Spis treści

1. Wstęp

1.1. Uzasadnienie i wybór tematu

Wzrost znaczenia aplikacji internetowych jako narzędzi pracy dla różnorodnych firm czy też przedsiębiorstw wciąż rośnie. Firmy produkcyjne, dystrybucyjne oraz sklepy korzystają z tych rozwiązań z uwagi na szybkość wymiany informacji oraz jej globalny zasięg. Ponadto brak konieczności instalacji dodatkowych bibliotek również przemawia za wyborem takiej, a nie innej formy wspomagania działalności. Również korzyści finansowe wynikające z uproszczonego modelu dystrybucji oprogramowania, oszczędność czasu i wzrost efektywności stanowią o sile aplikacji internetowych.

Obecnie jednym z najpopularniejszych zastosowań języka Java jest tworzenie komercyjnych aplikacji internetowych. Nie wystarczające są już rozwiązania znane z początków istnienia tego języka, z uwagi na wciąż rosnące potrzeby i wymagania klientów. Rezultatem tego były narodziny nowej gałęzi programów, a dokładniej ich zbioru - szkieletów aplikacji programistycznych, których głównym celem jest odciążenie programisty oraz wzrost jakości dostarczanych rozwiązań. Popyt generuje podaż - to zdanie jest wciąż bardzo aktualne ponieważ potrzeba lepszej jakości rozwiązań była motorem napędowym ich tworzenia i obecnie na rynku istnieje ponad 50 framework'ów, z publicznie dostępnym kodem źródłowym oraz dobrej jakości dokumentacją techniczną, napisanych w języku Java i przeznaczonych dla aplikacji biznesowych.

Obecnie na rynku istnieje co najmniej kilkanaście rozwiązań dedykowanych w mniejszym lub większym stopniu dla warsztatów samochodowych. Wspólnym mianownikiem dla wyżej wymienionych aplikacji jest zestaw funkcji przez nich realizowanych. Planowanie pracy warsztatu: data, godzina, czas realizacji, przypisany mechanik czy też lista czynności do wykonania przy pojeździe klienta, historia zleceń, kartoteka klientów, samochodów to jedynie niewielka część z nich. Niemniej wszystkie znalezione aplikacje łączy również brak otwartego kodu źródłowego oraz brak w pełni funkcjonalnych wersji dostępnych bez opłat. Część z nich to także programy działające na komputerze użytkownika, a nie w przeglądarce internetowej.

Czy potrzeba wypisać programy które znalazłem ?

Wybrany temat obrazuje użycie jednego ze szkieletów aplikacji dla przygotowania kompleksowego rozwiązania wspierającego warsztat samochodowy, które rozwiązywałoby te problemy. Program, będący częścią praktyczną pracy dyplomowej, będzie aplikacją dostępną przez przeglądarkę internetową, bez konieczności instalacji dodatkowych bibliotek lub aplikacji, z otwartymi źródłami oraz dostępną bez opłat.

1.2. Cel i zakres pracy

Głównym celem pracy jest poznanie wybranego szkieletu aplikacji internetowych, zrozumienie zasad pracy z nim w celu przygotowania aplikacji internetowej wspierającej misję przedsiębiorstwa - warsztatu samochodowego.

Pod kątem funkcjonalności, szczególna uwaga zostanie poświęcona zagadnieniom związanym z tworzeniem efektywnego modelu danych oraz bezpieczeństwu wrażliwych informacji takich jak dane osobowe, numery rejestracyjne oraz numery VIN pojazdów, utrzymanie i dostęp do kartoteki pojazdów, mechaników oraz klientów.

Drugim celem pracy jest dostarczyć wiedzy o projektowaniu oraz implementacji aplikacji biznesowych w języku Java przy wykorzystaniu Spring'a. Z tego powodu zagadnieniom takim jak: minimalizacja ilości kodu realizującego warstwę danych oraz ich walidacja, skupienie się na właściwej funkcjonalności aplikacji, przygotowanie generycznych modułów upraszczających dalszy rozwój aplikacji, jej utrzymanie oraz elastyczny model konfiguracji poświęcona zostanie szczególna uwaga. Ponadto przygotowany program ma służyć jednocześnie stworzeniu kilku potencjalnych modułów, które można by potem wykorzystać w innych projektach do podobnych celów.

2. Wymogi funkcjonalne aplikacji

2.1. Głowna funkcjonalność biznesowa

Analiza istniejących rozwiązań pozwoliła na ustalenia, że minimalna funkcjonalność programu realizujące wsparcie dla przedsiębiorstwa prowadzącego warsztat samochodowy nie powinna się ograniczać jedynie do wspomagania procesów: wymiany płynów eksploatacyjnych czy też napraw. Prowadzenia ewidencji klientów, pojazdów oraz ich wzajemnych powiązań przekłada się na efektywność pracy, dając jednocześnie dostęp do danych historycznych. Jest to szczególnie użyteczne w przypadku okresowych wymian oleju silnikowego czy też płynu chłodniczego, kiedy można odwołać się do danych z poprzedniej wizyty danego klienta i uzupełnić ewentualne braki magazynowe. Funkcjonalność programu dla warsztatów samochodowych to między innymi pomoc dla następujących obszarów pracy warsztatu samochodowego:

1. **klienci**:
   a) ewidencja, możliwość rejestracji nowych klientów,
   b) powiązania z samochodem
2. **samochody**:
   a) historia własności,
   b) historia wizyt
3. **zlecenia serwisowe**:
   a) wprowadzenia nowych zleceń,
   b) edytowania zleceń które nie zostały jeszcze zamknięte (nie minął termin ich wykonania)
   c) tworzenie listy zadań dla każdego zlecenia,
   d) usuwania zleceń,
   e) powiązania zlecenie - samochód - klient
4. **powiadomienia** - wewnętrzny system wiadomości:
   a) powiadomienia o nowym zleceniu,
   b) powiadomienie o zbliżającym się zleceniu,
   c) powiadomienia o zmianie stanu zlecenia

2.2. Architektura aplikacji

Aplikacja została zrealizowana z architekturze trójwarstwowej według modelu MVC[1] z uwagi na rozmiar aplikacji oraz mnogość funkcji implementowanych przez program. Niejed-

[1] Model-View-Controller

nokrotnie, z uwagi na szczegółowość modelu danych, wygodniejsze okazuje się przetwarzanie danych na poziomie serwera, do dodatkowo przemawia za wybraną architekturą. Dzięki warstwie kontrolerów można uzyskać wydajny sposób na przekazywanie informacji na linii klient-serwer, wykonanie operacji zgodnych z jej treścią i odesłanie wyniku. Skomplikowana logika biznesowa, gdzie na poszczególne wydarzenia należy reagować inaczej zależnie od parametrów jest powodem dla którego praca z danymi na poziomie klienta mogłaby się okazać niemożliwa lub trudniejsza z uwagi na konieczność pobrania dodatkowych danych z serwera, czyli wykonania następnych zapytań. Złożone przypadki można wydajniej i dokładniej obsłużyć mając dostęp do pełnego modelu danych oraz interfejsów dla operacji bazodanowych, niż pisać skomplikowany kod JavaScript. Niemniej logika biznesowa nie jest jedynym powodem dla którego wybrana została architektura MVC. Zawiłość po stronie serwera jest odzwierciedlona po stronie interfejsu użytkownika, który musi odwzorować związki między obiektami modelu danych oraz zaprezentować możliwe do wykonania akcja.

2.3. Minimalizacja własnego kodu

W momencie projektowanie nowej aplikacji bardzo często początki pracy to tworzenia kodu potrzebnego do wykonywania powtarzalnych operacji takich jak zapis i odczyt z bazy danych, reagowanie na zdarzenia interfejsu użytkownika czy też walidacja danych wejściowych. Nie jest to problematyczne dla małych projektów. Niestety, wraz ze wzrostem ilości linii kodu, złożoności klas czy też skomplikowania logiki biznesowej, okazuje się, że początkowo proste moduły rozrastają się w niekontrolowany sposób. Odwrotnie proporcjonalnie zmniejsza się możliwość na utrzymaniu i rozbudowę kodu aplikacji. Dodatkowe problemy pojawią się wraz z wprowadzeniem do programu cache'owania, konwertowanie danych między niekompatybilnymi typami, transakcji operacji bazodanowych. W tym momencie wzrost ogólnego skomplikowania kodu nie będącego częścią właściwej funkcjonalności jest jedynie jednym z problemów. Rosnąca zależność klas miedzy sobą, głębokości drzewa dziedziczenia dla klas oraz konieczność objęcia wspomnianymi elementami kolejnych obszarów programu przekłada się na zmniejszenie jakości kodu. Z tego powodu głównym założeniem części praktycznej jest wykorzystanie możliwie jak najwięcej funkcji oferowanych przez wybrane biblioteki do realizacji powtarzalnych operacji:

- generowanie zapytań SQL,
- operacje zapisu i odczytu z/do bazy danych,
- obsługa transakcji bazodanowych,
- walidacja danych pod kątem logiki biznesowej,
- konwertowanie danych miedzy niekompatybilnymi typami,
- efektywne reagowania na zdarzanie interfejsu użytkownika,
- minimalizacja ilości kodu linii JSP, skupienie się na właściwych stronach dostarczających interfejs,
- zmniejszenia kohezji między klasami,
- zmniejszenie głębokości drzew dziedziczenia

Podniesienie jakości kodu oraz możliwości jego późniejszego utrzymaniu oraz rozszerzenia o nowe funkcjonalności stanowi o sile każdej aplikacji, z tego powodu był to główny wyznacznik przy projektowaniu i implementacji części praktycznej pracy dyplomowej.

2.3.1. Generyczne moduły

Generyczne moduły są odpowiedzią na podniesienie optymalności oraz możliwości późniejszego rozszerzenia aplikacji. Koszt zaprojektowania niezależnych elementów programu realizujących pewne zadania jest akceptowalny z uwagi na następujące korzyści. Istniejący moduł przeznaczony do realizowania pewnego zadania, gdzie dane wejściowe to obiekty modelu danych, można następnie użyć dla nowych obiektów tego typu bez konieczności modyfikacji, lub nieznacznej zmiany istniejącej funkcjonalności.

2.4. Przewodniki - kreatory nowych obiektów

Biorąc pod uwagę funkcjonalność w zakresie realizacji celów biznesowych jednym z nadrzędnych celów jest dostarczenie możliwości tworzenia nowych obiektów modelu danych. Z uwagi na stopień skomplikowania modelu, jego wzajemnych relacji pojedyncze formularze to zły wybór. Głównym powodem tego powodem jest ilość danych potrzebnych do wprowadzenia, gdzie umieszczenie ich na wspólnej stronie mogłoby być kłopotliwe dla osoby używającej aplikacji. Podzielenia całego procesu na sekwencję kroków według zasady wprowadzenia logicznie ze sobą związanych danych odpowiadających pewnym cechą danego obiektu modelu danych pozwala na reagowania, w kolejnych krokach przewodnika, na dane wejściowe z danego i poprzednich kroków. Ponadto przetwarzanie danych wprowadzonych w danym kroku przewodnika odbywa się na serwerze, dzięki czemu istnieje dostęp do istniejących mechanizmów walidacji oraz innych danych na podstawie których możliwa jest wstępna weryfikacja obiektu na poziomie powiązań z innymi obiektami. Tym samym, możliwość wprowadzenia danych niepoprawnych, których ewentualna walidacja, polegająca na sprawdzeniu, na przykład, długości łańcuchów znakowych, mogłaby nie przechwycić, zostaje zmniejszona.

2.5. Terminarz - organizacja wizyt

Z uwagi na specyfikę aplikacji, jako narzędzia służącego przede wszystkim organizacji pracy warsztatu, terminarz jest funkcją której nie można pominąć. Intuicyjne używanie kalendarza jako terminarza, gdzie zapisywane są poszczególne spotkania, wizyty czy też zadania do wykonania, gdzie ważna jest data jest przydatne tutaj. Terminarz przygotowany w części praktycznej jest posiadać będzie 3 tryby widoku: **dzienny**, **miesięczny**, **tygodniowy**. Z uwagi na częstotliwość wizyt, ilość mechaników a co ważniejsze ilość klientów, a tym samym ich samochodów, 3 tryby pracy z terminarzem są koniecznością. Z poziomu kalendarza będzie można tworzyć nowe wizyty, zmieniać ich godziny lub je edytować.

3. Zaplecze technologiczne - wykorzystane biblioteki

3.1. Wybór narzędzi do pracy

Złożoność aplikacji, opisana w rozdziale 2, wymagała narzędzia lub narzędzi potrzebnych do sprostania poszczególnym wymogom funkcjonalnym. Z prostej przyczyny opcja wielu zewnętrznych bibliotek została odrzucona. Zbyt wielka ich ilość z czasem mogłaby stać się nie zaletą a problemem z uwagi na ewentualne konflikty między nimi lub wzrost liczby zależności przechodnich[1]. Spośród wielu dostępnych szkieletów aplikacji zdecydowano się w pierwszej kolejności na wybór wspierających **Dependency Injection** oraz **Inversion Of Control**. Obie techniki pozwalają znacząco podnieść jakość kodu oraz zminimalizować stopień kohezji oraz zależności między klasami. Jest to możliwe ponieważ odpowiedzialność za sterowanie przepływem programu zostaje przeniesiona na kod użytej biblioteki, co odciąża programistę od konieczności kontrolowania tychże aspektów. Po analizie takich rozwiązań jak **JBoss Seam Framework**, **Google Guice**, **PicoContainer**, **Spring Framework** okazuje się, że wszystkie z nich wspierają wymienione techniki. Niemniej stopień w jakim można by użyć jednego z nich do kompleksowego wsparcia aplikacji praktycznej nie jest już jednakowy. Zaczynając od **PicoContainer** oferującego wsparcia jedynie dla **Dependency Injection**, a kończąc na **Spring Framework** oraz **JBoss Seam Framework** posiadających najbogatszy wachlarz możliwości, najlepszym wyborem okazuje się być **Spring**. Z uwagi na popularność w środowisku programistów stanowi on doskonały kompromis między tym co oferuje, a tym czego wymaga. Pozwalając na przygotowania aplikacji bez żadnego wyraźnego nacisku na technikę czy też inne technologię jest wysoce konfigurowalnym fundamentem dla dowolnego programu.

3.2. Znaczenie szkieletów aplikacji

Internetowe szkielety aplikacji nie są same w sobie bibliotekami programistycznymi. Stanowią one raczej ich zbiór oraz nierzadko i narzędzi mających na celu ułatwienie programiście implementacji własnego rozwiązanie. Bardzo często są one również praktyczną implementacją standardów (tak jak **Seam Framework**) i tak zwanych **best practices**[2]. Jest to szczególnie użyteczne ponieważ nierzadko zdarza się, że programista popełnia błąd

[1] Zależność przechodnia - zależność od jednej biblioteki od innej, a w efekcie zależność aplikacji od tej ostatniej z bibliotek w łańcuchu zależności.

[2] Zalecane i pożądane sposoby realizacji często spotykanych problemów

na pewnym etapie projektowanie lub implementacji pewnego modułu, którego późniejsze konsekwencje wymagają stworzenie niepotrzebnego i nadmiarowego kodu, czego dałoby by się uniknąć, gdyby podążano już wyznaczonymi ścieżkami. Prawdziwe w takim wypadku staje się również zdanie, że jeden błąd generuje kolejne, a te mogą być zalążkiem następnych.

To czym są szkielety aplikacji u samego źródła ich istnienia jest zapobiegać takim sytuacjom poprzez proponowanie już gotowych modułów, które są przetestowane i ciągle modyfikowane przez doświadczone osoby celem dostarczenia jeszcze lepszych rozwiązań[3].

Oprogramowanie zorientowane obiektowo jest doskonałym zobrazowaniem koncepcji wykorzystanie szkieletu jako fundamentu do budowy własnego rozwiązania. Na najniższym poziomie szczegółowości każdy program czy też moduł większej części jest zbiorem klas posiadającym jasno określony zbiór ról - obowiązków, a których obiekty współpracują ze sobą celem dostarczenie gotowego wyniku lub jego części. Wspólnie te obiekty reprezentują pewną koncepcję, dla której został utworzone. W kontekście szkieletu aplikacji internetowych można więc wyróżnić klasy przeznaczone dla kooperacji z bazą danych, odpowiedzialnych za walidacji informacji czy też pomocnych w momencie renderowania widoku. Warto nadmienić, że te zasady są równie ważne dla małych systemów, jak i dla dużych. Niemniej w pierwszym przypadku, gdzie poziom skomplikowania jest niski nie ma potrzeby definiowania wielu poziomów abstrakcji ułatwiających określone czynności, jak na przykład wcześniej wymienione walidacje danych. Niestety z czasem, początkowo prosty system, staje się coraz bardziej skomplikowany i bardzo często programista nie jest już wtedy w stanie zapanować na chaosem oraz dostarczyć zunifikowanego sposobu rozwiązywania powtarzalnych czynności. Z tego powodu dobry framework charakteryzuje się jasno, ale nie sztywno, zdefiniowanymi granicami między poszczególnymi zbiorami funkcjonalnymi. Wprowadzone poziomy abstrakcji, często więcej niż jeden dla pojedynczego celu jak na przykład sposób interakcji systemu i jego klientów, są wynikiem wieloletnich zmian podczas kiedy zidentyfikowano wiele wspólnych problemów i dla których znaleziono rozwiązanie w postaci ram projektowych czy też **best pratices**, będących ostatecznie właściwą esencją znaczenie szkieletu aplikacji[9].

Dobrymi przykładami tutaj będą z pewnością warstwy abstrakcji dla obsługi operacji bazodanowych. Zawierając konkretne implementacje, które już posiadają funkcjonalność odpowiedzialną za wykonanie tychże operacji na praktycznie elementarnym poziomie, a zostawiając właściwą warstwę logikę w kontekście tworzonej aplikacji, odciążają one programistę od przysłowiowego wynajdowania koła od nowa.Praktyczną realizacją tej koncepcji jest na przykład **Spring Data**, które pozwala na napisania kodu, którego głównymi zaletami będzie odseparowanie logiki biznesowej od wybranej bazy danych oraz wyraźny podział na klasy odpowiedzialne za operacje **CRUD** na danych, jak i te wykonujące operacje biznesowe. Inne przykłady to między innymi **EJB**

books.bib

czy też moduł innego szkieletu programistycznego **GWT**

books.bib

wykonującego identyczne zadanie. Warto nadmienić, że również warstwy odpowiedzialne za tworzenie i zarządzanie widokiem (warstwa prezentacji), czy też takie których nadrzędnym celem jest pośredniczenie między widokiem, a danymi są potencjalnymi kandy-

datami do wyodrębnienie pewnego zbioru funkcjonalności jako części składowych gotowe szkieletu aplikacji.

3.2.1. Funkcjonalność szkieletów aplikacji

Do najczęściej implementowanych funkcjonalności szkieletów aplikacji internetowych należą:

- internacjonalizacja oraz lokalizacja tworzonych stron w dowolnym języku oraz wsparcia dla efektywnego przełączanie między nimi
- wsparcia dla technologi widoku innych niż strony **JSP** lub takich, które je wykorzystują ale definiują inny sposób korzystania z nich
- integracja z językiem szablonów innym niż **JSTL**
- walidacja danych
- mapowanie żądań HTTP do tzw. **kontrolerów**[3]
- wsparcie dla popularnych języków transportu danych jak JSON czy też jego odpowiednik XML.

3.2.2. Problemy szkieletów aplikacji

Mimo że szkielety aplikacji znacząco podnoszą jakość kodu aplikacji oraz obniżają późniejsze koszty jej utrzymania nie są doskonałym narzędziem. Większość niedociągnięć, które można zaobserwować związane jest z:

- *złożonością klas* - obiekty klas zaimplementowane w szkielecie aplikacji współpracują ze sobą, wielokrotnie w więcej niż jednym kontekście. Definiowania funkcjonalności danej klasy poprzez użycie pojedynczej klasy abstrakcyjnej lub interfejsu jest rozwiązanie zbyt sztywnym ponieważ często większa część zdefiniowanych metod nie będzie wykorzystana w innym miejscu,
- *skupieniem się na szczególe* - w momencie projektowania klas, tj. kreowania późniejszego celu istnienia ich obiektów, zdarza się, że gubi się obraz całości zbytnio skupiając się na poszczególnych przypadkach,
- *złożoność współpracy* - mechanizmy współpracy obiektów odpowiadających za, na przykład, komunikacją klient-serwer mogą stać się zbyt skomplikowane,
- *trudnością użycia* - brak drobiazgowej dokumentacji może skutkować użyciem szkieletu w sposób niezamierzony przez jego tworców, co może skutkować implementowaniem tzw. **work arounds**[4] lub błędami funkcjonalnymi[9].

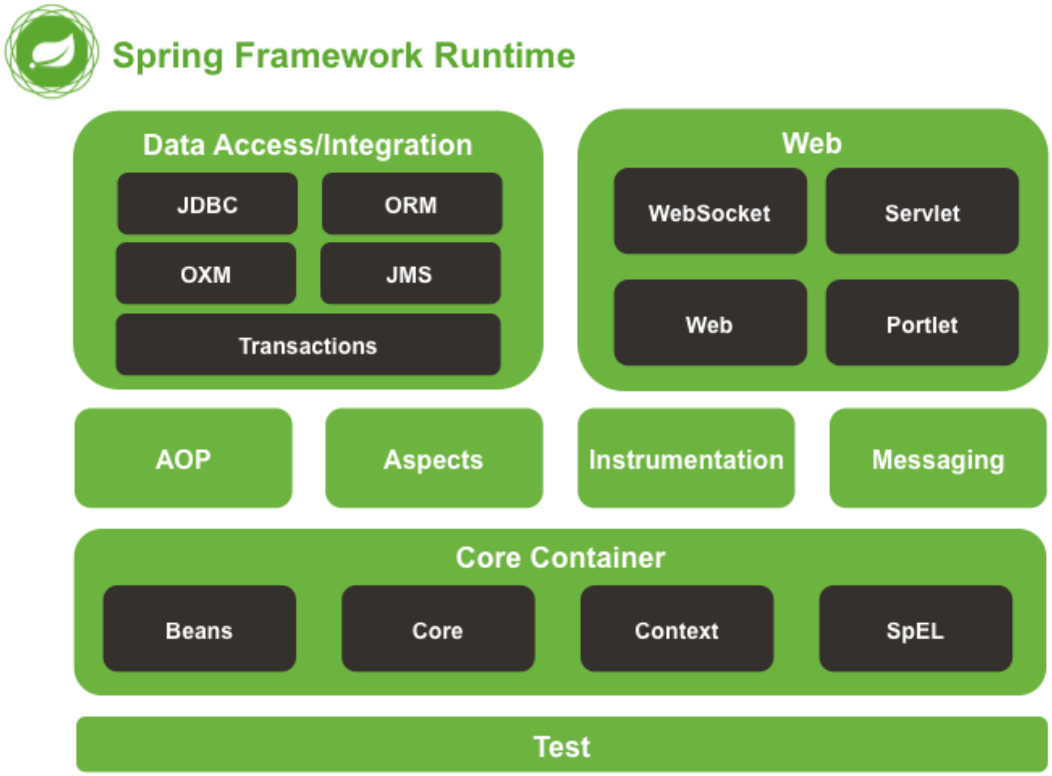
3.3. Spring Framework

Spring jest szkieletem tworzenia aplikacji w języku Java dla platformy Java (Standard Edition oraz Enterprise Edition) opisywanym jako *lekki szkielet aplikacji*. Lekkość

[3] Kontrolery należy rozumieć jako tzw. **POJO** które same w sobie są zwykłymi klasami, ale w kontekście framework'a nabierają konkretne znaczenie jako wykonawcy pewnej logiki właściwej dla danej aplikacji

[4] Technika programistyczna, której celem jest naprawa jakiegoś błędu bądź uzyskanie zamierzonego celu podczas gdy używany framework lub biblioteka nie pozwalają na dotarcie doń.

szablonu odnosi się tutaj nie do rozmiarów całości, ale do filozofii, jaka przyświecała i cały czas przyświeca **Spring'owi**, która nie wymusza konkretnego stylu programowania czy też używania konkretnych zewnętrznych bibliotek, jednocześnie dając możliwość integracji praktycznie dowolnej. Dobrym przykładem jest mnogość opcji do wyboru w przypadku pisania warstwy widoku aplikacji internetowej. Spring oferuje wsparcia czystego JSP (ze wsparciem tagów JSTL) jednocześnie dając możliwość użycia takich bibliotek jak Velocity, FreeMarker, XSLT czy Apache Tiles. **Spring** składa się z ponad 20 samodzielnych modułów



Rysunek 1: *Kontener Spring wraz z modułami*
źródło: [11]

pogrupowanych zgodnie z obszarem, które wspierają:

- **Core Container** - fundament szkieletu na którym oparte są pozostałe modułu. Definiuje funkcjonalność **Dependency Injection** oraz odwróconego sterowania[5] oraz posiada definicję obiektów takich jak **Bean**, **Context**. Ostatecznie zawiera w sobie klasy niezbędne do ładowania zasobów, lokalizacji[6] oraz **Expression Language** - manipulowania obiektami poprzez wyrażanie zapisane czystym tekstem, które później są tłumaczone na odpowiednie wywołania programowe,
- **Data Access/Integration** - określa sposób dostępu dla takich źródeł danych jak bazy danych, pliki XML, zdalne źródła danych dostępne przez protokół JMS. Najważniejszą

[5] IoC - Inversion Of Control
[6] Lokalizacja - internacjonalizacja aplikacja, wsparcie dla więcej niż jednego języka

zaletą jest maksymalizacja wykorzystanie spójnych interfejsów do dostępu do danych i ukrycie ich źródła.
- **Web** - jego zadaniem jest umieszczenia aplikacji działającej w kontenerze Spring w kontekście kontenera servlet'ów[7].
- **AOP** - dostarcza sposoby oraz środki do programowania aspektowego[8].

Modularna budowa jest praktyczną realizacją koncepcji odseparowania obszarów funkcjonalnych. Dzięki temu podejściu **Spring Framework** można użyć w dowolnej konfiguracji, korzystając jedynie z wstrzykiwania zależności lub odwróconego sterowania, ale także możliwe jest dodania kolejnych jego elementów odpowiedzialnych za architekturę MVC dla aplikacji trójwarstwowych lub **AOP**, jeśli w konkretnym przypadku użycia istnieje konieczność dostarczenie pewnego kodu realizującego funkcje tak jak walidacja dostępu na podstawie uprawnień. Implementacja bez użycia **AOP** wymagała by wprowadzenia do każdej z metod kodu sprawdzającego dostęp i podniosła niechciany współczynnik ścisłej zależności między klasami.

3.3.1. Użyte moduły Spring'a

Spring MVC MVC[9] jest wzorcem programistycznym właściwym dla implementowania warstwy interfejsu użytkownika, gdzie nacisk położony jest na ścisłe odseparowanie warstw **widoku**,**dostępu do danych** oraz **logiki biznesowej**. **Spring MVC** jest zorganizowany wokół centralnego servletu `DispatcherServlet` oraz klas z adnotacjami`@Controller` lub `@RestController` - **kontrolerów**. Adnotacje są implementacją konfigurowalnego mapowania adresów. Korzystając z takiego podejścia, programista nie jest zmuszony do definiowania kilku, bądź kilkunastu oddzielnych servletów, z których każdy odpowiada innemu przypadkowi użycia[10] lub pisania własnego silnika, który pozwalał by na generyczne i automatyczne wywołania konkretnych metod lub klas w zależności od podanego adresu lub jego części. Oprócz kontrolerów, `DispatcherServlet` wspiera także klasy, których zadaniem jest implementacja walidacji obiektów używanych w formularzach. Same obiekty nie podlegają żadnym konkretnym regułom wymuszonym przez framework. Są to tak zwane **POJO**. Spring jest odpowiedzialny za ewentualne walidacja, transformacje (z/do XML'a lub JSON'a) lub konwersje. Ostatecznie programista nie jest zmuszony do wykorzystanie konkretnej warstwy widoku, ponieważ **Spring MVC** daje możliwości korzystania z wielu różnych widoków w wielu różnych kontekstach. Zależnie więc od przypadku użycia widok może być rozumiany jako zwykły plik JSP, jako część lub konkretny widok w technologii **Apache Tiles**, a nawet plik PDF. Ponadto istnieje wsparcie dla widoków ładowanych przez **Ajax**, co bezpośrednio przekłada się na zoptymalizowanie częściowe ładowania stron.

[7] Servlet Container - komponent serwera aplikacji internetowej zarządzający między innymi cyklem życia servlet'ów, mapowaniem adresów URL

[8] Aspect Oriented Programming - sposób na zmniejszenie wzajemnej zależności klas oraz objęciem pewną funkcjonalnością pewnych obszarów aplikacji bez jawnego wykorzystania w nich konkretnych klas. Logika zdefiniowana w pojedynczym miejscu jest wykorzystywana w wielu miejscach

[9] **M**odel-**V**iew-**C**ontroller

[10] **Przypadek użycia** (ang. usecase) jest sposobem opisu wymagań aplikacji na poziomie interakcji między użytkownikiem końcowym (aktorem), a systemem

Spring Data - Spring Data JPA Spring Data jest praktycznym rozwiązaniem problemu związanego z implementacją warstwy dostępu do danych. Ów problem odnosi się do pisania szablonowego kodu, którego głównym zadaniem jest wykonanie operacji określanych skrótem **CRUD**[11]. Przy jednoczesnym zapewnieniu prostoty użytkowania, bardzo wysokim poziomie abstrakcji jest to biblioteka dzięki której ilość właściwego kodu, a przez to jego efektywność, realizującego specyficzne wymagania danej aplikacji jest maksymalnie niska. Warto w tym miejscu zwrócić uwagę na generyczne API, które przekłada się na wspomniany poziom abstrakcji, dzięki któremu możliwe jest korzystania z praktycznie dowolnego źródła danych przez zunifikowany interfejs. Nie ważne staje się, czy dane przechowywane są w bazie danych **MySQL** lub **Oracle**, czy też w bazach nierelacyjnych jako **Mongo**. Dzięki temu otrzymany kod jest wysoce przenośny, a zmiana źródła danych wiążę się jedynie z poprawkami w definicji modelu danych. Ponadto elementy takie ujednolicona hierarchia wyjątków, rozszerzalność stanowią wspólnie o sile **Spring Data**.

Spring Data JPA jest pod modułem **Spring Data**, które zawiera klasy oraz interfejsy szczególnie użytecznie jeśliźródło danych aplikacji jest relacyjną bazą danych jak na przykład MySQL. Jednym z tych elementów są repozytoria.Repozytorium jest niczym innym jak obiektem w naszej aplikacji dzięki któremu uzyskujemy faktyczny dostęp do danych i możemynimi zarządzać dzięki wspomnianym operacjom CRUD. Co ważniejsze pojęcie to jest znacznie szersze niż mogłobysię wydawać, zwłaszcza w kontekście operacji wyszukiwania. Poniższy przykład kodu pokazuje jedynie klasę **JpaRepository**. Istniejące tam deklaracje metod są jedynie rozszerzeniem tych zdefiniowanych w dwóch interfejsach, kolejne z siebie dziedziczących, z których **JPA Repository** rozszerza. Niemniej widać, że nawet na wyższym poziomie programiści **Spring Data** zadbali o bardzo wiele możliwych przypadków użycia, co przekłada się na końcową produktywność programisty.

```
@NoRepositoryBean
public interface JpaRepository<T, ID extends Serializable>
        extends PagingAndSortingRepository<T, ID> {

    List<T> findAll();

    List<T> findAll(Sort sort);

    List<T> findAll(Iterable<ID> ids);

    <S extends T> List<S> save(Iterable<S> entities);

    void flush();

    T saveAndFlush(T entity);

    void deleteInBatch(Iterable<T> entities);

    void deleteAllInBatch();

    T getOne(ID id);
}
```

Listing 1: **JpaRepository** interfejs dla operacji bazodanowych na relacyjnej bazie danych w **Spring Data**

[11] CRUD - *Create-Read-Update-Delete* - zbiór czterech podstawowych funkcji w aplikacji korzystających z pamięci trwałej jako nośnika przechowywania danych, które umożliwiają zarządzanie nimi.

. Ponadto programista nie jest zmuszony do implementacji takiego interfejsu, a jedynym jakie zadanie jest stworzenie własnego interfejsu, który będzie posiadał typy generyczne zgodne z jego przeznaczeniem. Klasa zostanie zaimplementowana podczas działania programu poprzez proxy. Repozytoria posiadają także inną bardzo interesującą cechą - automatyczne mapowaniem metod na kwerendy. Jest to pewna alternatywa dla znanych ze standardu **JPA** nazywanych kwerend (named queries). Definicja zapytania SQL pobierana jest z nazwy metody, co oczywiścia wymusza pewną konwencję nazewnictwa. Niemniej jest to koncepcja ciekawa i idealnie nadaje się do tworzenia zapytań odnoszących się do 1 lub 2 atrybutów danego obiektu. Wywołanie jest silnie typizowane dlatego programista ma pewność, że obiekt typu, który go interesuje zostanie zwrócony. Dla bardziej skomplikowanych kwerend istnieje możliwość zadeklarowanie metody i oznaczenia ją adnotacją *Query* z kodem JPQL[5] dla interesującego nasz przypadku[7].

Jak widać **Spring Data** jest dobrze zaprojektowanym modułem, dzięki któremu programista może oszczędzić dużo czasu, którego potrzebowałby na przygotowanie całej warstwy abstrakcji potrzebnej do wykonywania wszelakiego rodzaju operacji, bez skupienia się na źródle swojego problemu - logice aplikacji.

Spring Security W momencie pisania aplikacji w technologii **Java EE**[12] nie można zapomnieć o problemie nieautoryzowanego dostępu do strony lub do niektórych jej części. Sposób uzyskania takiej funkcjonalności jest zależny od kontenera w którym działamy. Inaczej to zagadnienie rozwiązywane jest w przypadku **Apache Tomcat**,a inaczej w przypadku **JBoss**. Oba z nich są serwerami aplikacji Javy, niemniej tak samo jak identyczny jest cel ich istnienia, tak samo różna jest implementacja kwestii autoryzacji. Dzięki **Spring Security** programista może korzystać z niezależnego od kontenera wysoce konfigurowalnego mechanizmu kontroli dostępu do zasobów. W tym miejscu warto nadmienić, że moduł można dostosować do korzystanie weryfikacji użytkowników zarówno z wykorzystaniem bazy danych, stałej listy. Ponadto niewielkim nakładem pracy można dodać mechanizm kontroli znany pod nazwą **Access Control List**. Jest to koncepcja gdzie prawa dostępu (zarówno zapisu, odczytu czy tez modyfikacji) związane są z konkretnym typem obiektu. Wszystkie wyżej wymienione cechy czynią **Spring Security** doskonałym wyborem do ochrony wrażliwych elementów aplikacji internetowej.

Spring HATEOAS [13] Jest to praktyczna implementacja paradygmatu znanego jako **HATEOAS - Hypermedia as the Engine of Application State** dla aplikacji wykorzystujących inny ze znanych koncepcji - **REST**. W przypadku **HATEOAS** duży procent funkcjonalności aplikacji jest oferowany w postaci hiperłączy powiązanych z obiektami. To co wyróżnia to podejście jest fakt, że klienci nie muszą znać funkcjonalności do momentu kiedy jest ona im prezentowana na stronie internetowej. Jest to możliwe ponieważ poza łączami przekierowującymi lub uruchamiającymi konkretne funkcje z serwera przekazywana są etykiety, które w czytelny sposób sygnalizują efekt akcji. W kontekście **Spring HATEOAS** należy wspomnieć, że rozwiązane jest zintegrowane z **Spring Data** i korzysta z tzw. repozytoriów danych jako źródła przez które udostępnia akcje - łącza dla każdego z obiektów. Przykładowa odpowiedź w rozumieniu tego paradygmatu mogłaby wyglądać następująco:

[12] Java Enterprise Edition

```
{
    "name"                : "John",
    "lastname"   : "Doe",
    "links"               : [
        {
            "rel" : "self",
            "href": "http://localhost/customer/1"
        },
        {
            "rel" : "parent",
            "href": "http://localhost/customer/1/parent"
        }
    ]
}
```

Listing 2: Przykładowa odpowiedź serwera na akcję w rozumieniu paradygmatu HATEOAS

Spring Web Flow

"Przemieszczenia się kogoś, czegoś, przekazywanie, obieg czegoś."[10]

SWF[13] jest szczególnie użyteczne gdy aplikacja wymaga powtarzalności tych samych kroków w więcej niż jednym kontekście. Czasami taka sekwencja operacji jest częścią większego komponentu co desygnuje je do wyodrębnienia ich jako re używalnego komponentu. Najlepszym przykładem użycia są w tym wypadku różnego rodzaju formularze służące do rejestracji użytkowników czy tez kreatory nowych obiektów, gdzie umieszczenie wszystkich wymaganych pól na jednej stronie mogłoby zaciemnić obraz i uniemożliwić użytkownikowi zrozumienie działania. Z racji, ze **SWF** jest modułem Spring, jest on w pełni zintegrowany z platformę **Spring MVC**3.3.1 oraz silnikiem walidacji i konwersji, dzięki czemu programista nie jest zmuszony do tworzenia własnych rozwiązań tego typu.

Flow[12] - jest centralnym obiektem modułu, w którym definiowane są kolejne kroki przepływu. Dzięki deklaratywnemu językowi XML definicje są czytelne, a możliwości których dostarcza **SWF** pozwala na kreowania sekwencji w dowolny sposób, łączenie kroków z modelem danych, korzystania z podstawowych jak i zaawansowanych mechanizmów implementacji akcji. Akcje zawierają właściwą logikę biznesową dla konkretnej fazy działania, w której możemy definiowania operacje takie jak pobranie danych wejściowych czy też obsługa wyjątków.

```
<view-state id="entity" view="ui.wizard.NewReportWizard.PickEntity" popup="true">
  <on-render>
    <evaluate expression="pickEntityFormAction.setupForm"/>
  </on-render>
  <transition on="next" bind="true" validate="true" to="pickColumns">
    <evaluate expression="pickEntityFormAction.bindAndValidate"/>
  </transition>
  <transition on="cancel" to="cancel" history="invalidate"/>
</view-state>
```

[13] Skrót od Spring Web Flow

Listing 3: *../NewReportWizard/flow.xml* - deklaratywna deklaracja stanu - kroku dla przepływu w rozumieniu **Spring Web Flow**

. Przykład **??** pokazuje kod XML, który definiuje jeden z kroków - stanów. Powyższy przykład korzysta z klasy org.springframework.webflow.action.FormAction. Przykładowo metoda setupForm może służyć między innymi do wprowadzenia danych wejściowych do kontekstu przepływu.

```
@Override
public Event setupForm(final RequestContext context) throws Exception {
    final MutableAttributeMap<Object> scope = this.getFormObjectScope().getScope(context);
    final Set<RBuilderEntity> entities = this.getReportableEntities(context);
    scope.put(ENTITIES, entities);
    scope.put(ASSOCIATION_INFORMATION, this.getReportableAssociations(entities));
    return super.setupForm(context);
}
```

Listing 4: *PickEntityFormAction#setupForm* - metoda *setupForm* wykorzystywana w definicji kroku **??** do umieszczenia danych w kontekście przepływu.

Spring Test Testowanie w aplikacjach, zarówno biznesowych, ale także i tych których zasięg ograniczony jest do lokalnych odbiorców, jest krytycznym punktem każdego programu. Test, w rozumieniu języków programowania, należy rozumieć jako kod, a dokładniej pojedynczą metodą lub ich zbiór, które wywołują inne metody i weryfikują poprawność zwróconych danych pod kątem zadanych danych wejściowych. Jeśli wyniki różnią się od oczekiwanych oznacza to, że test nie wykonał się poprawny. Dobry test jednostkowy charakteryzuje się:

- powinien być zautomatyzowany i wykonywalny wielokrotnie,
- powinien być łatwy w implementacji,
- raz napisany, powinien być wykorzystywany,
- nie powinien pozostawiać w testowanym systemie pozostałości swojego działania,
- powinien wykonywać się w miarę szybko.

[8].

Spring Test zostało stworzone aby uprościć proces testowania kodu aplikacji. Oferuje wsparcia dla wielu różnych framework'ów przeznaczonych do testów jednostkowych takich jak **JUnit** lub **TestNG**, czy też bibliotekami, których celem jest dostarczanie rozwiązań umożliwiających tak zwane *mockowanie*[14].

Z racji tego, że omawiany tutaj moduł jest częścią **Spring Framework** należy wspomnieć o pełnej integracji z poszczególnymi funkcjami. Elementy takie jak wsparcie dla *Dependency Injection*, pozwalają pisać testy dla obiektów działających w środowisku **MVC** czy też serwisów bazodanowych[15].

[14] Mock - obiekt, który symuluje działanie prawdziwego obiektu. Tworzony jest aby kontrolować przebieg testu w symulowanych warunkach

[15] Serwis bazodanowy - Klasa opatrzona adnotacją *Service*. Różni się ona od klas obsługujących operacje **CRUD** z uwagi na to, że ich głównym zadaniem jest dostarczanie funkcjonalności logiki biznesowej

3.4. JPA - Java Persistance API

JPA - Java Persistance API jest standardem określającym reguły opisu mapowania obiektowo-relacyjnego dla języka **Java**. W aplikacji demonstracyjnej **JPA** została zastosowana do: opisania wszystkich obiektów należących do modelu danych na poziomie nazw tabel, nazw oraz ograniczeń kolumn oraz wzajemnych powiązań między różnymi obiektami, czyli innymi słowy relacji klucz główny-klucz obcy na poziomie tabel w bazie danych. Użycie standardu jako sposobu specyfikacji modelu danych, zostało podyktowane wykorzystaniem tego samego standardu przez inne biblioteki, a w konsekwencji moduły aplikacji.

3.5. Hibernate - Object Relational Mapping

Wykorzystanie **Hibernate** w aplikacji jest w głównej mierze transparentne. Dzięki wykorzystaniu **Spring Data (3.3.1)** silnik ORM może zostać całkowicie wyłączony z operacji bazodanowych. Niemniej jego cecha jako wykonawcy mapowania obiektowo relacyjnego jest wciąż wykorzystywana. W swojej najnowszej wersji **Hibernate**, pomijając część wyjątkowych sytuacji specyficznych dla siebie, opiera się o standard **JPA**(**??**). Takie podejście pozwala ponownie na zaprojektowanie modelu danych którego przenośność jest wysoka, a migracja do innego szkieletu aplikacji ORM ogranicza się do wybrania takiego, który obsługiwałby standard.

3.6. c3pO

c3p0 jest łatwą do użycia biblioteką zaprojektowaną dla **Java**, której głównym zadaniem jest realizacja postulatów zdefiniowanych przez specyfikacją **jdbc3**. Dzięki powyższej bibliotece można w łatwy sposób zdefiniować n - połączeń z bazą danych, gdzie kolejne z nich będą wykorzystywane jeśli kolejka żądań do innych będzie już pełna. Także elementy takie jak zarządzania zajętymi zasobami, ich zwalnianie są obsługiwane przez **c3p0**. Dzięki wsparciu dla szkieletu aplikacji **Spring** konfiguracja okazuje się trywialna i polega na zadeklarowaniu odpowiedniego **bean'a** w konfiguracji XML lub Java [2].

3.7. QueryDSL

QueryDSL jest projektem *OpenSource*, którego użyciu wynika z korzystania z **Spring Data JPA (3.3.1)**. Jego główną zaletą jest możliwość interakcji z bazą danych na wysokim poziomie abstrakcji, gdzie nie jest znane, ponieważ nie ma takiej potrzeby, z jakiego silnika bazy danych korzysta aplikacja. Ponadto, w porównaniu z API Hibernate, **QueryDSL** pozwala na konstruowania silnie typizowanych zapytań lub bardziej ogólnie kwerend **HQL**[16] konstruowanych w oparciu o typ znakowy JAVA.LANG.STRING. Dzięki temu programista nie jest zmuszony do rzutowania oraz wcześniejszego sprawdzania, czy obiekt który chce

[16] Hibernate Query Language - język kwerend SQL zorientowany obiektowo

uzyskać odpowiada jego wymaganiom. Podczas pracy z **QueryDSL** najistotniejszym elementem jest model, który generowany jest z modelu danych podczas kompilacji przez **ATP**. Ważną cechą meta modelu jest jego niezależność od faktycznie używanego silnika danych, a najważniejszą zaletą jest to, że uzyskanym efektem są zwykłe klasy Java, dzięki czemu programista ma możliwość konstruowania kwerend ze wsparciem uzupełniania składni.

3.8. Ehcache

Ehcache jest biblioteką *OpenSource* dostarczającą funkcjonalność pamięci podręcznej dla aplikacja Java oraz Java Enterprise. Główną zaletą posiadania takiego rozwiązania jest odciążanie bazy danych, ponieważ część zapytań oraz ich wyników zapisana jest w pamięci lub w systemie plików. Użyteczność tej biblioteki potwierdza zasada znana, jako **zasada Pareto**, czyli stosunku 80:20. Jeśli weźmiemy pod uwagę 20% obiektów (np. rekordów z bazy danych),które używane są przez 80% czasu działania aplikacji to używając pamięci podręcznych możemy poprawić wydajność aplikacji o koszt uzyskania 20% obiektów.

W ogólnym zarysie idea działania pamięci podręcznej opiera się na tablicy asocjacyjnej, gdzie każdemu z unikatowych kluczy odpowiada pewna wartość. Podczas umieszczanie obiektu do pamięci obliczana jest unikatowa wartość klucza, po której owa wartość będzie identyfikowana. Samą pamięć można opisać jako, miejsce gdzie umieszcza się obiekty, które duplikują inne, ale do których dostęp jest bardziej długotrwały i wymaga dostępu np. do bazy danych, lub do obiektów, które są wynikami pewnych obliczeń. Podczas próby pobrania elementu z cache'u można mówić o pojęciu **hit** - element dla danego klucza zostaje znaleziony oraz o pojęciu **miss**, kiedy element o danym kluczu nie istnieje w pamięci podręcznej.[14]

3.8.1. Warstwa abstrakcji Spring dla cache'owania obiektów

W aplikacji demonstracyjnej **Ehcache** nie jest używany bezpośrednio, pomijając tworzenie domyślnych cache'ów. Cała funkcjonalność odnosząca się do ewentualnego pobierania, umieszczania oraz usuwania obiektów z odpowiadających im pamięci podręcznych realizowana jest na poziomie adnotacji:

Tabela 1: Adnotacje Spring opisujące poziom abstrakcji cache

Adnotacja	Funkcjonalność
Cacheable[17]	Adnotacja odnosi się do konkretnej metody lub całej klasy (w tym wypadku wszystkich zdefiniowanych w niej metod) i wskazuje, że ich argumenty mają być użyte od obliczenie klucza a wartości zwracane przez te metody będą odpowiadać wartościom klucza. Faktyczne umieszczenie obiektu w pamięci podręcznej może być pominięte jeśli istnieje on już w cache'u
Następna strona...	

[17] `http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/cache/annotation/Cacheable.html`

Tabela 1 – kontynuacja...

Adnotacja	Funkcjonalność
CachePut[18]	Działa podobnie jak powyższa adnotacja z tą różnicę, że każdorazowe wywołania metody opatrzonej tą adnotacją powoduje umieszczenie obiektu w pamięci podręcznej
CacheEvict[19]	W przeciwieństwie to *CachePut* oraz *Cachable* wskazuje, że wywołania metody nią opatrzonej spowoduje usunięcie z pamięci podręcznej obiektu o danym kluczu

3.9. Apache Tiles

Apache Tiles to biblioteka umożliwiająca dekompozycję widoku aplikacji na wiele niezwiązanych ze sobą bezpośrednio elementów - płytek [20]. Płytki można potem dowolnie łączyć w konkretne widoki definiując je na poziomie plików XML. Główną zaletą korzystanie z podejścia ze wzorca kompozycji jest wyeliminowanie powielania się elementów stron i zastąpienie ich szablonami gotowymi do użycia w dowolnym miejscu. Dodatkową zaletą użycia tej biblioteki było gotowe wsparcie dla modułu Spring'a – Spring Webflow**??**, gdzie jedną z preferowanych technologii widoku jest właśnie Apache Tiles, a także możliwość prostszego wsparcia dla **partial rendering**[21] stron, gdzie podczas przechodzenia do innego adresu w rzeczywistości zamiast ładować całość strony wraz ze wszystkimi plikami *CSS* oraz *JavaScript*, ładuje się jedynie konkretną zawartość.

3.10. Jasper Reports/Dynamic Jasper

Jasper Reports to kompleksowe rozwiązanie dla języka **Java** wspierające tworzenie oraz generowanie reportów biznesowych dla wielu różnorodnych formatów wyjściowych: **PDF**, **XLS**, **CSV** i wielu innych. Jego główną zaletą jest pojedynczy format przechowywania raportu oraz mnogość formatów reprezentacji, a także ogromna ilość narzędzi oraz bibliotek wspierających tworzenie, modyfikacje raportów.

Z drugiej strony wiele tych narzędzi jest aplikacjami uruchamianymi na komputerach użytkowników, a nie zaprojektowanych do wykorzystania w środowisku aplikacji WEB. Z tego powodu właściwą biblioteką, która została użyta celem utworzenia raportu jest **Dynamic Jasper**. Działając po stronie serwera oraz bazując na danych wejściowych uzyskanych

[18] `http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/cache/annotation/CachePut.html`

[19] `http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/cache/annotation/CacheEvict.html`

[20] Z angielskiego tiles może oznaczać płytkę, w kontekście technologii ApacheTiles należy rozumieć to wyrażenie, jako element, który można wykorzystać w dowolnym szablon

[21] Partial rendering należy rozumieć, jako usunięcie konieczności przeładowania całej strony WEB, a jedynie jej konkretnej części.

od użytkownika pozwala na kompilację do pliku **.jasper*. Możliwe jest ustalenie takich właściwości jak nagłówki, styl, ilość kolumn oraz typ danych w nich przechowywanych.

3.11. Dandelion Datatables

Dandelion-Datatables jest biblioteką zaprojektowaną dla języka **Java**, której zadaniem jest wsparcie dla tworzenie tabel korzystając z użyciem tagów JSP. Instrukcje dostarczane tą drogą uruchamiają proces generowania kodu JavaScript dla wtyczki **jQuery - DataTables**. Nie ma konieczności bezpośredniego pisania kodu JS co pozwala na budowania responsywnych tabel bez znajomości języka JavaScript. **Dandelion Datables** pozwala na bezproblemowe sortowanie, filtrowanie oraz eksportowanie danych.

4. Aplikacja demonstracyjna

4.1. Opis aplikacji wspomagającej warsztat samochodowy

Aplikacja przygotowana jako część praktyczna pracy dyplomowej powstała z uwagi na kilka czynników. Głównym powodem była chęć dostarczenia aplikacji, której zadaniem jest wsparcia dla przedsiębiorstwa prowadzącego warsztat samochody. Ponadto chęć poznania framework'a jakim jest **Spring**, zrozumienia zasad jakimi rządzi pisania się dużej aplikacji zdecydowały o wyborze takiego, a nie innego tematu.

Nadrzędnym celem aplikacji jest wsparcia dla misji przedsiębiorstwa prowadzącego warsztat samochody zajmujący się serwisowaniem samochodów, prowadzącym naprawy oraz przegląd oraz dokonującym okresowych czynności eksploatacyjnych jak na przykład wymiana oleju czy filtrów. Z tego powodu w kolejnych modułach aplikacji została zrealizowana część zarówno serwerowa jak i kliencka, które dostarczają funkcjonalność pozwalających na tworzenie, edycję oraz usuwaniu obiektów biznesowych jak i również przeglądanie informacji o nich.

Duży nacisk został położony na zrealizowanie warstwy serwerowej z uwagi na jej krytyczne znaczenie. Jest ona odpowiedzialna za realizacji postulatów logiki biznesowej, zarządzania prawami dostępu, walidację i konwersję danych. Z uwagi na wymienione punkty wiele elementów zostało dopracowanych, a poszczególne autorskie implementacje pewnych problemów zastąpione lepiej przetestowanymi i wydajniejszymi bibliotekami lub framework'ami.

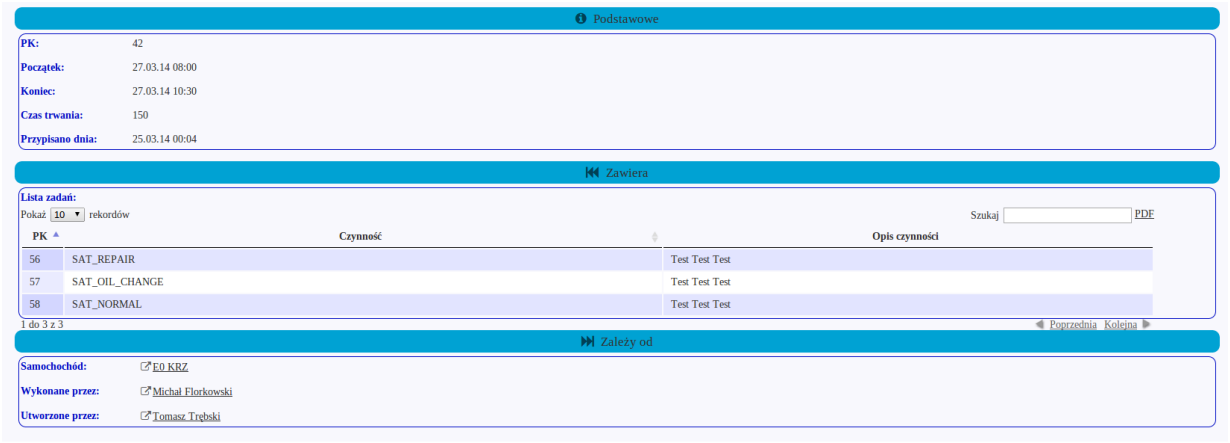
4.1.1. Funkcjonalność aplikacji

Istniejąca funkcjonalność aplikacji pozwala na:

1. dostęp do niektórych funkcji aplikacji dostępny jest jedynie dla osób upoważnionych:
   - aplikacja rozpoznaje czy użytkownik jest zalogowany czy nie dostosowując ilość dostępnych funkcji, w zależności od grupy (grup) w których użytkownik się znajduje,
   - na obecną chwilę weryfikacja następuje jedynie po adresie URI który uruchamia daną akcję
2. możliwe jest tworzenie nowych spotkań oraz raportów,
3. możliwe jest przeglądania terminarza spotkań,
4. istnieje generyczna warstwa stron obiektów domenowych, gdzie jedynym wymogiem jest istnienie obiektu w bazie oraz poprawność adresu

InfoPage - Strony domenowe

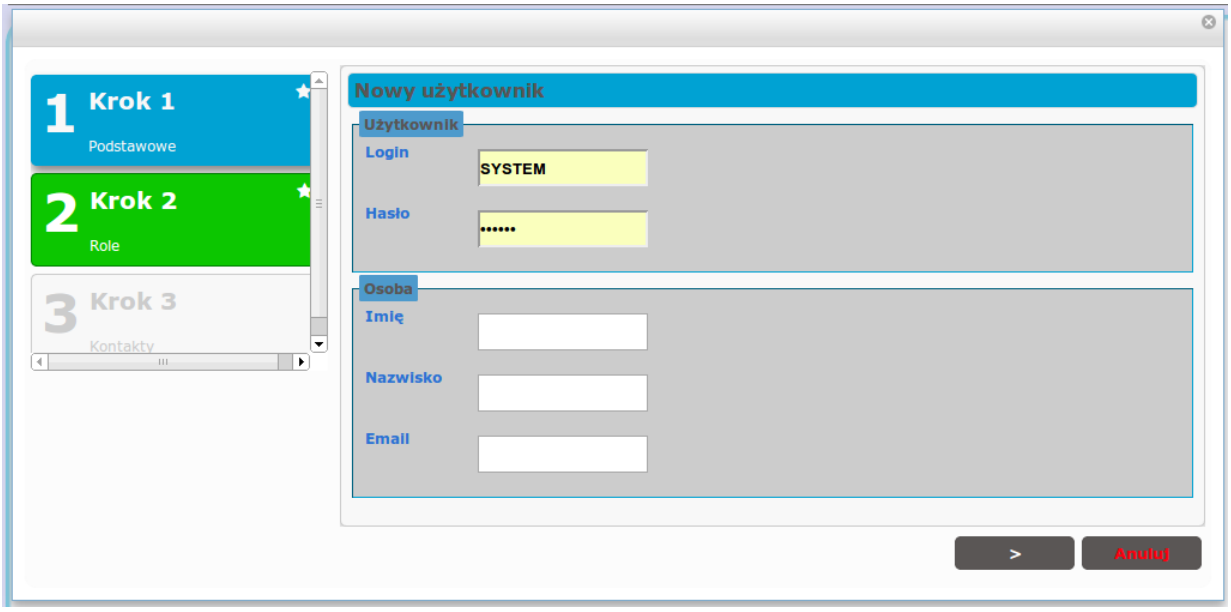
InfoPage jest komponentem prezentującym atrybuty danego obiektu domenowego w postaci strony HTML. Dzięki wykorzystaniu informacji pobranych z meta modelu obiektów domenowych utworzenie nowej strony sprowadza się do zaimplementowania rozszerzenie specjalnej klasy *EntityInfoPageComponentBuilder*. Uzyskanie dostępu do danych i wkomponowanie ich do struktury drzewa nie wymaga w tym miejscu dalszej implementacji kodu o ile dany atrybut istnieje w klasie **Java**. W innym wypadku jest on atrybutem dynamicznym i należy podać kod który zwróci dlań wartość.



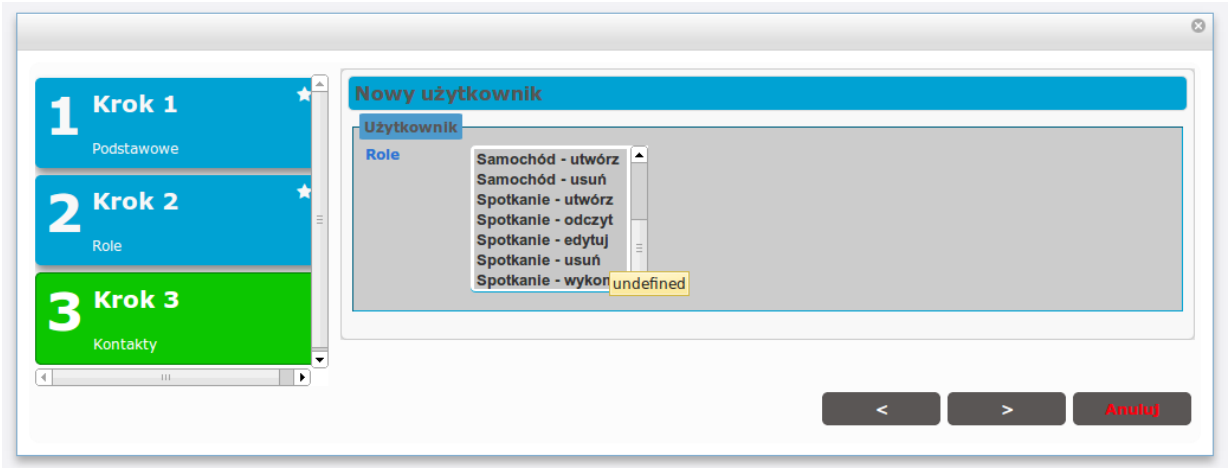
Rysunek 2: *Strona domenowa dla spotkania*

Kreator nowego użytkownika

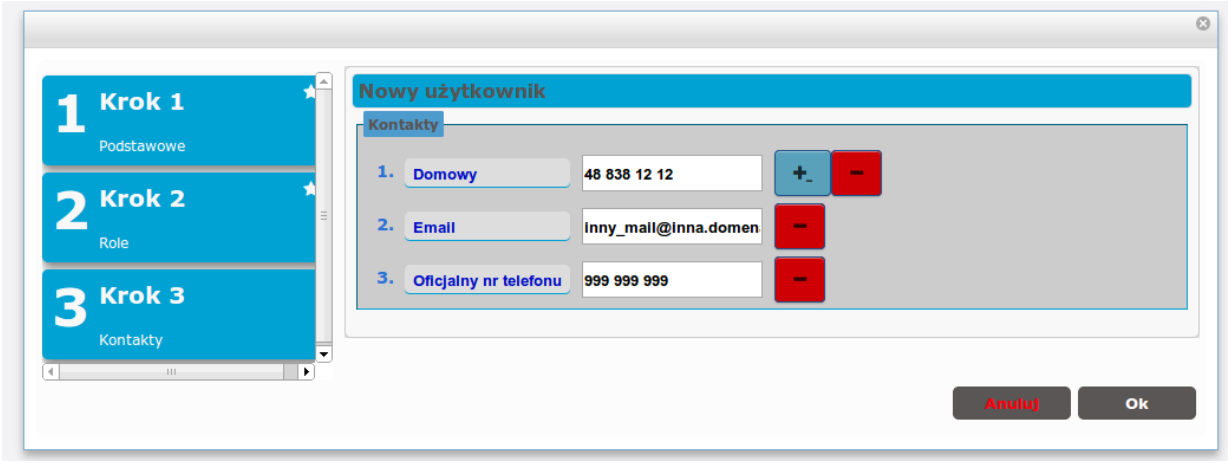
Kreator nowego użytkownika pozwala na tworzenie nowych użytkowników systemu dla uprawnionych osób. Kwestia uprawnień jest tutaj szczególnie ważna z uwagi na to, że w przewodniku wybierany jest zestaw ról. Role do których przypisany jest użytkownik stanowią późniejszą bazę do weryfikacji dostępności funkcji systemu dla poszczególnych użytkowników.



Rysunek 3: *Kreator nowego użytkownika - dane podstawowe*



Rysunek 4: *Kreator nowego użytkownika - uprawnienia użytkownika*



Rysunek 5: *Kreator nowego użytkownika - dane kontaktowe*

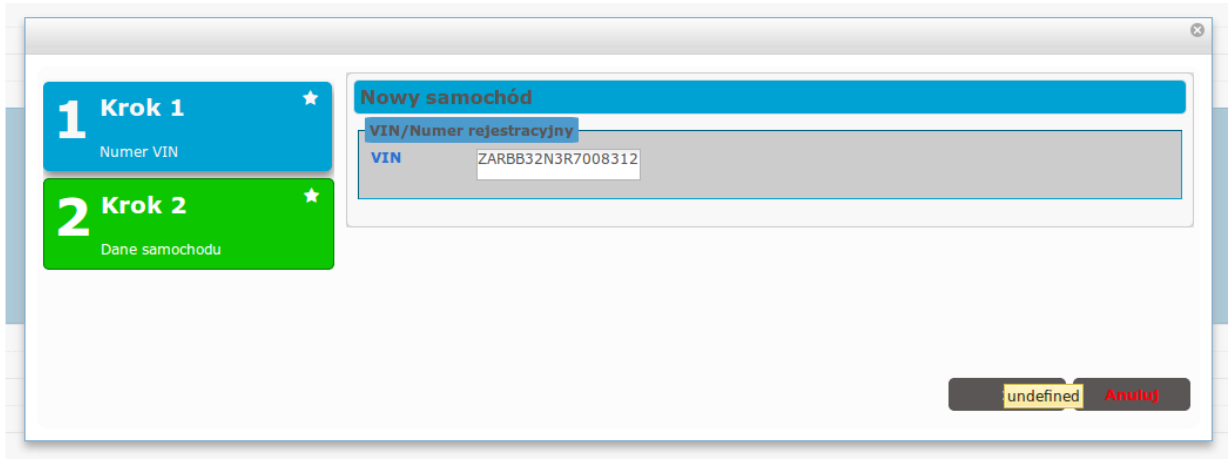
Kreator nowego samochodu

Kreator nowego samochodu został zaprojektowany aby wprowadzić do systemu nowy samochód (centralny obiekt danych). Pierwszym krokiem jest podanie **numeru VIN**, z którego system odczytuje wszelkie możliwe informacje, które da się odkodować korzystając z informacji dostępnych publicznie. Na obecną chwilę są to:

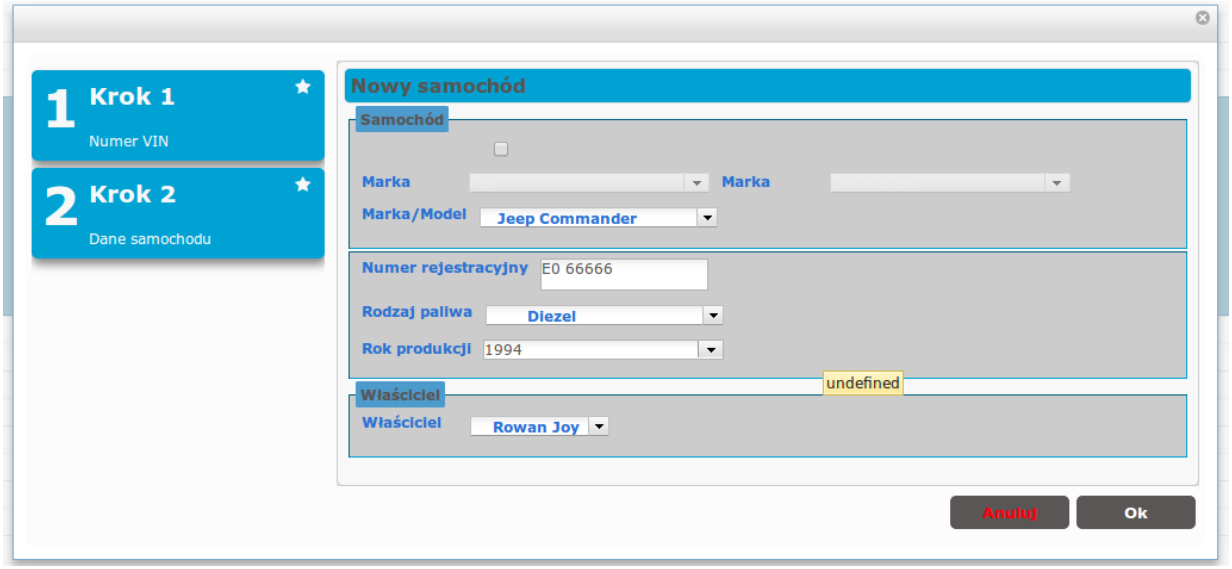
- rok produkcji, zwracany jako lista lat w których samochód mógł być wyprodukowany,
- kraj, w którym samochód został wyprodukowany.

W drugim kroku kreator nowego samochodu podaje takie informacje jak:

- markę oraz model,
- numer tablicy rejestracyjnej,
- rok produkcji,
- rodzaj paliwa,
- właściciela.



Rysunek 6: *Kreator nowego samochodu - numer VIN*

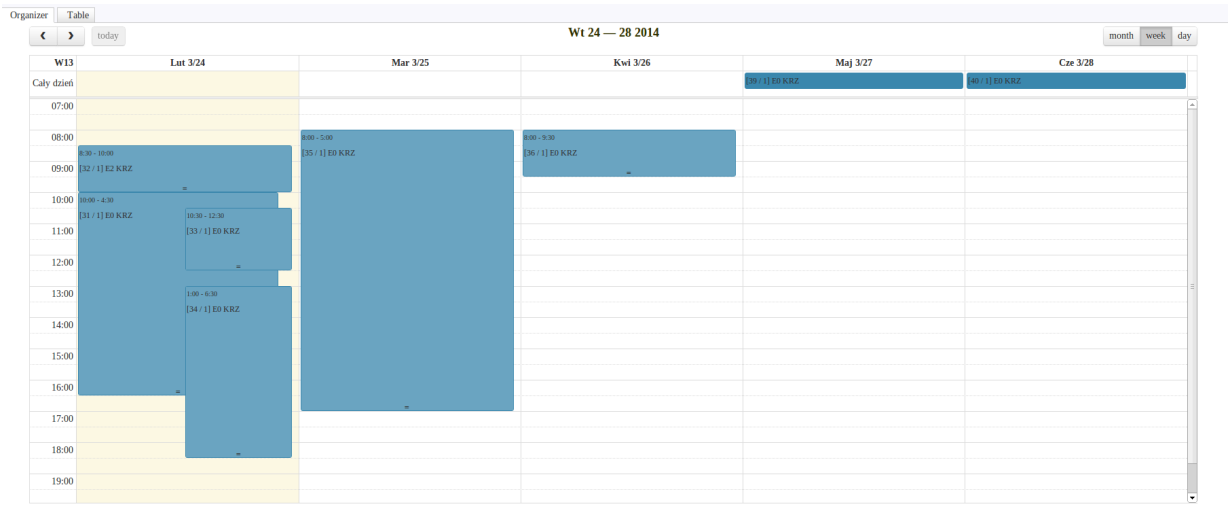


Rysunek 7: *Kreator nowego samochodu - pozostałe dane*

Organizator spotkań

Organizator spotkań jest komponentem wspierającym zarządzania tym typem obiektów. Pozwala na wgląd w listę wszystkich spotkań na 4 rożne sposoby:

- widok w kontekście wybranego dnia,
- widok w kontekście wybranego tygodnia,
- widok w kontekście wybranego miesiąca,
- widok tabelaryczny wszystkich spotkań

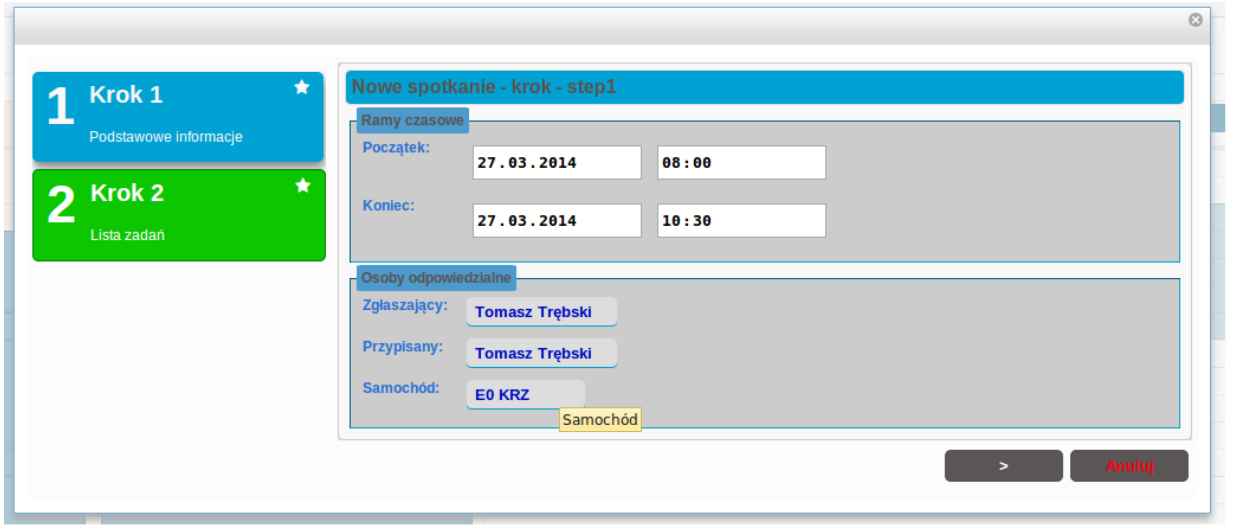


Rysunek 8: *Komponent kalendarza wspierajacy organizację spotkań - orgainizer*

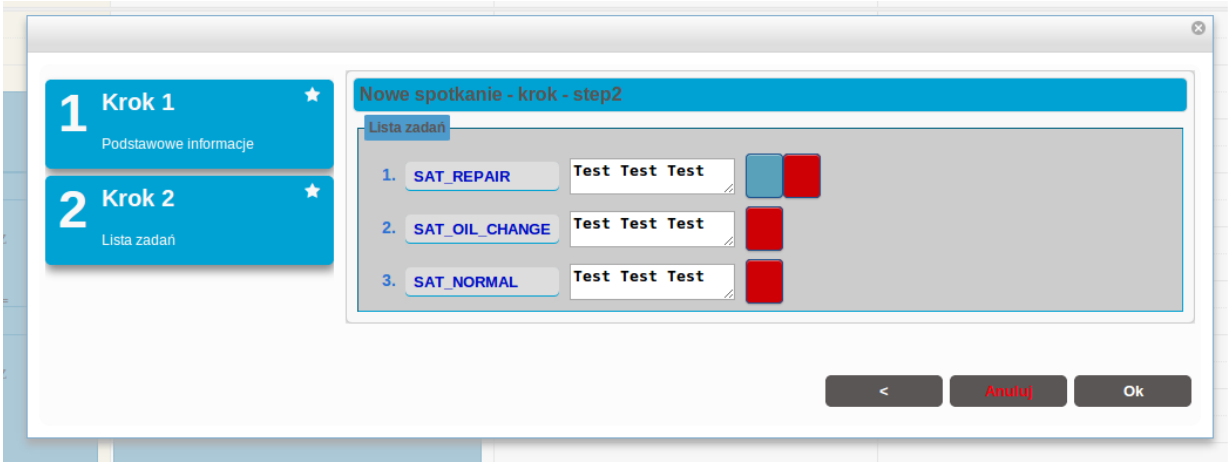
Rysunek 9: *Komponent kalendarza wspierajacy organizację spotkań - orgainizer*

Z poziomu organizera możliwe jest natomiast otwarcie strony domenowej dla wybranego spotkania dostarczającej znacznie więcej informacji niż sam organizator oraz uruchomienie przewodnika utworzenia nowego spotkania. Ważną cechą tego przewodnika jest to, że zakres dat wybrany w organizatorze jest zachowany. Sam kreator został napisany w oparciu o technologię **Spring Web Flow** dzięki czemu na poziomie jednego komponentu można wprowadzić danego takie jak:

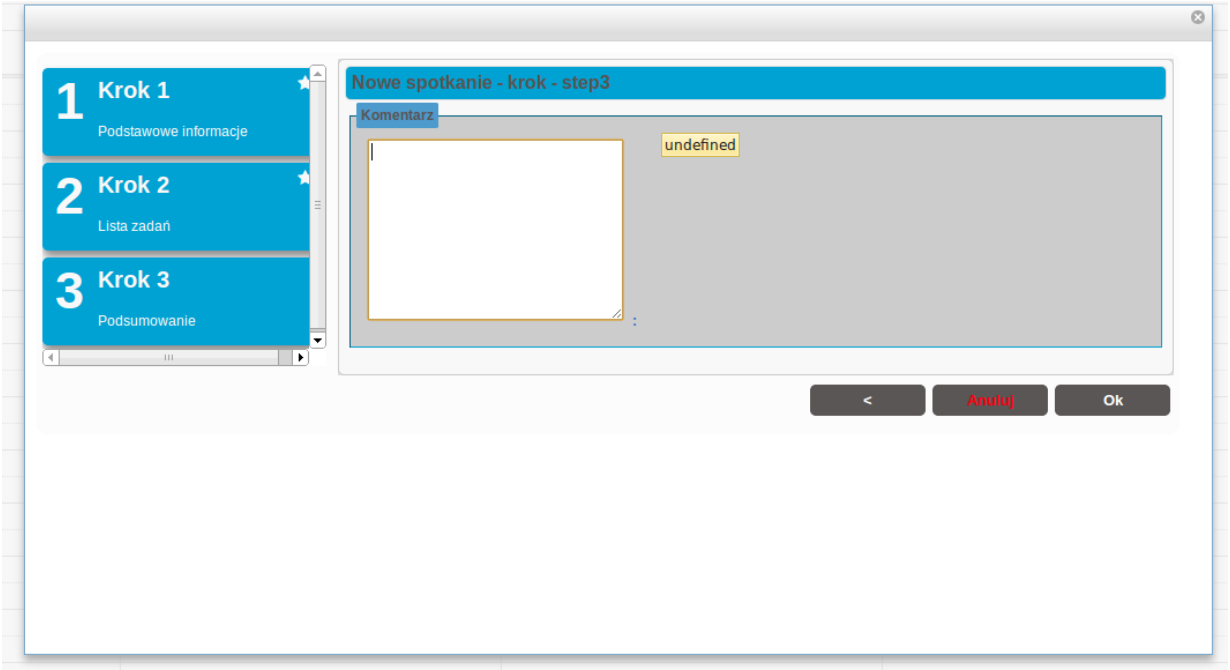
- mechanik, który będzie odpowiedzialny za wizytę,
- mechanik raportujący dane spotkanie, niemniej taką możliwość posiadają jedynie osoby uprawnione do tego,
- wybrany samochód,
- listę zadań,
- opcjonalny komentarz dla wizyty



Rysunek 10: *Kreator nowego spotkania - krok 1*



Rysunek 11: *Kreator nowego spotkania - krok 2*



Rysunek 12: *Kreator nowego spotkania - krok 3*

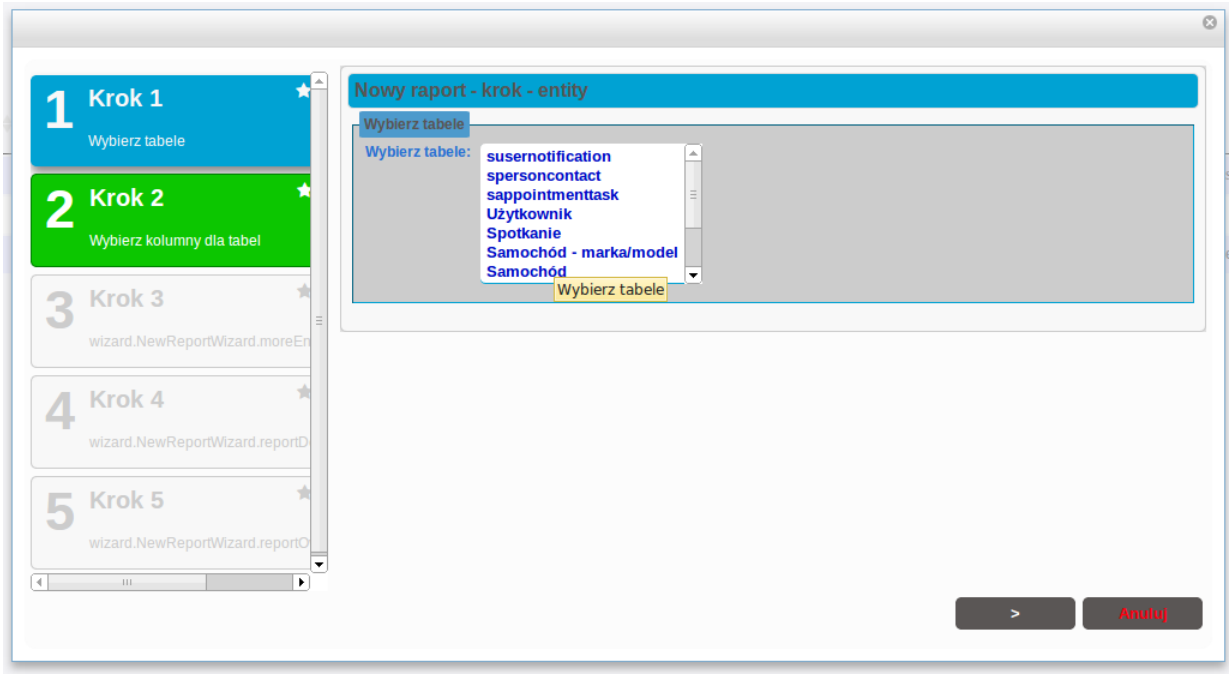
Dane wejściowe są walidowane pod kątem logiki biznesowej:

- termin spotkania:
  - — podane godziny (oraz czas) muszą mieścić się w wybranym zakresie [7-21],
  - — koniec spotkania nie może nastąpić później niż jego początek,
  - — zbyt krótkie (30 minut) oraz zbyt długie (10 dni),
  - — pokrywa się z co najmniej jednym spotkaniem dla mechanika wybranego jako wykonawca
- wybrany samochód:
  - — jeśli z właścicielem samochodu przypisane są jakiekolwiek problematyczne informacje wyświetlane jest ostrzeżenie

ReportBuilder

ReportBuilder jest narzędziem będącym obecnie w fazie rozwoju, niemniej pozwalającym już teraz na tworzenie raportów biznesowych. Dedykowany dla użytkownika daje mu możliwość wybrania zbioru interesujących go tabel, kolumn które zebrane razem stanowią logiczny zbiór używany w dalszej kolejności do konstrukcji zapytania do bazy danych i utworzenia gotowego raportu. Nie udało się znaleźć żadnego rozwiązania które można by wykorzystać bezpośrednio w aplikacji WEB. Byłoby do tej pory źródłem wielu problemów wynikających z konieczności wprowadzenie odpowiedniego, pośredniego, modelu danych przekazujących dane pobrane od użytkownika i kompilacji raportu. Przewodnik składa się z 4 wymaganych kroków:

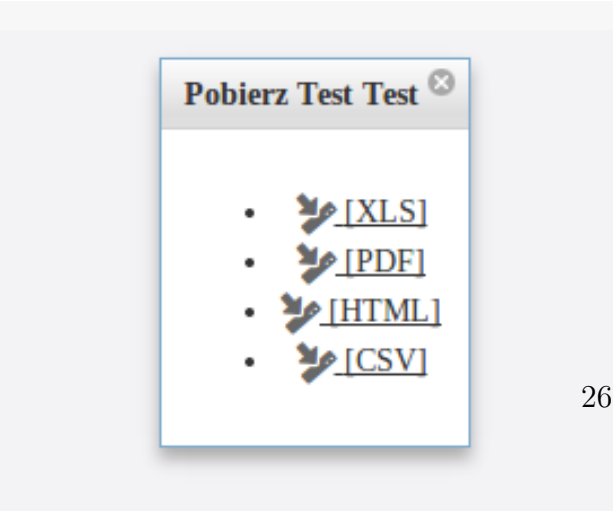
1. wybranie tabeli dla której wygenerowany zostanie raport
2. dostosowanie formatowania kolumn,
3. podania danych opisujących raport: tytuł, podtytuł, opis



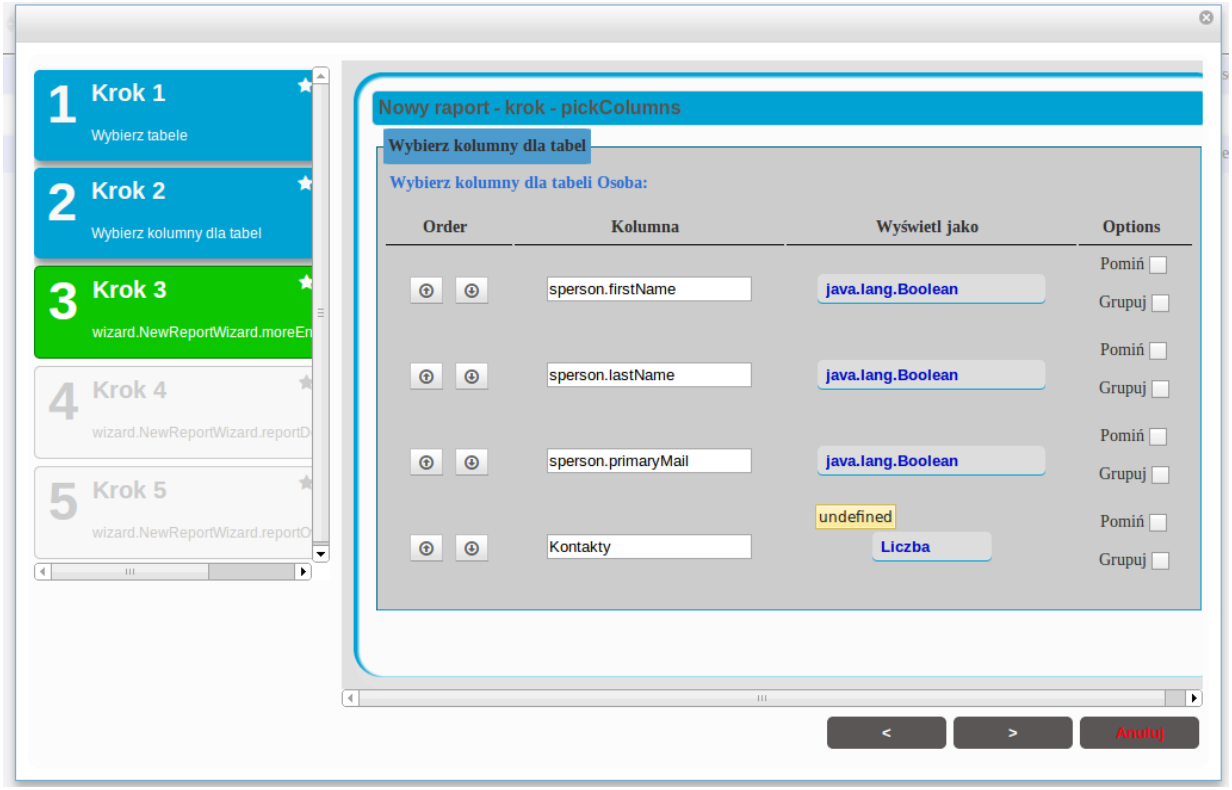
Rysunek 13: *Kreator nowego raportu - krok 1*

W momencie zapisania raportu zapisywane są:

- skompilowany raport w postaci pliku **.jasper*,
- deskryptor raportu w postaci pliku **.json*,
- obiekt domenowy raportu zawierający ścieżki do powyższych plików oraz pozostałe informacje biznesowe jak na przykład tytuł itp.



Rysunek 16: *Generowanie nowego raportu -*

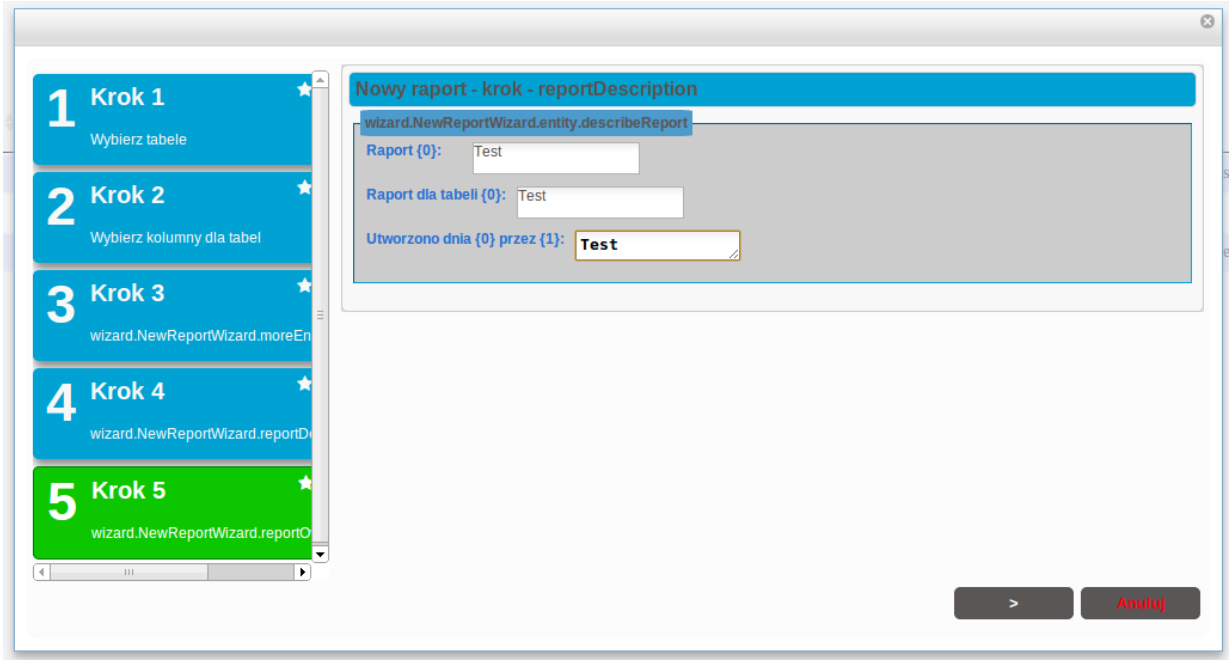


Rysunek 14: *Kreator nowego raportu - krok 2*

Dzięki możliwości wykonywania akcji na poszczególnych wierszach tabeli udało się przypisać dla każdego raportu link uruchamiający jego generowania oraz usunięcie. Usunięcie jest operacją trywialną z punktu widzenia użytkownika ponieważ nie widzi on niczego poza końcowym rezultatem. Z drugiej strony w momencie kliknięcia na przycisk **Generuj** użytkownika dostaje możliwość wybrania końcowego formatu w jakim chciałby zobaczyć swoje dane. Dalsza część funkcjonalności, czyli faktycznego zrenderowania gotowych danych została zaimplementowana z użyciem biblioteki wspierającej **JasperReports** z frameworka **Spring**. Przykładowy raport wygląda w następujący sposób:

4.1.2. Architektura MVC

Aplikacja została zrealizowana w architekturze MVC. Architektura MVC jest znanym wzorcem projektowym szczególnie użytecznym w procesie projektowania warstwy widoku. Korzyści takie jak odseparowanie logiki biznesowej, warstwy modelu danych czy też ostatecznie faktycznej funkcjonalności aplikacji udostępnionej poprzez serwisy. Model danych jest odpowiedzialny za zamknięcie i przenoszenie danych. W rozumieniu omawianego paradygmatu widok służy jedynie do zaprezentowania danych Warstwa środkowa - kontrolery



Rysunek 15: *Kreator nowego raportu - krok 3*

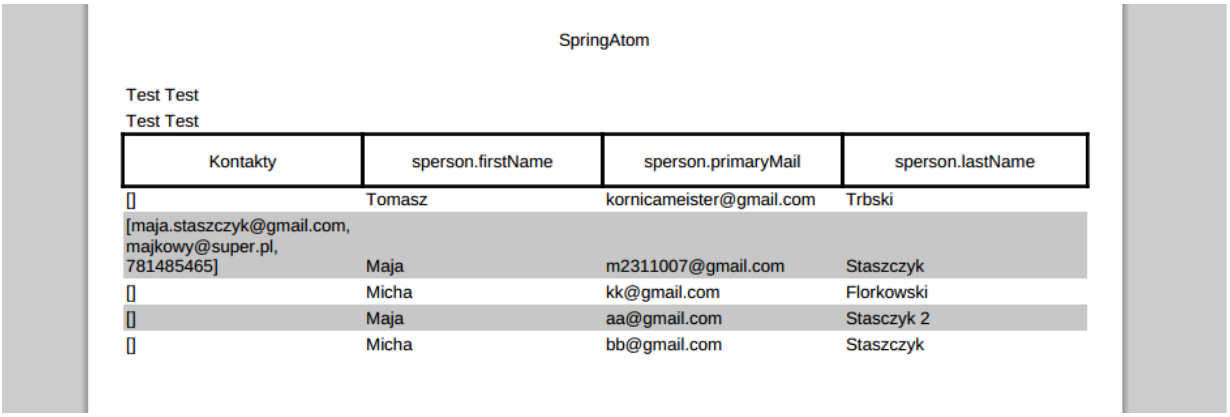
jest często utożsamiana jak najważniejsza część z uwagi na jej położeniu. Kontrolery odbierają żądania wysyłane przez klientów i wywołania serwisów celem uzyskania danych do wygenerowania poprawnej odpowiedzi, aby mogły być zaprezentowane w warstwie widoku[4].

W użytym szkielecie aplikacji - **Spring MVC**3.3.1 - model jest zbiorem *domain objects*[1], które przetwarzana są w warstwie serwisów (service layer), a zapisywane/odczytywane z bazy danych z wykorzystaniem warstwy bazodanowej. Widok jest najczęściej plikiem JSP, kompilowanym później do serwletu, zawierającym tagi JSTL.

Model - Warstwa danych

Warstwa modelu danych, domain objects, została zamodelowana z użyciem adnotacji 3.4 **JPA**. Użycie standardu stanowczo podnosi przenośność aplikacji i zwiększa możliwości modyfikacji kodu w późniejszych etapach rozwoju aplikacji. Wspomniane adnotacja byłyby bez znaczenie bez kolejnego frameworka. 3.5 **Hibernate**. Mimo że **Hibernate** został zaprojektowany do większej ilości celów, niż jedynie zarządzania serializacją oraz deserializacją obiektów z/do bazy danych, jest wykorzystywany jedynie w tym kierunku. Dostarczona funkcjonalność do tworzenia zapytań bazodanowych niestety ma kilka wad, które w kontekście aplikacji były nie do akceptowania. Po pierwsze tworzone kwerendy nie są typizowane co wymusza konieczność rzutowania typów i niskopoziomowych operacji na zwracanych strukturach danych. Na dłuższą metę jest to rozwiązanie nieefektywne i generuje zbyt duża ilość zbędnego kodu, który można by zastąpić lepszym pochodzącym z innych bibliotek. Ostatecznie problematyczne zarządzania sesją oraz błędy wynikające

[1] **Domain Object** - obiekt biznesowy będący encją zawierającą informacje użyteczne z punktu widzenia logiki biznesowej w aplikacji wielopoziomowej



Rysunek 17: *Gotowy raport utworzony przez komponent* ***RBuilder***

z przedwcześnie zamkniętej sesji lub nieistniejącej generują dalsze problemy w momencie serializacji danych do formatu przenośnego przez sieć internet takim jak na przykład JSON. Niemniej czynnikiem decydującym był brak możliwości integracji z pozostałymi modułami szkieletu aplikacji **Spring**.

Powyższe problemy został rozwiązane przy użyciu kolejnych bibliotek. Jest to praktycznie wykorzystanie reguły mówiącej o tym, żeby nie wynajdować koła od nowa. Poniższe punkty opisują wybrane biblioteki i korzyści jakie z tego wynikają.

Spring Data JPA Głównym celem jaki przyświeca istnieniu **Spring Data JPA** jest usprawnienie pracy z repozytorium danych niezależnie od jego rodzaju - baza relacyjna, nierelacyjna. Ponadto programista zostaje odciążony od konieczności wykonywania powtarzalnych czynności i może skupić się na rozwiązywaniu faktycznego problemu.

Spring Data Envers Envers wspierają tak zwane wersjonowanie obiektów, dzięki któremu programista ma dostęp do pełnej, a także konfigurowalnej, historii ich zmian. Dużym plusem w korzystaniu z tej technologii jest to, że wywołania logiki pobierającej instancję danego obiektu z bazy przekłada się na uzyskania jego ostatniej nie zmodyfikowanej wersji. Dzięki temu bez dodatkowego nakładu pracy możliwe jest zaprezentowania użytkownikowi ostatniej iteracji danego obiektu domenowego.

QueryDSL Mimo że **QueryDSL** nie jest faktycznie modułem **Spring**'a, ani nie jest też bezpośrednio związana z firmą odpowiedzialną za niego, to jest na tyle interesująca, że doczekała się swojej integracji z nim. Wsparcie dla takich elementów jak silnie typizowane kwerendy, które można tworzyć bez znajomości języka SQL czy też nużących i podatnych na błędy łączenia czy też dzielenia łańcuchów znakowych i ostatecznie znacznie niższa wrażliwości na zmiany przeprowadzane w modelu danych, czynią z tego frameworka wręcz idealna narzędzie do pracy z bazą danych. Ponadto wspomniana integracja ze szkieletem aplikacji **Spring** nie wymaga implementacji istniejących interfejsów czy też przerabiania swojego kodu. Jedyne co należy zrobić to rozszerzyć interfejs swojego repozytorium o ten odpowiedzialny za wywoływanie kwerend w rozumieniu **QueryDSL**, a Spring wykona resztę sam.

Model danych jest w niektórych przypadkach wersjonowany. Oznacza to, że dla wybranych obiektów **Hibernate** trzyma ich historię zmian w bazie danych. Modyfikacja każdego lub jedynie wybranych pól, jest to zależne od konfiguracji danego obiektu domenowego, powoduje nie tyle zapisywanie zmian do bazy co utworzenie nowych rekordów w tabelach:

- **revinfo** - zawiera następujące kolumny:
  - — datę utworzenia rewizji
  - — login użytkownika, który dokonał zmiany
- **revchanges** - zawiera następujące kolumny:
  - — numer rewizji
  - — nazwę (pełna nazwa klasy) modelu
- **nazwa_tabeli_history** - zawiera ona te pola, które został wybrane jako kandydaci do prowadzenia dziennika zmian dla danego typu obiektu domenowego

Model - Warstwa repozytoriów

Repozytoria w aplikacji demonstracyjnej są niczym więcej jak jedynie zdefiniowanymi w odpowiedni sposób interfejsami. Odpowiedni sposób oznacza, że dla każdego z nich, pierwszym interfejsem w drzewie dziedziczenie jest Repository. Jest to kluczowe ponieważ w ten sposób szkielet aplikacji **Spring** rozpoznaje repozytoria podczas skanowania **classpath** i tworzy z nich, poprzez zastosowanie proxy, konkretne obiekty dostępne później do użycia w sposób niejawny. Programista jest zobowiązany jedynie, w przypadku chęci skorzystania, do utworzenia w swojej klasie pola typu interfejsu, a właściwa referencja do obiektu zostanie umieszczona poprzez **dependency injection**. Poniższy przykład kodu pokazuję deklarację repozytorium.

```
 */

@Qualifier(SCarMasterRepository.REPO_NAME)
@RepositoryRestResource(itemResourceRel = SCarMasterRepository.REST_REPO_REL, path = SCarMasterRepository.REST_REPO_PA
public interface SCarMasterRepository
        extends SBasicRepository<SCarMaster, Long> {
    String REPO_NAME      = "CarMasterRepository";
    String REST_REPO_REL  = "repo.carmaster";
    String REST_REPO_PATH = "car_master";

    @RestResource(rel = "byBrandAndModel", path = "brandAndModel")
    SCarMaster findByManufacturingDataBrandAndManufacturingDataModel(
            @Param("brand") final String brand,
            @Param("model") final String model
    );

    @RestResource(rel = "byBrand", path = "brand")
    Page<SCarMaster> findByManufacturingDataBrand(
            @Param(value = "brand") final String brand,
            final Pageable pageable
    );

    @RestResource(rel = "byModel", path = "model")
    Page<SCarMaster> findByManufacturingDataModel(
            @Param(value = "model") final String model,
            final Pageable pageable
    );

    @RestResource(rel = "byBrandOrModelContaining", path = "brandOrModel_contains")
    @Query(name = "byBrandOrModelContaining", value = "select cm from SCarMaster as cm where cm.manufacturingData.brand like %
    List<SCarMaster> findByManufacturingDataBrandContainingOrManufacturingDataModelContaining(
            @Param("arg") final String searchArgument
    );
}
```

Listing 5: **SCarMasterRepository** - interfejs repozytorium pokazujący użycie metod mapowanych na kwerendy oraz bardziej skomplikowanego zapytania z użyciem adnotacji QUERY

Repozytoria został dostosowane do wymagań aplikacji celem umożliwiania wyszukiwania obiektów w konkretnych rewizjach lub ich zakresach. Z tego powodu zostało wprowadzone abstrakcyjne repozytorium SREPOSITORY.

```
public interface SRepository<T, ID extends Serializable, N extends Number & Comparable<N>>
        extends SBasicRepository<T, ID>,
                RevisionRepository<T, ID, N> {
    /**
     * {@code findInRevision} returns {@link Revision} of the target underlying target entity in the given revision
     * described in {@code revision} param
     *
     * @param id
     *         id of the entity {@link org.springframework.data.domain.Persistable#getId()}
     * @param revision
     *         the revision number
     *
     * @return {@link Revision} for passed arguments
     *
     * @throws EntityInRevisionDoesNotExists
     * @see SRepository#findInRevisions(java.io.Serializable, Number[])
     */
    Revision<N, T> findInRevision(final ID id, final N revision);

    /**
     * {@code findInRevisions} does exactly the same job but for multiple possible {@code revisions}.
     *
     * @param id
     *         id of the entity {@link org.springframework.data.domain.Persistable#getId()}
     * @param revisions
     *         varargs with revision numbers
     *
     * @return {@link Revisions}
     */
    @SuppressWarnings("unchecked")
    Revisions<N, T> findInRevisions(final ID id, final N... revisions);

    Revisions<N, T> findRevisions(final ID id, final DateTime dateTime, final Operators before);

    /**
     * Returns how many revisions exists for given {@link org.springframework.data.domain.Persistable#getId()} instance
     *
     * @param id
     *         the id
     *
     * @return revisions amount
     */
    long countRevisions(ID id);
}
```

Listing 6: **SRepository** - abstrakcyjne repozytorium wspierające dostęp do rewizji

Z uwagi na to, że funkcjonalność tego interfejsu nie jest dostępna w klasach dostarczonych przez **Spring Data JPA**, musiałaby być ona zaimplementowana. Niemniej nie odniosło by to skutku bez wprowadzenie dodatkowo elementu - fabryki, która nadpisywała by proces tworzenia repozytoriów. Poniższa klasa SRepositoriesFactoryBean działa wybiórczo, tj. dobiera konkretne implementacja zależnie od faktycznego typu repozytorium. Rozstrzygającym kryterium jest czy dane repozytorium jest przeznaczone dla obiektów wersjonowanych czy też nie.

```
public class SRepositoriesFactoryBean<T extends SBasicRepository<S, ID>, S, ID extends Serializable>
        extends JpaRepositoryFactoryBean<T, S, ID> {

    private Class<?> revisionEntityClass;

    public void setRevisionEntityClass(Class<?> revisionEntityClass) {
        this.revisionEntityClass = revisionEntityClass;
    }

    @Override
    protected RepositoryFactorySupport createRepositoryFactory(final EntityManager entityManager) {
        return new SRepositoryFactory(entityManager, this.revisionEntityClass);
    }

    private static class SRepositoryFactory
            extends JpaRepositoryFactory {
        private static final Logger LOGGER = Logger.getLogger(SRepositoryFactory.class);
        private final Class<?>                      revisionEntityClass;
        private final RevisionEntityInformation     revisionEntityInformation;
        private final LockModeRepositoryPostProcessor lockModePostProcessor;

        public SRepositoryFactory(final EntityManager entityManager,
                                  final Class<?> revisionEntityClass) {
            super(entityManager);
            this.lockModePostProcessor = LockModeRepositoryPostProcessor.INSTANCE;

            this.revisionEntityClass = revisionEntityClass == null ? AuditedRevisionEntity.class : revisionEntityClass;
            this.revisionEntityInformation = DefaultRevisionEntity.class
                    .equals(revisionEntityClass) ? new AuditingRevisionEntityInformation() : new ReflectionRevisionEntityInformation(this.revisio

            if (LOGGER.isTraceEnabled()) {
                LOGGER.trace(String.format("Created %s with arguments=[em=%s,rec=%s,rei=%s]",
                        SRepositoryFactory.class.getSimpleName(),
                        entityManager,
                        this.revisionEntityClass,
                        this.revisionEntityInformation));
            }
        }

        @Override
        @SuppressWarnings({"unchecked", "UnusedDeclaration"})
        protected <T, ID extends Serializable> JpaRepository<?, ?> getTargetRepository(final RepositoryMetadata metadata, final EntityMar
            final JpaEntityInformation<T, Serializable> entityInformation = (JpaEntityInformation<T, Serializable>) getEntityInformation(me
                    .getDomainType());
            final Class<?> repositoryInterface = metadata.getRepositoryInterface();
            SimpleJpaRepository repository;
            if (!ClassUtils.isAssignable(SRepository.class, repositoryInterface)) {
                repository = new SBasicRepositoryImpl(entityInformation, entityManager);
            } else {
                repository = new SRepositoryImpl(entityInformation, revisionEntityInformation, entityManager);
            }
            repository.setLockMetadataProvider(this.lockModePostProcessor.getLockMetadataProvider());
            return repository;
        }

        @Override
        protected Class<?> getRepositoryBaseClass(final RepositoryMetadata metadata) {
            final Class<?> repositoryInterface = metadata.getRepositoryInterface();
            if (!SRepository.class.isAssignableFrom(repositoryInterface)) {
                return SBasicRepositoryImpl.class;
            }
            return SRepositoryImpl.class;
        }
    }
}
```

Listing 7: **SRepositoriesFactoryBean** - fabryka repozytoriów dla implementacji własnej funkcjonalności

Model - Warstwa serwisów

Serwisy stanowią w aplikacji element realizujący logikę biznesową. Nie odnoszą się one jednak jedynie do operacji na modelu danych. Również inne moduły aplikacji korzystają z własnych serwisów, jako miejsc gdzie funkcjonalność została zebrana i jest gotowa do użycia.

```
public interface SPersonService
      extends SService<SPerson, Long, Integer> {

   SContact<SPerson> newContactData(
         @NotNull
         final String contact,
         final long assignTo,
         @NotNull
         final SContact assignToContact) throws EntityDoesNotExistsServiceException;

   List<SPersonContact> findAllContacts(final Long idClient);

   List<SPerson> findByFirstName(
         @NotNull
         final String firstName);

   List<SPerson> findByLastName(
         @NotNull
         final String lastName);

   SPerson findByEmail(
         @NotNull
         final String email);
}
```

Listing 8: **SPersonService** - interfejs serwisu dla modelu SPerson

Serwisy został zaprojektowane aby korzystać z repozytoriów danych. Ma to swoje pozytywne skutki i nie jest wcale oznakę nadmiarowości kodu czy też jego duplikowania. Z uwagi na fakt, ze serwisy odwołują się do danych poprzez interfejsy repozytorii, podnosi to znacząco możliwości późniejszych zmian w postaci silnika bazy danych. Dodatkową korzyścią jest podniesienie testowania interesujących funkcjonalności bez konieczności posiadania działającego połączenia z bazą danych.

Kontroler - Warstwa kontrolerów

Kontrolery stanowią spoiwo całej architektury MVC. Bez nich nie byłoby możliwe efektywne przekazywanie żądań do warstw modelu z wykorzystaniem innych warstw oraz nie można byłoby przekazywać odpowiedzi. W aplikacji demonstracyjnej warstwa kontrolerów nie wyróżnia się niczym szczególnym są to tak zwane **POJO** oznaczone odpowiednimi adnotacji zarówno na poziomie samych klas jak i poszczególnych metod, które dopiero w kontekście szkieletu aplikacji **Spring** nabierają właściwego znaczenia.

Widok - Warstwa widoku

Warstwa widoku została zaprojektowana z wykorzystaniem standardowej biblioteki tagów **JSTL** oraz plików **JSP**. Niemniej z uwagi na powtarzalność większości elementów na

stronie stało się nieefektywne, aby powtarzać pewne stałe elementy w każdym pliku JSP lub opierać się na względnych ścieżkach podczas zawierania innych plików **JSP** w innych. Doskonałym rozwiązaniem okazało się skorzystanie z biblioteki **Apache Tiles** 3.9. **Tiles** rozumiane są tutaj jako podstawowe elementy, z których można składać gotowy widok dostępny później pod konkretną nazwą, którą można zwrócić w kontrolerze Spring.

```
<tiles-definitions>

   <definition name="ui.core.Page" template="/ui/core/page.jsp">
      <put-attribute name="head" value="ui.meta.Head"/>
      <put-attribute name="css" value="ui.meta.head.CSS"/>
      <put-attribute name="js" value="ui.meta.head.JS"/>
      <put-attribute name="navigator" value="ui.nav.Navigator"/>
      <put-attribute name="header" value="/ui/core/c-header.jsp"/>
      <!-- to override by concrete page -->
      <put-attribute name="content" value=""/>
      <!-- to override by concrete page -->
   </definition>

</tiles-definitions>
```

Listing 9: Definicja *tile* - podstawowy element widoku w rozumieniu technologi Apache Tiles

Listing **??** pokazuje deklarację abstrakcyjnej *płytki*. Abstrakcyjność jest tutaj kwestią umowną ponieważ można by tę *płytkę* zwrócić z kontrola i została by ona zrenderowana do poprawnego widoku HTML. Elementem, który jest w tej definicji zadeklarowany, lecz nie zdefiniowany jest *content* - faktyczna zawartość danej strony. Mimo to plik **JSP** odpowiadający tej płytce zawiera kod, który umieści ją w ostatecznej strukturze DOM.

```
<%@ page session="true" language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>
<html>
<head>
   <tiles:insertAttribute name="head"/>
   <tiles:insertAttribute name="css" flush="true"/>
   <tiles:insertAttribute name="js" flush="true"/>
</head>
<body class="tundra">
<div id="page">
   <div class="content-wrapper">
   <header id="header" class="main">
         <s:message code="label.dashboard.header" htmlEscape="true" var="headerLabel"/>
         <p>${headerLabel}</p>

         <div id="authentication-control">
            <security:authorize access="isFullyAuthenticated()" var="userAuthenticated"/>
            <c:choose>
               <c:when test="${!userAuthenticated}">
                  <tiles:insertDefinition name="springatom.tiles.auth.action.login" flush="true"/>
               </c:when>
               <c:otherwise>
                  <tiles:insertDefinition name="springatom.tiles.auth.action.logout" flush="true"/>
               </c:otherwise>
            </c:choose>
         </div>
      </header>
      <tiles:insertAttribute name="header"/>
      <div class="content">
         <tiles:insertAttribute name="content"/>
      </div>
      <footer id="footer" class="main">
         <p>Footer goes here</p>
      </footer>
   </div>
   <tiles:insertAttribute name="navigator"/>
</div>
</body>
```

Listing 10: Plik **JSP odpowiadający płytce ??**

Widok - Warstwa komponentów

W momencie realizacji części praktycznej okazało się koniecznie stworzenie oddzielnego modułu, którego zadaniem byłaby budowa struktury obiektów interfejsu użytkownika po stronie serwera w sposób generyczny z uwagi na duża ilość obiektów biznesowych. Specjalnie zaprojektowany interfejs **ComponentBuilder**, będąc korzeniem całej hierarchii klas, wyznacza zakres obowiązków wszystkich klas go implementujących.

```
public interface ComponentBuilder<COMP extends Serializable> {

    String getId();

    Class<?> getBuilds();

    ComponentBuilds.Produces getProduces();

    ComponentDataResponse<?> getData() throws ComponentException;

    COMP getDefinition() throws ComponentException;

    void init(ComponentDataRequest componentDataRequest);
}
```

Listing 11: **ComponentBuilder** - korzeń hierarchii modułu komponentów, który wyznacz rolę tego rodzaju obiektów w systemie.

Szczególnie ważne są tutaj następujące metody:

- **ComponentBuilder#getDefinition()** - wywołanie następuje w momencie kiedy użytkownik otwiera stronę, w strukturze DOM, gdzie zapisana jest informacja o obiekcie biznesowym pobrana z aktualnego adresu URL.
- **ComponentBuilder#getData()** - jest to metoda zaprojektowania do wygenerowania asynchronicznej odpowiedzi **Ajax** na żądania pobrania gotowego widoku renderowanego komponentu.

Komponenty istnieją w dwóch różnych wariantach. Wariant 1 służy do generowania stron dla obiektów biznesowych, zwanych dalej **info page**, natomiast zadaniem wariantu drugiego jest generowania zarówno konfiguracji jak i danych dla tabel. Oba rodzaje został zaprojektowane, aby zminimalizować ilość potrzebnych do przesłania danych. Powodem istnienia dwóch odmiennych klas jest odmienność konstrukcji zwykłej strony HTML od tabeli. Dodatkowo w przypadku tabeli pewne charakterystyczne elementy, takie jak zmienne opisujące zachowanie kolumn czy też całej tabeli wymuszone zostały przez zastosowanie biblioteki **Dandelion Datatables**3.11.

Elementem, który kontroluje całość jest kontroler, czyli klasa będąca implementacją funkcjonalności architektury MVC, znajdującą się między modelem danych, a widokiem. Z uwagi na dwa oddzielne odnogi **ComponentBuilder**'ów w strukturze dziedziczenie oraz odmiennych wymoglów co do konstrukcji końcowego widoku prezentowanego użytkownikowi, istnieją także dwa kontrolery. Jeden z nich został zaprojektowany specjalnie dla **info page**, gdzie rolą drugiego jest umożliwić generowanie tabel.

Poniższe listingi pokazują implementacje metod w warstwie kontrolerów odpowiednio wywołujących funkcje **ComponentBuilder#getData()** oraz **ComponentBuilder#getDefinition()**.

```
@RequestMapping(value = "/{path}/{id}", method = RequestMethod.GET)
public ModelAndView getInfoPageView(@PathVariable("path") final String path,
                                    @PathVariable("id") final Long id) throws InfoPageNotFoundException {
    LOGGER.debug(String.format("/getInfoPageView/path=%s/id=%s", path, id));
    final SInfoPage page = this.getInfoPageForPath(path);
    if (page != null) {
        LOGGER.trace(String.format("Resolved infoPage => %s for path => %s", page, path));
        final ModelMap modelMap = new ModelMap();

        modelMap.put(InfoPageConstants.INFOPAGE_PAGE, page);
        modelMap.put(InfoPageConstants.INFOPAGE_VIEW_DATA_TEMPLATE_LINK, this.getViewDataTemplateLink());
```

```
            return new ModelAndView(
                    this.getViewForPage(page),
                    modelMap
            );
        }
        throw new InfoPageNotFoundException(path);
    }
```

Listing 12: Obsługa żądania w kontrolerze, które wywołuje metodą **ComponentBuilder#getDefinition()**

dla **info page**

```
    @ResponseBody
    @RequestMapping(value = "/data/{id}")
    public DatatablesResponse<?> getBuilderData(
            @PathVariable("id") final String builderId,
            final ComponentTableRequest tableRequest,
            final WebRequest request) throws ControllerTierException {
        LOGGER.info(String.format("/getBuilderData -> builder=%s, tableRequest=%s", builderId, tableRequest));

        final ComponentBuilder<?> builder = this.builders.getBuilder(builderId, new ModelMap(ComponentConstants.REQUEST_BEAN, tab
        try {
            if (builder != null && builder instanceof TableComponentBuilder) {
                LOGGER.trace(String.format("Found builder %s:%s:%s", builderId, builder.getId(), builder.getBuilds()));
                return DatatablesResponse.build((DataSet<?>) builder.getData().getValue(), tableRequest.getCriterias());
            }
        } catch (Exception e) {
            LOGGER.error("/getBuilderData threw exception", e);
            throw new ControllerTierException(e);
        }
        return null;
    }
```

Listing 13: Obsługa żądania w kontrolerze, które wywołuje metodą **ComponentBuilder#getData()**

dla tabeli

Dla tabel istnieje możliwość zdefiniowania atrybutów do zaprezentowania użytkownikowi, które bezpośrednio nie istnieją w klasie odpowiadającej danemu obiektowi. Należy wtedy przeciążyć metodę *handleDynamicColumn*, której zadaniem jest zwrócenie wartości dla danego **dynamicznego** atrybutu. Poniższy listinig pokazuje przykładową implementację:

```
@Override
protected Object handleDynamicColumn(final SReport object, final String path) throws DynamicColumnResolutionException {
	Object retValue = super.handleDynamicColumn(object, path);
	if (retValue != null) {
		return retValue;
	}
	switch (path) {
		case "generate-action": {
			try {
				final Link link = linkTo(ReportBuilderController.class).slash(object.getId()).withSelfRel();
				retValue = new PopupAction().setType(RequestMethod.POST)
						.setUrl(link.getHref());
			} catch (Exception exception) {
				final String message = String.format("Error in creating link over path=%s", path);
				this.logger.error(message, exception);
				throw new DynamicColumnResolutionException(message, exception);
			}
		}
		break;
		case "delete-action": {
			try {
				final Link link = linkTo(ReportBuilderController.class).slash(object.getId()).withSelfRel();
				retValue = new AjaxAction().setType(RequestMethod.DELETE)
						.setUrl(link.getHref());
			} catch (Exception exception) {
				final String message = String.format("Error in creating link over path=%s", path);
				this.logger.error(message, exception);
				throw new DynamicColumnResolutionException(message, exception);
			}
		}
	}
	return retValue;
}
```

Listing 14: Uzyskania wartości dynamiczego atrybutu w momencie tworzenia odpowiedzi dla tabeli

Zadaniem **??** jest zwrócenie akcji dla poszczególnych wierszy, które użytkownik będzie mógł wykonać na obiekcie odpowiadającym danemu rekordowi w tabeli.

Model/Widok - RBuilder

RBuilder jest praktyczną realizacją, będącej na wczesnym etapie realizacji, koncepcji zaprojektowania generycznego silnika raportowaniu dla aplikacji internetowej. Głównymi założeniami tego komponentu są:

- możliwość generowania raportów przez użytkowników systemu,
- możliwość zapisywania raportów do bazy danych celem późniejszego ich użycia,
- możliwość edycji istniejących raportów,
- możliwość usuwania istniejących raportów,
- wsparcia dla zabezpieczenie akcji możliwych do wykonania na raportach,
- eksport raportów do formatów:
  - PDF
  - XLS
  - HTML
  - CSV

Jest to obecnie najbardziej skomplikowany komponent aplikacji, którego złożoność jeszcze wzrośnie, z uwagi na założenie, że raporty mogą być definiowane przez użytkowników.

Definiowanie raportu przebiega według następujących kroków:

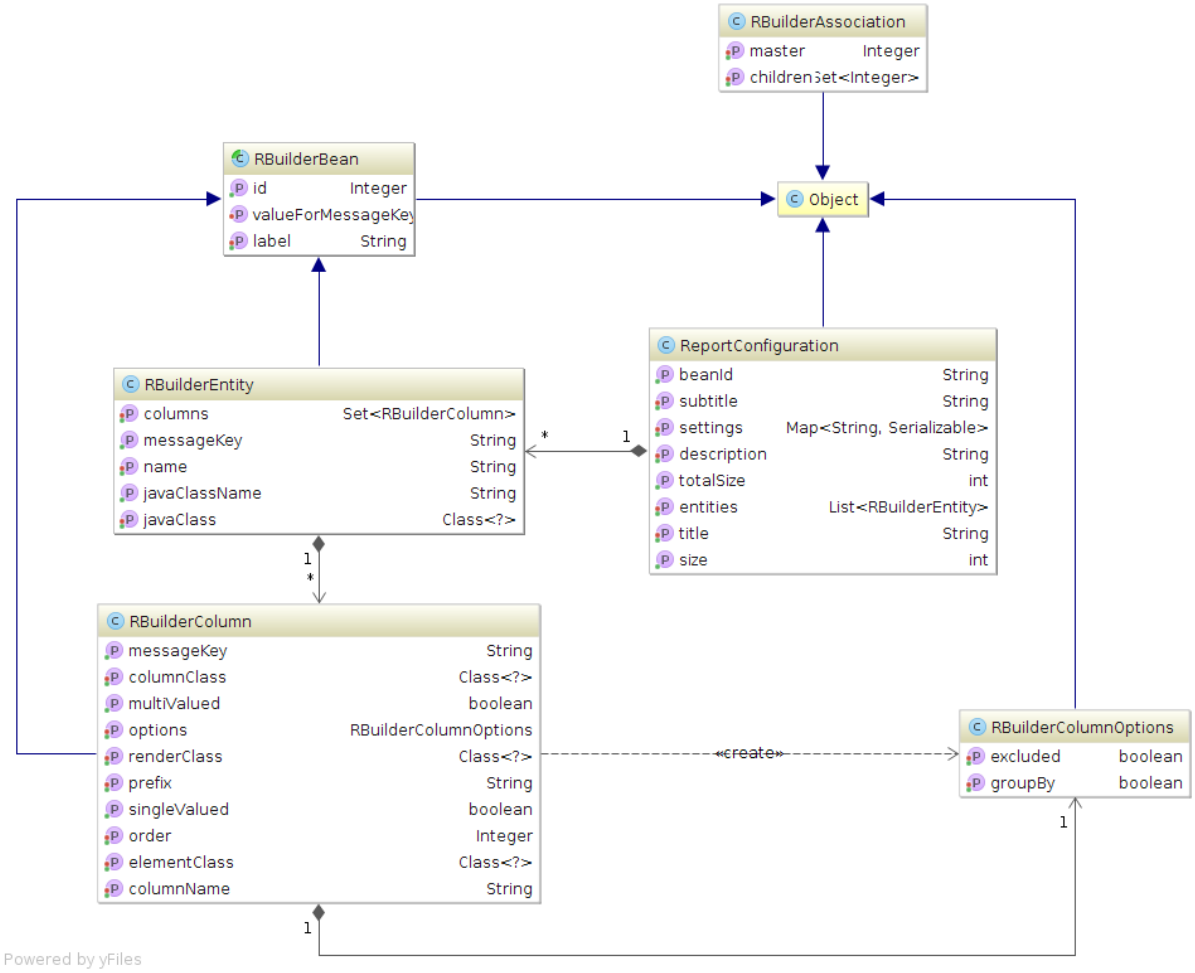
1. Użytkownik wybiera z listy tabele dla których chce wygenerować raport,
2. Dla wybranych tabel użytkownik wybiera kolumny oraz typy w jakich chce aby zostały one zrenderowane,
3. Użytkownik podaje informacje takie jak:
   - nazwa
   - opis
4. Sterowanie jest przekazywane do serwisu odpowiedzialnego za:
   - zdecydowanie o rodzaju raportu: dla jednej tabeli, dla wielu tabel,
   - utworzenie obiektu domenowego raportu zawierającego informację takie jak tytuł,
   - utworzenie, kompilacja i zapisanie zserializowanego obiektu klasy **DynamicReport** do systemu plików,
   - zwrócenie sterowania do **Spring Web Flow**
5. Generowanie raportu jest dostępne z tabeli zawierającej listę wszystkich raportów

Report należy w tym miejscu rozumieć jako obiekt domenowy, który służy do późniejszego do zapisanie wymaganych informacji do bazy danych oraz jako obiekt **Dynamic Jasper Report**. Celem dla którego tworzony jest ten typ obiektu leży w wykorzystanej bibliotece **DynamicJasper**. Aby raport mógł być zaprezentowany użytkowniki musi on zostać w pierwszej kolejności skompilowany do pliku **.jasper*. Załadowania, a dokładniej odczytanie tego obiektu z zserializowanego pliku zapisanego w systemie plików jest wymogiem koniecznym i absolutnym aby sterowanie procesem tworzenie jednej z reprezentacji przekazać do szkieletu aplikacji **Spring**. Niemniej nie jest to jedyny wymóg. W specjalnej mapie, pod jasno określonymi kluczami, musi znaleźć się co najmniej źródło danych dla raportu spójne z jego budową. Powyższe powody generują wiele niedogodności jak konieczność przechowywania dodatkowych plików na serwerze oraz trudności w uogólnieniu niektórych aspektów projektowania nowego raportu.

Model Model danych przeznaczony dla **RBuilder** jest pewnym rozszerzeniem informacji o modelu danych biznesowych (domain object). Zawarte w nim informacje pozwalają na stwierdzenie następujących właściwości obiektów domenowych:

- nazwa tabeli,
- lokalizowana nazwa tabeli,
- atrybuty,
- powiązania z innymi modelami,

Późniejsze instancje obiektów zaprezentowanych na diagramie UML są wykorzystywane jako do przechowywania informacji niezbędnych do prawidłowego utworzenia konfiguracji raportu **ReportConfiguration**.



Rysunek 18: *Diagram UML modelu danych* ***RBuilder***

Serwisy Serwisy **RBuilder**'a dostarczają metod dzięki którym możliwe jest wykonanie przygotowanie gotowego reportu w wybranej przez użytkownika reprezentacji, ale także utworzenie danych wymaganych przez kolejnego kroki przewodnika. Dostarczone usługi to:

- ustalenie możliwych typów w których można zrenderować wartość danej komórki / kolumny,
- tworzeniem obiektu domenowego w zależności od konfiguracji raportu,
- generowaniem instancji reprezentatywnego raportu

Funkcjonalność tej grupy klas dla komponentu **RBuilder** można podzielić na następujące bloki:

Tabela 2: Bloki funkcjonalne modelu serwisów **RBuilder**

Grupa	Funkcjonalność
Operation Management	Grupa **Operation Management** odpowiedzialna jest za instacjonowanie obiektu **SReport** w zależności od ilości wybranych dla konkretnego raportu. Lista klas: • RBuilderOperation • RBuilderCreateOperation • SingleEntityRBuilderCreateOperation • MultipleEntitiesRBuilderCreateOperation
Data Management	Klasy z grupy **Data Management** zostały zaprojektowane do pobierania danych takich jak: • informacje o typach obiektów domenowych, które można uwzględnić w raportach. Takie klasy adnotowane są przez adnotacje *ReportableEntity*, a ich lista udostępniana jest poprzez interfejs *ReportableEntityResolver*, • listę kolumn wraz z ich właściwościami takimi jak jej nazwą, nazwa dla wybranej lokalizacji językowej, typ danych przechowywanych w odpowiadającej jej polu w klasie, odpowiednie typu na które można rzutować dany typ. Informacje tego typu udostępniane są poprzez interfejs *ReportableBeanResolver*, • listą powiązań między modelami w uproszczonej formie na potrzeby wybierania tabel podczas projektowania raportu. Na obecną chwile możliwe jest utworzenie jedynie nieprzechodnich powiązań opisanych na bazowym poziomie przez relacje klucz główny - obcy. Dane tego typu udostępnione są przez interfejs *ReportableAssociationResolver*.
Dynamic Jasper Operation	*JasperBuilderService* jest jedyną klasą tej grupy posiadającą dostarczającą możliwości utworzenia skompilowanego raportu typu **JasperReport**, który potem jest serializowany do systemu plików. Niemniej jego głównym zadaniem jest wkomponowanie w obiekty wyżej wymienionego typu takich danych jak: • tytuł, • podtytuł, • opis, • język, • szerokość odpowiednich sekcji jak nagłówek, stopka itp., • lista kolumn, • lista kolumn według których dane mają być grupowane.

Następna strona...

Tabela 2 – kontynuacja...

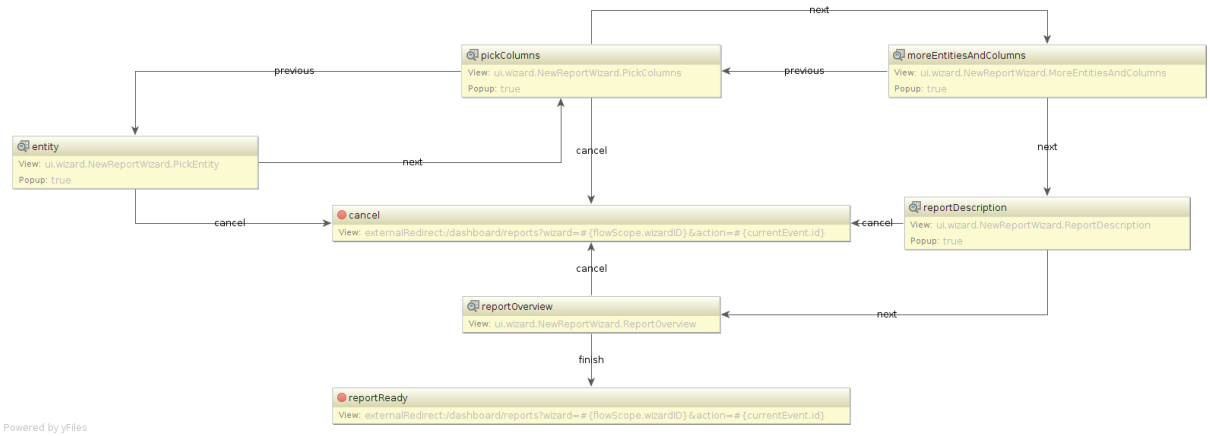
Grupa	Funkcjonalność
View helper	*ReportBuilderService* jest interfejsem serwisu, którego celem istnienia jest przekazania sterowania do modułu **Operation Management** celem utworzenia instancji obiektu domenowego **SReport** oraz wsparcie dla operacji renderowania raportu w konkretnej reprezentacji. Kiedy pierwsza z funkcji jest trywialna w kontekście złożoności służąc jedynie separacji zadań i zmniejszeniu kohezji klas, druga z wymienionych metod jest dużo bardziej złożona. Jej celem jest wykonanie następujących operacji celem uzyskania danych wymaganych przez moduł **Spring** renderującego raport w wybranej reprezentacji (PDF, HTML, CSV, XLS): • pobranie obiektu domenowego z bazy danych dla danego numeru raportu, • deserializacja skompilowanego pliku **.jasper* z systemu plików, • utworzenie źródła danych na podstawie informacji takich jak lista kolumn, ich typ, wybranych typ reprezentacji danych w kolumnie

Widok Na tę część składają się następujące elementy:
- tabela z istniejącymi raportami ,
- przewodnik tworzenia nowego raportu ,
- specjalnie skonfigurowanego ViewResolver'a [2].

Tabela (4.1.2) jest obiektem należącym do warstwy widoku, który został utworzony z użyciem komponentu tabeli. Zawiera ona dodatkowo akcje, umożliwiające operacje na raportach.

Przewodnik (4.1.2) został zaprojektowany na podstawie **Spring Web Flow**. Dzięki temu zbiór formularzy, na których definiuje się poszczególne elementy składowe gotowego obiektu ReportConfiguration stanowiącego bazę do generowania wybranej reprezentacji 4.1.2 reportu, jest logicznie połączony. Dodatkową korzyścią jest uzyskanie bezproblemowego wsparcia dla przetwarzania uzyskanych danych wejściowych po stronie serwera bez konieczności pisania własnej logiki do zarządzania komunikacją Ajax oraz możliwość przekazywania danych między kolejnymi krokami.

[2] ViewResolver - interfejs, które implementują specjalne klasy zadaniem których jest ładowanie widoków poprzez odwołanie m.in. do plików JSP, logicznych nazw widoków itp.



Rysunek 19: *Logiczne połączenie kroków w przewodniku dla **RBuilder***

Każdy z kroków przewodnika wspiera odpowiednia klasa Java będąca specjalizowanym rozszerzeniem FormAction zarządzającą ustawieniem danych wejściowych dla formularza, walidacji i konwerterów. Ostatecznie rzecz jest szczególnie ważna ponieważ praktycznie wszystkie formularze zostały napisana aby wspierać wprowadzanie danych dla więcej niż jednego obiektu. Z uwagi na brak natywnego wsparcia ze strony szkieletu aplikacji **Spring** należało napisać odpowiednie metody samemu celem utworzenia z prostych łańcuchów znakowych poprawnych obiektów w danym kontekście.

```
private abstract class BaseConverter
        extends MatcherConverter {
    protected Set<RBuilderEntity> doConvert(final Set<String> list) {
        LOGGER.trace(String.format("converting with selected clazz=%s", list));
        Preconditions.checkNotNull(list);
        Preconditions.checkArgument(!list.isEmpty());
        final Set<RBuilderEntity> reportedEntities = Sets.newHashSet();
        for (final String beanIdentifierString : list) {
            final Integer identifier = Integer.valueOf(beanIdentifierString);
            final RBuilderBean bean = reportableBeanResolver.getReportableBean(identifier);
            if (ClassUtils.isAssignable(RBuilderEntity.class, bean.getClass())) {
                reportedEntities.add((RBuilderEntity) bean);
            }
        }
        return reportedEntities;
    }
}
```

Listing 15: Bazowy konwerter dla operacji **PickEntityFormAction** który tworzy z łańcuchów znakowych obiekty **RBuilderEntity**

Globalna - Warstwa konwersji

Warstwa konwersji typów jest modułem **Spring**, który wykorzystywany jest w momencie kiedy z obiektu typu A programista chce uzyskać obiekt typu B. Dobrym przykładem jest najczęściej moment w którym w kodzie JSP umieszczamy bezpośrednio nasz obiekt, korzystaj z biblioteki tagów dostarczonej przez **Spring**. W tym momencie uruchamiany jest proces mający na celu konwersję obiektu do reprezentatywnej postaci łańcucha znakowego, który można będzie wkomponować do drzewa DOM. Jest to rozwiązanie efektywne niemniej

posiadające jedno uchybienie. W przypadku gdy programista chciałby uzyskać selektywny sposób, zależny od kontekstu w jakim znajduje się obiekt lub też chęci uzyskania wartości jednego z atrybutów, nie jest on w stanie osiągnąć zamierzonego rezultatu z uwagi na sposób w jaki działa konwersja typów. Znalezienie pierwszego konwertera, który jest w stanie przeprowadzić żądaną transformację kończy proces wyszukiwania. Nie oznacza to wcale, że uzyskany wynik będzie zgodny z oczekiwanym. Z tego powodu aplikacja demonstracyjna rozszerza istniejące funkcjonalność przez umożliwienie selektywnego konwertowania między poszczególnymi typami. Na obecną chwilę zostało to zaimplementowane dla obiektów domenowych, a klasą kontrolującą selektywny proces jest **PersistableConverterPicker**.

```
@LazyComponent
@Role(BeanDefinition.ROLE_SUPPORT)
public class PersistableConverterPicker {

	@Autowired
	private Set<PersistableConverter<?>> converters;

	@SuppressWarnings("unchecked")
	public <T extends Persistable> Converter<T, String> getConverterForSelector(final String key) {
		final Optional<PersistableConverter<?>> match = FluentIterable.from(this.converters).firstMatch(new Predicate<PersistableCo
			@Override
			public boolean apply(@Nullable final PersistableConverter<?> input) {
				assert input != null;
				final PersistableConverterUtility annotation = input.getClass().getAnnotation(PersistableConverterUtility.class);
				return annotation != null && AnnotationUtils.getValue(annotation, "selector").equals(key);
			}
		});
		if (match.isPresent()) {
			return (Converter<T, String>) match.get();
		}
		return new DefaultPickedConverter<>();
	}

	@SuppressWarnings("unchecked")
	public <T extends Persistable> Converter<T, String> getDefaultConverter(final TypeDescriptor sourceType) {
		final Optional<PersistableConverter<?>> match = FluentIterable
				.from(this.converters)
				.filter(new Predicate<PersistableConverter<?>>() {
					@Override
					public boolean apply(@Nullable final PersistableConverter<?> input) {
						assert input != null;
						return input.matches(sourceType, TypeDescriptor.valueOf(String.class));
					}
				})
				.firstMatch(new Predicate<PersistableConverter<?>>() {
					@Override
					public boolean apply(@Nullable final PersistableConverter<?> input) {
						assert input != null;
						final PersistableConverterUtility annotation = input.getClass().getAnnotation(PersistableConverterUtility
						return annotation != null && String.valueOf(AnnotationUtils.getValue(annotation, "selector")).isEmpty(
					}
				});
		if (match.isPresent()) {
			return (Converter<T, String>) match.get();
		}
		return new DefaultPickedConverter<>();
	}

	private class DefaultPickedConverter<T extends Persistable>
			extends PersistableConverterImpl<T> {

		@Override
		public boolean matches(final TypeDescriptor sourceType, final TypeDescriptor targetType) {
			return true;
```

```
}

@Override
public String convert(final Persistable source) {
        if (source == null) {
                return "";
        }
        return source.toString();
}
```

Listing 16: **PersistableConverterPicker** - koordynator selektywnej konwersji typów

PersistableConverterPicker posiada metody które pozwalają na wybór selektywnego konwertera do wykonania operacji transformacji. Zostało również zapewnione wsparcia dla istniejącej funkcjonalności, bez użycia selektora. Ta część realizowana jest w metodzie **PersistableConverterPicker#getDefaultConverter(...)**. Selektywne konwertery różnią się od normalnych jedynie użyciem specjalnej adnotacji, która pozwala na ustalenie, że:

- konwerter jest domyślny jeśli nie został zdefiniowany klucz,
- konwerter jest selektywny jeśli istnieje zdefiniowany klucz.

4.1.3. Plany rozwojowe

Aplikacja demonstracyjna jest obecnie na etapie dalszego rozwoju. Posiada szeroko zdefiniowane modułu zaprojektowane aby wspierać takie rejony jak:

- generyczna warstwa operacji bazodanowych,
- warstwa logiki biznesowej udostępnionej przez serwisy,
- moduł komponentów dla budowy tabel oraz stron obiektów domenowych,
- selektywna warstwa konwersji,
- tagi oddelegowane dla przewodników opartych o Spring Web Flow,
- model danych wraz z jego abstrakcyjną warstwą (interfejsy) do użytku zewnętrznego,
- moduł obsługujący funkcjonalność raportowania

W większości przypadków uzyskana funkcjonalność jest jednak na etapie implementacji i niektóre z niedopracowanych elementów ulegną zmianie celem uproszczenia zarządzaniem złożonością projektu oraz usunięciem nadmiarowych klas oraz słabo konfigurowalnych części. Plany rozwojowe aplikacji zostały zestawione w poniższej tabeli.

Tabela 3: Plany rozwojowe aplikacji demonstracyjnej

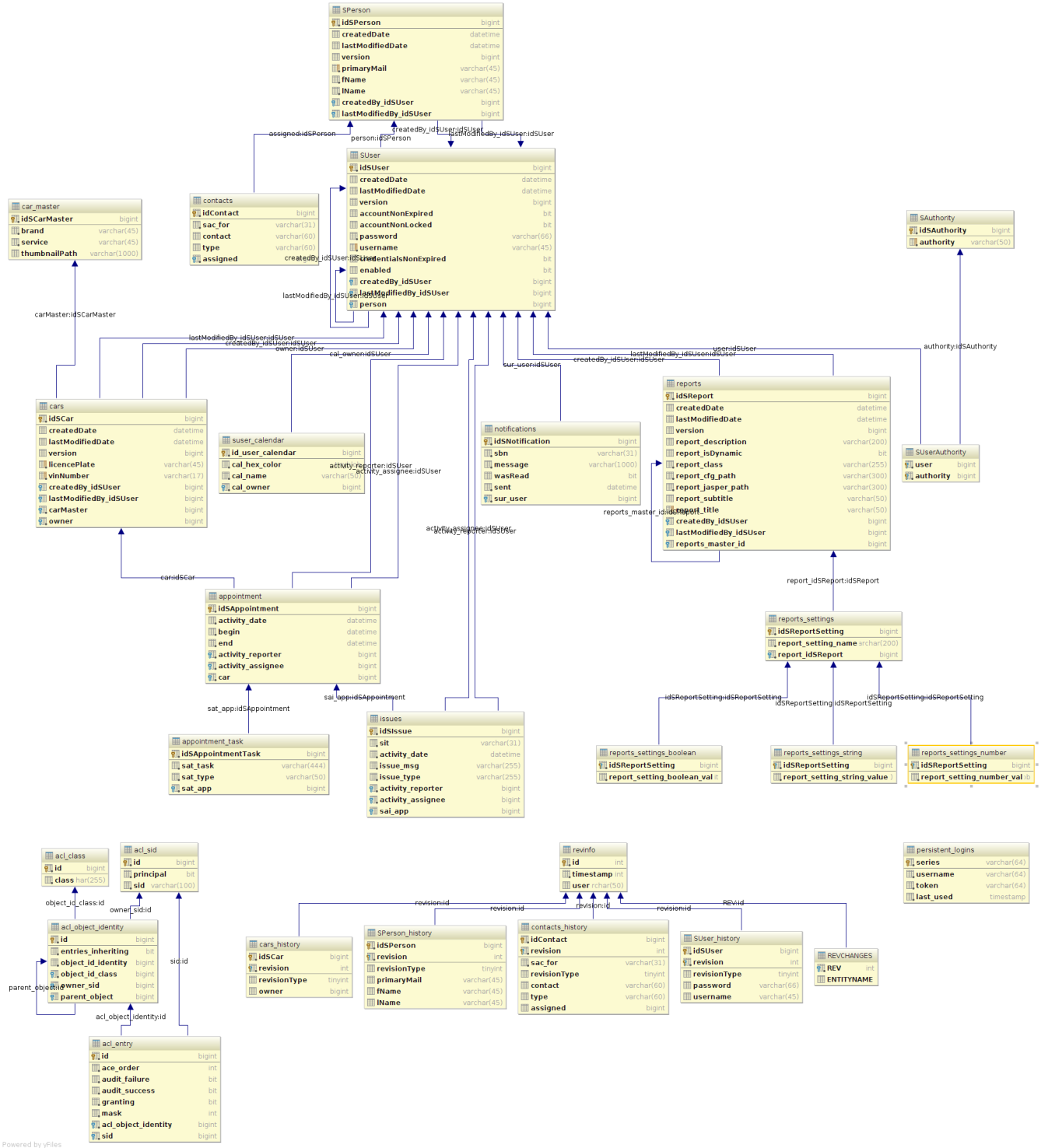
LP	Moduł	Opis zmian
1	**ComponentBuilder**	• optymalizacja kodu oraz wsparcie dla cache'owania raz załadowanych • przeniesienie definicji stron do deklaratywnego języka XML. Będzie to możliwe po usprawnieniu działania selektywnych konwerterów oraz zwiększy możliwość zmian zgodnie z wymaganiami bez konieczności zmian w plikach Java, • prawdopodobne przeprojektowanie operacji ładowania strony obiektu domenowego. Na obecną chwilę ładowany jest gotowy widok, jednakże z uwagi na możliwie duży narzut przesłanych danych bardziej optymalnym rozwiązaniem jest załadowanie pojedynczo każdego z panelu zdefiniowanego na danej stronie, • przeniesienie kodu widoku stron domenowych do łatwiejszego w zarządzaniu oraz utrzymaniu kodu ExtJS
2	**ComponentBuilder**	Połączenie kontrolerów odpowiedzialnych za obsługę żądań pochodzących od komponentów typu **tabele** oraz **strony domenowe**. Celem tego działania jest ujednolicenie adresów sugerujące, że oba moduły wywodzące się z tej samej rodziny klas oraz służą podobnemu celowi. Oddelegowania właściwego uzyskania wyniku zarówno w kwestii danych jak i definicji przez oddelegowane klasy.
3	**ComponentBuilder**	Wprowadzenie automatycznego mechanizmu generującego przekierowania do stron obiektów domenowych
4	**WebFlow**	Zaprojektowanie biblioteki wspierającej funkcjonalność **Spring Web Flow** dla biblioteki ExtJS. Decyzja podyktowana jest chęcią zminimalizowania użycia różnorodnych bibliotek JavaScript. Gotowa biblioteka byłaby udostępniona na licencji OpenSource.
5	**Selektywne konwertery**	Rozszerzenie możliwości selektywnych konwerterów o obiekty inne niż domenowego oraz wsparcie dla możliwości definiowana powtarzających się selektorów - kluczy w kontekście globalnym, ale unikatowych w kontekście danego obiektu podlegającego konwersji.
	Następna strona...	

Tabela 3 – kontynuacja...

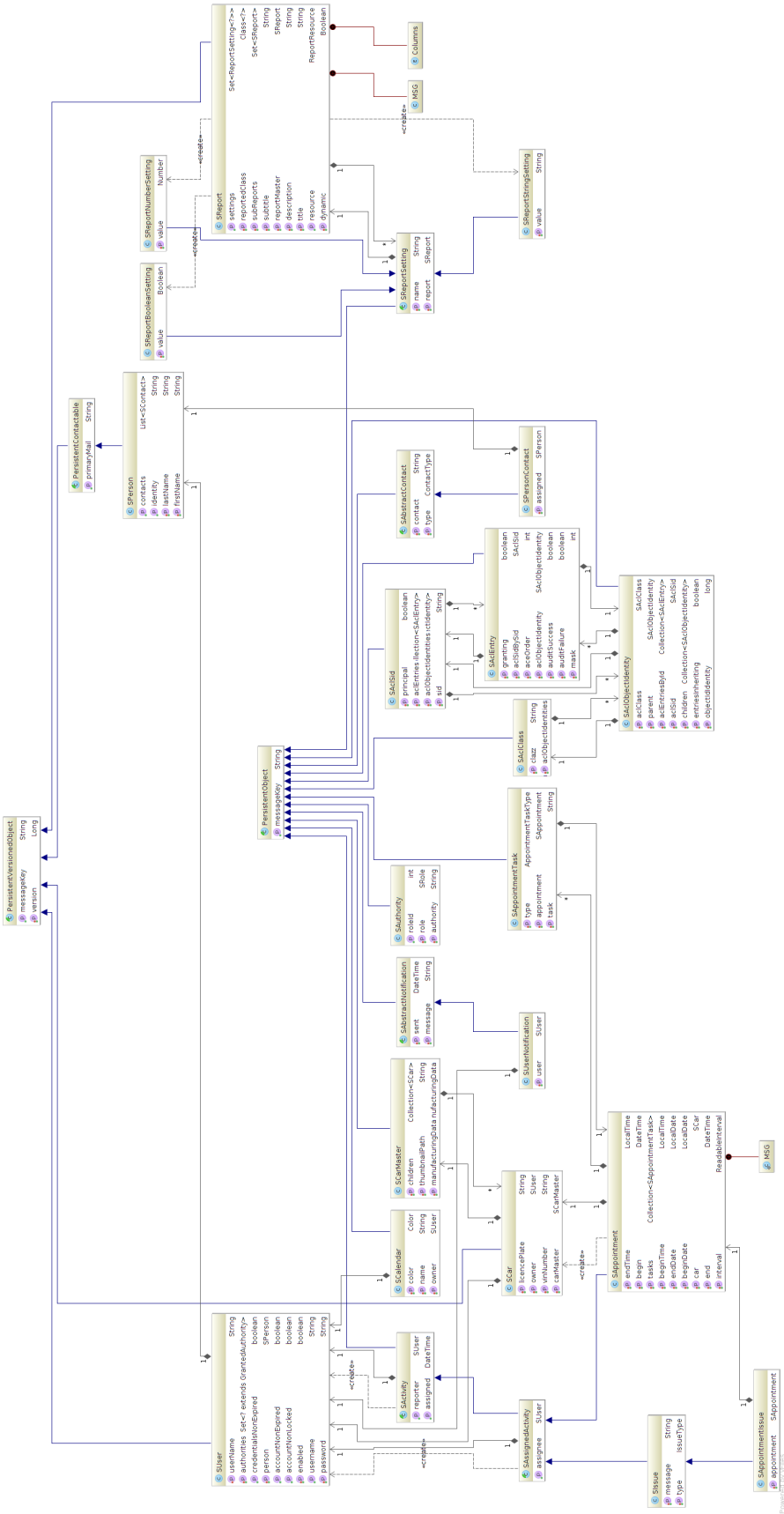
LP	Moduł	Opis zmian
6	**Kod ogólny**	Usunięcie pozostałości nadmiarowego kodu, który pozostał po uaktualnieniu aplikacji do korzystania z najnowszej (4.0.0) wersji szkieletu aplikacji Spring. Dzięki wprowadzonym poprawkom oraz nowym funkcjom część istniejącego kodu możliwa będzie do zastąpienie przez kod szkieletu co zmniejszy stopień bezpośredniej zależności między aplikacją a framework'iem.
7	**Widok**	Wsparcia dla ExtJS - migracja istniejących komponentów warstwy widoku użytkownika do szkieletu aplikacji ExtJS. Prawdopodobnym wyborem będzie wykorzystanie ExtJS w wersji 4.x.x z uwagi na lepszą wydajność i większe możliwości biblioteki.
8	**ReportBuilder**	Zrezygnowania z ciężkiego w użyciu oraz utrzymaniu sposobu generowania raportów w aplikacji poprzez bibliotekę **DynamicJasper**. Przeniesienie bezpośredniego generowania raportów do tabelek oraz inwestygacja możliwych technologii do wykorzystania w przypadku eksportowania raportu do formatów takich jak HTML, CSV, PDF oraz XLS.
9	**Wizard'y**	Ukończenie wszystkich wymaganych kreatorów zarówno modyfikacja jak i tworzenia nowych obiektów domenowych
10	**Terminarz spotkań**	Ukończenie prac nad terminarzem spotkań. Obecna funkcjonalność pozwala na tworzenie obiektów, które następnie można przeglądać z użyciem kalendarza (podobny do używanego w programie Outlook).
11	**Dekodowane numeru VIN**	Rozszerzenie obecnych możliwości dekodera numeru VIN o pobierania informacji takich jak: • wytwórca samochodu, • marka oraz model, • typ nadwozia, • typ paliwa, • typ silnika

4.2. Diagramy UML, schemat bazy danych

4.2.1. Schemat bazy danych

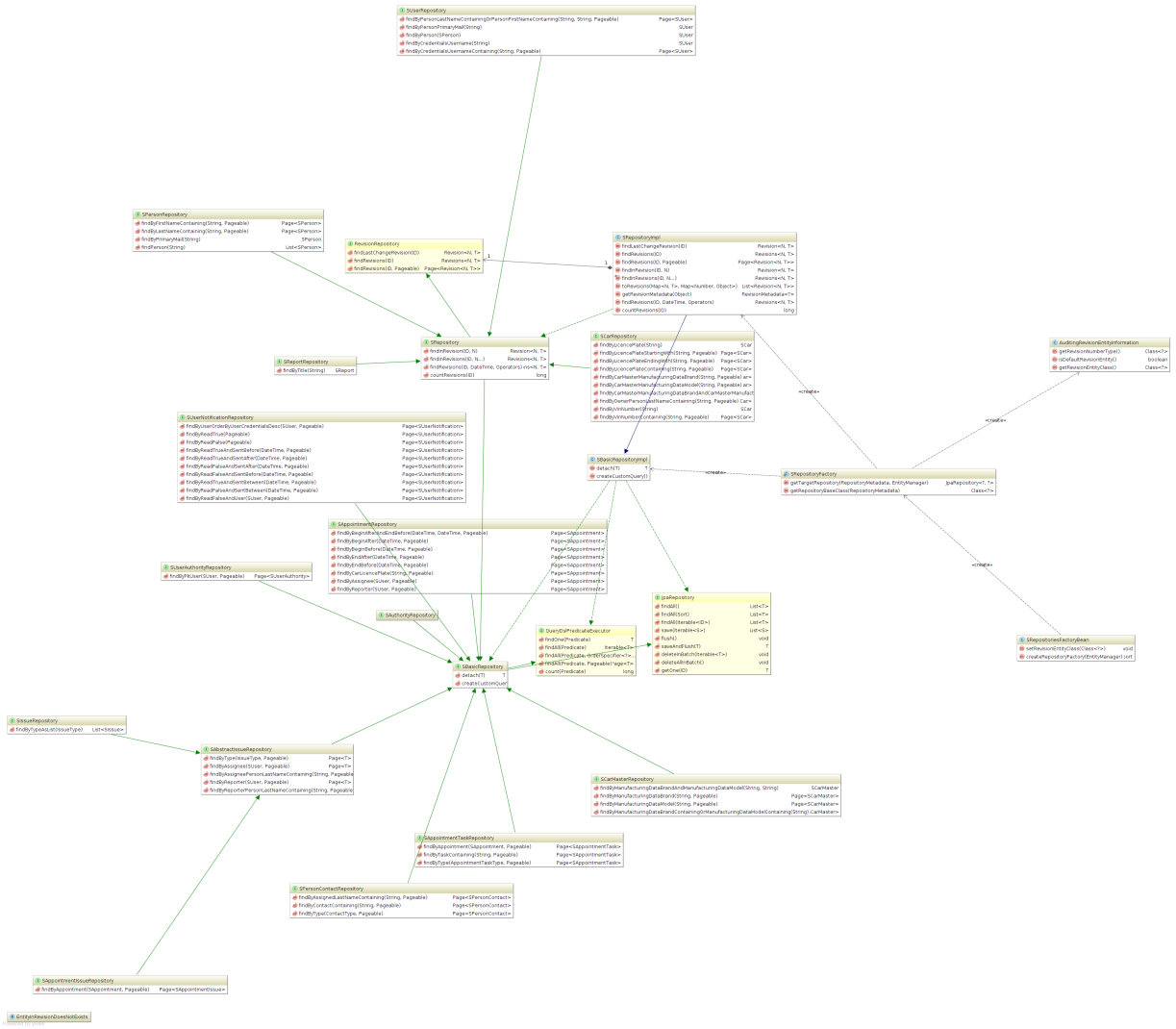


Rysunek 20: *Schemat bazy danych aplikacji demonstracyjnej*



Rysunek 21: *Obiektowy model danych odpowiadający który odpowiada schematowi bazy danych*

4.2.2. Schemat warstwy repozytoriów



Rysunek 22: *Schemat repozytoriów aplikacji demonstracyjnej*

Add the rest of UMLs

4.3. Metryki kodu

Kod aplikacji jest zbiorem funkcjonalności zaimplementowanych zarówno dla części serwerowej jak i klienckiej. Z tego powodu liczba linii kodu została podzielona na odpowiednie grupy zgodne z językami użytymi do stworzenia aplikacji demonstracyjnej.

4.3.1. Liczba linii kodu aplikacji

Tabela 4: Liczba linii kodu według języka programowania

Język	**LOC**	**LOC (SC)**	**LOC (C)**	**LOC (EL)**
Java	29555	16628	9136	3791
JS	1044	?	?	?
JSP	2122	?	?	?
Razem	32721	16628	9136	3791

LOC	Całkowita liczba linii kodu
LOC (SC)	Liczba linii kodu źródłowego
LOC (C)	Liczba linii komentarzy
LOC (EL)	Liczba linii pustych

4.3.2. Java

Zadaniem tabel 4.3.1 (dodać resztę) jest zobrazowanie złożoności projektu, którego celem prócz dostarczenie finalnej funkcjonalności, jest zaprojektowanie kilku modułów do późniejszej publikacji oraz wykorzystania w innych projektach. Projekt składa się z 5 głównych modułów:

- **AOP** - funkcjonalność opierająca się o *Aspect Oriented Programming*, działająca globalnie na całą aplikację i wspomagające niektóre z funkcjonalności, bez konieczności implementacji metod w konkretnych metod lub wykorzystywaniu statycznych metod różnych klas pomocniczych. Działanie klas tego modułu jest wysoce transparentne a funkcje wykorzystywane i odsperarowane od reszty aplikacji są wykonywane wspierając ją. Niemniej jest pewna znacząca różnica. Nie ma potrzeby bezpośrednio wprowadzania zależności między klasami wykorzystujący programowanie aspektowe a klasami objętymi ich działanie,
- **Core** - zbiór artefaktów (klas, klas abstrakcyjnych, interfejsów oraz typów wyliczeniowych) przeznaczonych do wykorzystanie w innych miejscach aplikacji celem dostarczenie właściwej funkcjonalności,
- **Server** - zadaniem klas tego modułu jest dostarczenie modelu danych, wsparcia jego walidacji oraz wersjonowania, definicje interfejsów repozytoriów oraz serwisów odpowiedzialnych za logikę biznesową oraz funkcjonalność dla operacji na plikach XML.
- **Web** - artefakty tego modułu stanowią zbiór zarówno klas wspomagających funkcjonalność warstwy MVC (wszelakiego rodzaju modelowania akcji dla warstwy widoku, pełna definicja modułu komponentów, definicje tagów wspierających przewodniki definiowane na bazie **Spring Web Flow**, moduł raportowania, czy też ostatecznie wszelakiego rodzaju klasy pomocnicze przeznaczone dla tak zwanych *interceptor*'ów[3].
- **WebMVC** - zbiór wszelakich artefaktów przeznaczonych do implementacji warstwy MVC. W zbiorze artefaktów tego modułu znajdują się między innymi kontrolery, konwertery danych, wspomniane już wcześniej *interceptor*'y (korzystające z klas zdefiniowanych w modelu **Web** celem dostarczania danych dla widoków, definicje klas zajmują-

[3] Interceptor - specjalne klasy które można wpiąć w dowolny etap wywołania danego widoku po adresie URI celem wprowadzenia danych bądź zmiennych

cych się dostarczaniem definicji oraz danych dla komponentów takich jak tabele i strony domenowe obiektów.

Strukturalne metryki kodu

Tabela 5: Liczba klas / Liczba linii kodu modułów

Moduł	LK	LOC	CD	D	TD
Aop	3/3	192/192	0	0	0
Core	13/1.86	628/89.71	0	0.25	0.25
Server	187/3.07	10806/183.15	0.82	3.32	23.48
Web	188/2.89	10803/166.20	0.3	3.28	14.01
WebMVC	37/2.85	2262/174	0	4.27	37.73

LK	Liczba klas/Średnia liczba klas
LOC	Liczba linii kodu/Średnia liczba linii kodu
D	Średnia liczba cyklicznych zależności
CD	Średnia liczba zależności
TD	Średnia liczba zależności przechodnich

Metryka Chidamber Kemerer

Zadaniem metryki jest analiza następujących metryk kodu [6]:

- **WMC** - liczba metod zdefiniowanych w klasie,
- **DIT** - głębokość drzewa dziedziczenia,
- **SUB** - liczba bezpośrednich potomków w hierarchii dziedziczenia,
- **CBO** - stopień zależności od pozostałych artefaktów,
- **RFC** - responsywność klasy, czyli możliwość jej odpowiedzi na wywołania przez inne obiekty innych klas,
- **LCOM** - stopień kohezji, im wyższy oznacza większą zależność między poszczególnymi elementami

Tabela 6: Metryka Chidamber - Kemerer

Moduł	CBO	DIT	LCOM	RFC	SUB	WMC
Aop	0	1.00	2.33	12.00	0	6
Core	2.25	1.50	1.00	34.60	0.62	5.12
Server	6.50	2.35	1.64	282.30	0.77	7.16
Web	5.78	2.03	1.74	194.33	0.64	7.47
WebMVC	5.00	2.94	1.33	213.22	0.18	5.03
Razem	5.78	2.22	1.65	222.44	0.62	6.98

Na uwagę zasługują w tym miejscu niskie wartości takich współczynników jak **DIT** gdzie średnia wartość nie przekroczyła wartości 3 począwszy od korzenia wszystkich klas definiowanych w języku Java - **Object**. Przyjmuje się, że wartość graniczna dla większości aplikacji wynosi 5. Niemniej wartość średnia nie oddaje pojedynczych przypadków nadużyć. Większość przypadków występujących aplikacji demonstracyjnej, które przekraczają graniczną wartość, odnosi się do klas rozszerzających standardowe możliwości szkieletu aplikacji celem dostosowania ich do konkretnych przypadków użycia. Dzieje się tak w przypadku wprowadzanie wsparcia dla biblioteki **Apache Tiles 3** dla częściowego ładowania stron w szczególni dla modułów opierających się na **Spring Web Flow**. Inną grupą przypadków jest głębokość dziedziczenia przekraczająca przyjętą wartość w klasach opisujących biznesowy model danych. Fakt ten można pominąć z uwagi na to, że wspomniane artefakty służą wsparciu dla dziedziczenia wspólnych atrybut dla konkretnych gałęzi klas oraz tym, że są to w klasy definiujące, prócz wspomnianych już pól, metody dostępowe takie jak popularnie nazywane **gettery** oraz **settery**. W nielicznych przypadkach część funkcjonalności biznesowej została zamknięta w obiektach domenowych z uwagi na rozbieżność w sposobie przechowywania danych a typami jakie są udostępnienie zewnętrznym klasom.

W tym miejscu warto nadmienić o wartości jaką uzyskano dla wskaźnika **SUB**, który jest blisko związany z poprzednio omawianym **DIT**. Podczas gdy **DIT** opisuje głębokość drzewa dziedziczenia, co przekłada się na zwiększenie zarówno ilości atrybut jak i metod będących kandydatami do ponownego wykorzystania (nadpisania), **SUP** odnosi się do szerokości drzewa dziedziczenia, czyli ilości dzieci będących bezpośrednimi potomkami analizowanej klasy. Przyjęto, że niska wartość **DIT** jest zdecydowania lepsza od **SUB**. Tak też jest w przypadku aplikacji demonstracyjnej, gdzie wartości tych dwóch współczynników charakteryzują się następującymi wartościami **DIT=2.22** oraz **SUB=0.62**. Widać wyraźnie, że zdecydowana większość klas definiuje swoją rolę poprzez mechanizm polimorfizmu.

Największym problem aplikacji okazała się wysoka wartość współczynnika **RFC**. Im jest ona wyższa tym bardziej aplikacja narażona jest na błędy, a istniejąca złożoność utrudnia zrozumienie oraz testowanie aplikacji.

Metryka MOOD

Zadaniem metryk **MOOD** zostały zaprojektowane do mierzenia globalnej jakości projektu OOP[1]. Analiza projektu tymi metrykami jest szczelnie użyteczna dla dużych projektów zaprojektowanych z użyciem technik programowania obiektowego. Duża ilość klas projektu oraz istniejące na tym etapie plany rozwojowego sugerujące dalszy wzrost ilości linii kodu sprawiają, że powyższe analizy stanowią dobry wybór.

Tabela 7: Metryka MOOD

Projekt	**AHF**	**AIF**	**CF**	**MHF**	**MIF**	**PF**
X	100.0%	0.00%	0.00%	53.85%	0.00%	100.0%

AHF	Współczynnik enkapsulkacji pól klas
AIF	Współczynnik dziedziczenia atrybutów
CF	Współczynnik powiązań
MHF	Współczynnik enkapsulkacji metod
MIF	Współczynnik dziedziczenia metod
PF	Współczynnik polimorfizmu

Analiza wyników Na istotną uwagę zasługują dwa wskaźniki **PF=100%** oraz **MHF=53.85%**. Pierwszy z wyników odnosi się do polimorfizmu. Wynik jest z pewnością wskazuje na dobry projekt systemu ponieważ projekt wykazuje dużą abstrakcyjność co usprawnia późniejsze modyfikacje na poziomie zmiany sposobów realizacji konkretnych bloków funkcjonalnych. Jedyną stałą takiej zmiany jest interfejs opisujący rolę danej implementacji. Drugi ze wskaźników odnosi się współczynnika enkapsulacji metod. Obliczany jest z następującego wzoru:

$$MHF = 1 - \frac{\sum MV}{(C-1)}$$

, gdzie **MV** - liczba klas gdzie dana metoda jest widoczna oraz **C** - ilość klas. Nie ma jednoznacznie przyjętej poprawnej wartości tej metryki, niemniej uznaje się, że im wyższa wartość tym jakość kodu jest większa, a potencjalne błędy skupione i łatwe do zlokalizowania. Z drugiej strony wysoka wartość oznacza wysoką specjalizację klas przy jednocześnie niskim poziomie funkcjonalności, która rozsiana jest między poszczególnymi elementami systemu.

5. Podsumowanie

Głównymi celami pracy było zaprojektowania oraz przygotowanie aplikacji wspierającej warsztat samochodowy w zakresie zarządzania terminarzem wizyt, bazą klientów oraz informacjami o samym warsztacie oraz poznanie szkieletu aplikacji Spring jako kompleksowego narzędzia wspierającego tworzenie rozbudowanych aplikacji **Java EE**. Znaczenia oraz korzyści jakie przynosi korzystania z tego typu aplikacji nie sposób nie zauważyć. Dobrze zaprojektowana jest cennym dodatkiem wspomagającym pracę przedsiębiorstwa w zakresie zarówno chwili obecnej jak i analizy danych historycznych. Program obejmujący swym zasięgiem globalny aspekt misji danej firmy przy jednoczesnym dostępie do poszczególnych elementów.

W kontekście aplikacji demonstracyjnej nie udało się zrealizować wszystkich zamierzonych postulatów. Niemniej część z brakujących elementów, z uwagi na pracę oraz czas poświęcony na zaprojektowania niewidocznych dla użytkownika elementów, co wcale nie umniejsza ich znaczenie, będzie możliwa do szybkiego wprowadzenia do gotowego rozwiązania. Pozostałe braki wynikają głównie z problematyki rozwiązania poszczególnych problemów i dla celu aplikacji demonstracyjnej zostały pominięte. Mimo że aplikacja przygotowana w ramach projektu inżynierskiego nie jest w pełni funkcjonalna to wiele z jej modułów oraz komponentów będzie stanowić o jej sile w momencie gdy braki zostaną uzupełnione. Generyczny kod, który znaleźć można praktycznie w dowolnym miejscu oraz nacisk położony na wykorzystanie zestawu bibliotek dodatkowych sprawiają, że późniejsze zmiany czy też usprawnienia nie będą trudne.

Udało się natomiast dobrze poznać podstawowe prawa i reguły rządzące pisaniem rozbudowanego projektu zarówno w sensie ogólnym oraz w kontekście framework'a Spring. Rzeczy, które mogą się wydawać trywialne, jak dobry dobór bibliotek, projekt przed implementacją, testy jednostkowe, a które zostały zaczerpnięte z zestawu **best practices** dla Spring pozwoliły zrozumieć na co zwracać szczególną uwagę oraz jak ważne mogą stać się małe pomyłki.

Bibliography

[1] Fernando Brito e Abreu. *MOOD Metrics*. URL: http://www.aivosto.com/project/help/pm-oo-mood.html.

[2] *c3p0 - JDBC3 Connection And Statement Pooling*. URL: http://www.mchange.com/projects/c3p0/.

[3] Neal Ford. *Art of Java Web Development*. Manning Publications, 2004.

[4] Daniel Rubio Gary Mak Josh Long. *Spring Recipies*. ISBN: 978-1-4302-2499-0. Apress, 2010.

[5] *Java Persistence Query Language*. URL: http://docs.oracle.com/javaee/7/tutorial/doc/persistence-querylanguage.htm.

[6] Chidamber Kemerer. *Chidamber Kemerer object-oriented metrics suite*. URL: http://www.aivosto.com/project/help/pm-oo-ck.html.

[7] Thomas Risber Jonathan L. Brisbin Michael Huner Mark Pollack Olivier Gierke. *Spring Data: Modern Data Access For Enterprise Java*. ISBN: 978-1-449-32395-0. O'Reilly, 2013.

[8] Roy Osherove. *The Art of Unit Testing*. ISBN: 978-1-933988-27-6. Manning Publications, 2009.

[9] Dirk Riehle. "Framework Design: A Role Modelling Approach". PhD thesis. ETH Zrich, 2000.

[10] Wydawnictwo Naukowe PWN SA. *S?ownik J?zyka Polskiego*. URL: polish_dictionary.

[11] Pivotal Software. *Spring Framework Reference Documentation*. 2014. URL: http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/.

[12] Pivotal Software. *Spring Web Flow Reference Documentation*. 2014. URL: http://docs.spring.io/spring-webflow/docs/2.4.x/reference/htmlsingle/.

[13] Pivotal Software. *Understanding HATEOAS*. 2013. URL: http://spring.io/understanding/HATEOAS.

[14] Terracotta. *Ehcache - Documentation*. 2014. URL: http://ehcache.org/documentation/index.

Spis tabel

Spis rysunków

Kody źródłowe