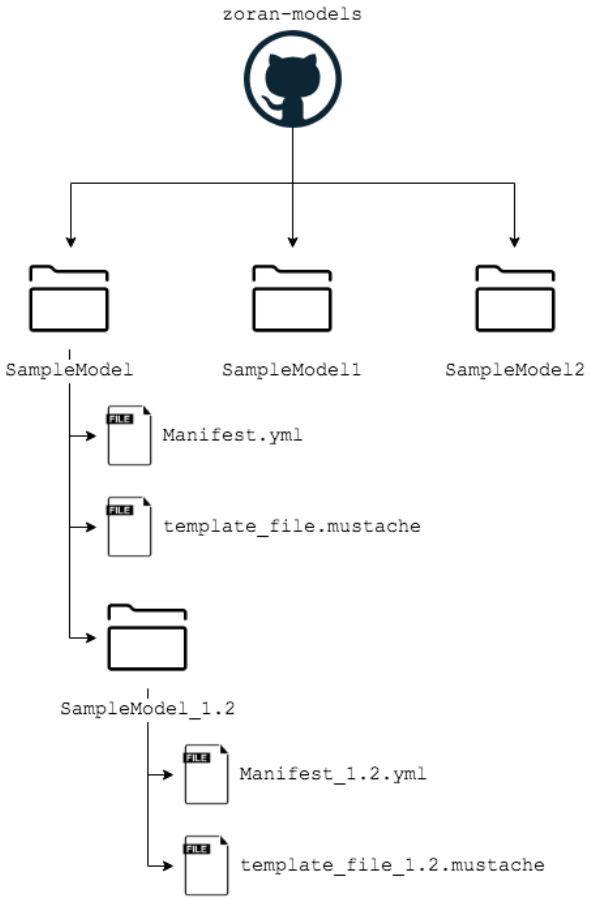
Połączenie bazy danych do aplikacji serwerowej odbywa się w pełni automatycznie wykorzystując funkcjonalność auto konfiguracji SpringBoot. Klasa konfigurująca `ZoranCoreConfiguration` została wzbogacona o adnotację `@EnableMongoRepositories("io.zoran")`, do której jako parametr została przekazana ścieżka do auto skanowania ścieżki zawierające zadeklarowane repozytoria. Parametry połączeń do bazy danych są przekazywane w momencie uruchamiania aplikacji jako zmienne środowiskowe:

```
SPRING_DATA_MONGODB_URI=mongodb://mongodb:27017/local
```

Ponieważ baza danych pracuje w kontenerze, którego serwis nazywa się „*mongodb*” w zmiennej przekazane jest URI zawierające identyfikator kontenera. W wersji demonstracyjnej aplikacji, dostęp do danych jest nieograniczony oraz nie chroniony hasłem. Takie rozwiązanie jest niedopuszczalne w wersji produkcyjnej, opis możliwego rozwiązania tego problemu jest opisany w rozdziale “7.2.1 Zarządzanie Sekretami”.

6.2.2 Dane w serwisie GitHub

Oprócz standardowej warstwy danych znajdujących się w systemie DBMS, aplikacja integruję się z zewnętrznym portalem GitHub.com, gdzie składowane są wersjonowane pliki szablonów. Struktura danych przedstawiona jest na rysunku 6.



Rysunek 6 Model Danych Szablonów w serwisie GitHub; Opracowanie własne

Na rysunku 6 pojawiają się obiekty typu *Manifest* i *Template* (szablon). Są to kluczowe pojęcia, których zrozumienie jest wymagane w celu poprawnej konfiguracji silnika szablonowego. Definicje tych pojęć dostępne są poniżej:

- *Szablon*, w kontekście systemu jest plikiem szablonowym w technologii Mustache. Jest częścią modelu.

- *Manifest* jest plikiem konfiguracyjnym szablonu zawierającym informacje zdefiniowane przez autora szablonu. Elementami manifestu są informacje dla użytkownika końcowego, takie jak nazwa, opis i wersja, a także informacje techniczne, takie jak zależności, spis plików szablonów wraz z ich *hashem*. Rysunek 7 przedstawia przykładowy manifest. Manifest jest elementem modelu.

```
name: WebSecurityConfigurer
lead: Template that configures basic websecurity
version: 0.0.1
owner: FakeName FakeSurname
path: /SampleModel
visibility: PUBLIC
type: CLASS
dependencies:
- lombok
template:
  - name: WebSecurityConfigurer
    filename: WebSecurityConfigurer.java.mustache
    preferredLocation: INFRASTRUCTURE
    context:
      - name: package_name
        value:
      - name: file_name
```

Rysunek 7 Przykład definicji manifestu dla szablonu klasy WebSecurityConfigurer*; Opracowanie własne*

- *Model*, jest katalogiem zawierającym pliki szablonowe, plik manifest oraz potencjalnie inne modele.

Struktura modelu danych w sposób naturalny mapuję się na graf acykliczny, gdzie węzłami są modele. Wykorzystanie danego węzła przy tworzeniu nowego zasobu, powoduje, iż wszystkie węzły przodki, czyli te występujące na ścieżce przed wybranym węzłem, zostają również zaimportowane. Ten model pozwala w prostu sposób modelować zależności pomiędzy modelami. W przypadku, gdy model jest zależny od modelu nie znajdującego się w sposób naturalny na ścieżce, istnieje możliwość jego zaimportowania poprzez zdefiniowanie zależności w pliku *manifest*.

6.3 Aplikacja serwerowa

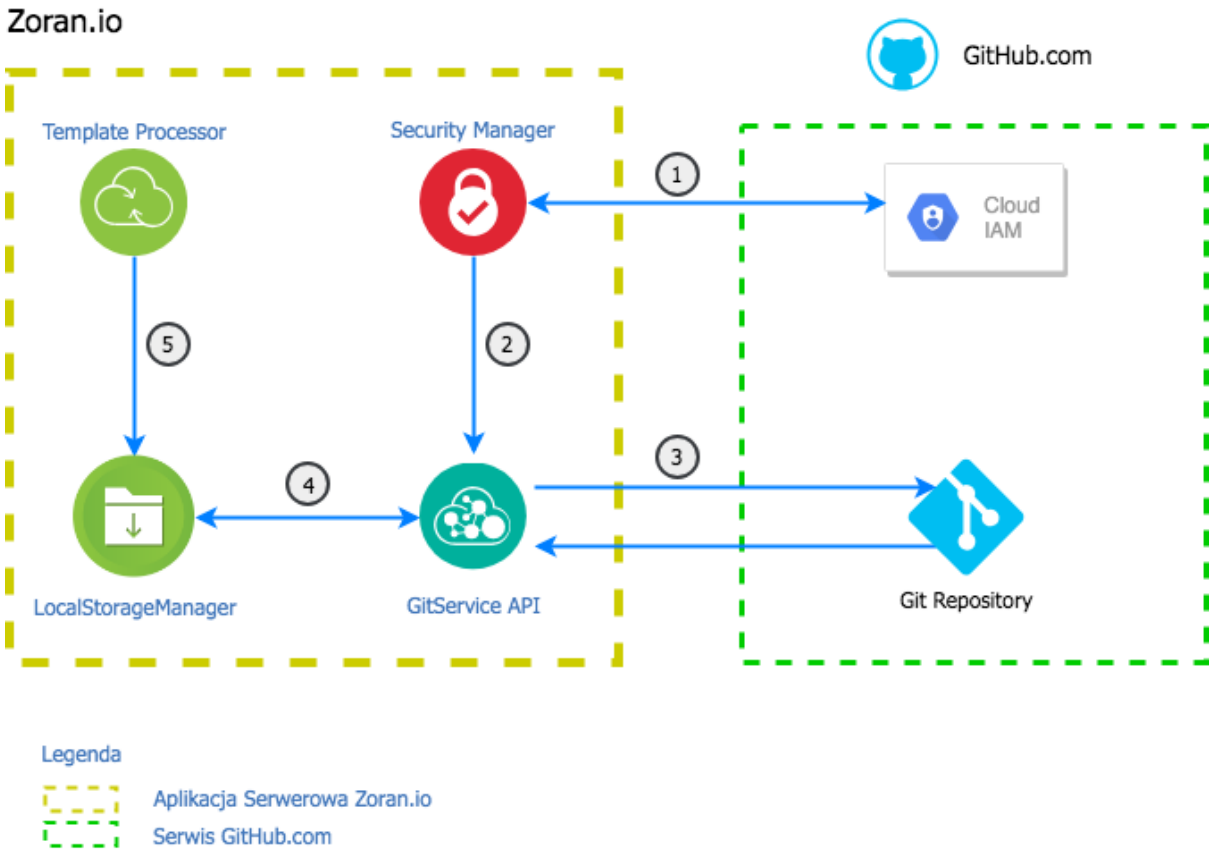
Poniższy rozdział zawiera opis poszczególnych, kluczowych elementów aplikacji serwerowa. Cała aplikacja serwerowa zawiera około 7 tysięcy linijek kodu oraz około tysiąc linijek kodu testów opisujących około 30% krytycznych funkcjonalności aplikacji.

6.3.1 Integracja z serwisem GitHub.com

Poniższy rozdział zawiera opis architektury modułu integracyjnego z zewnętrzną usługą GitHub.com. Serwis udostępnia także system Identity Access Management, wykorzystany do autoryzacji i autentykacji użytkowników wykorzystując protokół OAuth 2.0 („Rozdział 5.1.5 Identity Access Management"). Opis funkcjonalności szablonów opisany jest dokładnie w rozdziale „5.3.3 Silnik Szablonowy", opis funkcjonalności bezpieczeństwa i zarządzania użytkownikami opisany jest w rozdziale „5.4 Identity Access Management".

6.3.1.1 Architektura modułu integracyjnego

Architektura modułu integracyjnego przedstawiona jest na rysunku 8.



Rysunek 8 Komunikacja pomiędzy serwisami; Opracowanie własne

Przepływ logiki aplikacyjnej oznaczony jest na Rysunku 8 numerami 1-5

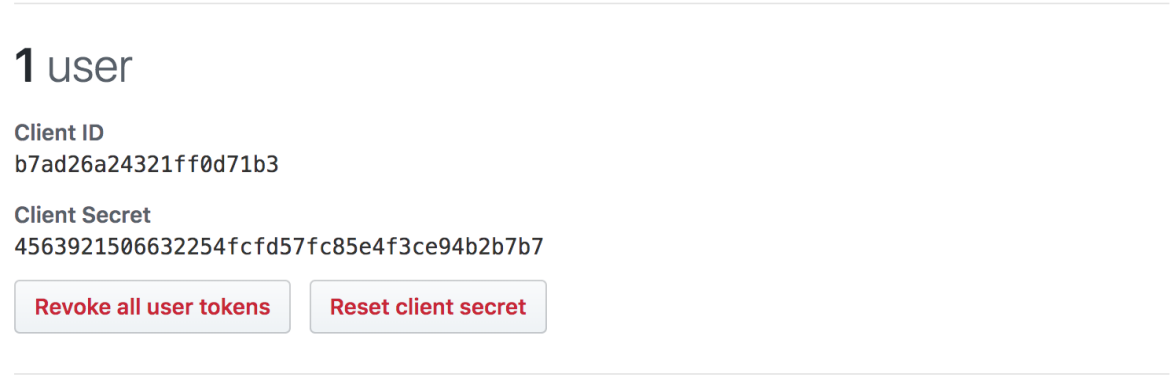
1. Autoryzacja aplikacji Zoran.io z serwisem GitHub.com.
2. Przekazanie otrzymanego klucza API do serwisu `GitService`
3. Sklonowanie repozytorium zawierającego dane szablonów.
4. Zapis danych do repozytorium lokalnego w pamięci trwałej systemu.
5. Dane repozytorium mogą zostać wykorzystane przez procesor modeli szablonów.

Szczegółowy opis podprocesów znajduję się w rozdziale “5.3.1.2 Opis procesu integracji”.

6.3.1.2 Opis procesu integracji

Integracja odbywa się z wykorzystaniem klucza do API serwisu GitHub.com, wygenerowanym w panelu zarządzania. W tym celu, założone zostało techniczne konto, w którym została zarejestrowana aplikacja, jako „*OAuth App*”. Umożliwia to wygenerowanie użytkownika GitHub API, tworząc dwa

klucze: `client_id` i `client_secret`. Rysunek 9 przedstawia widok opisanych danych w konsoli zarządzania GitHub.



Rysunek 9 Wygenerowane klucze dostępu aplikacji w serwisie GitHub; Opracowanie własne

ID oraz sekret klienta są dodane w postaci ciągów znaków do pliku z właściwościami aplikacji. Podobnie jak w przypadku bazy danych, jest to rozwiązanie tylko i wyłącznie dopuszczalne w wersji demonstracyjnej. Kod źródłowy aplikacji jest przechowywany w publicznym repozytorium, więc sekret jest widoczny dla każdego użytkownika. Opis rozwiązania tego problemu opisany jest w rozdziale "7.2.1 Zarządzanie Sekretami".

Ponieważ autentykacja aplikacji z GitHub oparta jest o protokół OAuth 2.0, dla uproszczenia może zostać wykorzystane rozszerzenie struktury Spring - Spring Security. Wymagane moduły są konfigurowane wykorzystując zasadę auto konfiguracji zależności. W tym celu, klucze pobrane z serwisu *GitHub.com*, dodane są do pliku z właściwościami aplikacji. W trakcie startu aplikacji, będą one automatycznie przeczytane i odpowiedni zestaw obiektów implementujących proces integracji zostanie stworzony. Rysunek 10 przedstawia zrzut ekranu z edytora kodu, prezentując zawartość pliku.

```
1  spring:
2    security:
3      oauth2:
4        client:
5          registration:
6            github:
7              client-id: 2807563e28e21e5e1494
8              client-secret: 8f81db99cb88a5f37c40a76589a269ff1b4b7a92
9              scope: repo
10             model_git_url: https://github.com/zoran-core/zoran-models.git
```

Rysunek 10 Konfiguracja wartości połączenia z serwisem GitHub; Opracowanie własne

Interfejs integracyjny z serwisem GitHub obsłużony jest przez rekomendowaną bibliotekę Java, `org.eclipse.egit.github.core`, będącą implementacją REST API serwisu GitHub w wersji 3. Pozwala to na wykorzystywanie zasobów zewnętrznego serwisu, wykorzystując obiekty Java'owe,

bez potrzeby manualnego tworzenia klienta HTTP i odpytywania dostępnych punktów dostępu. Redukuje to poziom skomplikowania aplikacji i zmniejsza ilość potencjalnych błędów, związanych z nieprawidłową konfiguracją klienta HTTP. Biblioteka jest wykorzystywana w następujących procesach:

1. klonowania zdalnego repozytorium do repozytorium lokalnego,
2. wykonywania operacji git: COMMIT i PUSH,
3. tworzenia nowych, zdalnych repozytoriów.

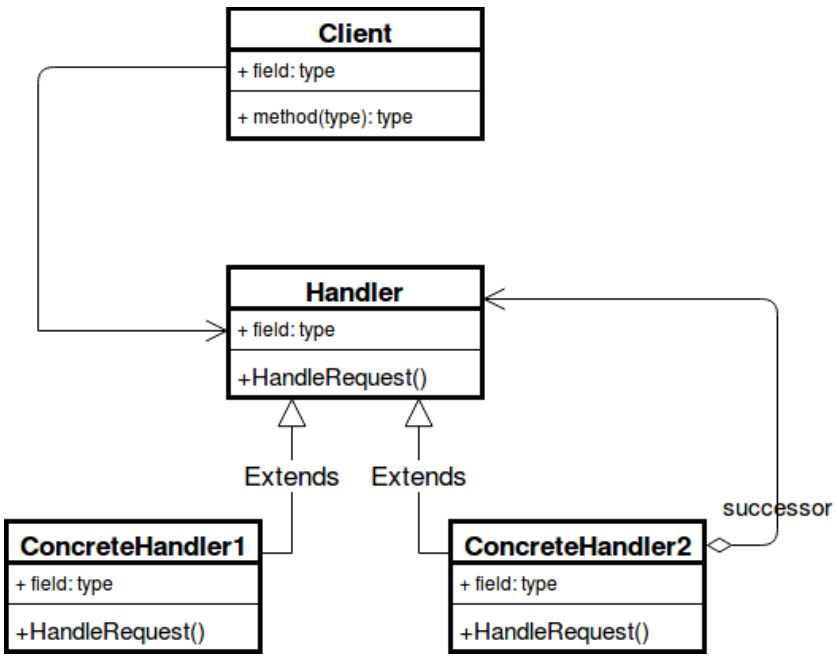
Wymienione funkcjonalności systemu wersjonowania git są wykorzystywane przez silnik zarządzania szablonami. Szczegóły implementacyjne są opisane w rozdziale “5.3.2 Silnik Szablonowy”.

6.3.2 Procesowanie strumieniowe ciągów zadań

Poniższy rozdział zawiera opis implementacji silnika do procesowania zadań w postaci ciągów wykonawczych. Jest to funkcjonalność umożliwiająca tworzenie oraz łączenie zadań wykonywanych sekwencyjnie.

6.3.2.1 Model strumieniowy

Model strumieniowy jest oparty o implementacje wzorca projektowego *Chain of Responsibility*. Poszczególne, konkretne implementację zadań rozszerzają wspólną warstwę abstrakcji, zgodnie z rysunkiem 11:



Rysunek 11 Implementacja wzorca projektowego "Łańcuch odpowiedzialności"; Źródło: [35]

Strumień jest łańcuchem zadań wykonywanych w trybie sekwencyjnym. Oznacza to, że każde zadanie jest wykonane po zakończeniu poprzedniego zadania. Egzekucja strumienia jest operacją asynchroniczną, wykonaną w oddzielnym wątku.

Asynchroniczność jest osiągnięta poprzez wykorzystanie specjalistycznych metod oferowanych przez bibliotekę Spring. W tym celu należy włączyć wsparcie komunikacji asynchronicznej, poprzez oznaczenie klasy konfiguracyjnej adnotacją `@EnableAsync`. Oznacza to, że system będzie wspierać wykonywanie metod oznaczonych jako `@Async` z wykorzystaniem zbioru wątków działających w tle. Ten zbiór jest specjalnie sparametryzowanym egzekutorem w postaci wstrzykiwanym jako Spring Bean. Dodatkowo wymagana jest odpowiednia konfiguracja unikatowego egzekutor, określający liczbę wątków wykorzystanych dla metod asynchronicznych, maksymalną wielkość kolejki zadań oraz prefiks, umożliwiający jego zidentyfikowanie w logach konsolowych. W przypadku konfiguracji deweloperskiej aplikacji, egzekutor posiada dwa wątki. Rozmiar zbioru rdzeni nie powinien przekraczać liczby fizycznych rdzeni procesora. Rysunek 12 prezentuje przykładową implementację wspomnianego egzekutora:

```
@EnableAsync
@Configuration
class ExecutorConfiguration {

    @Bean(name = "pipelineProcessorExecutor")
    public Executor pipelineThreadPool() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(2);
        executor.setMaxPoolSize(2);
        executor.setQueueCapacity(30);
        executor.setThreadNamePrefix("pipelineProcessor-");
        executor.initialize();
        return executor;
    }

}
```

Rysunek 12 Implementacja klasy Executor w systemie Zoran.io; Opracowanie własne

Zadania asynchroniczne zwracają instancję interfejsu Java, `CompletableFuture`. Umożliwia on proste łączenie wielu zadań asynchronicznych w pojedynczą operację asynchroniczną. Implementacja obiektów zadań dziedziczy z abstrakcyjnej klasy `AbstractPipelineTask`. Klasy reprezentujące implementacje zadań wykonywanych strumieniowo są opatrzone adnotacja `@Handler`. Umożliwia to zarejestrowanie nowego zadania w silniku procesowania strumieniowego wykorzystując automatyczną konfigurację komponentów Spring. Każde zadanie jest opatrzone

dodatkowymi metadanymi, zamieszczonymi w pliku z właściwościami YAML, zawierających kanoniczną nazwę zadania, krótki opis funkcjonalności, którą realizuje, identyfikator zadania oraz opcjonalne parametry. Parametry są dodatkowymi informacjami, które wypełnia użytkownik w momencie zdefiniowania swojego strumienia.

Poszczególne zadania mają możliwość rejestrowania artefaktów, czyli informacji, które są wynikiem egzekucji zadania. Artefakty, które są ścieżkami do nowoutworzonych zasobów, użytkownik może pobrać w formie skompresowanego archiwum ZIP.

6.3.2.2 Zaimplementowane zadania

W opisywanym systemie zaimplementowane zostały trzy przykładowe zadania spełniające główne wymagania projektowe. Pierwszym z zadań jest stworzenie nowego zasobu, projektu, zgodnie z wybraną definicją. W tym celu wykorzystywany jest silnik szablonowy, którego implementacja opisana jest w rozdziale 5.3.3 Silnik Szablonowy.

Drugim zadaniem, również bazującym na funkcjonalności silnika szablonowego jest wygenerowanie zasobów z wybranych szablonów z repozytorium szablonów wygenerowanych przez użytkownika.

Trzecim zaimplementowanym zadaniem, dostępnym tylko w przypadku, gdy autentykacja użytkownika jest włączona, jest stworzenie zdalnego repozytorium w serwisie Github.com oraz przesłanie wygenerowanych zasobów.

W tym celu, system najpierw inicjalizuje repozytorium lokalne, do którego zapisywane są artefakty poprzednich zadań, lub pozostaje ono puste. W drugim przypadku silnik dodaje pojedynczym plikiem README.md do nowego repozytorium lokalnego. Następnie system dodaje wszystkie nowe pliki z repozytorium lokalnego do systemu wersjonowania. Odpowiednik metody w terminalu to:

```
$ git add .
```

Śledzone zmiany są zatwierdzane i przesłane do repozytorium zdalnego. Poniżej znajduje się analogiczna komenda wykonująca wspomniane zadania poprzez terminal:

```
$ git commit && push origin
```

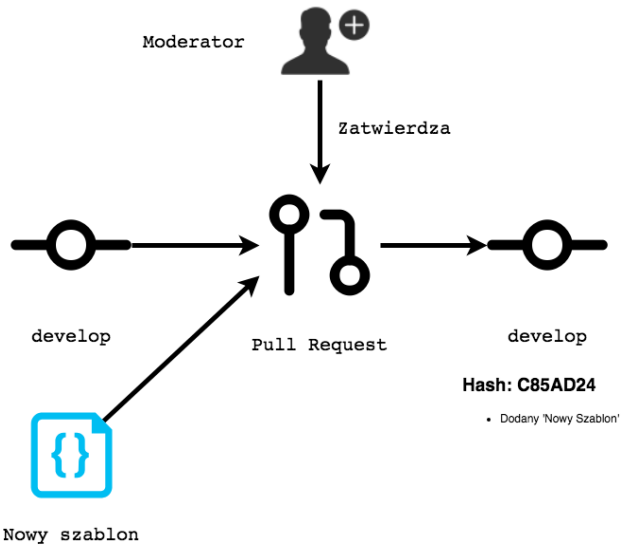
System można rozszerzyć o własne zadania. Należy dodać do źródeł aplikacji własnych implementacji klasy abstrakcyjnej, wraz z plikiem z właściwościami zawierającym metadane opisujące dodane zadania.

6.3.3 Silnik szablonowy

Jednym z najważniejszych funkcji aplikacji jest możliwość procesowania plików szablonów. Tak jak zostało to opisane w rozdziale “5.1.4 Obsługa szablonów”, do pliku szablonowego dodaje się plik zwany „*hash*”, zawierający wartości wstawiane w sam szablon. Poniższy rozdział zaprezentuję architekturę oraz implementację systemu szablonowego.

6.3.3.1 Przechowywanie i synchronizacja szablonów

Szablony są przechowywane w wersjonowanym repozytorium, hostowanym w serwisie GitHub.com. Jest to publicznie dostępna usługa, implementująca funkcjonalność zdalnego repozytorium git. Pozwala to na przetrzymywanie wersjonowanych plików oraz łatwą kolaborację pomiędzy wieloma użytkownikami w ramach jednego projektu. Wykorzystanie systemu kontroli wersji w opisanym projekcie pozwala na implementację systemu socjalnego, czyli umożliwia użytkownikom łatwe dzielenie się i wykorzystywanie cudzych szablonów. Rysunek 13 przedstawia podstawowy schemat kolaborowania w ramach systemu git.



Rysunek 13 Diagram prezentujący proces kolaboracji nad szablonami; Opracowanie własne

Git umożliwia tworzenie osobnych gałęzi, będący wskaźnikami na daną wersję danych zawartych w projekcie. Repozytorium zawiera domyślną gałąź „*develop*”, będącą aktualną, najświeższą wersją szablonów. Użytkownicy mogą do tej wersji zgłaszać prośby o dołączenie nowych zmian, poprzez zgłaszanie *pull request*. W tym momencie, desygnowany kurator, mający dostęp do platformy ma możliwość zaakceptowania bądź odrzucenia zmian, zapewniając moderowanie danych i kontrolę nad systemem. Próby jednoczesnych zmian tych samych zasobów będą powodować konflikty,

uniemożliwiając możliwość dołączenia nowych zmian. W przypadku zaakceptowania zmian, są one dołączane do repozytorium i stają się możliwe do wykorzystania przez system.

Dane ze zdalnego repozytorium są synchronizowane z repozytorium lokalnym okresowo. Jest to pewne ograniczenie funkcjonalności systemu, którego potencjalne rozwiązanie omówione jest w rozdziale “7.2.2 Wykrywanie zmian i synchronizacja szablonów”. Okres synchronizacji może być zdefiniowany w pliku z właściwościami lub przekazana jako zmienna środowiskowa. Domyślnie synchronizacja odbywa się co 5 min (300 sekund). W celu konfiguracji mechanizmu synchronizacji wykorzystywany jest specjalistyczny podzespół Spring, Spring Scheduler.

Klasa odpowiedzialna za synchronizację nazywa się `IndexModelPopulator`. Posiada pojedynczą metodę, która uruchamia proces klonowania repozytorium zewnętrznego oraz indeksowanie repozytorium lokalnego. Do tej metody zostały dodane następujące adnotacje:

- `@PostConstruct` adnotacja bezpośrednio nawiązująca do cyklu życia obiektu. Powoduje, że funkcja jest uruchamiana tuż po inicjalizacji obiektu Spring Bean. Metoda jest wywoływana wyłącznie jednorazowo.
- `@Scheduled(fixedDelay=30000, fixedRate=300000)` adnotacja definiuje nowe zadanie dla Spring Scheduler, który uruchamia się dokładnie raz na każde 300000 milisekund (5 minut), z 30 sekundowym początkowym opóźnieniem. Zastosowanie opóźnienia pozwala na bezpieczne zakończenie operacji czyszczenia poprzedniej iteracji modeli.

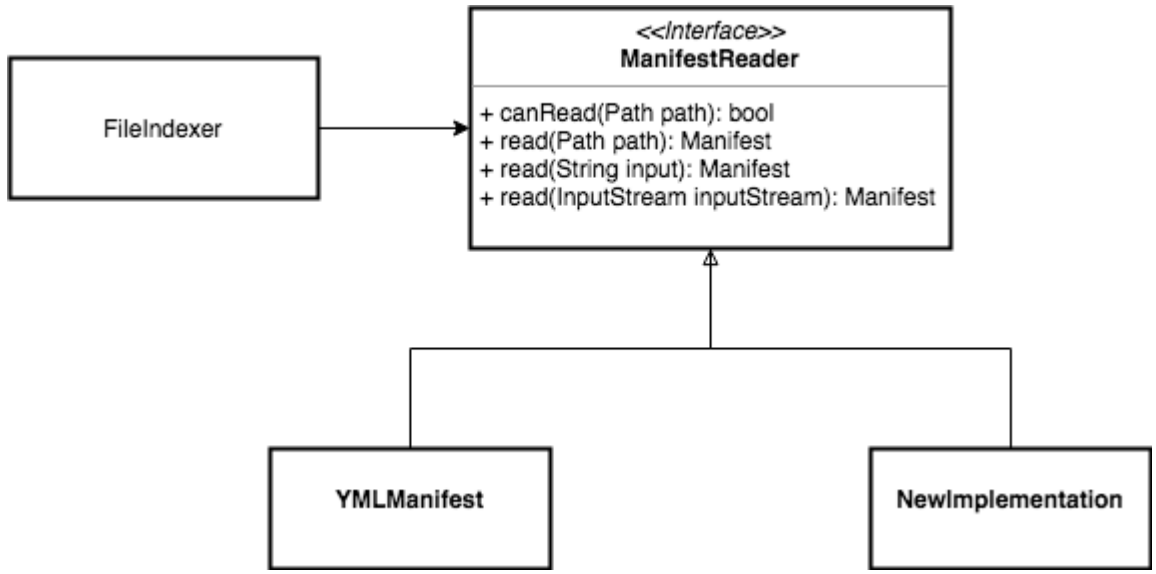
6.3.3.2 Indeksowanie modeli

Indeksowaniem modeli, w kontekście projektu, nazywamy analizę świeżo zsynchronizowanego zdalnego repozytorium oraz mapowanie struktury plików do obiektowego grafu acyklicznego. Po synchronizacji, w lokalnym repozytorium znajduje się luźny system plików. Indeksowanie jest potrzebne, aby silnik szablonowy był świadomy, jakie posiada szablony i w jaki sposób je procesować. W tym celu system czyta wszystkie pliki zawarte pod ścieżką do którego zostało zsynchronizowane repozytorium i oznacza je w zależności od typu pliku. Algorytm czytanie jest zaimplementowany według poniższej listy zadań:

- jeżeli plik jest katalogiem, dodaj węzeł do grafu.
- przeczytaj następny plik,
- jeżeli plik jest typu *file* i nie katalogiem lub pustą referencją,
- sprawdź, czy jest plikiem typu manifest.
- przeczytaj manifest,
- dodaj manifest do węzła, katalogu, w którym się znajdował.
- przeczytaj następny plik.

Indeksowanie kończy się w momencie, gdy wszystkie pliki zostaną przeczytane i uporządkowane, lub gdy operacja dodawania manifestu zawiedzie. Błąd w indeksowaniu będzie oznaczał, iż funkcjonalność generowania kodu z szablonów będzie niedostępna.

Domyślnie manifesty są plikami typu „.yml", umożliwiając prosty odczyt zawartości dla użytkowników. Klasą odpowiedzialną za czytanie plików tego typu, jest klasa `YMLManifestReader` implementująca interfejs `ManifestReader`. Utworzenie poziomu abstrakcji nad czytnikami manifestu pozwalają na personalizacje technologii obsługującej manifesty. Umożliwia to dodanie własnych standardów przechowywania informacji, na przykład bazując na standardzie JSON. W tym celu należałoby zaimplementować interfejs `ManifestReader` wraz z implementacją logiki wszystkich metod abstrakcyjnych. Schemat obiektowy modułu manifestów zaprezentowany jest poniżej:



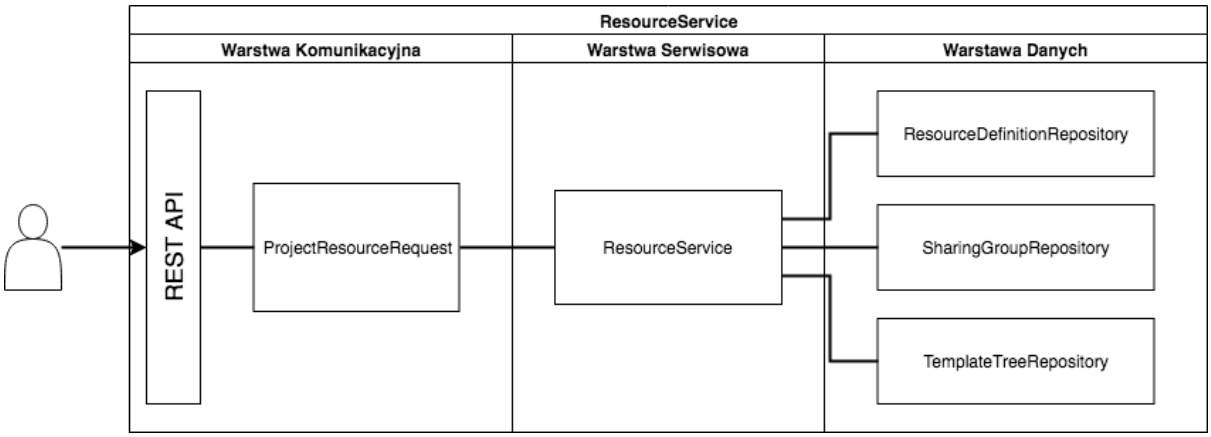
Rysunek 14 Schemat klas ManifestReader implementujący wzorzec projektowy mostu; Opracowanie własne

Jest to implementacja wzorca projektowego mostu [36], umożliwiając odłączenie warstwy abstrakcji od konkretnych implementacji pozwalając na ich niezależne wykorzystanie. Dodatkowe modułu rejestrowane są w kontekście aplikacji podczas fazy inicjalizacji i są wykorzystywane przez silnik manifestu.

6.3.3.3 Generowanie nowego zasobu przez użytkownika końcowego

Pojęcie generowania nowego zasobu, należy rozumieć jako wygenerowanie nowego projektu programistycznego, bazując na metadanych dostarczonych przez użytkownika. Silnik generujący zasoby podzielony jest warstwowo, zapewniając klarowne rozdzielenie pomiędzy warstwami danych, warstwą serwisową a warstwą komunikacyjną.

Rysunek 15 prezentuje wysokopoziomowy schemat wspomnianego rozwarstwienia usługi.



Rysunek 15 Diagram prezentujący podział warstw serwisu zarządzającym zasobami; Opracowanie własne

6.3.3.3.1 Warstwa komunikacyjna

W warstwie komunikacyjnej zdefiniowane są metadane zawierające informacje określające pożądany język programowania, pełną nazwę nowego projektu, wykorzystane szablony i możliwe zależności, które mają zostać dodane do projektu. Te metadane składają się na definicję nowego projektu. Definicje są przesyłane wykorzystując REST-API, przez co są opisane z wykorzystaniem standardu JSON. Rysunek 16 prezentuje schemat definicji nowego projektu.

```
ProjectResourceRequest { 
    artifactId          string
    bootVersion         string
    description         string
    gitUrl              string
    groupId             string
    javaVersion         string
    lead                string
    licenseKey          string
    name                string
    projectLanguage     string
    resourceVisibility  string
                        Enum:
                         [ PRIVATE, SHARED_FRIENDS, PUBLIC, ALL ]
    tags                string
    templatesUsed
                         [string]
    type                string
                        Enum:
                         [ CLASS, TEMPLATE, PROJECT, MAVEN_PROJECT, GRADLE_PROJECT, MAVEN_POM, GRADLE_CONFIG ]
    version             string
}
```

Rysunek 16 Definicja nowego zasobu w formacie JSON; Opracowanie własne

Nowe zasoby są wersjonowane zgodnie z systemem wersjonowania semantycznego MAJOR.MINOR.PATCH. Jest to standardowa metoda opisywania wersji systemów informatycznych wykorzystywana oraz zalecana przez międzynarodowe stowarzyszenie zrzeszające inżynierów technologii wokół internetowych IETF [37]. Wersję swojego zasobu, użytkownik definiuje przy pomocy parametru `version`.

Nowy zasób może zostać opisany, pozwalając potencjalnym użytkownikom dowiedzieć się więcej na temat danej definicji projektu. W tym celu można wykorzystać poniższe parametry:

1. *name* zawierający kanoniczną nazwę zasobu, jest parametrem wymaganym, ograniczonym do 15 znaków alfanumerycznych.
2. *lead* będący, krótkim wstępem do opisu zasobu,
3. *description* będący pełnym opisem zasobu, którego dana definicja dotyczy, pełniąc podobną rolę do pliku README. System wspiera także język Markdown, umożliwiając łatwe i bogate narzędzia do formatowania tekstu,
4. *tags* słowach klucze, umożliwiające łatwe wyszukiwanie definicji zasobów,
5. *licenceKey*, identyfikator typu licencji, która ma zostać dołączona do zasobu.

W obecnej wersji prototypu wspierane są projekty bazujące na językach wykorzystujących JVM: Java, Groovy, Kotlin wykorzystujące Gradle lub Maven, oraz projekty nie wymagające wygenerowania struktury plików i folderów. System może zostać rozszerzony o nowe funkcjonalności umożliwiające budowanie projektów bazujących na innych językach programowania. Wykorzystując parametr `projectLanguage` użytkownik może zdefiniować język projektu.

W językach JVM'owych, standardowa struktura folderów w projekcie jest zdefiniowana następująco:

```
nazwa projektu > src > main > java | resources > path
```

Następnie, użytkownik musi ustawić domyślną ścieżkę pakietów, która jest konkatenacją, czyli połączeniem ciągów znaków, z dwóch parametrów zgodnie z poniższą regułą:

```
{groupId}.{artifactId}
```

Te parametry są wykorzystane do stworzenia podstawowej struktury folderów, a także są umieszczone w plikach budujących Maven lub Gradle (w zależności od wyboru użytkownika). Wybór zdefiniowany jest przez ustawienie konkretnej wartości w `type` czyli typie generowanego zasobu. Podział typów na rodzaj budowanego zasobu można podzielić w sposób następujący:

- Budowanie nowego projektu programistycznego:
  - `MAVEN_PROJECT` - projekt wykorzystujący technologię Apache Maven jako system do zarządzania zależnościami i budowaniem projektu.

        - `GRADLE_PROJECT` - projekt wykorzystujący technologię Gradle jako system do zarządzania zależnościami i budowaniem projektu.
    - Budowanie plików konfigurujących systemy do zarządzania zależnościami:
        - `MAVEN_POM` - plik konfigurujący dla technologii Apache Maven
        - `GRADLE_CONFIG` - plik konfigurujący dla technologii Gradle
    - Budowanie pojedynczych plików z szablonów:
        - `TEMPLATE`
    - Budowanie plików nieokreślonych lub innych:
        - `CLASS` - budowanie plików klas, dla języków obiektowych
        - `PROJECT` - budowanie plików innych typów, niewspomnianych w żadnym powyższym punkcie.

W przypadku, gdy budowany zasób ma wykorzystywać szablony w trakcie procesu budowania, wykorzystywany jest parametr `templatesUsed` jako tablice identyfikatorów szablonów to użycia. Wypełniona definicja nowego zasobu jest potem przekazana warstwie serwisowej.

6.3.3.3.2 Warstwa serwisowa

Warstwa serwisowa jest serią klas i pakietów odpowiedzialnych za logikę tworzenia produktu końcowego, czyli gotowych plików, a także jest wspiera operacje dodawania, edycji, usuwania definicji zasobów. Tworzenie gotowych plików odbywa się w dwóch trybach:

1. Generowanie plików z szablonów wykorzystując metadane szablonu,
2. Generowanie nowego projektu wykorzystując język i technologie zawarte w definicji nowego zasobu.

Generowanie pliku z szablonu jest procesem łączenia pliku szablonowego wraz jego *hashem*, który zawiera konkretne wartości zdefiniowane przez użytkownika. Domyślna technologia wykorzystana w projekcie to, wspomniane wcześniej, Mustache. Istnieje możliwość rozszerzenia systemu o dodatkowe technologie szablonowe. Dodatkowe serwisy muszą implementować interfejs `TemplateProcessor` oraz zarejestrować się jako procesor szablonów w kontekście aplikacji poprzez dodanie adnotacji `@Processor` na klasie. Interfejs zawiera dwie metody:

1. `void compile(TemplateClassContext tc);` gdzie `TemplateClassContext` jest *hashem*. Metoda jest odpowiedzialna za utworzenie pliku końcowego z szablonu,
2. `boolean canProcess(Path tc);` metoda zwracająca flagę czy dany procesor szablonów może obsłużyć dany plik szablony. Decyzja podejmowana jest na podstawie

rozszerzenia pliku danej technologii szablonowej. Na przykład pliki z rozszerzeniem „.mustache", będą wykonywane przez procesor szablonów w technologii Mustache.

Domyślnie zaimplementowany system szablonów Mustache wykorzystuje informacje zawarte w manifeście jako metadane szablonu, a także wartości otrzymane od użytkownika końcowego w *hash*. Identyfikatory szablonów są pobierane z definicji nowego zasobu i po kolei wyszukiwane w grafie przechowywanych szablonów. System wykonuje poszukiwanie odpowiedniego manifestu zawierającego wspomniane metadane wykorzystując implementację algorytmu przeszukiwania w głąb *(DFS)*.

Użytkownik końcowy ma możliwość dodania swoich wartości, do konkretnego szablonu. Każde pole przeznaczone do wypełnienia ma poniższą formę:

```
{{nazwa_pola}}
```

Hash podany przez użytkownika jest mapą, zawierającą nazwę wartości szablonu (nazwa_pola) jako klucz i wartość, która ma zostać wstawione w dane miejsce. Metadane szablonu, wraz z *hashem* zapisywane są w obiekcie `TemplateClassContext` i przesyłane dalej do odpowiedniego procesora. Silnik skanuje plik szablonowy i wypełnia go, zgodnie z danymi zawartymi w tablicy. Następnie podejmowana jest decyzja o finalnym rozszerzeniu pliku. Zaproponowane rozwiązanie zakłada, że odpowiedzialność na opatrzeniu pliku końcowego odpowiednim rozszerzenie spoczywa na twórcy szablonu. Autor wie jaki powinien być typ pliku wynikowego, na przykład „`.java", „.kt"` czy bez żadnego rozszerzenia. W tym celu pliki szablonowe powinny być nazywane zgodnie z poniższym schematem:

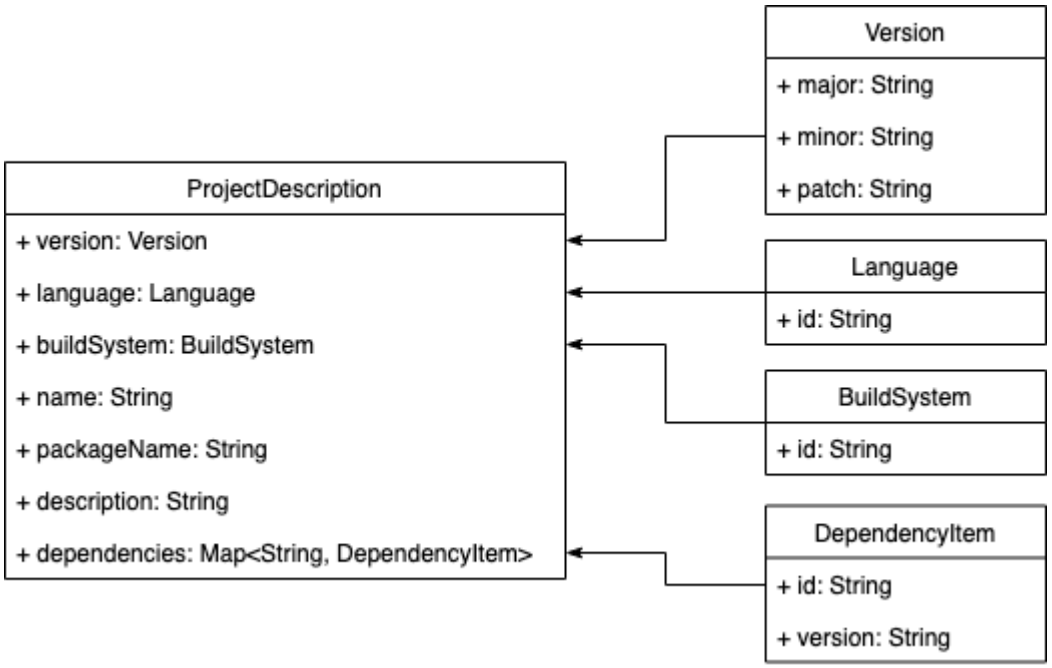
```
nazwa_kanoniczna.docelowe_rozszerzenie.rozszerzenie_technologii_szablonowej
```

Na przykład:

```
ApplicationTest.java.mustache
```

W procesie generowania pliku, rozszerzenie technologii szablonowej jest usuwane pozostawiając oryginalną nazwę pliku wraz z docelowym rozszerzeniem.

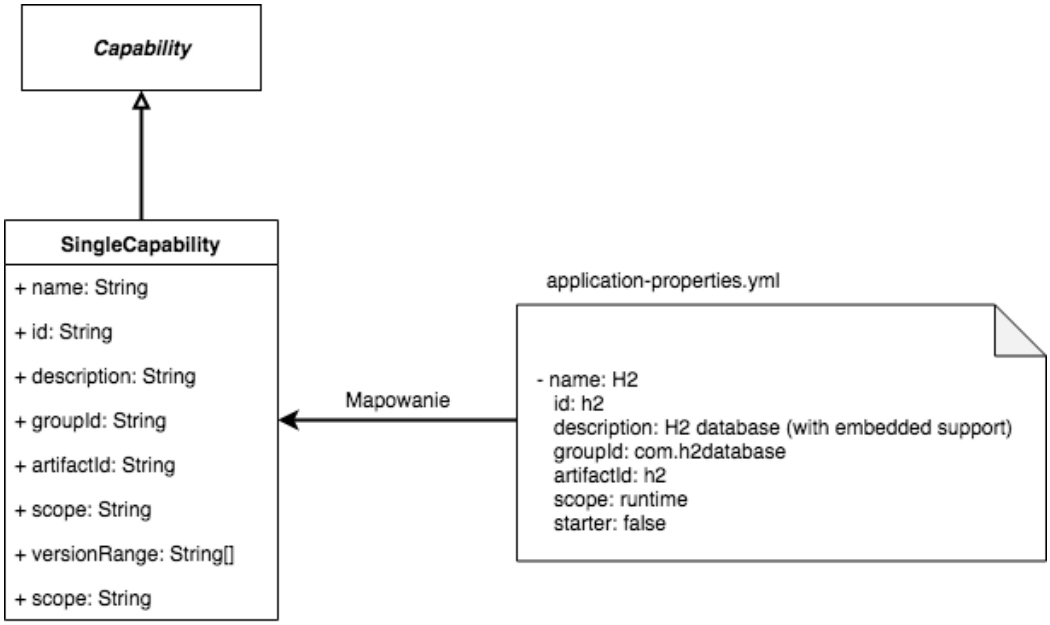
Alternatywną usługą jest generowanie nowego projektu z definicji projektu zakładającą stworzenie podstawowej struktury folderów wykorzystując: język programowania, wersję, system budowania i zależności zdefiniowane przez użytkownika końcowego. W tym celu zaimplementowany został serwis obsługujący wspomnianą funkcjonalność. Rysunek 17 prezentuje diagram klas realizujący strukturę projektu.



Rysunek 17 Implementacja struktury projektu; Opracowanie własne

Wartości, z których inicjalizowane są powyższe obiekty są pobierane z definicji nowego zasobu zdefiniowanego przez użytkownika końcowego, omówionych w rozdziale „5.3.3.3.2 Warstwa Serwisowa”. Poszczególne parametry są korelowane z lokalnym repozytorium dostępnych wartości. Lista wspieranych elementów, takich jak wspierane wersje lub zależności, jest przechowywana w postaci pliku z właściwościami. Ich umiejscowienie poza kodem źródłowym aplikacji, powoduje, że mogą być w łatwy sposób rozszerzone. Dodanie nowych zależności lub pakietów odbywa się bez potrzeby ponownego wdrażania aplikacji. Możliwe wartości zawarte w pliku z właściwościami mapują się na encje `Capability` i są wykorzystywane przez silnik w celu wygenerowania nowego projektu.

Właściwości systemu, czyli obiekty typu `Capability` mapują się w sposób następujący z pliku z właściwościami, na przykładzie zależności bazy danych H2:



Rysunek 18 Mapowanie dostępnych zależności z pliku z właściwościami; Opracowanie własne

W celu mapowania, wykorzystana została adnotacja `@ConfigurationProperty()` umożliwiająca przekazanie ścieżki YML w postaci parametru. Umożliwia to automatyczną konfigurację obiektów typu `Capability` podczas inicjalizacji aplikacji.

Każdy nowy zasób typu „nowy projekt" składa się ze struktury folderów, których ścieżka określana jest według wcześniej wspomnianego wzoru:

```
nazwa projektu > src > main > java | resources > path
```

gdzie nazwa projektu jest konkatenacją wersji zasobu zgodnie ze standardem IETF: MAJOR.MINOR.PATCH. W lokalnym systemie plików, stworzona jest ścieżka ustawiająca podstawową strukturę folderów, za pośrednictwem java.nio API Path. W przypadku języków JVM-owych, czyli domyślnie wspieranych przez ten projekt, dodany zostaje standardowy plik zawierający metodę:

```
void main(String[] args),
```

czyli punkt wejściowy aplikacji, nazwany zgodnie z nazwą projektu podaną w definicji projektu.

Do struktury plików dodany jest także system umożliwiający budowanie aplikacji: Gradle lub Apache Maven. Wybór jest dokonany poprzez interpretację podanego typu projektu i mapowanie go na obiekt typu `BuildSystem`. W zależności od wyboru, do folderu głównego należy dodać:

1. dla Apache Maven:
    a. plik pom.xml zawierający definicję projektu wraz z deklaracjami zależności, czyli zewnętrznych bibliotek,
    b. pliki wykonawcze mvnw i mvnw.cmd umożliwiające wykorzystanie API Apache Maven bez potrzeby deklarowania go w ścieżce systemowej systemu-klienta.
2. dla Gradle:
    a. build.gradle zawierający definicję projektu wraz z deklaracjami zależności, czyli zewnętrznych bibliotek,
    b. gradle i gradle.cmd umożliwiające wykorzystanie API Gradle bez potrzeby deklarowania go w ścieżce systemowej systemu-klienta.

O ile powyższe pliki wykonawcze są jednakowe, niezależnie od projektu, zawartość plików pom.xml i build.gradle w dużej mierze zależy od definicji zasobu. Standardowy plik budujący wykorzystujący technologię Apache Maven zawiera informację dotyczące:

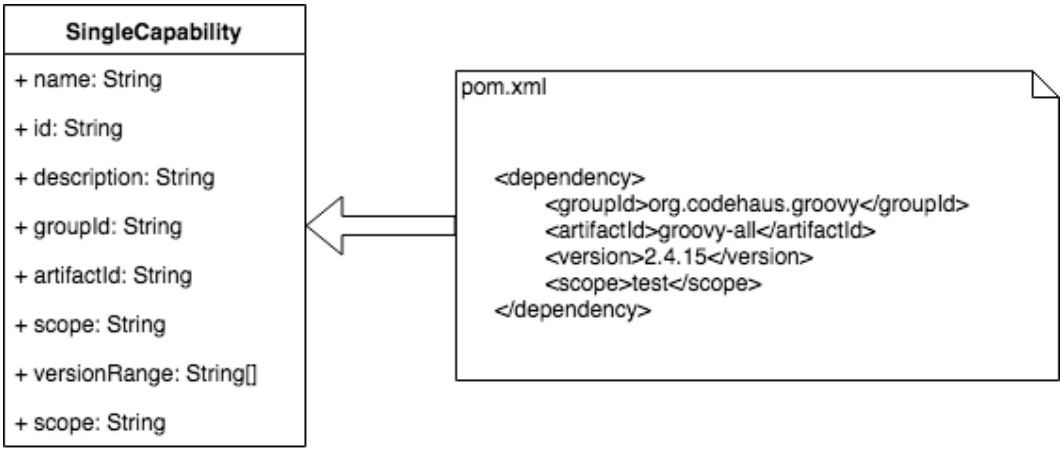
- kanonicznej nazwy projektu, wybranym przez użytkownika,
- wersji projektu, zgodnie z wcześniej wspomnianym standardzie IETF,
- opis projektu,
- typ pakowania projektu: JAR lub WAR,
- repozytorium bibliotek skąd mają być pobierane zależności,
- listę zależności dodanych do projektu,
- listę plug-in, czy dodatkowych narzędzi, które mogą zostać wykorzystane przez system budowania.

Rysunek 19 zawiera przykładowy plik pom.xml.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>io.zoran</groupId>
    <artifactId>core</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>core</name>
    <description>Zoran Core</description>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.0.RELEASE</version>
        <relativePath/>
        <!-- lookup parent from repository -->
    </parent>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <spring-cloud.version>Finchley.SR1</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/libs-snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
    </repositories>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </build>
        <profiles></profiles>
    </project>
```

Rysunek 19 Przykładowy plik pom.xml z zależnością na moduł spring.cloud; Opracowanie własne

Można zauważyć, że obiekty typu `Capability`, przechowujące dane dotyczące informacji zależności, w prosty sposób mapują się na kolejne węzły `<dependency>` w pliku XML.



Rysunek 20 Mapowanie zależności w pliku pom.xml na encję SingleCapability*; Opracowanie własne*

Poszczególne wartości parametrów obiektów `SingleCapability` są dopisywane do pliku pom.xml jako kolejne węzły tekstowe `<dependency>`.

W przypadku, gdy w nowym projekcie wymagane będzie dodanie plików wygenerowanych z współdzielonych szablonów, zostaną one dodane według flagi zawartej w manifeście. Parametr `template.preferredLocation` zawiera wyliczenie określające docelową lokacje pliku w projekcie. Wyliczenie ma następującą formę:

```
public enum Location {
    INFRASTRUCTURE,
    API,
    DOMAIN,
    SERVICES,
    PRESENTATION,
    DATA
}
```

Rysunek 21 Definicja umiejscowienia nowego zasobu; Opracowanie własne

Każda wartość określa warstwę, do której dany plik ma przynależeć. Do ścieżki doklejana jest wartość parametru z manifestu, tworząc nowy folder w projekcie. To rozwiązanie posiada wadę, wymuszając utworzenie nowego folderu bądź dodanie wygenerowanego pliku do już istniejącego folderu, bez możliwości zmiany docelowej ścieżki przez użytkownika końcowego.

6.3.3.3.3 Warstwa danych

Warstwa danych wspierających silnik szablonowy składa się z trzech repozytoriów. Służą one do przechowywania i serwowania informacji o zasobach stworzonych przez użytkowników systemu. Poniżej znajduję się krótki opis funkcjonalności i zadań wspomnianych magazynów danych:

ResourceRepository jest repozytorium bazującym na nierelacyjnej bazie danych *MongoDB*, przetrzymującym definicje zasobów stworzonych przez użytkowników.

SharingGroupRepository jest repozytorium przetrzymującym dane dotyczące uprzywilejowanych użytkowników do dostępu do danego zasobu (więcej na temat współdzielenia dostępu do zasobów w rozdziale poświęconym bezpieczeństwie, „5.3.4 Systemy Zabezpieczeń"). Informacje są również przechowywane w postaci dokumentów w bazie danych *MongoDB*.

TemplateTreeRepository jest repozytorium realizowanym w postaci lokalnie przechowywanych plików szablonów. Jest to docelowe miejsce, do którego synchronizowane są szablony z repozytorium zdalnego.

6.3.4 Systemy zabezpieczeń

Jednym z wymogów systemu, którego celem jest integracja w firmowe środowisko informatyczne, jest posiadanie odpowiedniego systemu zabezpieczeń. Poniższy rozdział opisuje typy przechowywanych danych wraz z identyfikacja danych poufnych i tajnych oraz metody ich zabezpieczenia [38].

Ogólnie pojęte bezpieczeństwo aplikacji bazuje na trzech pryncypiach:

1. Poufność (*Confidentiality*), czyli zabezpieczenie informacji przed nieautoryzowanym dostępem osób trzecich,
2. Spójność **(***Integrity*), zapewnienie, że informacje są niemutowane przez osoby trzecie,
3. Dostępność (*Availability*) gwarantujący dostęp do zasobów, kiedy autoryzowani użytkownicy ich potrzebują.

Podstawową metodą zabezpieczenia systemu, realizującą powyższe punkty jest wprowadzenie systemu umożliwiającego dostęp do zasobów tylko osobom autoryzowanym poprzez bramkę logowania. Umożliwia odizolowanie użytkowników systemu od osób trzecich a także umożliwia korelację zasobów z ich właścicielami. Rozdzielenie warstw zasobów od użytkowników, pozwala na dostosowanie dostępu użytkowników tylko do potrzebnych informacji.

Warstwa bezpieczeństwa realizowana jest przy pomocy rozszerzenia Spring Boot Security. Opis implementacji opisany jest w rozdziale „5.3.4.4 Implementacja systemu zabezpieczeń".

6.3.4.1 Przechowywane dane

W poniższym podrozdziale, znajdują się rozpisane przechowywane typy danych wraz z analizą ich znaczenia w kontekście bezpieczeństwa.

Definicje nowych zasobów generowane przez klientów aplikacji, definicje mogą dotyczyć systemów tajnych i ich zawartość powinna być objęta zabezpieczeniami.

Szablony są zasobami jawnymi przechowywanymi w systemie Github.com, dostępnymi dla wszystkich użytkowników.

W celu wprowadzenia systemu logowania do aplikacji oraz dodania możliwości identyfikowania zasobów z ich właścicielami, wprowadzone zostały trzy dodatkowe typy obiektów. Dane tych encji są przetrzymywane w systemie w sposób nieulotny:

Dane audytowe zawierające informacje dotyczące operacji wykonanych w systemie. Wprowadzenie śledzenia operacji zapewnia możliwość wykrycia operacji wykonanych przez osoby nieautoryzowane.

Dane użytkownika, czyli informacje dotyczące podstawowych danych użytkownika, jak email, login czy obraz profilowy (*avatar)*.

Dane współdzielenia zasobów, czyli dane korelujące użytkowników wraz z zasobami. Ponieważ połączenie tych dwóch encji jest relacją typu wiele-do-wiele, wielu użytkowników może mieć dostęp do wielu zasobów. W tym celu wprowadzona została encja pomocnicza. Zawiera ona informacje o użytkownikach, którzy mają dostęp do danego zasobu, wraz z zakresem ich przywilejów, takich jak prawo do edycji czy oglądania.

6.3.4.2 System logowania

Prezentowany system został wyposażony w bramkę logowania, opartą o usługę serwisu Github.com, *Identity Access Management* wspomnianą w rozdziale „5.1.5 Identity Access Management”. Umożliwia ona autentykacje w docelowym systemie poprzez autoryzację użytkownika z serwisem Github.com. Proces integracji został opisany w rozdziale „6.3.1.2 Opis procesu integracji”.

W trakcie logowania poprzez Github.com, opisywany system otrzymuje podstawowe informacje o użytkowniku, przedstawione na rysunku 22:

```
"user": {
  "login": "octocat",
  "id": 1,
  "node_id": "MDQ6VXNlcjE=",
  "avatar_url": "https://github.com/images/error/octocat_happy.gif",
  "gravatar_id": "",
  "url": "https://api.github.com/users/octocat",
  "html_url": "https://github.com/octocat",
  "followers_url": "https://api.github.com/users/octocat/followers",
  "following_url": "https://api.github.com/users/octocat/following{/other_user}",
  "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
  "organizations_url": "https://api.github.com/users/octocat/orgs",
  "repos_url": "https://api.github.com/users/octocat/repos",
  "events_url": "https://api.github.com/users/octocat/events{/privacy}",
  "received_events_url": "https://api.github.com/users/octocat/received_events",
  "type": "User",
  "site_admin": false
}
```


Rysunek 22 Informacje dotyczące zalogowanego użytkownika; Opracowanie własne

Powyższe informacje są mapowane na obiekt `ZoranUser`, a następnie zapisywane w repozytorium danych w użytkownika. Dane są przechowywane w formie czystego tekstu, co jest zagrożeniem dla spójności systemu. Hasło użytkownika jest wymieniane na token w standardzie OAuth 2.0 zawierający zakres (*scope*) umożliwiający tworzenie nowych repozytoriów zdalnych oraz przesyłania zasobów i to on jest wykorzystywany w celu autentykacji użytkownika z systemem. Ponieważ tokeny mają swój termin ważności (*expiry time*), po jego wygaśnięciu jest wymagane, aby użytkownik zalogował się ponownie. Nowa sesja użytkownika jest połączona z istniejącymi danymi wykorzystując adres e-mail pochodzący z serwisu IAM.

6.3.4.3 Współdzielenie zasobów

W celu ograniczenia dostępu do definicji nowych zasobów osobom trzecim, musi zostać wprowadzony system umożliwiający korelacje zasobów wraz z ich właścicielami. Ponieważ dostęp do pojedynczego zasobu może zostać przyznany wielu użytkownikom z różnym zakresem przywilejów, należy określić rolę dla indywidualnego zasobu. Role dzielą się na następujące kompetencje:

- **VIEW**, umożliwiający oglądanie definicji zasobu,
- **EDIT** pozwalający na edycję parametrów definicji,
- **OWNER**, domyślnie twórca szablonu, posiadający możliwość nadawania lub odbierania poszczególnych kompetencji. Właściciel może być tylko jeden, aczkolwiek istnieje możliwość zmiany właściciela,
- **REVOKED** oznaczający brak dostępu dla danego użytkownika.

Informacje dotyczące poziomu dostępu do poszczególnych elementów zapisywane są w postaci obiektu typu `SharingGroup`. Klasa `SharingGroup` jest w specjalnej relacji agregacyjnej, z klasą

zawierającą informacje dotyczące definicji nowego obiektu `ProjectResource`. Oznacza to, że każdy obiekt typu `ProjectResource`, posiada obiekt typu `SharingGroup`, który jest określany jako obiekt składowy.

Grupy użytkowników zaimplementowane są na strukturze słownika zawierającej identyfikatory użytkowników jako klucze oraz poziom dostępu jako wartości. W przypadku, gdy zabezpieczenia systemu są włączone, zapytania dotyczące definicji zasobów są najpierw analizowane przez serwis odpowiedzialny za mechanizm współdzielenia zasobów. Serwis odczytuje obiekt typu `ProjectResource` z repozytorium definicji projektowych i z niego wybiera obiekt składowy `SharingGroup`. Ponieważ w tym obiekcie znajduje się lista uprawnionych użytkowników, wystarczy pobrać identyfikator użytkownika korzystającego w tym momencie z funkcjonalności. W tym celu, system patrzy na aktualną sesję użytkownika przechowywaną w kontekście bezpieczeństwa. Jest to specjalny obiekt należący do struktury *Spring Security*. Zawiera on informacje dotyczące użytkownika operującego w zakresie danego wątku. W przypadku, gdy w kontekście sesji znajduje się obiekt typu `ZoranUser`, identyfikator użytkownika jest porównywany z identyfikatorem z obiektu zawierającego dane współdzielenia zasobów `SharingGroup`. Jeśli w kontekście sesji nie ma użytkownika lub w informacji współdzielenia zasobów nie widnieje użytkownik o danym identyfikatorze lub jego przywileje są niewystarczające, podnoszony jest wyjątek i zapytanie jest odrzucane.

6.3.4.4 Zdefiniowane role użytkowników

W aplikacji zdefiniowane są trzy role, umożliwiające różny dostęp do zasobów aplikacyjnych. Role prezentują się następująco:

Użytkownik Anonimowy - użytkownik, który nie został zautoryzowany i nie posiada ważnej autentykacji przez zewnętrzny serwis IAM. Anonimowy użytkownik nie ma dostępu do danych wymienionych w rozdziale „6.3.4.1 Przechowywane dane”, z wyjątkiem danych szablonów, które są publiczne.

Użytkownik definiowany jest w kontekście bezpieczeństwa jako użytkownik systemu, który został pozytywnie zautoryzowany przez IAM, oferowanym przez serwis Github.com. Użytkownik ma wtedy dostęp do usług umożliwiających przeglądanie i tworzenie nowych zasobów oraz definiowanie i egzekucję strumieni wykonawczych („6.3.2 Procesowanie strumieniowe ciągów zadań”).

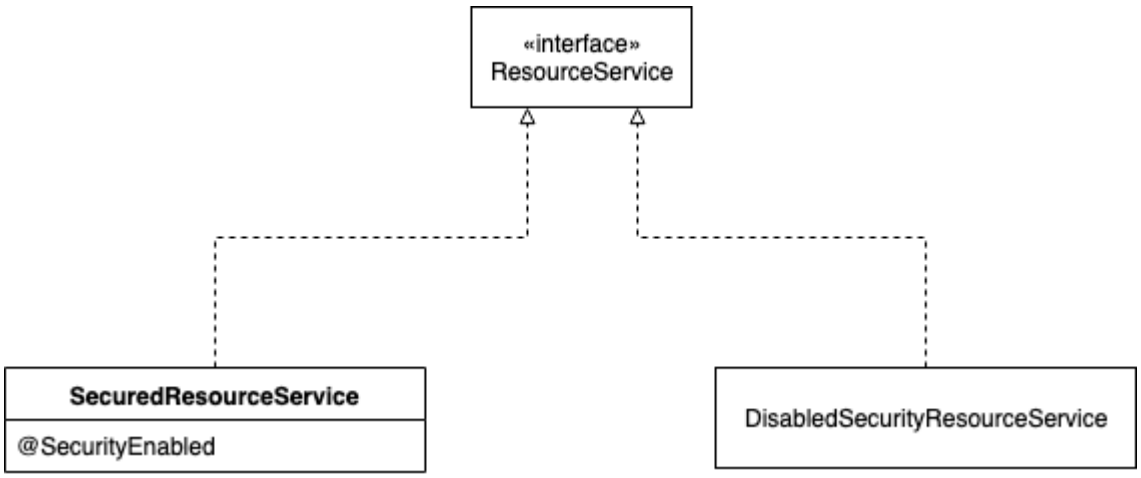
Administrator jest specjalnym typem użytkownika, pełniącym także rolę moderatora. Jego zadaniem jest zapewnienie spójności danych znajdujących się w repozytorium zdalnym git. Dodatkowo, posiada on możliwość przeglądania danych audytowych. Z perspektywy projektu systemu, nie ma dedykowanej

implementacji tej roli. Użytkownik o statusie Administrator posiada hasła, umożliwiające mu dostęp do bazy danych oraz serwisu GitHub.com.

6.3.4.5 Implementacja systemu zabezpieczeń

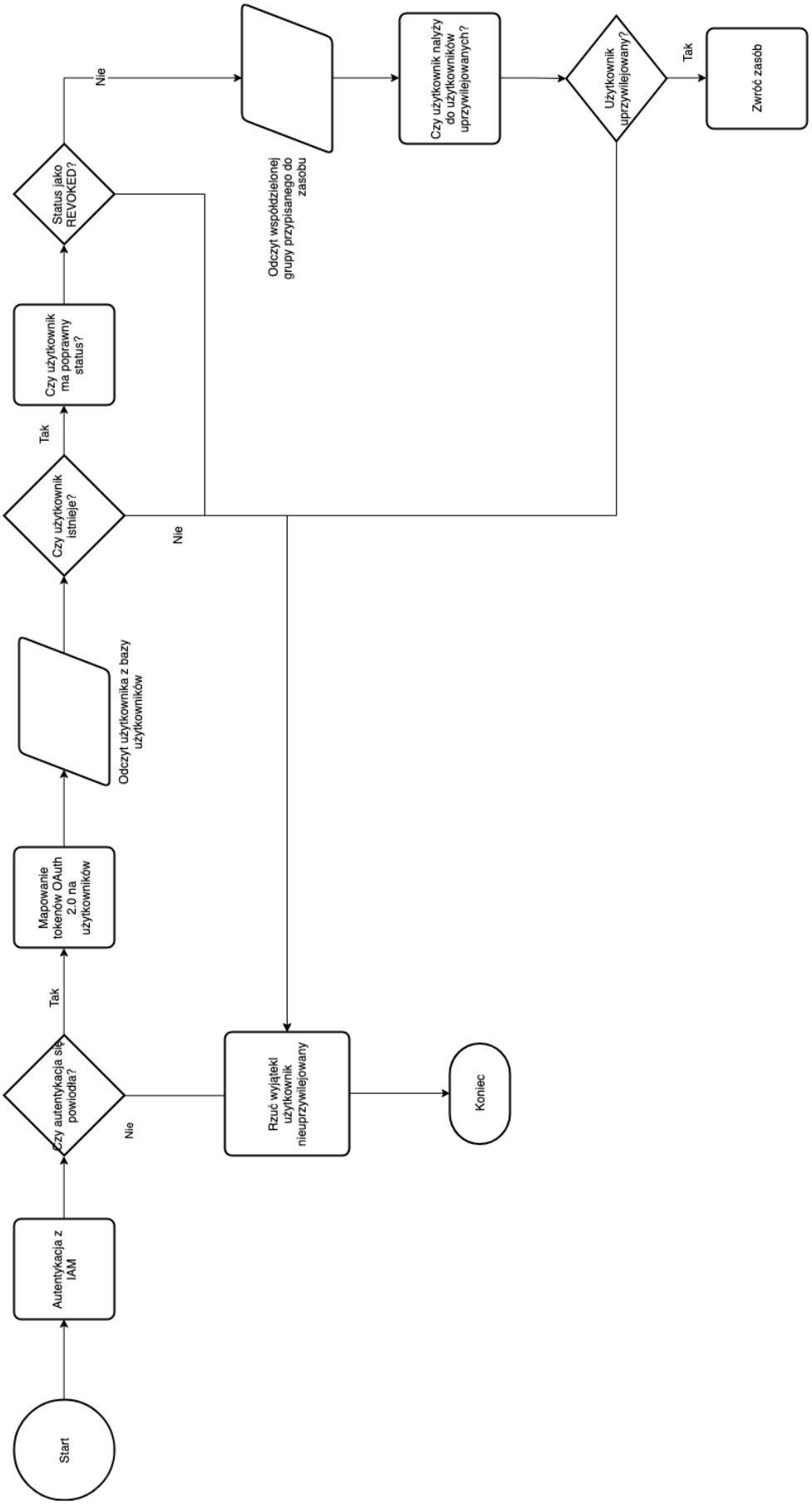
Warstwa bezpieczeństwa została zaimplementowana w oparciu o rozszerzenie Spring Boot Security. Z jej pomocą, można zabezpieczyć aplikacje przeciw większości ataków z zewnątrz i jest standardem w produkcji aplikacji opartych o Java. System umożliwia autentykację i autoryzację użytkowników wykorzystując w tym celu jeden z wielu dostępnych standardów. Opisywany system wykorzystuję technologię OAuth 2.0 w połączeniu z zewnętrznym dostawcą usług IAM, Github.com.

Warstwa bezpieczeństwa została zaimplementowana w sposób pozwalający na uruchomienie aplikacji w jednym z dwóch trybów, z włączonym lub wyłączonym systemem bezpieczeństwa. Funkcjonalność jest sterowana poprzez parametr `application.security.enabled` znajdujący się w pliku z właściwościami. Ustawienie flagi na stan `false`, powoduje, że usługi oraz komponenty odpowiedzialne za implementacje warstwy bezpieczeństwa, czyli klasy oznaczone adnotacją `@SecurityEnabled`, są pomijane w trakcie inicjalizacji kontekstu aplikacji. Implementacja adnotacji wykorzystuje adnotacje `@ConditionalOnProperty(name, value)` z biblioteki Spring. Parametrem adnotacji jest ścieżka pożądanej właściwości oraz oczekiwana wartość. Podczas inicjalizacji kontekstu aplikacji, warunek jest sprawdzany. W przypadku, gdy wynik warunku jest prawdziwy, obiekt jest inicjalizowany do kontekstu, w przeciwnym wypadku jest on pomijany. Rysunek 23 prezentuje sposób, w jakim wprowadzenie do warstwy serwisowej abstrakcji w celu umożliwienia płynnej zmiany stanu bezpieczeństwa aplikacji:



Rysunek 23 Diagram klas prezentujący dualizm serwisów; Opracowanie własne

Serwisy wykorzystujące powyższy mechanizm, to serwisy zajmujące się serwowaniem zasobów, współdzieleniem zasobów oraz użytkownikami. Podczas gdy warstwa bezpieczeństwa jest włączona, użytkownik musi prawidłowo wykonać sekwencję przedstawioną na rysunku 24:



Rysunek 24 Proces dostępu do zasobów; Opracowanie własne

Proces rozpoczyna się od zapytania o dany zasób. W przypadku, gdy zasób jest dostępny pod zabezpieczonym punktem końcowym, wymagana jest autentykacja użytkownika z zewnętrznym IAM. Jeśli proces autentykacji zakończony jest pomyślnie, użytkownik zostaje przekierowany z powrotem do punktu końcowego, gdzie autoryzuje się nowo nadanym tokenem. Token jest mapowany na wewnętrznego użytkownika, a dane sesji zapisywane są do kontekstu bezpieczeństwa. Następnie, system wczytuje z repozytorium dane dotyczące współdzielenia zasobu i sprawdza czy aktualny użytkownik należy do uprzywilejowanej grupy. Jeśli należy, zwracany jest pożądany zasób, w przeciwnym wypadku, system podnosi wyjątek.

Informacje dotyczące danych użytkownika w obecnej sesji są przechowywane w postaci kontekstu bezpieczeństwa. Jest on określany jako obiekt typu `SecurityContext`, czyli interfejs zawierający minimum informacji dotyczących obecnego wątku wykonawczego w formie obiektu typu `Authentication`. Jest to obiekt oznaczony jako `ThreadLocal`, co oznacza, że jest on dostępny tylko i wyłącznie dla obecnego wątku. Takie zabezpieczenie jest konieczne, gdyż są w nim zawarte tajne dane użytkownika. Kontekst jest budowany podczas zapytania usług webowych API. Jeżeli w nagłówkach zapytania HTTP znajduję się klucz dostępowy (*access token*), jest on wykorzystany jako parametr `@RegisteredOAuth2AuthorizedClient` w celu identyfikacji użytkownika. Jeżeli parametr jest nieobecny, użytkownik jest przekierowywany do strony IAM w celu zalogowania i uzyskania klucza dostępu.

Dostęp do kontekstu jest dostępny poprzez obiekt `SecurityContextHolder`, zawierający statyczną metodę `SecurityContext getContext();`. System wybiera informacje o użytkowniku, `Principal` i próbuje ten obiekt zmapować na wewnętrzny obiekt zawierający informacje dotyczące aktualnego użytkownika systemu. W przypadku, gdy obiekt jest innego typu i mapowanie nie może zostać przeprowadzone, oznacza to, iż jest to użytkownik anonimowy i dostęp do zasobów jest blokowany.

Dodatkową warstwą zabezpieczeń, jest ograniczenie dostępu nieautoryzowanego użytkownika do punktów końcowych usług API. W tym celu, została dodana dodatkowa klasa konfiguracyjna, rozszerzająca klasę `WebSecurityConfigurerAdapter`. Poprzez przeładowanie metody `protected void configure(HttpSecurity http)` można określić dokładną konfigurację dostępu do usług. Rysunek 25 pokazuje implementację konfiguracji:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
            .authorizeRequests()
            .antMatchers("/build-info",
                    "/swagger-ui.html",
                    "/swagger-resources", "/userinfo").permitAll()
            .anyRequest().authenticated()
            .and()
            .oauth2Login()
            .userInfoEndpoint()
            .userService(userService)
            .and()
            .failureUrl("/401")
            .and()
            .exceptionHandling()
            .accessDeniedPage("/401");
}
```

Rysunek 25 Implementacja konfiguracji dostępu do punktów końcowych aplikacji; Opracowanie własne

Powyższa konfiguracja definiuje zbiór predykatów, ustawiających politykę bezpieczeństwa dla aplikacji. Dotyczy ona dostępności punktów końcowych aplikacji dla ruchu zewnętrznego, wymagając poprawnej autoryzacji dla wszystkich funkcjonalności wystawionych poprzez REST-API. W tym celu obiekt `HttpSecurity` jest skonfigurowany, w sposób, aby wszystkie zapytania podlegały procesowi autentykacji, poprzez wywołanie metody parametryzującej:

```
HttpSecurity  http.anyRequest().authenticated();
```

W systemie uwzględnione są także punkty niewymagające autentykacji, dodane jako parametry metody

```
.antMatchers(...);
```

Nie wymagające autentykacji są punkty końcowe biblioteki *Swagger* [39], dającej możliwość prostego przedstawienia zbioru punktów końcowych, wraz z wymaganymi parametrami w przystępny dla człowieka sposób. Poprzez interfejs dostępne są metody pobierające dane o aktualnie zalogowanym użytkowniku i informacje dotyczące zainstalowanej iteracji aplikacji.

Ponieważ aplikacja wykorzystuje wspomniany wcześniej standard OAuth 2.0 w celu autentykacji użytkownika, wymagane jest zdefiniowanie konfiguracji w obiekcie `HttpSecurity`. Na konfigurację składają się trzy metody: inicjalizująca wykorzystanie standardu OAuth 2.0, metodę uruchamiającą punkt końcowy oferujący możliwość wyświetlania swoich danych oraz metodę

definiującą serwis tłumaczący token zawarty w obiekcie `OAuth2UserRequest` na wewnętrzny obiekt użytkownika `ZoranUser`.

Serwis dodatkowo zapewnia tworzenie nowych użytkowników, logujących się po raz pierwszy, a także umożliwia dostęp użytkownikom powracającym, korzystającym z usług systemu wcześniej. W przypadku gdy dostęp do aplikacji danego użytkownika jest zablokowany, aplikacji podnosi wyjątek `UserDeniedAuthorizationException(UNAUTHORIZED_MESSAGE);` i użytkownik nie ma dostępu do funkcjonalności aplikacji, poza treściami nie wymagającymi autoryzacji.

6.4 Aplikacja prezentacyjna

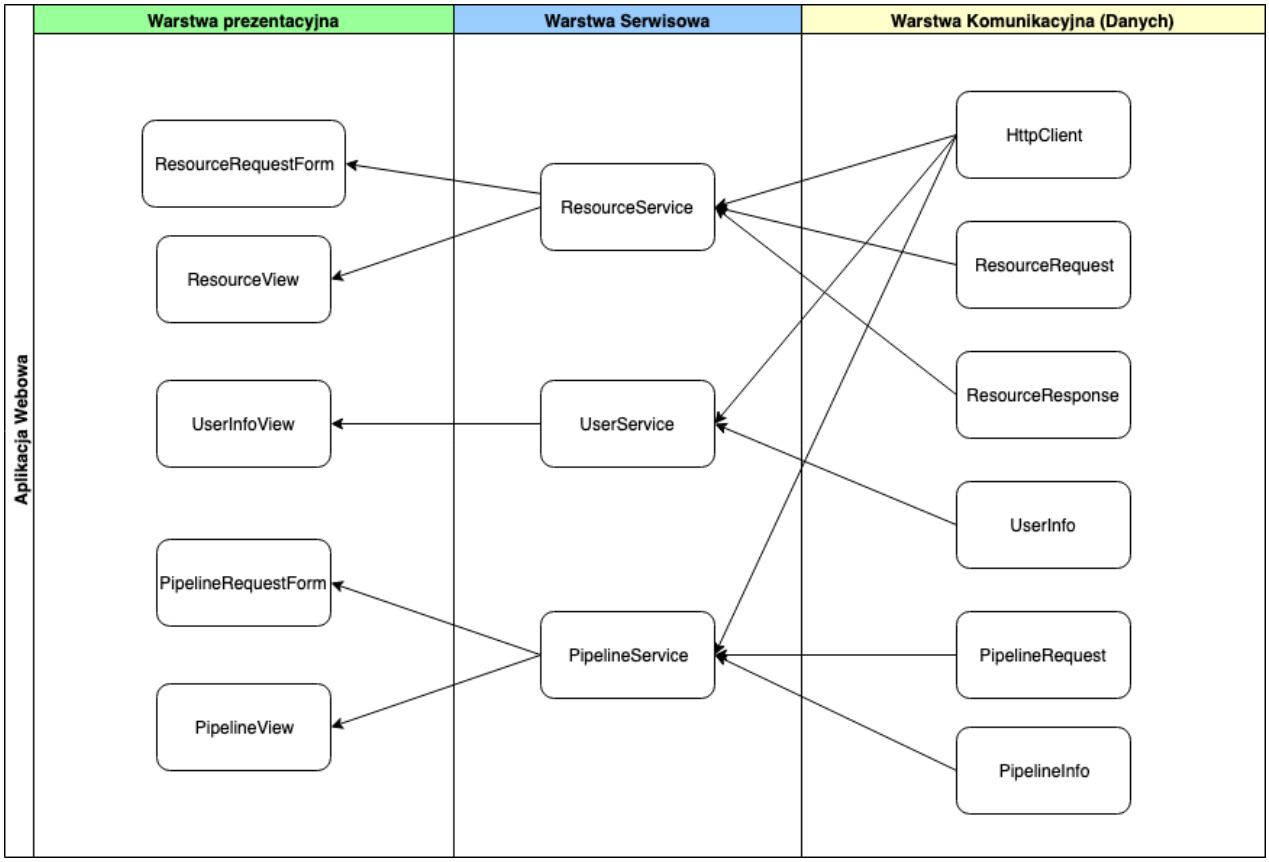
Poniższy rozdział prezentuje implementację aplikacji prezentacyjnej. Zgodnie z ogólnym diagramem architektury, znajdującej się w rozdziale "5.1 Wysokopoziomowa Architektura Rozwiązania", projekt, oprócz aplikacji webowej, posiada jeszcze graficzny interfejs użytkownika, w postaci aplikacji frontend'owej, działającej w przeglądarce użytkownika końcowego. Celem tej funkcjonalności jest zapewnienie prostego w obsłudze a także przejrzystego widoku na dane pobierane poprzez punkty końcowe aplikacji serwerowej.

6.4.1 Architektura aplikacji frontend'owej

Aplikacja webowa została zaimplementowana na podstawie zmodyfikowanego wzorca projektowego "*Business Logic Component (BLoC)*", autorstwa firmy Google [40]. Główną ideą tego wzorca jest rozdzielenie warstwy danych biznesowych pochodzących z REST API, od obiektów dostępnych po stronie warstwy prezentacyjnej. Komunikacja pomiędzy użytkownikiem końcowym a serwisem odbywa się poprzez warstwę pośrednią, w oryginalnym założeniu zaimplementowaną na strumieniach akcji (Event-ów). W opisywanym projekcie, warstwa pośrednia bazuje na encjach pośrednich, będących abstrakcją nad obiektami wykorzystywanymi w celu wyświetlania danych na interfejsie użytkownika. Wprowadzenie wspomnianej abstrakcji redukuje zależności pomiędzy warstwami i umożliwia ułatwioną migracje interfejsu końcowego do innej architektury, na przykład do architektury systemów dostępnych na urządzeniach mobilnych.

6.4.1.1 Wysokopoziomowa architektura rozwiązania

W poniższym rozdziale zaprezentowany jest diagram prezentujący wspomniany wcześniej podział aplikacji webowej na warstwy [Rysunek 26].



Rysunek 26 Wysokopoziomowa architektura aplikacji webowej; Opracowanie własne

6.4.1.2 Warstwa prezentacyjna

Aplikacja webowa wykorzystuje technologię AngularDart, będącą wersją popularnej technologii *Front End*, Angular, bazującej na języku obiektowym Dart. W Angular, aplikacja składa się z modułów, w których treści wyświetlane są poprzez sparametryzowane szablony, oraz logikę odpowiadającą na interakcję użytkownika. Te elementy są komponentami aplikacji. Komponenty są modularne, przez co mogą zostać wykorzystane wielokrotnie. Agregują one logikę zawartą w plikach Dart-owych, dokumentach zawierający szablon HTML oraz styli. Rysunek 27 zawiera przykładową definicję komponentu.

```dart
import 'package:angular/angular.dart';
import 'package:angular_components/angular_components.dart';
import 'package:angular_router/angular_router.dart';

@Component(
  selector: 'welcome-page',
  templateUrl: 'welcome_page_component.html',
  directives: const [
    coreDirectives,
    routerDirectives,
    MaterialIconComponent,
    MaterialButtonComponent
  ],
  styleUrls: const ['welcome_page_component.scss.css'],
  providers: const [
    materialProviders,
  ],
)
class WelcomePageComponent {
  final Router router;

  WelcomePageComponent(this.router);

  static List<ModuleDto> moduleList = [
    new ModuleDto("Resources", "developer_mode",
        "Add new Resources or Edit existing ones.", "RESOURCES", "resources"),
    new ModuleDto("User Profile", "perm_identity",
        "View and edit your user details.", "ACCOUNT", "user"),
    new ModuleDto("Browse Templates", "search",
        "Search facility allows to browse shared resources.", "BROWSE", "resource_browser_component"),
    new ModuleDto("Resource Management", "settings",
        "Manage your intergations, deployments and such..", "MANAGE RESOURCES", "management"),
    new ModuleDto("Place for additional card/module", "ADD_NEW", null, null, ""),
  ];

  void navigate(String url) {
    router.navigate(url);
  }
}
```

Rysunek 27 Definicja komponent strony głównej Zoran.io w bibliotece Angular; Opracowanie własne

Warto zauważyć, że klasy zawierające definicję komponentów są adnotowane dyrektywą `@Component()`. Adnotacja jest sparametryzowana poprzez przekazanie następujących parametrów:

- Identyfikatora komponentu oznaczonego jako `selector`,
- Ścieżki do pliku zawierającego szablon HTML,
- Listy plików zawierających style, które będą wykorzystane podczas interpretacji szablonu przez przeglądarkę
- Zależności (dyrektyw) wykorzystanych w komponencie
- Listy dostawców zależności

Komponenty posiadają architekturę drzewiastą, oznacza to, że poszczególne elementy mogą być enkapsulowane w innych. Dodatkowo pozwala to na zastosowanie mechanizmu zwanego *„tree*

shaking" [41], optymalizującego aplikację poprzez usunięcie kodu, który nie będzie mógł zostać wykonany.

Stan aplikacji może zostać przekazywany poprzez serwisy lub przekazywany pomiędzy komponentami w parametrach nawigacyjnych. Nawigacja pomiędzy komponentami odbywa się poprzez implementację obiektu `Router`. Umożliwia on wyświetlanie odpowiednich widoków zgodnie aktualnym stanem stosu nawigacyjnego. Miejsce wyświetlenia widoku z *router*'a określony jest w szablonie HTML komponentu, tagiem `router.outlet`. Dostępne ścieżki nawigacyjne są zdefiniowane jako statyczne ciągi znaków będące adresem URL.

6.4.1.3 Warstwa serwisowa

Zgodnie z architekturą przedstawioną w rozdziale "5.4.1.1 Wysokopoziomowa architektura rozwiązania", aplikacja posiada warstwę serwisową będącą warstwą pośrednią pomiędzy komponentami Angular'owymi, a warstwą komunikacyjną. Klasy serwisowe mapują wyniki w formie JSON otrzymane od REST API na encje wykorzystywane w aplikacji webowej. Manipulacje danych przez użytkownika końcowego odbywają się na zmapowanych obiektach biznesowych.

Dane z aplikacji webowej są przesyłane do klienta w standardzie JSON, który jest dekodowany z wykorzystaniem standardowej metody oferowanej przez język Dart i mapowany na obiekt `Map<String, dynamic>`. Zdekodowana mapa jest wykorzystywana jako parametr w konstruktorze odpowiednich encji. Encje są oznaczone adnotacją `@JsonSerialize()`, oznaczający, że kompilator wygeneruje metody mapujące obiekty `Map<String, String>` na encję. Adnotację można sparametryzować przekazując flagę `createToJson: true`, oznaczającą, że kompilator wygeneruje także metody umożliwiające serializacje encji do obiektów typu `Map`. Ponowne wykorzystanie wbudowanej obsługi JSON pozwala w prosty sposób przekształcić obiekt `Map` na docelowy format JSON. Zaenkodowane dane są następnie wysyłane do aplikacji serwerowej.

6.4.1.3 Warstwa komunikacyjna (danych)

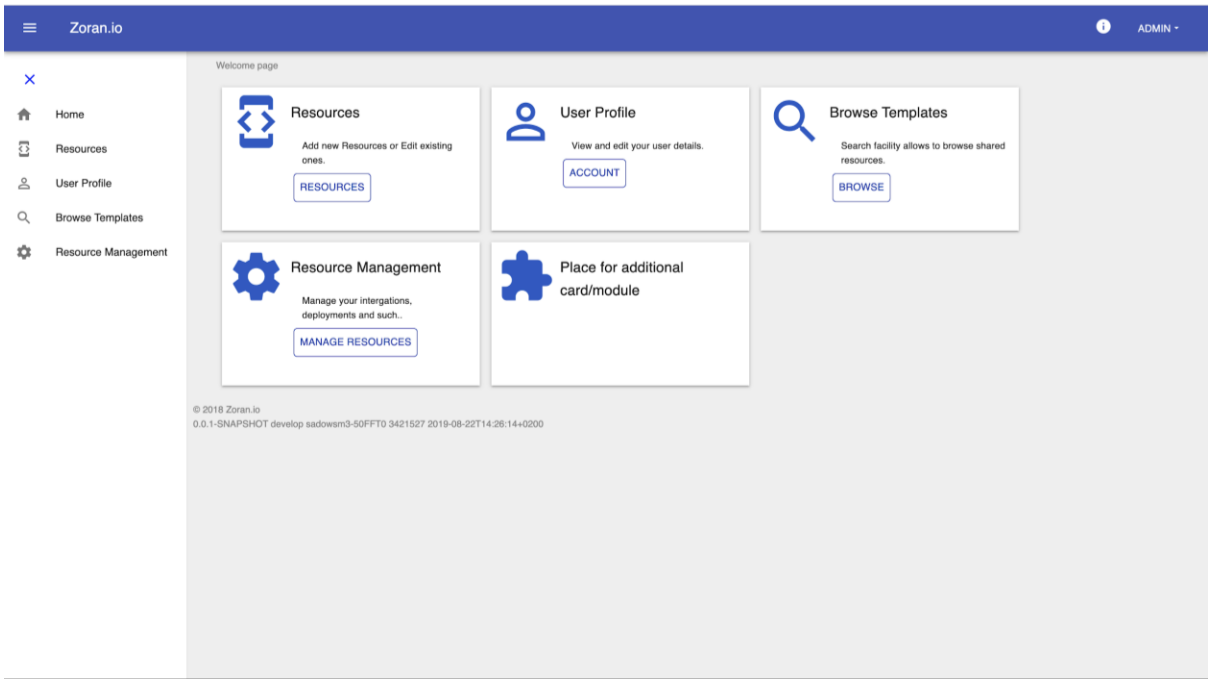
Warstwa komunikacyjna odpowiada za poprawną komunikację pomiędzy aplikacją serwerową a aplikacją webową. W tym celu wykorzystane są serwisy będące implementacją klienta HTTP. Eksponują one generyczne metody mapujące się na punkty końcowe usług wystawionych przez aplikację serwerową umożliwiając komunikację pomiędzy klientem a serwerem.

7. Przykład funkcjonowania systemu

Poniższy rozdział zawiera prezentację wykorzystania systemu w celu zrealizowania opisywanego procesu biznesowego.

7.1 Strona główna

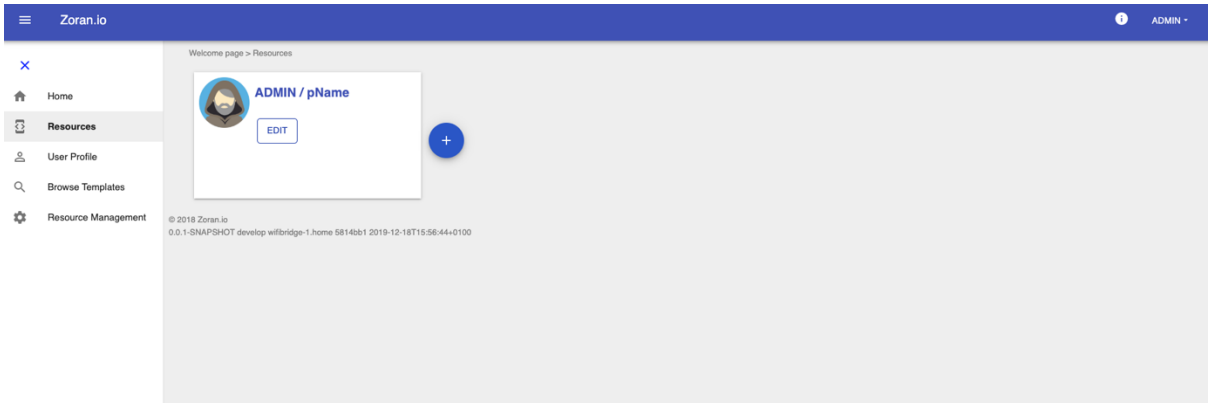
Proces rozpoczyna się w momencie odwiedzenia strony systemu, prezentującego graficzny interfejsu użytkownika. Użytkownik jest prezentowany stroną główną, przedstawiającą listę dostępnych widoków. Rysunek 28 prezentuje opisany widok. W prawym górnym rogu, użytkownik może przejść do swojego profilu lub się wylogować z aplikacji. Menu dostępne po lewej stronie jest elementem komponentu nadrzędnego, dzięki czemu jest dostępne w każdym widoku aplikacji.



Rysunek 28 Strona głowna interfejsu graficznego Zoran.io; Opracowanie własne

7.2 Zarządzanie zasobami

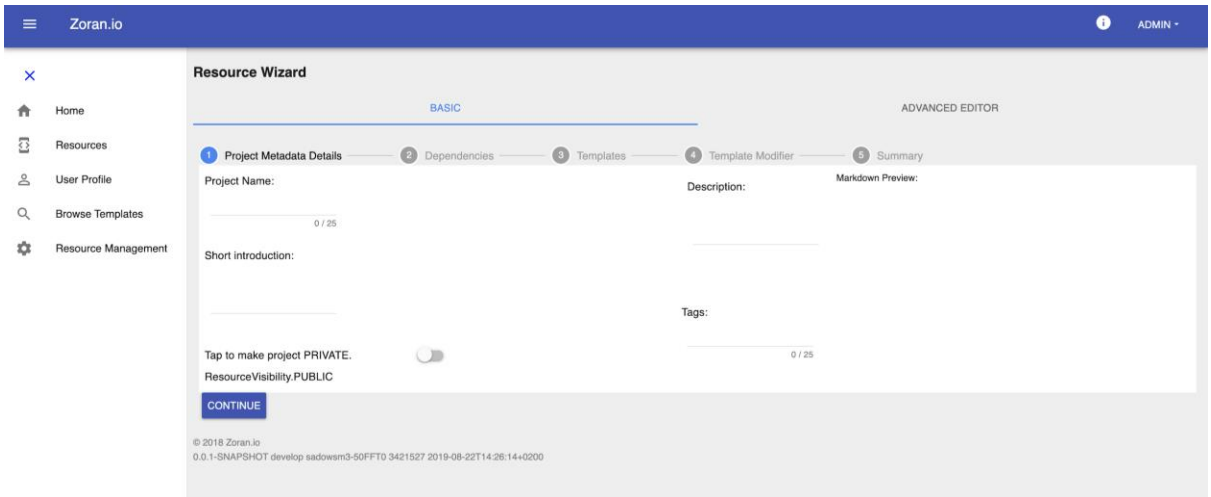
Następnym krokiem w procesie jest przejście do widoku przeglądania i tworzenia nowych definicji zasobu. Ekran ten jest dostępny poprzez nawigacje do zakładki `Resources`. Dostęp do tej strony jest możliwy poprzez kliknięcie przycisku na ekranie głównym lub w menu bocznym. Następnie użytkownik ma możliwość przejrzenia dostępnych zasobów bądź stworzenia nowej definicji zasobu.



Rysunek 29 Ekran przedstawiający listę dostępnych zasobów; Opracowanie własne

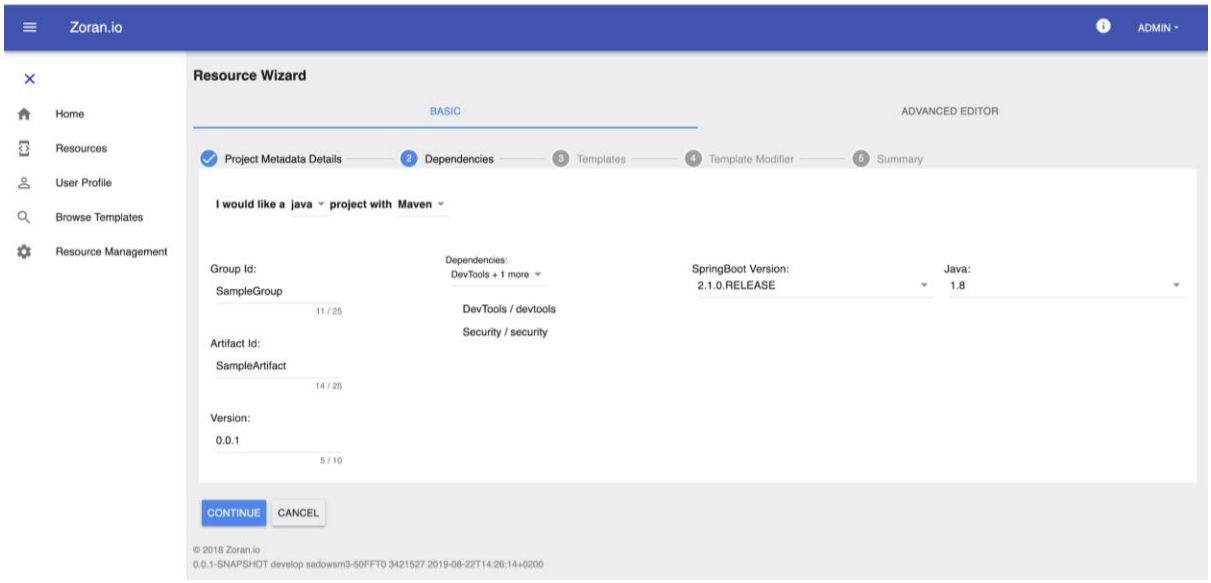
Kliknięcie okrągłego przycisku z ikoną „+", przekierowuje użytkownika do edytora nowych zasobów. System tworzenia definicji składa się z czterech ekranów, w których użytkownik parametryzuje nową definicję zasobu.

Na pierwszym ekranie, użytkownik definiuje nazwę projektu, nazwę grupy oraz identyfikator artefaktu. Dodatkowo, użytkownik może dodać opis zasobu będący dodatkową informacją o funkcjonalności szablonu. Opis może zostać napisany z wykorzystaniem języka Markdown, nadając mu składniejszą strukturę. Aplikacja zawiera wbudowany interpreter, umożliwiający natychmiastowy podgląd opisu.



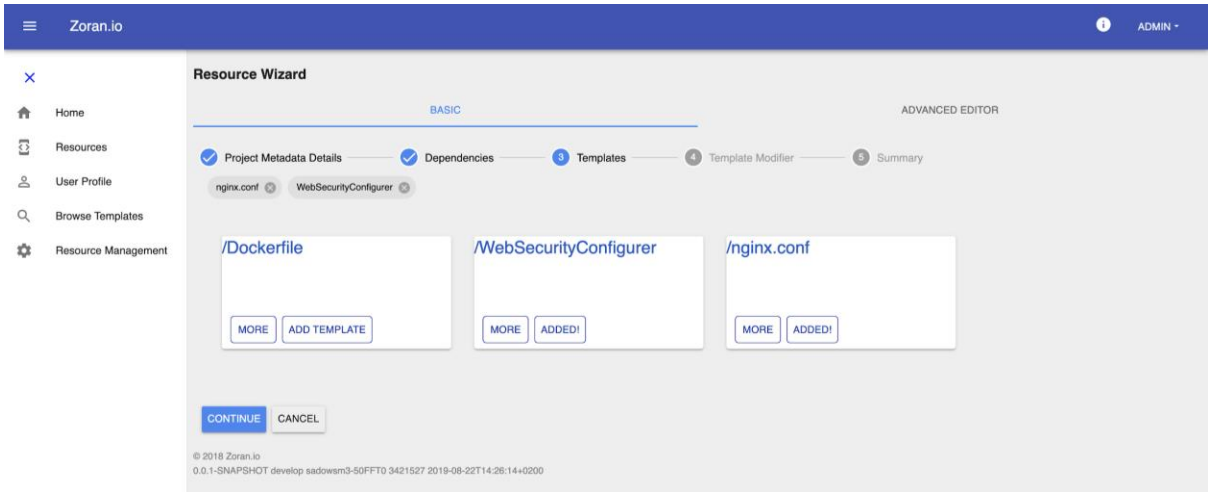
Rysunek 30 Ekran umożliwiający zdefiniowanie podstawowych informacji dotyczących zasobów; Opracowanie własne

Na następnym ekranie, użytkownik ma możliwość zdefiniowania języka zasobu oraz wyboru zależności danego zasobu. Użytkownik może także określić wersję języka oraz wersję wybranych zależności. Po zatwierdzeniu wprowadzonych informacji, użytkownik zostaje przekierowany do następnego ekranu.



Rysunek 31 Ekran umożliwiający zdefiniowanie podstawowych informacji dotyczących zasobów, strona druga; Opracowanie własne

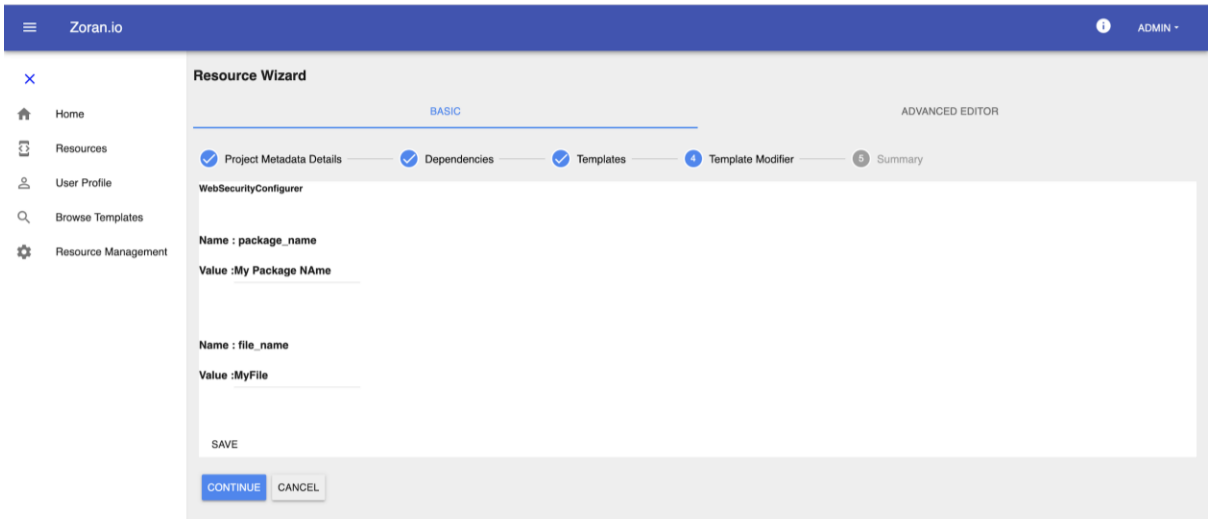
Użytkownik dostaje możliwość wyboru szablonów, które mają zostać wykorzystane w nowym zasobie. Lista pobierana jest dynamicznie z repozytorium szablonów aplikacji serwerowej. Dodanie szablonu do zasobu wymaga od użytkownika kliknięcia przycisku „ADD TEMPLATE”. Dodane szablony są prezentowane w postaci listy chipsów, prezentowanej w górnej części widoku. Istnieje możliwość ich usunięcia, klikając krzyżyk. Po akceptacji szablonów, użytkownik zostaje przekierowany na następną stronę konfiguratora.



Rysunek 32 Ekran umożliwiający zdefiniowanie podstawowych informacji dotyczących zasobów, wybór szablonów; Opracowanie własne

Jeżeli został wybrany jeden lub więcej szablonów, użytkownik ma możliwość ich sparametryzowania. W widoku dostępne są wszystkie pola zdefiniowane w manifeście zawierającego

metadane szablonu. Dodanie wartości do odpowiednich pól pozwala na zdefiniowanie *hash*, czyli słownika wartości, które będą wpisane w odpowiadające kluczom miejsca w szablonie. W wypadku, gdy użytkownik nie wybrał żadnych szablonów lub gdy wszystkie szablony zostały uzupełnione, użytkownik zostaje przekierowany do ekranu, na którym zaprezentowane zostały wszystkie wartości.

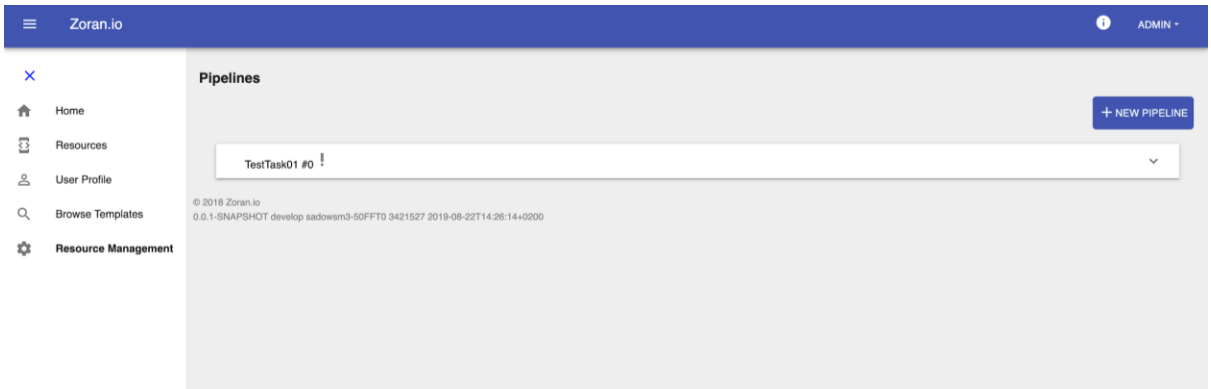


Rysunek 33 Ekran umożliwiający zdefiniowanie podstawowych informacji dotyczących zasobów, definiowanie wartości szablonowych; Opracowanie własne

W przypadku, gdy dane są poprawne oraz zaakceptowane, definicja zasobu jest wysyłana do aplikacji serwerowej w celu zapisania. Następnie użytkownik ma możliwość powrotu do ekranu `Resource Management`, zapewniającego widok na wszystkie zasoby aktualnie przeglądającego użytkownika.

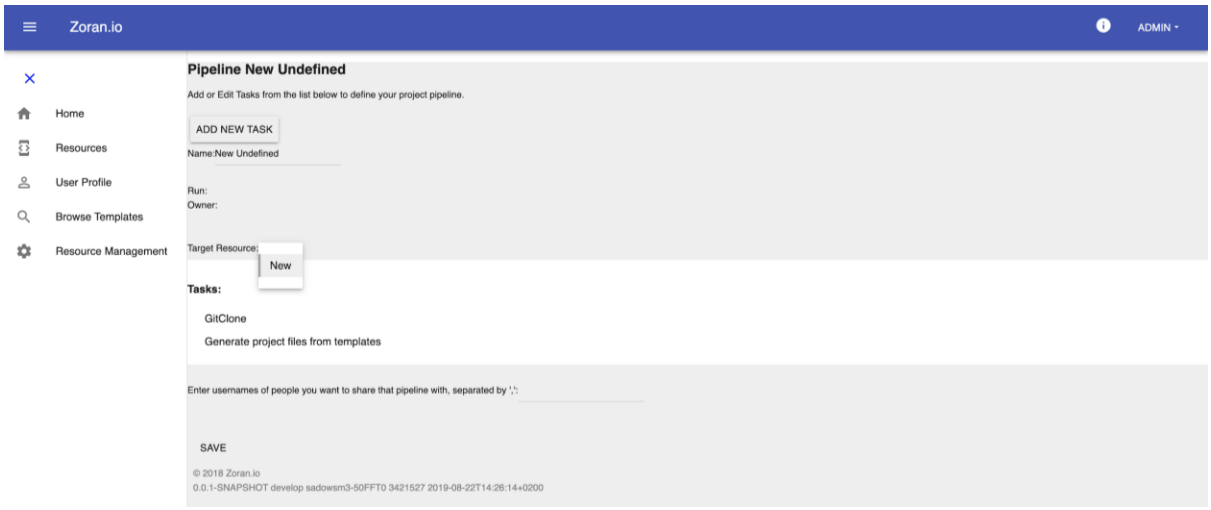
7.3 Zarządzanie ciągami wykonawczymi

W zaprezentowanym oknie użytkownik jest prezentowany listą definicji swoich ciągów wykonawczych, do których posiada dostęp. Zrzut ekranu tego widoku dostępny jest na Rysunku 34.



Rysunek 34 Ekran prezentujący listę dostępnych ciągów wykonawczych (Pipeline'ów); Opracowanie własne

Ciągi wykonawcze opatrzone są informacją dotyczącą ich nazwy, daty ostatniego uruchomienia a także liczbą prezentującą ilość uruchomień danego ciągu. Kliknięcie na wybrany wiersz eksponuje dodatkową opcję, umożliwiającą przekierowanie do widoku edycji ciągu. W przypadku, gdy użytkownik chciałby wygenerować nowy ciąg wykonawczy, ma możliwość przejścia do kreatora poprzez kliknięcie przycisku „NEW PIPELINE” w prawym, górnym rogu widoku.



Rysunek 35 Ekran edycji parametrów ciągu wykonawczego; Opracowanie własne

W kreatorze, użytkownik musi zdefiniować metadane nowego ciągu:

1. nazwę nowego ciągu, po której będzie można zidentyfikować ciąg;
2. unikalny identyfikator zasobu, dla którego dany ciąg jest konfigurowany;
3. listę zadań, które mają zostać wykonane.

Zadania wykonywane są sekwencyjnie, zgodnie z kolejnością wyboru w kreatorze. Lista wybranych zadań jest zaprezentowana w liście w widoku kreatora. W przypadku, gdyby któreś z zadań przyjmowało listę parametrów, odpowiednie pola pojawią się poniżej zadania, umożliwiając ich sparametryzowanie. Dodatkowo użytkownik ma możliwość dodania innych użytkowników, którzy także będą mieli dostęp do opisanego ciągu wykonawczego, identyfikując ich po nazwie użytkownika. Możliwe jest przekazanie tablicy nazw, należy wtedy rozdzielić nazwy przecinkiem. Po zakończeniu kreowania, użytkownik klika przycisk „`Save`” i ciąg jest zapisywany.

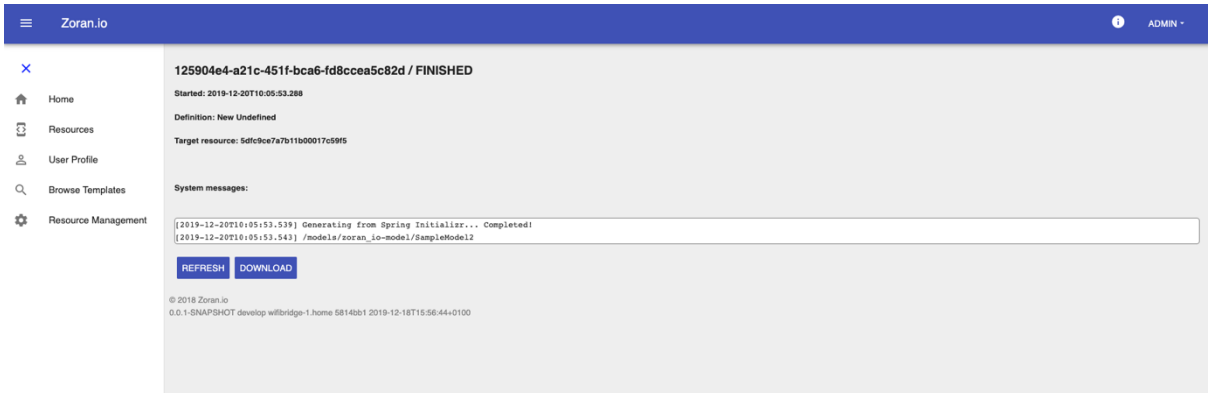
7.4 Wykonanie ciągów wykonawczych

Ciągi są uruchamiane z ekranu głównego `Resource Management`. Z listy dostępnych ciągów, użytkownik wybiera ten, który ma zostać uruchomiony. Po wybraniu, z menu należy kliknąć przycisk “`Run`”. Ciąg wykonawczy jest uruchomiony w aplikacji serwerowej, przekierowując

użytkownika do ekranu zawierającego status obecnego zadania oraz unikalny identyfikator zadania jako parametr w URL. Ponieważ jest to operacja asynchroniczna, widoczny jest także aktualny status wykonania. Cykl życia każdego wykonania składa się z czterech stanów:

1. QUEUED, które oznacza, że zadanie jest zakolejkowane; jest to domyślny, pierwszy stan;
2. IN_PROGRESS, sygnalizujący, że zadanie jest aktualnie procesowane;
3. COMPLETED, stan oznaczający zakończenie wykonywania;
4. FAILED, w momencie, gdy zadanie nie jest zakończone sukcesem.

Aktualny stan zadania widoczny jest w panelu zadania. W konsoli widoczne są informacje zwracane przez poszczególne zadania wykonane wewnątrz aplikacji serwerowej. Przycisk "`Refresh`" umożliwia odświeżenie informacji na ekranie.

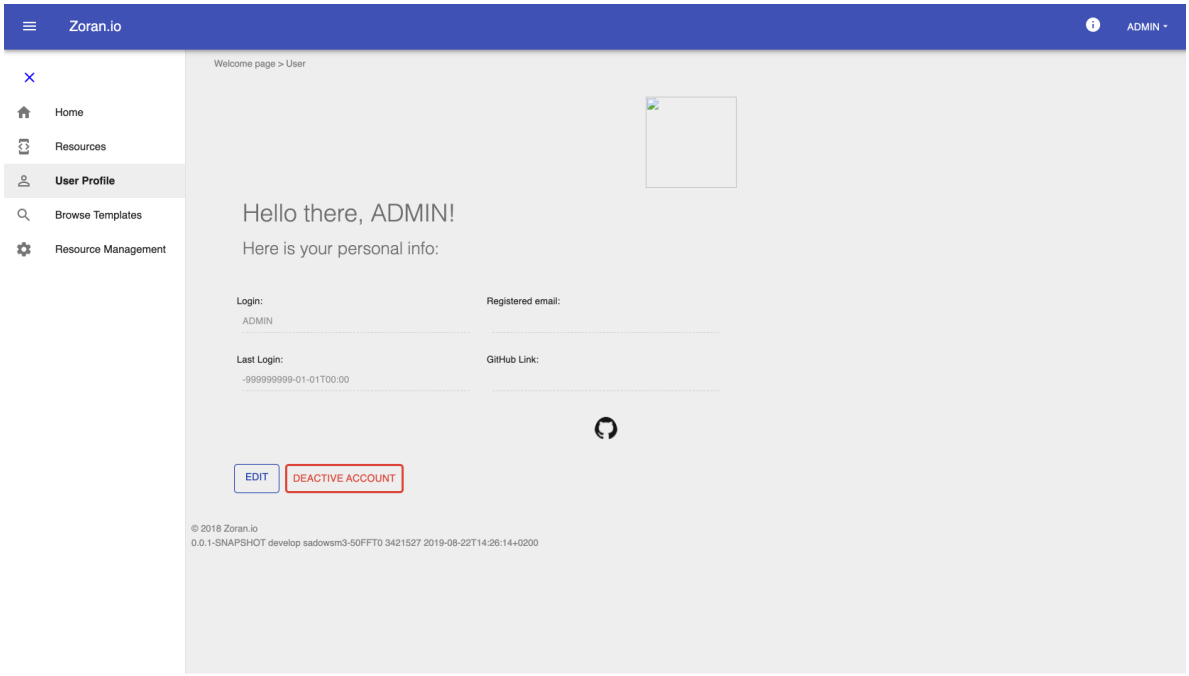


Rysunek 36 Ekran prezentujący status procesowania danego ciągu wykonawczego; Opracowanie własne

Po zakończeniu wykonywania wszystkich zadań w ciągu, na ekranie pojawia się link, umożliwiający użytkownikowi końcowemu ściągnięcie wszystkich wygenerowanych artefaktów. W przypadku, gdy zadania w ciągu nie generują artefaktów, link będzie niedostępny.

7.5 Ekrany pomocnicze

Aplikacja webowa zawiera także ekrany pomocnicze, które nie są częścią procesu biznesowego. Pierwszym ekranem jest ekran informacji o aktualnie zalogowanym użytkowniku. Informacje są wynikiem danych zawartych w encji `ZoranUser`, zawierającym informacje z zewnętrznego IAM.



Rysunek 37 Ekran prezentujący dane aktualnego użytkownika; Opracowanie własne

Drugim ekranem, jest widok na dostępne szablony, pobrane z lokalnego repozytorium modeli. Jest to widok powielony z kreatora nowych definicji.

8. Podsumowanie i wizja rozwoju aplikacji

Poniższy rozdział prezentuję możliwe kierunki rozwoju aplikacji, w przypadku, gdyby projekt byłby kontynuowany. Projekt w obecnym etapie jest w pełni funkcjonalny, realizujący swoje założenia biznesowe. Jednak jest to system zaprojektowany i zaimplementowany spełniając wzorzec projektowy Otwarty-Zamknięty, oznaczający, że istnieje możliwość rozszerzenia jego funkcjonalności.

8.1 Kierunki rozwoju

Aplikacja może być rozwijana w wielu płaszczyznach, począwszy od podłoża infrastrukturalnego poprzez rozszerzenie wspieranych technologii do zwiększenia ilości wykonywanych zadań.

1. Zagadnienia Infrastrukturalne - aplikacja została zaprojektowana z myślą o jak najprostszym procesie wdrażania w firmowe środowisko informatyczne. Technologią, która została wykorzystana w tym celu to Docker i Docker-Compose. W ramach usprawnienia tego systemu, należałoby wzbogacić proces wdrożeniowy o możliwość wdrożenia aplikacji na infrastrukturę chmurową. Zaletami byłoby przeniesienie odpowiedzialności systemowej na serwis, co umożliwiłoby automatyczną skalowalność aplikacji a także redukcję kosztów związanych z serwerami.

2. Rozszerzenie wspieranych technologii - system był stworzony z myślą o wsparciu dla języków autorowi najbliższych, czyli języków bazujących na JVM takich jak Java, Kotlin czy Groovy. W przyszłości system mógłby zostać rozszerzony o wsparcie dla innych języków programowania i ich technologii. Zwiększyłoby to grupę docelową aplikacji oraz umożliwiłoby to wykorzystanie omawianego systemu w wdrażaniu skomplikowanych, wielomodułowych aplikacji bazujących na szerokim stosie technologicznym.

3. Zwiększenie ilości wspieranych operacji - sercem logiki systemu jest serwis procesujący ciągi wykonawcze. W pierwszej iteracji systemu, zaimplementowane są trzy oddzielne zadania, jednak system może zostać w prosty sposób rozszerzony o dodatkowe. Mogą one dotyczyć nie tylko zadań działających na lokalnych danych, jak procesory szablonów, ale mogą być to zadania integracyjne. Przykładem takich zadań może być budowanie projektu z wykorzystaniem Travis-Ci, wdrożenie aplikacji na platformę chmurową czy rejestracja nowej grupy Active Directory czy Vault.

4. Wprowadzenie wyższego stopnia personalizacji aplikacji do użytkownika - ponieważ system wspiera logowanie, jest to szansa na maksymalne zwiększenie indywidualności spełnianych procesów. W przyszłości system mógłby się integrować z systemami służącymi do zarządzania projektami, takimi jak Atlassian Jira, w celu automatycznego aktualizowania statusu zadań oraz zdawania raportów w sprintcie.

Poza kierunkami wspomnianymi powyżej, system powinien wspierać rolę administratora systemu, dając mu możliwość moderowania treści, bez potrzeby manualnej ingerencji w dane bezpośrednio w bazie danych. Przykładowym rozwiązaniem mógłby być panel zarządzania, dający uprzywilejowanemu użytkownikowi prawa nadzoru nad operacjami systemowymi.

8.2 Sugerowane usprawnienia

Opisywany system został stworzony z wysoką dbałością o szczegóły i wysoką jakość oprogramowania. Niemniej jednak, istnieją tematy, które znalazły się poza spektrum projektowym, a są pilne i wymagają rozwiązania w następnych iteracjach systemu.

8.2.1 Zarządzanie sekretami

Pierwszym możliwym usprawnieniem, jest problem zarządzania hasłami, sekretami i potencjalnie licencjami. W obecnej wersji, baza danych przechowuje klucze sesji logowań użytkowników, wraz z ich danymi, w formie czystego tekstu. Sama baza, zabezpieczona jest hasłem, wczytywanym z pliku z właściwościami. To podejście jest niebezpieczne, gdyż istnieje wysokie prawdopodobieństwo sytuacji, w której hasła do bazy danych, lub zapytania HTTP zawierające poufne dane są przechwytywane przez osoby trzecie. W trakcie procesu zbierania informacji w przygotowaniu implementacji projektu przebadane zostały potencjalne rozwiązania tego problemu. Wyzwanie związane z przechowywaniem sekretów, może zostać rozwiązane poprzez wdrożenie rozwiązania od firmy HashiCorp Vault (https://www.vaultproject.io/). Vault umożliwia bezpieczne przechowywanie tajnych kluczy w aplikacjach wdrożonych na infrastrukturze wysokiego ryzyka, na przykład w publicznej chmurze. Każdy projekt zarejestrowany w usłudze, posiada swoją własną przestrzeń, w której przechowywane są hasła do systemów HTTP, CLI i UI. Posiada on wtyczkę umożliwiającą prostą integrację z systemami bazującymi na Spring Boot i jest licencjonowany w trybie Open Source.

8.2.2 Wykrywanie zmian i synchronizacja szablonów

Pliki szablonowe są synchronizowane z lokalnym repozytorium przy każdym starcie aplikacji oraz okresowo, raz na pięć minut. Na obecnym etapie jest to rozwiązanie spełniające założenia minimalne, jednak w najgorszym wypadku, użytkownik musi odczekać pięć minut, zanim będzie w stanie wykorzystać nowo dodane szablony. Rozwiązaniem tego problemu, byłaby implementacja konceptu *Web Hook*, czyli zarejestrowanie usługi w oczekiwaniu na zdarzenie z zewnętrznego serwisu GitHub. Przykładowo, system mógłby oczekiwać na zdarzenie typu *Merge*, czyli zdarzenie, w którym użytkownik dodaje intencję dodania nowych szablonów i jest ona zaakceptowana i wdrożona do repozytorium szablonów. W momencie otrzymania tego zdarzenia, system mógłbym rozpocząć procedurę synchronizacje szablonów. W tym przypadku, tylko użytkownicy pytający o dostępne szablony w trakcie synchronizacji, otrzymaliby nieważne dane, co jest ogromnym postępem względem oczekiwania do pięciu minut.

8.2.3 Generyczność

Pomimo faktu, iż system został zaprojektowany jako maksymalnie generyczny, implementacja zawiera kilka twardych zależności, które mogą potencjalnie ograniczyć użyteczność systemu. Pierwszą zależnością jest serwis GitHub.com. Opisywany system wykorzystuje serwis GitHub.com jako repozytorium zdalne szablonów, a także jako serwis uwierzytelniający użytkowników. Problem pojawia się w momencie, gdy potencjalny klient systemu nie chce wykorzystać domyślnej usługi, ale na przykład, system BitBucket lub GitLab. W tym momencie, wymagane są modyfikacje w kodzie aplikacji, co wiąże się z wysokim kosztem zmian, spowodowany wymogiem ponownych testów regresyjnych aplikacji. W przyszłości należałoby wprowadzić warstwę abstrakcji, umożliwiając użytkownikom na dodawanie własnych modułów autoryzacyjnych i zdalnego serwisu git. Mogłoby się to odbywać poprzez dodawanie własnych bibliotek wraz z potrzebną konfiguracją.

9. Wnioski

Praca przedstawiła problematykę nieefektywnego procesu startowania projektów informatycznych. Zaprezentowane zostały rozwiązania dostępne na rynku, wraz z opisem, zawierającym ich wady, ale i zalety. Po przeanalizowaniu wszystkich istniejących rozwiązań zaprezentowana została propozycja autorskiego podejścia do problemu. Zaproponowane rozwiązanie wyróżnia się niskim progiem wejścia, wykorzystując przystępny graficzny interfejs użytkownika oraz uniwersalnością, oferując wsparcie w wielu dziedzinach włącznie z tymi charakterystycznymi dla danych firm. Poprzez prosty system rozszerzalności, platforma jest bardzo elastyczna, mogąca wspierać nawet bardzo specjalistyczne procesy. Dzięki temu, że jest to aplikacja internetowa, nie wymaga ona instalowania pełnego środowiska informatycznego na maszynie użytkownika, a wykorzystanie systemu kontenerów aplikacyjnych znacznie redukuje poziom skomplikowania procesu wdrożenia.

System został zaprojektowany z myślą o dalszym rozwoju i jest o wysokim potencjale, realizując realistyczne cele biznesowe i optymalizując mało efektywne procesy. Jest to system, który mógłby konkurować z innymi rozwiązaniami obecnymi na rynku.

Bibliografia

[1] S. MF, Software Prototyping: Adoption, Practice and Management, London: McGraw-Hill, 1991.

[2] M. Revell, „OutSystems.com," OutSystems, 07 February 2019. [Online]. Available: https://www.outsystems.com/blog/what-is-low-code.html. [Data uzyskania dostępu: 23 March 2019].

[3] E. Ries, „Minimal Viable Product: a Guide," StartupLessons Learnt, 03 August 2009. [Online]. Available: http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html. [Data uzyskania dostępu: 23 March 2019].

[4] D. Nourie, „Java Technologies for Web Applications," November 2006. [Online]. Available: https://www.oracle.com/technical-resources/articles/javase/webapps-1.html. [Data uzyskania dostępu: 23 March 2019].

[5] Cisco Systems, „What Is Software as a Service," [Online]. Available: https://www.cisco.com/c/en/us/products/software/what-is-software-as-a-service-saas.html. [Data uzyskania dostępu: 23 March 2019].

[6] A. Kumawat, „Cloud Service Models (IaaS, SaaS, PaaS) + How Microsoft Office 365, Azure Fit In," CMS Wire, 10 July 2013. [Online]. Available: https://www.cmswire.com/cms/information-management/cloud-service-models-iaas-saas-paas-how-microsoft-office-365-azure-fit-in-021672.php. [Data uzyskania dostępu: 23 March 2019].

[7] W. F. /. H.-P. H. H. W. F. M. F. L. o. A. E. N. (. O. 2. I. M. C. (. M. 2. S. A. C. F. (. M. 2. I. D. O. (. M. 2. David Booth, „Web Services Architecture," W3C, 11 February 2004. [Online]. Available: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest. [Data uzyskania dostępu: 11 September 2019].

[8] Forrester Consulting, „he Total Economic Impact™ Of PowerApps And Microsoft Flow," Forrester Consulting, 2018. [Online]. Available: https://www.sqlsaturday.com/SessionDownload.aspx?suid=22896. [Data uzyskania dostępu: 2 Listopad 2019].

[9] G. Satell, „The Future of Software Is No-Code," inc.com, 21 April 2018. [Online]. Available: https://www.inc.com/greg-satell/how-no-code-platforms-are-disrupting-software.html. [Data uzyskania dostępu: 11 September 2019].

[10] G. Booch, w *Object Oriented Design: With Applications*, Benjamin Cummings, 1991, p. 209.

[11] C. Drumond, „12 Agile Manifesto principles: a culture, defined," Atlassian, [Online]. Available: https://www.atlassian.com/agile/manifesto. [Data uzyskania dostępu: 11 September 2019].

[12] J. v. d. Hoek, „Pursuing a Full Agile Software Development Life Cycle," 16 Maj 2016. [Online]. Available: https://www.mendix.com/blog/pursuing-a-full-agile-software-lifecycle/, . [Data uzyskania dostępu: 2019 Wrzesień 11].

[13] B. Reselman, „Why the promise of low-code software platforms is deceiving," 31 Styczeń 2018. [Online]. Available: https://searchsoftwarequality.techtarget.com/opinion/Why-the-promise-of-low-code-software-platforms-is-deceiving. [Data uzyskania dostępu: 11 Wrzesień 2019].

[14] K. I. M. D. J. W. Y. N. Paul Vincent, „Magic Quadrant for Enterprise Low-Code Application Platforms," Gartner., 2019.

[15] Gartner, „Gartner,” Gartner, 2019. [Online]. Available: https://www.jp-institute-of-software.com/439889666. [Data uzyskania dostępu: 11 Listopad 2019].

[16] Yeoman.io, „THE WEB'S SCAFFOLDING TOOL FOR MODERN WEBAPPS,” yeoman.io, [Online]. Available: https://yeoman.io. [Data uzyskania dostępu: 11 Wrzesień 2019].

[17] jhipster.tech, „What Is JHipster?,” jhipster.tech, [Online]. Available: https://www.jhipster.tech. [Data uzyskania dostępu: 11 Wrzesień 2019].

[18] Gartner., „Enterprise Application Software,” Gartner, [Online]. Available: https://www.gartner.com/en/information-technology/glossary/enterprise-application-software. [Data uzyskania dostępu: 11 Wrzesień 2019].

[19] Center for Food Safety and Applied Nutrition Center for Veterinary Medicine Office of Regulatory Affairs Center for Drug Evaluation and Research Center for Devices and Radiological Health Center for Biologics Evaluation and Research, „Part 11, Electronic Records; Electronic Signatures - Scope and Application,” 2003. [Online]. Available: https://www.fda.gov/regulatory-information/search-fda-guidance-documents/part-11-electronic-records-electronic-signatures-scope-and-application. [Data uzyskania dostępu: 11 Wrzesień 2019].

[20] Forrester, „The Forrester Wave™: Low-Code Development Platforms For AD&D Professionals, Q1 2019,” Forrester, 2019. [Online]. Available: https://www.outsystems.com/1/low-code-development-platforms-wave/. [Data uzyskania dostępu: 11 Wrzesień 2019].

[21] Appian, „Large Enterprises Succeeding With Low-Code,” 2019. [Online]. Available: https://www.appian.com/resources/forrester-large-enterprises-succeeding-with-low-code/. [Data uzyskania dostępu: 2 Listopad 2019].

[22] OutSystems, „Development With No Limits,” outsystems, [Online]. Available: https://www.outsystems.com. [Data uzyskania dostępu: 2 Listopad 2019].

[23] Salesforce, „Salesforce,” Salesforce, [Online]. Available: https://www.salesforce.com/eu/?ir=1. [Data uzyskania dostępu: 2 Listopad 2019].

[24] Go Java , „Java Powers Our Digital World,” 2019. [Online]. Available: https://go.java. [Data uzyskania dostępu: 2 Listopad 2019].

[25] StackOverflow, „Developer Survey Results 2019,” 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019. [Data uzyskania dostępu: 2 Listopad 2019].

[26] „Learn About Design Patterns Used in Spring Framework,” 1 Styczeń 2018. [Online]. Available: https://blog.eduonix.com/java-programming-2/learn-design-patterns-used-spring-framework/. [Data uzyskania dostępu: 2 Listopad 2019].

[27] Mustache, „https://mustache.github.io/,” [Online]. Available: https://mustache.github.io/ . [Data uzyskania dostępu: 2 Listopad 2019].

[28] E. D. Hart, „The OAuth 2.0 Authorization Framework,” Październik 2012. [Online]. Available: https://tools.ietf.org/html/rfc6749. [Data uzyskania dostępu: 2 Listopad 2019].

[29] MongoDB, „MongoDB,” [Online]. Available: https://www.mongodb.com. [Data uzyskania dostępu: 11 Listopad 2019].

[30] J. Schauder, „Why You Should Avoid JSF,” 3 Listopad 2014. [Online]. Available: https://dzone.com/articles/why-you-should-avoid-jsf. [Data uzyskania dostępu: 11 Listopad 2019].

[31] StateOfJs, „Front-end Frameworks - Overview,” [Online]. Available: https://2018.stateofjs.com/front-end-frameworks/overview/. [Data uzyskania dostępu: 11 Listopad 2019].

[32] Gartner, „Gartner Best Cloud Report 2018,” Gartner, 2018. [Online]. Available: https://www.gartner.com/reviews/customers-choice/public-cloud-iaas. [Data uzyskania dostępu: 12 01 2020].

[33] Docker Enterprise, „What is a Container?,” [Online]. Available: https://www.docker.com/resources/what-container. [Data uzyskania dostępu: 11 Listopad 2019].

[34] https://travis-ci.org, „Test and Deploy with Confidence,” [Online]. Available: https://travis-ci.org. [Data uzyskania dostępu: 11 Listopad 2019].

[35] JavaExplorer, „Chain of Responsibility Design Pattern,” Listopad 2016. [Online]. Available: http://javaexplorer03.blogspot.com/2016/11/chain-of-responsibility-design-pattern.html. [Data uzyskania dostępu: 13 Listopad 2019].

[36] R. H. R. J. J. V. Eric Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1993, p. 151ff.

[37] T. Preston-Werner, „https://semver.org/,” [Online]. Available: https://semver.org/. [Data uzyskania dostępu: 16 Styczeń 2020].

[38] Technopedia Staff, „The 7 Basic Principles of IT Security,” 19 Maj 2017. [Online]. Available: https://www.techopedia.com/2/27825/security/the-basic-principles-of-it-security. [Data uzyskania dostępu: 6 Styczeń 2019].

[39] Swagger.io, „Swagger,” [Online]. Available: https://swagger.io/. [Data uzyskania dostępu: 16 Styczeń 2020].

[40] S. Suri, „Architect your Flutter project using BLOC pattern,” Google, 26 Sierpień 018. [Online]. Available: https://medium.com/flutterpub/architecting-your-flutter-project-bd04e144a8f1. [Data uzyskania dostępu: 16 Styczeń 2020].

[41] P. Rubens, „Can Google Dart Solve JavaScript's Speed and Scale Problems?,” 3 Wrzesień 2013. [Online]. Available: https://www.cio.com/article/2382855/can-google-dart-solve-javascript-s-speed-and-scale-problems-.html. [Data uzyskania dostępu: 16 Styczeń 2020].

Spis Rysunków