

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2013/2014

Praca dyplomowa inżynierska

Andrzej Niedźwiedź

**System ewidencji zabytków
archeologicznych przy użyciu
technologii Vaadin**

Opiekun pracy:
dr inż. Jakub Janusz Koperwas

Ocena

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego



Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 12 grudnia 1991 r.

Data rozpoczęcia studiów: 1 października 2010 r.

Życiorys

Urodziłem się 12 grudnia 1991 roku w Zamościu. W 2010 roku ukończyłem Społeczne Liceum Ogólnokształcące im. Unii Europejskiej w Zamościu. W październiku 2010 roku rozpocząłem studia na kierunku Informatyka na Wydziale Elektroniki i Technik Informatycznych na Politechnice Warszawskiej. Od trzeciego roku studiów jestem na specjalizacji Inżynieria Systemów Informatycznych.

.....

podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dn.

Z wynikiem

Ogólny wynik studiów

Dodatkowe wnioski i uwagi Komisji

.....

Streszczenie

Przedmiotem niniejszej pracy jest stworzenie systemu wspierającego ewidencje i tworzenie dokumentacji badań archeologicznych. Aplikacja ta została zaimplementowana z użyciem technologii zaproponowanej przez promotora - szkieletu aplikacji Vaadin - której zbadanie jest równoległym celem. Początek pracy poświęcony jest opisowi zakresu pracy. Następnie przedstawiony został opis systemu oraz wymagania mu stawiane. W dalszej części pracy są wymienione i opisane technologie, które zostały użyte do stworzenia systemu. Dalej przedstawiony jest opis implementacji z naciskiem na architekturę rozwiązania. W kolejnej części znajduje się przedstawienie produktów, które powstały przy okazji tworzenia systemu. Ostatnia część pracy zawiera podsumowanie oraz wnioski płynące z prac nad systemem.

Słowa kluczowe: Vaadin, Archeologia, Dokumentacja Archeologiczna

Abstract

Title: *System to support the creation of archeological documentation created using Vaadin framework*

This thesis describes web application using three-tier architecture and Java programming language, which purpose was to support making documentation after archaeological research. The following chapters present stages of application preparation. Whole graphic interface was made in framework Vaadin - which is described in detail in one of the chapters

Key words: Vaadin, Archeology, Archaeological Documentation

Spis treści

1. Wstęp	5
1.1. Cele i zakres pracy	6
1.2. Układ pracy	6
2. Analiza dziedziny problemu	7
2.1. Wymagania biznesowe	7
2.2. Słownik dziedziny problemu	8
2.3. Model dziedziny problemu	9
2.4. Model przypadków użycia	11
3. Wykorzystane technologie	13
3.1. MySQL	13
3.2. Hibernate	14
3.3. Spring Transaction Management	15
3.4. Spring	15
3.5. Logback + SLF4J	16
3.6. AspectJ	16
3.7. SpringSecurity	17
3.8. LDAP - Apache Directory Server	17
3.9. Jasper reports	18
3.10. Tomcat	18
3.11. Vaadin	19
3.11.1. Zdalne wywoływanie procedury	20
3.11.2. Architektura	21
3.11.3. Aplikacja po stronie klienta	23
3.11.4. Nowe kontrolki	23
3.11.5. Style	23
3.11.6. Powiązanie danych	24
3.12. Podsumowanie	25
4. Architektura rozwiązania	26
4.1. Model Widok Prezenter	26
4.2. Podział na pakiety	27
4.3. Nawigacja między widokami	28
4.4. Maksymalna abstrakcyjność	28
5. Prezentacja aplikacji	35
6. Dodatkowe produkty pracy	41
6.1. Przykładowa klasa modelu danych	41
6.2. CRUDTable	43
6.3. ForeignField	44
6.4. DefaultForm	45
6.5. Szkielet aplikacji przetwarzania danych biznesowych	47
7. Podsumowanie	48
Bibliografia	49

1. Wstęp

W ostatnich latach, dzięki dynamicznemu rozwojowi informatyki, cyfryzacja jest widoczna w coraz to większej ilości gałęzi gospodarki. Nowoczesne przedsiębiorstwa zaczynają widzieć znaczne oszczędności dzięki wprowadzaniu do codziennej pracy systemów informatycznych. Szczególnym przykładem czynności, które mogłyby być a nawet powinny zostać zautomatyzowane są prace związane z wypełnianiem dokumentów, w których informacje wprowadzane przez pracownika są powielane w wielu miejscach. W bardziej popularnych branżach, rozwiązania usprawniające taką pracę są coraz częściej używane, jednak są także profile działalności, w których informatyzacja byłaby pożądana, ale nie jest wprowadzana.

Jednym z przykładów działalności, gdzie systemy informatyczne usprawniłyby pracę, jest firma prowadząca badania archeologiczne. Zwykle prace na stanowisku archeologicznym sprowadzają się do fizycznego przebadania pewnego obszaru, a następnie stworzenia dokumentacji podsumowującej wykonane prace, czyli opisującej obiekty archeologiczne oraz zabytki, które zostały znalezione w trakcie badań.

W ostatnich latach, polski rynek badań archeologicznych staje się coraz większy, dzięki inwestycjom związanym z Unią Europejską. Badania archeologiczne prowadzone są na coraz większym obszarze, a co za tym idzie budżety firm archeologicznych są coraz większe. Niestety, wraz ze wzrostem skali badań, wzrasta też ilość wykonywanej dokumentacji. Sytuację komplikuje fakt, że w różnych województwach wymagania są nieco inne. Dodatkowo, wraz ze wzrostem nakładów finansowych inwestowanych w badania archeologiczne zwiększyła się także liczba przedsiębiorstw, które rywalizują ze sobą w przetargach. Większa konkurencja wymusza oczywiście spadek stawek za przeprowadzenie badań, dlatego firmy archeologiczne zmuszone są szczególnie do szukania oszczędności, których może dostarczyć cyfryzacja.

Na szczęście wraz z rozwojem informatyki idzie rozwój narzędzi do tworzenia systemów informatycznych. Szczególną gałęzią informatyki, która dzięki rozwojowi internetu zyskała na znaczeniu, są technologie tworzenia aplikacji internetowych. Jedną z nich, która w niniejszej pracy została wybrana do stworzenia systemu, jest szkielet aplikacji Vaadin, który umożliwia szybkie tworzenie stron WWW.

1.1. Cele i zakres pracy

Głównym celem niniejszej pracy jest stworzenie systemu dla firmy "JN-Profil-badania archeologiczne i historyczne" wspierającego ją w dokumentowaniu badań archeologicznych. Drugim celem jest zapoznanie ze szkieletem aplikacji Vaadin, ułatwiającym tworzenie aplikacji internetowych. Dodatkowym, choć nieobowiązkowym celem jest stworzenie komponentów graficznych ułatwiających wyświetlanie list obiektów i ich cech a także usprawniających tworzenie formularzy.

System ewidencji zabytków archeologicznych tworzony w ramach pracy powinien umożliwiać wprowadzanie danych i generowanie dokumentacji archeologicznej w jak najbardziej przystępny i intuicyjny sposób. Aplikacja powinna także umożliwiać wprowadzanie danych w dowolnym miejscu, ze względu na charakter działalności firmy archeologicznej - praca w różnych miejscach kraju.

1.2. Układ pracy

Rozdział 2. zawiera opis systemu ewidencji zabytków archeologicznych od strony inżynierii oprogramowania.

W rozdziale 3. znajduje się opis użytych technologii z uzasadnieniem wyboru.

Rozdział 4. zawiera opis technologiczny szkieletu aplikacji Vaadin, którego zbadanie jest celem niniejszej pracy.

Rozdział 5. opisuje proces tworzenia aplikacji oraz architekturę rozwiązania.

W rozdziale 6. jest opisany proces testowania aplikacji.

Rozdział 7. opisuje produkty, które zostały dodatkowo wytworzone w trakcie budowy aplikacji.

Rozdział 8. zawiera wnioski wyciągnięte w procesie budowy systemu.

W rozdziale 9. znajduje się podsumowanie rezultatów pracy.

2. Analiza dziedziny problemu

W dzisiejszych czasach, przed firmami archeologicznymi stoi nie małe wyzwanie, polegające na przeprowadzaniu badań archeologicznych na coraz większych powierzchniowo obszarach. Przedinwestycyjne badania wykopaliskowe (w których specjalizuje się firma "JN-Profil- docelowy użytkownik oprogramowania stworzonego w ramach niniejszej pracy) polegają na przeprowadzeniu badań terenowych oraz stworzeniu dokumentacji podsumowującej rezultaty tych badań.

Dokumentacja archeologiczna zawiera w sobie dużą liczbę dokumentów, z których co najmniej część opiera się na tych samych danych, których proces przetwarzania można, a nawet powinno się zautomatyzować. Do tej pory tworzenie dokumentacji sprowadzało się do mozolnego kopiowania danych z jednego miejsca do drugiego, jednak dzięki współcześnie dostępnym technologiom informatycznym istnieje możliwość zautomatyzowania i przyspieszenia tego procesu - co w szerszej perspektywie prowadzi do zmniejszenia kosztów przeprowadzania badań archeologicznych.

2.1. Wymagania biznesowe

Przed systemem do prowadzenia ewidencji badań archeologicznych jest stawiany szereg wymagań, których spełnienie jest warunkiem poprawnego funkcjonowania oraz zadowolenia użytkowników. Część z nich jest wspólna dla wszystkich systemów informatycznych, natomiast reszta jest związana z charakterem pracy archeologa.

Aby system nadawał się do profesjonalnego użytku, konieczne jest, aby efekty pracy były zapisane w pamięci trwałej. Konieczne jest także, aby możliwe było równoległe korzystanie wielu użytkowników, z dodatkowym zastrzeżeniem, że równoległe sesje użytkowników mogą pracować na tych samych danych i system powinien mimo to gwarantować ich spójność.

Specyficzną cechą pracy archeologa jest konieczność wyjazdów i prowadzenia badań archeologicznych w różnych miejscach. System wspierający prowadzenie ewidencji badań archeologicznych powinien umożliwiać tworzenie dokumentacji z różnych miejsc i urządzeń.

2.2. Słownik dziedziny problemu

Na samym początku opracowywania systemu dla przedsiębiorstwa należy zdefiniować wszystkie elementy występujące w codziennej pracy, aby nie było wątpliwości co do zrozumienia tematu.

- Obiekt - obiekt archeologiczny, główny cel badań. Obiekt to prawdopodobny ślad działalności człowieka, bardzo często jest postaci pozostałości po palenisku.
- Zabytek wydzielony - zabytek większej wartości, dokumentowany i opisywany osobno. Na przykład - trzonek siekierki.
- Zabytek masowy - kilka zabytków, zgrupowanych w pozycjach dokumentacji jako jeden. Zazwyczaj składa się z niewielkich przedmiotów, np. krzemień czy ceramika o wielkości kilku centymetrów kwadratowych
- Rysunek - sposób dokumentowania zabytków i obiektów. Zazwyczaj rysunek przedstawia rzut albo profil obiektu lub obrys zabytku.
- Zdjęcie - sposób dokumentowania obiektów, rzadziej zabytków. Najczęściej powstają w trakcie eksploracji obiektów, tj zaraz po odsłonięciu obiektu (zdjęcie rzutu), następnie po wyeksplorowaniu połowy (profil) i na końcu zdjęcie wyeksplorowanego obiektu (negatyw).
- Ar - jednostka powierzchni wykopalisk archeologicznych. Punkt odniesienia dla wszelkich informacji w dokumentacji.
- Typ zabytku - Podział zabytków (masowych) ze względu na typ zabytku. Klasyfikacja najczęściej pojawiających się zabytków, np. fr. ceramiki, okruch (Krzemienia)
- Grupa zabytków - Bardzo ogólny podział zabytku, zazwyczaj podział jest na trzy grupy (Ceramika, Krzemień, Inne)
- Profil - przekrój wyeksploatowanego obiektu, jeden z dokumentowanych elementów
- Rzut - widok obiektu z góry, przed eksploatacją
- Calec - rodzaj otoczenia znajdującego się dookoła obiektu, jednak nie będący nim, np. less.
- Kultura - zespół stale współwystępujących ze sobą na pewnym terytorium i w pewnym czasie charakterystycznych form źródeł archeologicznych.
- Funkcja - określona w ramach badań obiektu jego obecność i przyczyna jego powstania
- Chronologia - pozwala na określenie wieku znaleziska
- Eksploracja obiektu - proces badania obiektu archeologicznego polegająca na wydobywaniu jego zawartości w międzyczasie udokumentowując kolejne etapy wykonywania badania

2.3. Model dziedziny problemu

W systemie wspierającym działalność firmy archeologicznej powinno się wyróżnić 6 najważniejszych elementów (opisanych w poprzednim podrozdziale):

- Zabytki Masowe
- Zabytki Wydzielone
- Obiekty Archeologiczne
- Zdjęcia
- Rysunki
- Ary

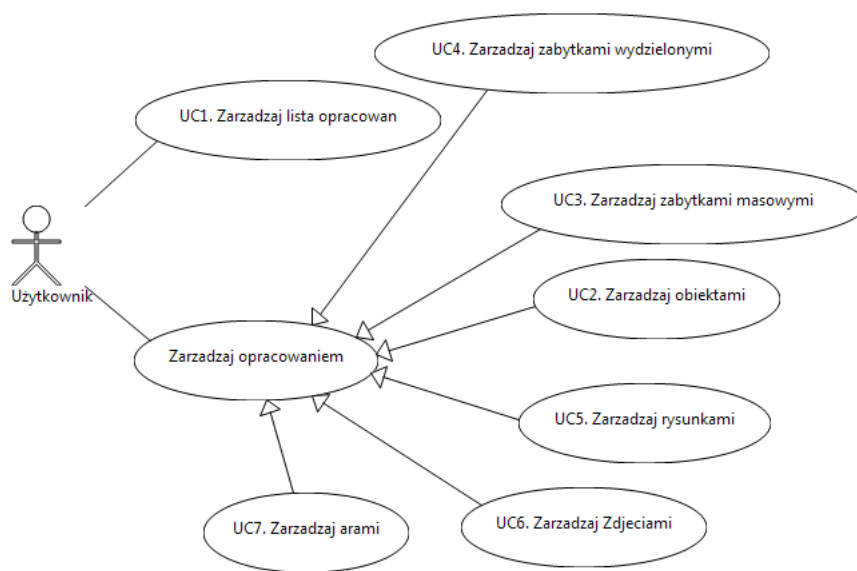
Wszystkie powyższe obiekty są produktem przeprowadzanego badania archeologicznego, dlatego są bytami, które mają sens tylko w konkretnym kontekście biznesowym (czyli opracowaniu). Wszystkie inne informacje wprowadzone do systemu są uniwersalne dla badań archeologicznych, dlatego są dostępne i edytowalne niezależnie od opracowania.



Jak widać, poza głównymi elementami bazy danych odpowiadającym elementom specyficznym dla badania występuje duża ilość encji słownikowych, co powoduje znaczne zwiększenie koniecznego nakładu pracy.

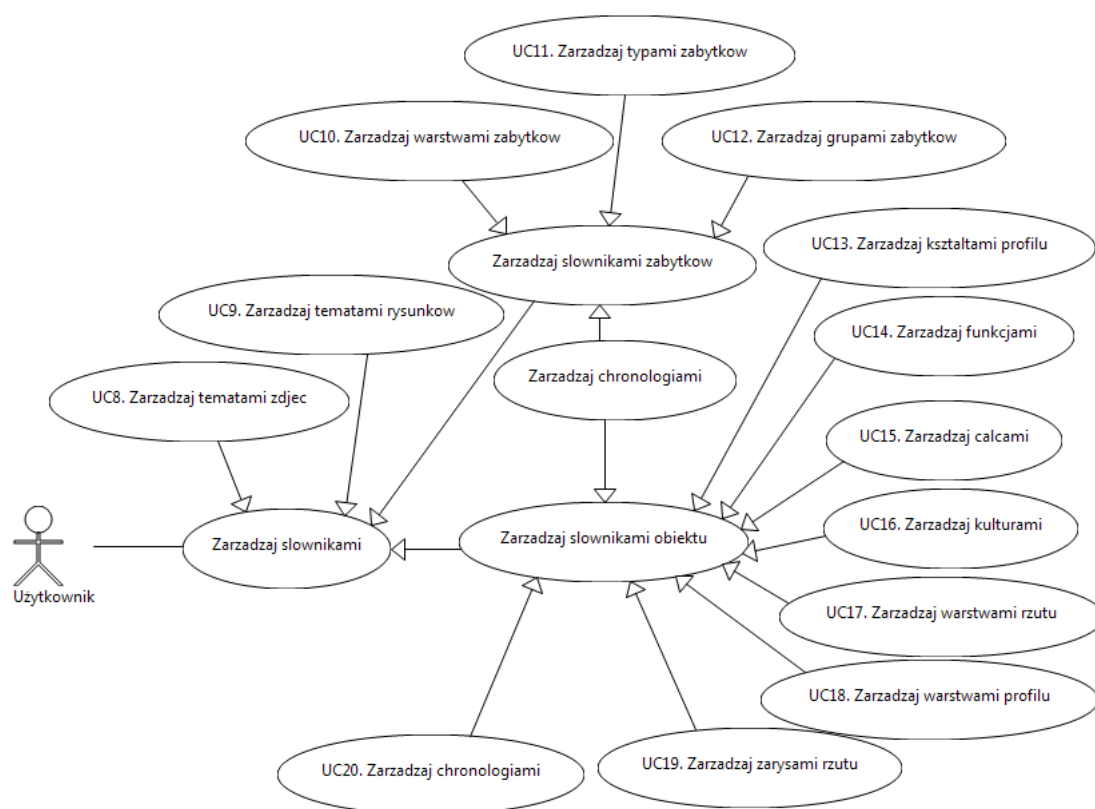
2.4. Model przypadków użycia

Zadaniem systemu wspierającego tworzenie dokumentacji archeologicznej powinno być umożliwienie wprowadzenia danych w łatwy i przejrzysty sposób oraz szybkie generowanie raportów, będących składnikami dokumentacji archeologicznej.



Rysunek 2.2. Diagram przypadków użycia dla obiektów specyficznych dla opracowania

Dodatkową cechą systemu będącego produktem niniejszej pracy jest to, że możliwa jest edycja słowników służących do wypełniania danych z poziomu okna edycji obiektu wypełnianego.

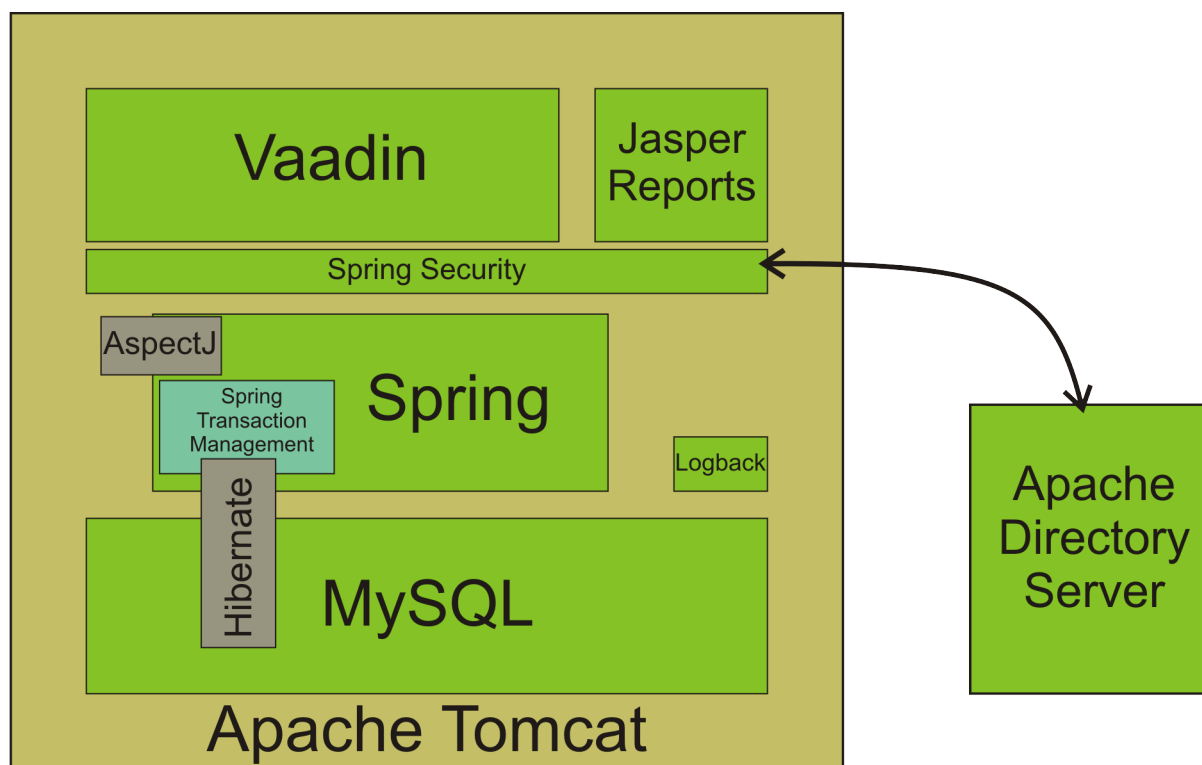


Rysunek 2.3. Diagram przypadków użycia dla obiektów uniwersalnych względem opracowania

Jak można zauważyć na podstawie diagramów przypadków użycia, każdy użytkownik ma takie same uprawnienia - tzn. może zarówno zarządzać listą opracowań, jak i modyfikować istniejące opracowanie, a także może edytować zawartość słowników.

3. Wykorzystane technologie

Na rynku istnieje bardzo wiele szkieletów aplikacji i bibliotek usprawniających tworzenie aplikacji, w szczególności systemów wspomagających prace przedsiębiorstw, dlatego wybór technologii nie jest oczywisty. Jednym ze wspólnych kryteriów stawianych technologiom przy wyborze do tego projektu, był wymóg nieodpłatności za ich używanie, dlatego całe poniżej wymienione oprogramowanie jest bezpłatne.



Rysunek 3.1. Stos technologiczny

3.1. MySQL

Jednym z najważniejszych wymagań spoczywających na systemie informatycznym jest utrwalanie pracy użytkowników i radzenie sobie z ich równoległą pracą. We współczesnej informatyce, rola ta jest zazwyczaj oddelegowywana do specjalnie dedykowanych do tego celu programów nazwanych ogólnie bazami danych. Na rynku jest wiele systemów baz danych dlatego wybór nie jest oczywisty.

Podstawowy podział baz danych, wg którego powinno rozpocząć się wybór oprogramowania to - bazy relacyjne i tzw. NoSQL. Wydaje się jednak, że nierelacyjne bazy danych mają przewagę w systemach, które są lub będą kiedyś duże, natomiast system ewidencji zabytków archeologicznych raczej nie ma szans być takim systemem. Z drugiej strony, relacyjne systemy baz danych były przez dłuższy czas uważane jako podstawowe, dlatego istnieje bardzo dużo informacji na ich temat w internecie, w przeciwieństwie to baz NoSQL, o których więcej informacji dopiero od niedawna.

Na rynku istnieje wiele relacyjnych systemów baz danych, jednak większość najpopularniejszych jest płatnych w rozwiązaniach komercyjnych. Wyjątkiem okazuje się MySQL, który pozwala korzystać w wersji bezpłatnej z bardzo szerokiej puli funkcjonalności, które w zupełności pokrywają potrzeby tworzonego przez autora systemu.

3.2. Hibernate

Aby aplikacja mogła korzystać z bazy danych konieczne jest skomunikowanie ich ze sobą. Na rynku istnieje wiele narzędzi wspierających pracę programisty, spośród których najpopularniejsze są JDBC i JPA. Ponieważ system będący wynikiem niniejszej pracy będzie operował na obiektach, warto byłoby użyć interfejsu programistycznego, który umożliwi zapisywanie całych obiektów.

Java Persistence API to oficjalny standard mapowania relacyjno-obiektowego, który pozwala na utrwalanie obiektów w relacyjnej bazie danych. Jedną z implementacji będącą jednocześnie najpopularniejszą jest Hibernate.

Tablica 3.1. Liczba wątków z pytaniami związanymi z implementacją JPA

Implementacja JPA	Ilość wątków
Datanucleus	682
EclipseLink	2745
Hibernate	37787
TopLink	240
OpenJPA	715

Ze względu na niewielką styczność autora z dostępem do danych przez JPA, wybór padł na najbardziej popularny szkielet aplikacji - Hibernate.

3.3. Spring Transaction Management

Elementem nierozłącznym z relacyjną bazą danych jest transakcyjne przetwarzanie danych. Również w tym temacie istnieje wiele rozwiązań, dlatego nie ma potrzeby ręcznego implementowania fragmentów kodu odpowiedzialnego za zatwierdzanie bądź odrzucanie transakcji. W przypadku systemu tworzonego w ramach niniejszej pracy logika transakcji nie jest skomplikowana, dlatego im prostsza implementacja, tym bardziej preferowana.

Wybrane przez autora rozwiązanie, Spring Transaction Management, umożliwia wpłatanie transakcji w istniejący kod w sposób przezroczysty. Po dodaniu do pliku konfiguracyjnego springa 4 linijek, programista jest w stanie tworzyć operacje na bazie danych w transakcjach po dopisaniu zaledwie jednej adnotacji definicją funkcji, w obrębie której odbywać się mogą operacje na bazie danych.

```
@Transactional
public void add(Figure f)
{
    getDaoObj().add(f);
}
```

W jaki sposób jest otrzymywany powyższy efekt? Spring tworzy obiekt zdefiniowany przez programistę i opakowuje je w obiekt pośredniczący, który deleguje wszystkie operacje do obiektu właściwego, natomiast operacje oznaczone adnotacją `@Transactional` opakowuje w kod odpowiedzialny za transakcyjne przetwarzanie danych.

Inną możliwością było użycie jednej z implementacji JTA jednak nie dałoby się wykorzystać jej funkcjonalności, ze względu na to, że wszystkie transakcje odbywają się na bazie danych.

3.4. Spring

Jeszcze niedawno dość dużym wyzwaniem było zarządzanie drzewem zależności obiektów wewnątrz aplikacji. W ostatnich latach została jednak stworzona koncepcja wzorca projektowego polegającego na wstrzykiwaniu zależności bezpośrednio do obiektów ich używanych. Implementacja tego wzorca jest jeszcze łatwiejsza, dzięki powstaniu wielu bibliotek wspierających tego typu podejście.

Spring jako szkielet aplikacji zyskał na popularności przede wszystkim właśnie dzięki kontenerowi IoC (ang. Inversion-of-Control), który pozwala na łatwe tworzenie i zarządzanie zależnościami między obiektami. Wzorzec odwrócenia sterowania, będący pojęciem szerszym niż wstrzykiwanie zależności, pozwala skupić się na właściwej implementacji logiki a pozostawić zarządzanie zależnościami kontenerowi Springa, co znacznie ułatwia pracę.

W dzisiejszych czasach Spring jest jednym z najpopularniejszych szkieletów aplikacji, dlatego jednym z argumentów będących za jego użyciem jest bardzo duża ilość stron internetowych opisujących go oraz rozwiązujących potencjalne problemy w trakcie pracy z nim.

3.5. Logback + SLF4J

Bardzo ważnym zagadnieniem, w kontekście tworzenia systemu informatycznego jest tworzenie logów aplikacji. Na rynku istnieje kilka rozwiązań, dlatego wybór nie jest oczywisty. Aby uniezależnić się od wybranego rozwiązania, oprócz implementacji biblioteki wspierającej logowanie, autor zdecydował się na przykrycie logowania warstwą abstrakcji umożliwiającą nieinwazyjne przełączenie się między realizacjami standardu logowania - SLF4J.

Wybór w kwestii implementacji szkieletu aplikacji zapewniającego logowanie odbył się pomiędzy dwoma najpopularniejszymi - log4j oraz logback. Okazało się jednak, że logback jest kilkukrotnie szybszy niż log4j i zużywa przy tym mniej pamięci (jak twierdzą autorzy tego pierwszego¹). Dodatkowo, przy zapoznawaniu się z SLF4J okazało się, że logback wspiera natywnie ten standard, dlatego nie ma potrzeby używania modułu pośredniczącego między slf4j i logbackiem, dzięki czemu można zyskać dodatkowo na szybkości działania aplikacji.

3.6. AspectJ

Programowanie aspektowe powstało, aby w łatwy sposób wpłatać pewne zachowania w wywołania metody. Najlepszym przykładem konieczności takiej funkcjonalności są transakcje, które Spring pod warstwą abstrakcji obsługuje właśnie w ten sposób w swoim module Spring Transaction Management, który został także użyty w systemie. Dodatkowo jednak zaistniała konieczność wychwytywania wyjątków wyrzucanych przez bazę danych w przypadku operacji zabronionych, takich jak łamanie unikalności lub innych ograniczeń nałożonych po stronie bazy danych.

¹<http://logback.qos.ch/reasonsToSwitch.html>

Na rynku najpopularniejszym rozwiązaniem jest AspectJ, który został użyty w niniejszej pracy do obsługiwanie wyjątków bazy danych. Poprzez zdefiniowanie klasy obsługującej błędy jako komponentu springa oraz zastosowaniu adnotacji definiującej punkt wplatania akcji autor uzyskał efekt obsługi błędu w jednym miejscu bez konieczności dostosowywania innych klas w tym celu.

3.7. SpringSecurity

Bardzo ważnym zagadnieniem, w implementacji systemów, szczególnie dostępnych w internecie, jest zapewnienie bezpieczeństwa. Aby dane gromadzone przez firmę nie wpadły w niepowołane ręce, a szczególnie nie uległy zniszczeniu, konieczny jest szczelny sposób autoryzacji i autentykacji. Także i w tej dziedzinie istnieje wiele rozwiązań wartych rozważenia.

Autor zdecydował się na SpringSecurity, będący odrębnym bytem w stosunku do rdzenia Springa, jednak w prosty sposób się z nim integrującym. Implementacja ogranicza się do dopisania kilku linijek do pliku konfiguracyjnego springa, stworzenia strony odpowiadającej za zalogowanie i dodanie filtru do deskryptora wdrożenia (plik web.xml). Zaletą wybranego szkieletu aplikacji jest mnogość źródeł danych służących w procesie uwierzytelniania.

Gdyby nie SpringSecurity, autor prawdopodobnie wybrał by rozwiązanie o nazwie Apache Shiro, jednak prostota integracji ze szkieletem aplikacji Spring zdecydowała o wyborze.

3.8. LDAP - Apache Directory Server

Innym zagadnieniem związanym z bezpieczeństwem jest miejsce przechowywania danych o użytkownikach, w szczególności ich loginów, haseł oraz uprawnień z nimi związanych. Standardowo, aplikacje przechowują takie informacje w bazie danych, jednak dla przedsiębiorstwa, którego liczba i skład pracowników zmienia się rzadko, mogą być także przechowywane w zewnętrznym serwerze takim jak LDAP.

Dodatkową korzyścią wynikającą z przechowywania danych użytkowników na serwerze LDAP jest możliwość posiadania centralnego punktu autoryzacji i autentykacji użytkowników w obrębie całej firmy.

3.9. Jasper reports

W systemie, który ma usprawnić tworzenie dokumentacji archeologicznej, kluczową sprawą jest generowanie raportów. Na rynku istnieje wiele rozwiązań nadających się do tego celu, jednak autor wahał się między użyciem iText oraz JasperReport. Obie biblioteki umożliwiają tworzenie raportów tabelarycznych, które będą tak naprawdę głównym rodzajem generowanej dokumentacji. W trakcie zapoznawania się z wyżej wymienionymi technologiami, okazało się, że JasperReports używa iText'a jako swojego silnika do generowania dokumentów w formacie PDF.

Argumentem przeważającym, który wpłynął na podjęcie decyzji było oprogramowanie wspierające tworzenie raportów w JasperReports, program iReport. Aplikacja ta pozwala generować wygląd dokumentów w edytorze z interfejsem graficznym. Tworzenie plików jest na zasadzie przeciągnij-i-upuść, do tego jest bardzo intuicyjne. Dodatkowo, narzędzie to jest w stanie komunikować się bezpośrednio z bazą danych, dzięki czemu można w szybki sposób sprawdzić czy stworzony wygląd raportu jest satysfakcjonujący czy wymaga jeszcze poprawek.

3.10. Tomcat

Standardowy proces wdrażania aplikacji internetowych wymaga oprogramowania, które dostarcza środowisko uruchomieniowe. Powstało wiele tzw. kontenerów aplikacji webowych, które pozwalają w prosty sposób uruchomić aplikacje napisane w języku Java, jednocześnie udostępniając je w sieci. Część z nich, zwana serwerami aplikacji udostępnia dużo większy zakres możliwości i funkcjonalności, jednak nie były brane pod uwagę, ze względu na stopień skomplikowania.

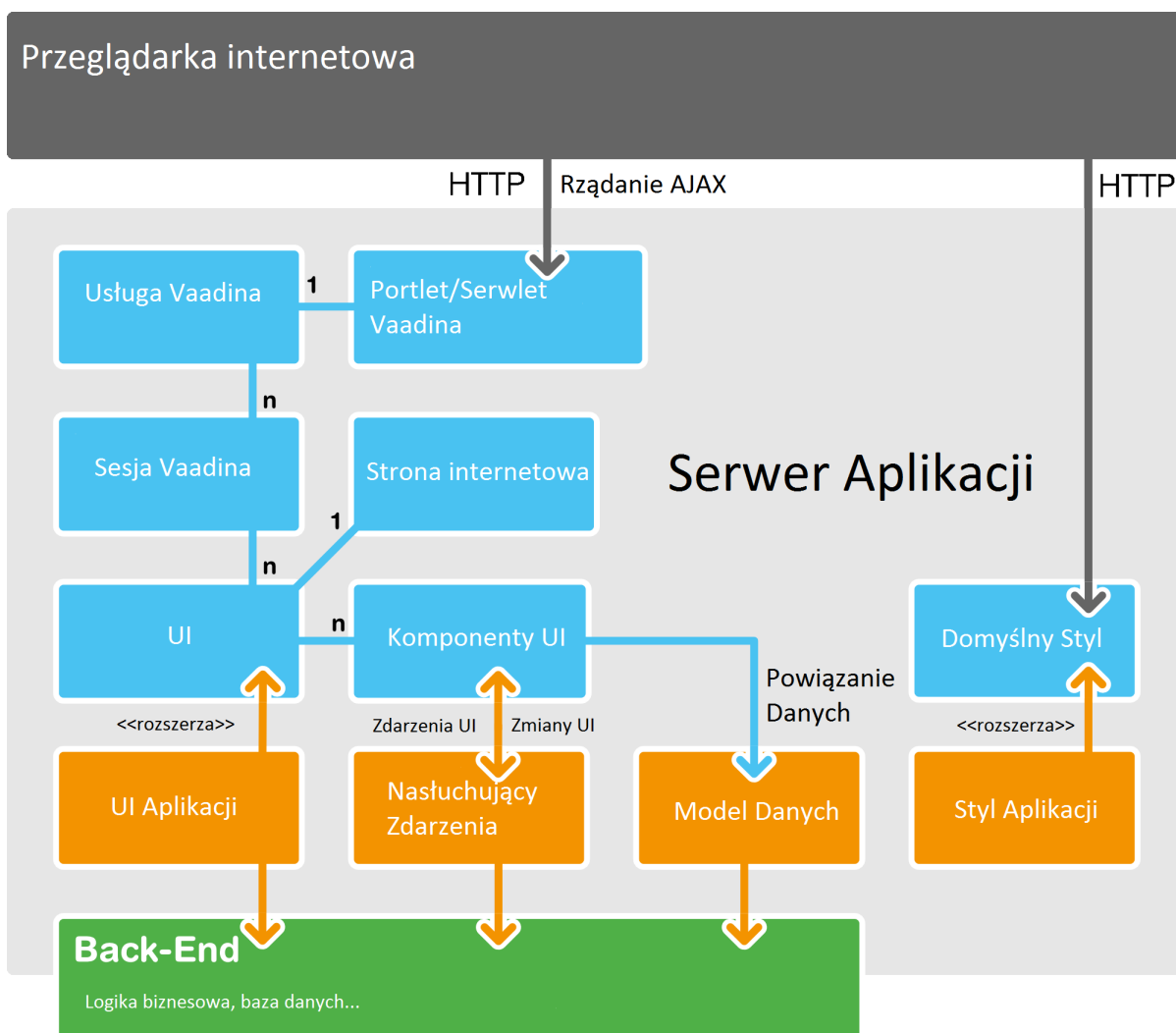
Wybór autora padł na Apache Tomcat będącym jednym z najbardziej popularnych kontenerem aplikacji webowych. Dużą zaletą tego oprogramowania, oprócz łatwości i prostoty rozwiązania, jest wsparcie środowiska programistycznego Eclipse, wybranego przez autora do stworzenia implementacji systemu, do szybkiego wdrażania oprogramowania jeszcze w trakcie jego rozwijania. Drugi kontener serwetów nad którym zastanawiał się autor to JBoss Application Server, jednak wersja serwera wspierana przez Eclipse jest już dosyć przestarzała. Dodatkowo, serwer aplikacji uruchamiał się dłużej, ze względu na o wiele większe możliwości, które jednak nie były konieczne w projekcie.

3.11. Vaadin

Standardowe szkielety aplikacji do tworzenia stron WWW oferują tworzenie oprogramowania opartego o podejście żądanie-odpowiedź. Jest to model zupełnie inny, niż standardowy sposób pisania tzw. aplikacji desktopowych (czyli programowanie sterowane zdarzeniami), od których zaczynają swoją informatyczną karierę początkujący programiści. Niestety, zmiana modelu pisania programów wymaga przestawienia swojego myślenia. Na szczęście istnieje jeszcze krok pośredni tzn. szkielety aplikacji, które umożliwiają tworzenie aplikacji internetowych poprzez programowanie sterowane zdarzeniami. Jednym z nich jest właśnie Vaadin - technologia zaproponowana przez promotora niniejszej pracy.

Vaadin jest szkieletem aplikacji, który umożliwia korzystanie z gotowych kontrolerek, aby tworzyć aplikacje internetowe korzystające z AJAX jedynie za pomocą jednego języka programowania (może to być Java, ale także inne języki działające na wirtualnej maszynie Javy). Podejście Vaadina polega na tym, że każde zdarzenie wywołane po stronie klienta (przeglądarka WWW), jest przetwarzane po stronie serwera – co powoduje, że część aplikacji znajdująca się po stronie klienta nie zawiera żadnej logiki. Rozwiązanie wykorzystane w Vaadinie do odbierania żądań i wysyłania odpowiedzi nie jest czymś nowym.

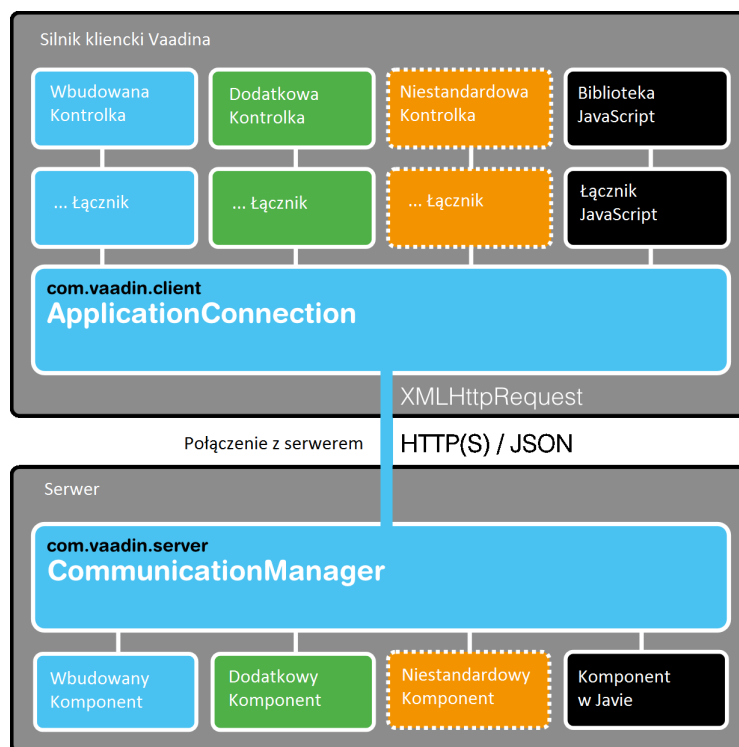
Elementem mającym styczność ze światem zewnętrznym jest serwlet (lub portlet) Vaadina, który tworzy obiekt usługi Vaadina. Do każdego połączenia usługa Vaadina przyporządkowuje sesję, w której obrębie przetrzymywane są zmienne. W ramach sesji użytkownik widzi wiele interfejsów (UI), które z kolei składają się z komponentów (omówione będą później), z którymi mogą być związani są nasłuchujący zdarzenia oraz model danych.



Rysunek 3.2. Architektura szkieletu aplikacji Vaadin po stronie serwera

3.11.1. Zdalne wywoływanie procedury

Cała tajemnica sukcesu Vaadin polega na zunifikowanym podejściu do obsługi zdarzeń. Każda kontrolka posiada swój łącznik, który jest podłączony do obiektu `ApplicationConnection` (jedna instancja na aplikacje) który odpowiada za komunikację z serwerem po stronie klienta. Z drugiej strony natomiast występuje swego rodzaju lustrzane odbicie – obiekt `CommunicationManager` jest połączony z każdym z komponentów.

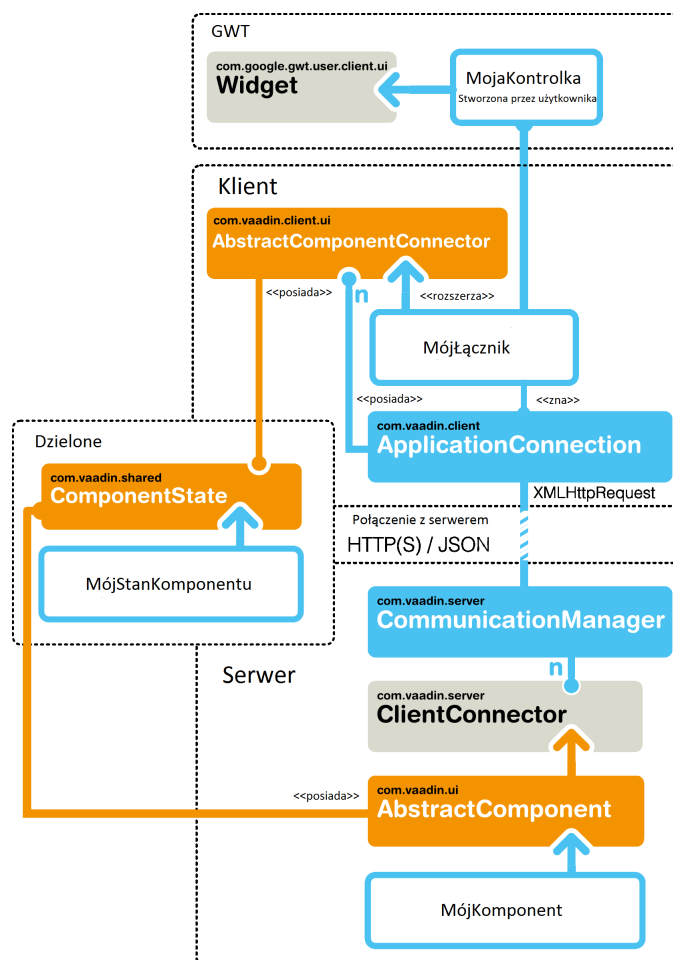


Rysunek 3.3. Komunikacja między kontrolką a komponentem

Efekt ten można uznać za odmianę zdalnego wywoływania procedury – tzn. kontrolka wykonuje akcje związaną z nią przez JavaScript (jest to przezroczyste dla programisty) – komunikuje się przez WebService z serwilem Vaadina, który obsługuje żądanie, i w zależności od rezultatu – albo wysyła nową stronę do klienta albo asynchronicznie zmienia zawartość aktualnej.

3.11.2. Architektura

Do tej pory nie zostało to jeszcze powiedziane w sposób jednoznaczny – każdej kontrolce, którą widzi użytkownik przyporządkowany jest po stronie serwera jeden komponent, który przechowuje stan kontrolki, a także potrafi obsługiwać zdarzenia przez nią wygenerowane.



Rysunek 3.4. Implementacja połączenia kontrolki i komponentu

Na powyższym schemacie przedstawiony jest sposób realizacji opisanego wcześniej podejścia. Z każdą kontrolką widzianą przez użytkownika związane są cztery klasy:

- Kontrolka – dziedzicząca po klasie `Widget`, definiująca co użytkownik widzi
- Łącznik – który odpowiedzialny jest za przesyłanie informacji o zaistniałym zdarzeniu do serwera i synchronizacji stanu kontrolki
- Komponent – klasa realizująca serwerową logikę kontrolki – realizuje reakcje na wygenerowanie zdarzenia, aktualizuje stan
- Stan – klasa zawierająca informacje o aktualnym stanie kontrolki.

Gdy zostanie wygenerowane zdarzenie przez kontrolkę po stronie klienta, łącznik przekazuje informacje o tym na stronę serwera do komponentu, który reaguje na tę zmianę. Możliwa jest zmiana stanu zarówno własnej kontrolki, jak i innych kontrolek. Każdorazowa zmiana którejś z właściwości w obiekcie stanu powoduje asynchroniczną aktualizację po stronie klienta.

3.11.3. Aplikacja po stronie klienta

Jedną z zalet Vaadina jest fakt, o którym była mowa wcześniej, że całość logiki znajduje się po stronie serwera. Twórcy szkieletu aplikacji Vaadin zdają sobie jednak sprawę z tego, że część aplikacji potrzebuje natychmiastowych odpowiedzi (czyli nie ma czasu na czasochłonną komunikację z serwerem przy każdym np. wciśnięciu klawisza) dlatego część zdarzeń powinna być obsługiwana po stronie klienta.

Bardzo pomocny w tym celu okazuje się fakt, że Vaadin został stworzony jako obudowa dla GWT (Google Web Toolkit). Okazuje się, że istnieje możliwość skorzystania z kontrolek Google Web Toolkit w ramach aplikacji (istnieje także grupa kontrolek Vaadina dedykowanych specjalnie do tego rodzaju problemu).

3.11.4. Nowe kontrolki

Wielką zaletą szkieletu aplikacji, który wybrałem jest spora liczba kontrolek będących w standardowym zestawie. Jeżeli jednak okazałoby się, że zbiór ten nie zawiera takiej, która jest potrzebna do zrealizowania funkcjonalności, warto udać się pod adres <https://vaadin.com/download>, pod którym do tej pory zostało opublikowanych prawie 400 nowych.

Oczywiście może się okazać, że i w tej liście nie znajduje się odpowiednia kontrolka – w tym przypadku nie pozostaje już nic innego jak stworzyć nową. W ramach szkieletu aplikacji można tworzyć 4 rodzaje nowych kontrolek:

- Kontrolka złożona – składająca się z kilku istniejących kontrolek
- Kontrolka rozszerzająca istniejącą (Vaadin) – zmieniająca właściwości / zachowanie tej kontrolki – np. zmiana tła pola tekstowego na niebieski
- Kontrolka rozszerzająca istniejącą (GWT) – zaimplementowanie własnej kontrolki na podstawie istniejącej kontrolki z GWT
- Całkiem nowa kontrolka, oparta o JavaScript/HTML – zaimplementowanie nowej kontrolki od zera również jest możliwe (choć skomplikowane).

3.11.5. Style

Autorzy stron internetowych z wykorzystaniem HTML mogą być przyzwyczajeni do możliwości wyprowadzania elementów dotyczących wyglądu aplikacji poza zawartość kodu strony (CSS). Twórcy Vaadina prawdopodobnie dostrzegli dużą zaletę w tego typu podejściu, ponieważ w tym szkielecie aplikacji również jest to możliwe. Mechanizm ten jest wierną kopią kaskadowych arkuszy stylów – można określać styl na każdym poziomie aplikacji, jednocześnie przykrywając go na poziomie niżej, np. kod powoduje, że wszystkie tła w aplikacji będą koloru żółtego:

```
.v-app {  
    background: yellow;  
}
```

Natomiast dopisując poniższy kod uzyskiwany jest efekt, polegający na tym, że wszystko oprócz przycisków będzie posiadać żółte tło, które będą mieć tło kolorowane na niebiesko:

```
.v-app .mybutton {  
    background: blue;  
}
```

3.11.6. Powiązanie danych

Główną rolą aplikacji internetowych jest przetwarzanie informacji. Ważnym elementem jest wyświetlanie danych wprowadzonych do bazy do tej pory, potrzebne jest także umożliwienie edycji / dopisania / usunięcia elementu. Tak samo jak każdy szanujący się szkielet aplikacji, Vaadin umożliwia uproszczenie tego procesu za pomocą powiązania danych (ang. Data Binding). W Vaadinie istnieją trzy wymiary, w których przechowywane są dane:

- Właściwość (ang. Property) – każda kontrolka posiada przypisaną do niej właściwość
- Pozycja (ang. Item) – każdy formularz lub wiersz w tabeli posiada odpowiadającą mu pozycję
- Pojemnik (ang. Container) – każdej tabeli odpowiada pojemnik przechowujący jej dane

Co daje powiązanie danych? Umożliwia to w prosty sposób edycje obiektów w bazie danych. Na przykład, wprowadzenie danych do formularza jest jednoznaczne z ustawieniem właściwości w obiekcie, dzięki czemu w wyniku np. kliknięcia przycisku dodaj, szkielet aplikacji dostarcza wypełniony już obiekt, gotowy do wprowadzenia do bazy danych.

Innym przykładem korzyści płynących z powiązania danych jest usunięcie wiersza z tabeli. Dzięki temu mechanizmowi minimalizowana jest praca polegająca na oprogramowywaniu przycisku usuń.

Ważnym elementem w kontekście powiązania danych jest proces walidacji danych (szczególnie w formularzu). Vaadin umożliwia (jak większość szkieletów aplikacji) sprawdzenie poprawności wprowadzonych informacji i w razie potrzeby wyświetla potrzebną informację zwrotną o błędzie użytkownikowi. Walidacja odbywa się zgodnie ze standardem Java Bean Validation (JSR-303).

3.12. Podsumowanie

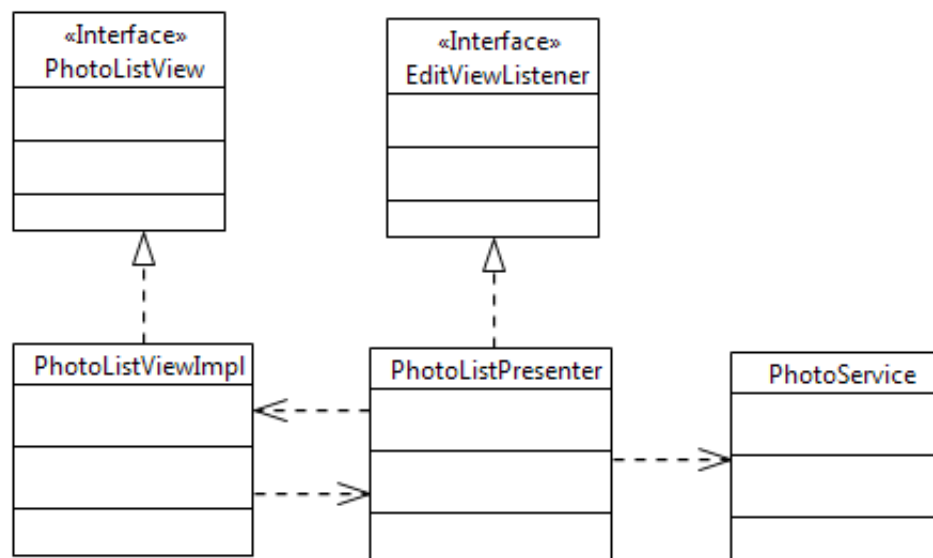
Wybrane przez autora technologie zostały dopasowane do potrzeb tworzonego systemu, ponieważ w pracy nad każdym oprogramowaniem do wyboru narzędzi wspierających budowę powinno się podchodzić indywidualnie.

Należy także pamiętać, że technologie użyte przez autora do stworzenia systemu będącego tematem pracy, nie są uniwersalne - a wręcz przeciwnie - były aktualne i dosyć powrzechne w chwili pisania tej pracy. Bardzo prawdopodobne jest jednak, że za kilka lat zostaną uznane w środowisku informatycznym za przestarzałe, a w ich miejsce powstaną nowe, lepsze i/lub wydajniejsze.

4. Architektura rozwiązania

4.1. Model Widok Prezentor

Aplikacje internetowe z reguły najlepiej wpisują się we wzorzec MVP (ang. Model-View-Presenter, Model-Widok-Prezentor), z którego autor postanowił skorzystać jako podstawie architektury systemu. Wzorzec ten polega na przekazaniu kontroli nad danymi prezentowanymi przez system do obiektu zwanego Prezentor. Mimo, że zdarzenia bezpośrednio pojawiają się w warstwie widoku, to natychmiast są one przekazywane do Prezentora, który w razie potrzeby odwołuje się do warstwy modelu i ewentualnie zmienia wygląd widoku prezentowanego użytkownikowi.



Rysunek 4.1. Przykład implementacji wzorca MVP

Powyższy rysunek przedstawia implementację wzorca MVP w systemie wspierającym prowadzenie ewidencji badań archeologicznych. Na przykładzie widoku oraz prezentora listy zewidencjonowanych zdjęć pokazana jest współpraca komponentów. Jest to odmiana wzorca nosząca nazwę "Passive view- w tej odmianie całą pracę wykonuje prezentor.

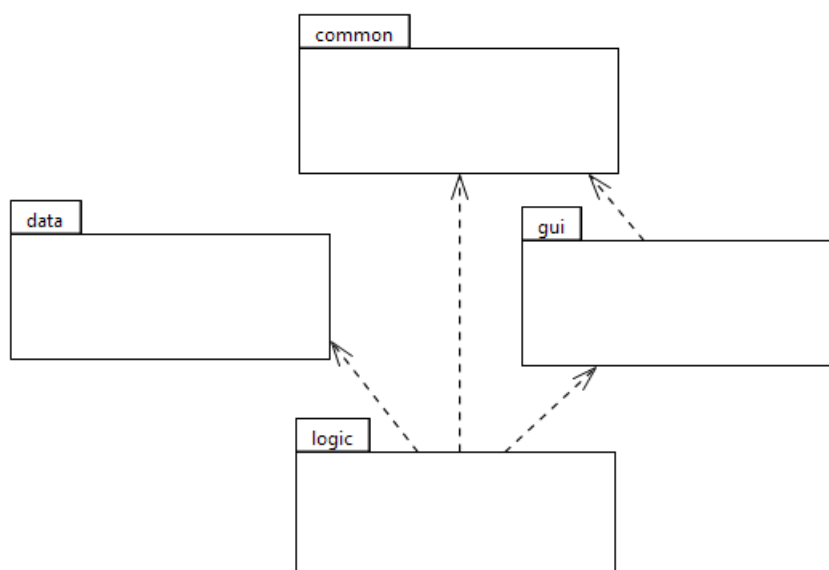
Aby maksymalnie rozluźnić zależności między obiektami zastosowane są interfejsy przez które komunikują się Widok z Prezenterem. Prezenter wystawia interfejs, który musi być implementowany przez widok, natomiast widok umożliwia jedynie rejestrację prezentera jako nasłuchiwanca zdarzeń.

W chwili wystąpienia zdarzenia, np. chęci dodania nowego zdjęcia do listy, widok wywołuje odpowiadającą temu zdarzeniu metodę na wszystkich obiektach zarejestrowanych jako nasłuchujące na obiekcie widoku. Prezenter otrzymuje w ten sposób notyfikację, która może powodować np. komunikację z bazą danych przez obiekt PhotoService (np. dodanie do bazy danych nowego zdjęcia). Na zakończenie przetwarzania prezenter musi powiedzieć widokowi, że zmieniła się lista zdjęć. Aby utrzymać maksymalne rozluźnienie zależności, widok jest powiadamiany przez interfejs który implementuje.

4.2. Podział na pakiety

Konsekwencją zastosowania wzorca MVP jest rozluźnienie zależności między obiektami, dzięki czemu w prosty sposób da się wyodrębnić luźno powiązane ze sobą moduły. W systemie zostały wyodrębnione 4 podstawowe pakiety:

- logic - pakiet zawierający definicje prezenterów oraz klas abstrakcyjnych z nimi powiązanych
- gui - pakiet zawierający definicje widoków oraz klas abstrakcyjnych z nimi związanych
- data - pakiet przechowujący kod procedur wykonujących operacje utrwalania danych
- common - pakiet zawierający klasy wspólne dla klas z pakietów logic i gui



Rysunek 4.2. Podział klas na pakiety

4.3. Nawigacja między widokami

Szkielet budowy aplikacji Vaadin sam w sobie posiada mechanizm przechodzenia między widokami, jest on jednak dosyć niewygodny w użyciu - istnieje konieczność rejestrowania widoków i ich adresów w momencie inicjalizacji obiektu UI. Na szczęście, dzięki wtyczce SpringVaadinIntegration możliwe jest przechodzenie między widokami jedynie po ich identyfikatorze w kontenerze Spring.

We wspomnianej przeze mnie wtyczce istnieje klasa implementująca powyższe zachowanie - `DiscoveryNavigator`. Wciąż jednak nie jest ona wystarczająca, jednak po zdefiniowaniu kilku metod okazuje się, że można dostosować ją do swoich potrzeb.

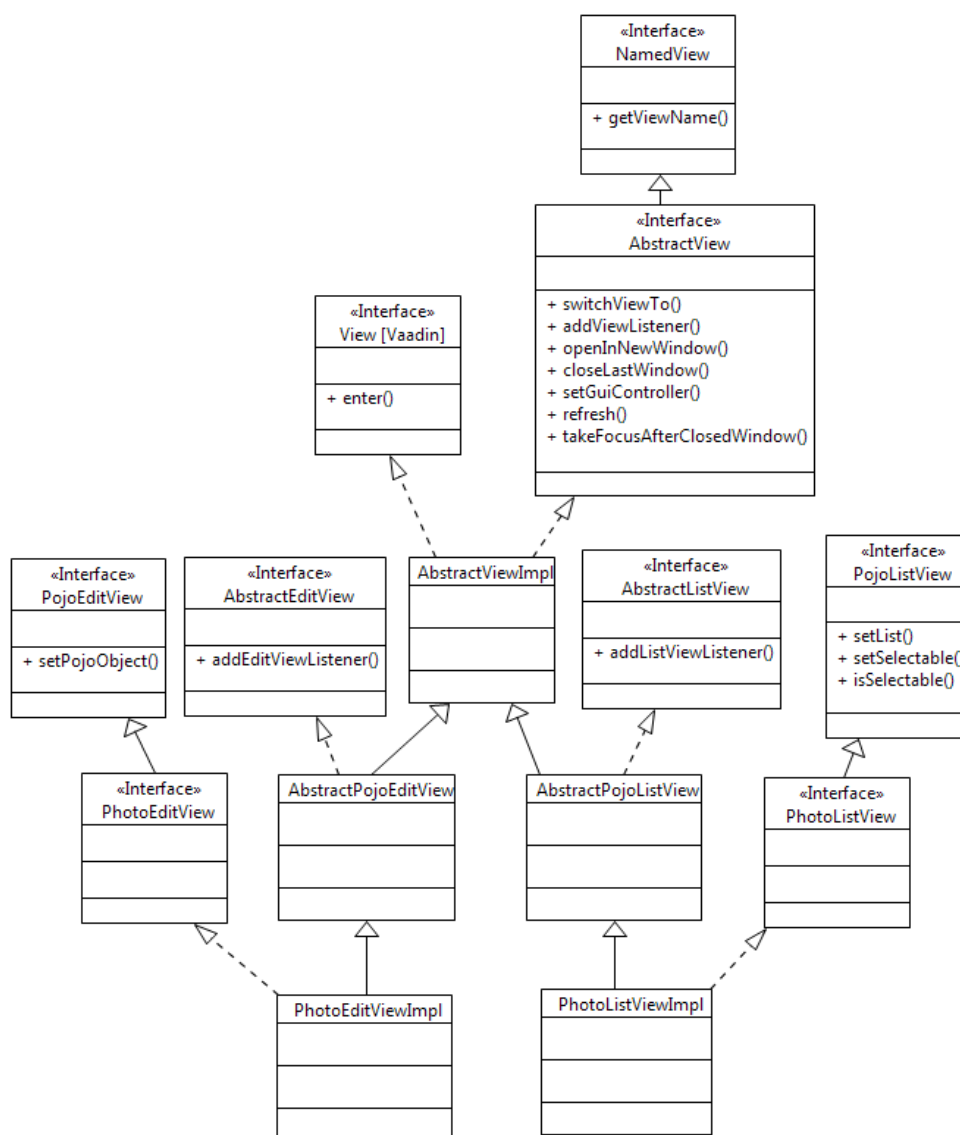
Rozwiązanie wykorzystane w systemie będącym produktem niniejszej pracy wymaga zarejestrowania widoku w kontrolerze sesji, który na podstawie nazwy widoku nadaje powiązanie z nim odpowiedniemu prezwenterowi. Prezwenter z kolei, rejestruje się jako słuchacz zdarzeń widoku, którym steruje.

4.4. Maksymalna abstrakcyjność

System wspierający prowadzenie ewidencji dokumentacji archeologicznej musi umożliwiać wprowadzanie wielu rodzajów danych. Wiąże się to z dużą ilością klas z kategorii widoków i prezwenterów. Szczególnie w przypadku słowników, istnieje wielka szansa na konieczność wielokrotnego powtarzania tych samych fragmentów kodu, stąd ambicją autora było stworzenie szkieletu aplikacji skracającego czas tworzenia komponentów odpowiedzialnych za wprowadzanie danych do systemu do minimum.

Szkielet aplikacji, o którym była mowa w poprzednim akapicie wymaga dwóch osobnych struktur hierarchii obiektów, jednej dla widoków i jednej dla prezwenterów. Konieczne jest także, aby te struktury wzajemnie były od siebie zależne, zgodnie ze wzorcem MVP opisanym w pierwszej części tego rozdziału.

Poniżej zostało zaprezentowane drzewo dziedziczenia widoków, powstałe w wyniku projektu aplikacji, który następnie w trakcie implementacji został delikatnie zmodyfikowany.



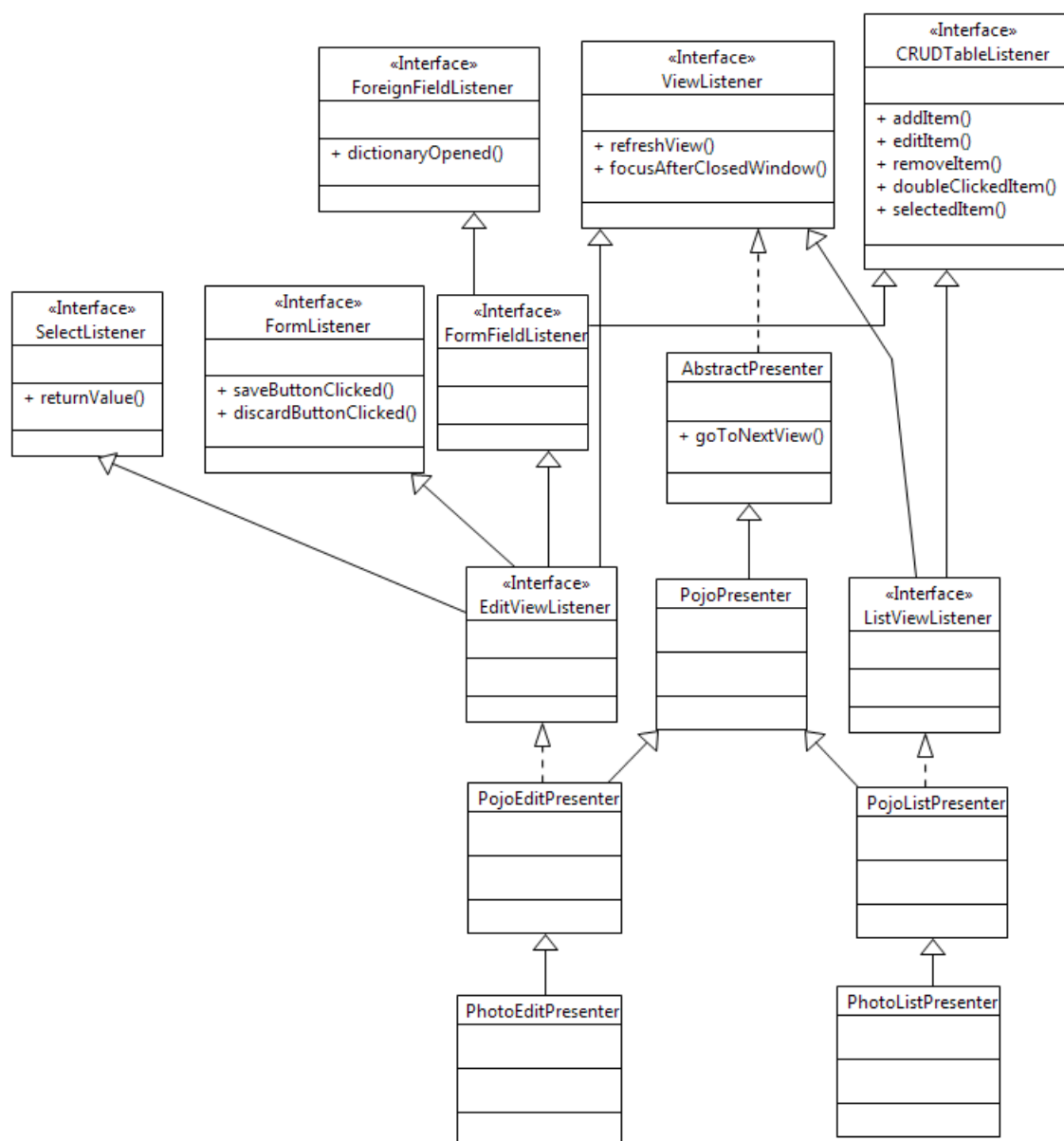
Rysunek 4.3. Hierarchia widoków

Na samej górze hierarchii znajduje się interfejs wymuszony przez technologię Vaadin (View), która jest konieczna do wyświetlenia różnych widoków w obrębie jednego obiektu UI. Drugim korzeniem hierarchii jest interfejs NamedView, który pozwala nadawać nazwy widokom, co z kolei utożsamiane jest z nazwą odpowiadającego komponentu Springa (ang. bean), a także przydzielanego fragmentu URI, doczepianego do adresu i pozwalającego przeglądarce zapamiętywać historie nawigacji po stronie.

Całe drzewo dziedziczenia wyraźnie podzielone jest na dwie części - obiekty widoków listy i widoków okna edycji wiersza. Jest to spowodowane oczywiście różnymi zadaniami stawianymi obiektom implementującym powyższe funkcje.

Rdzeń implementacji znajduje się w klasach `AbstractViewImpl` oraz `AbstractPojoEditView` i `AbstractPojoListView`. `AbstractViewImpl` jest klasą, z której bezpośrednio dziedziczą wszystkie standardowe widoki prezentujące zwykłą treść, np. lista słowników czy strona startowa. `AbstractPojoEditView` jest z kolei klasą, po której bezpośrednio dziedziczą wszystkie implementacje widoków okien edycji użyte w systemie. Implementuje ona wygląd okna edycji (za pomocą klasy `DefaultForm`, opisanej w rozdziale 7., która dynamicznie generuje wygląd formularza na podstawie adnotacji zawartych w klasie obiektu edytowanego) oraz całą obsługę przekazywania zdarzeń, które zostały wygenerowane przez użytkownika, czyli wywołuje metody na obiektach nasłuchiwalcy (czyli prezenterów).

Warto także dodać, że hierarchia jest skonstruowana w ten sposób, aby uniezależnić maksymalnie resztę klas wchodzących w skład pakietu obsługującego cały interfejs graficzny użytkownika, od technologii prezentacji, którą w tym przypadku jest Vaadin.



Rysunek 4.4. Hierarchia prezenterów

Powyżej została przedstawiona hierarchia prezenterów w szkieletcie aplikacji stworzonym przez autora w trakcie tworzenia systemu ewidencji zabytków archeologicznych. Na samym początku warto zaznaczyć, że hierarchia ta jest tak rozbudowana, ze względu na dopasowanie do komponentów stworzonych w trakcie pracy nad systemem - ForeignField oraz CRUDTable, które są opisane w rozdziale 7.

Dodatkowo, wszystkie widoki użyte w systemie muszą implementować interfejs ViewListener, który pozwala reagować na ogólne zachowania widoków, a mówiąc ściślej, pozwala wykonywać akcje przy okazji przejścia z jednego widoku do drugiego - np. odświeżenie zawartości.

Każdy prezydent w systemie powinien także dziedziczyć po klasie `AbstractPresenter`, implementuje elementy związane z nawigacją (pamięta, który prezydent jest jego rodzicem- czyli jest prezydentem widoku, który wywołał ten widok oraz implementuje przejście do następnego widoku).

Tak jak w przypadku hierarchii widoków, tak w przypadku prezydentów konieczne jest rozdzielenie na dwie główne gałęzie - prezydentów okna edycji oraz prezydentów listy obiektów. Gałęzie te mają swoją implementację kolejno w klasach: `PojoEditPresenter` i `PojoListPresenter`. Głównym zadaniem wymienionych klas jest implementacja reakcji na działania użytkownika, które zostały przekazane dalej przez widok poprzez interfejs nasłuchiwarza (zaimplementowany w tych właśnie klasach). Na samym dole hierarchii znajdują się klasy konkretne, odpowiadające prezydentom obiektów dziedziny problemu. Poniżej przykład implementacji prezydenta okna edycji:

```
@Component
@Scope("session")
public class FigureSubjectEditPresenter
    extends PojoEditPresenter<FigureSubject>
{
    public interface FigureSubjectEditView
        extends PojoEditView<FigureSubject>
    {
    }

    @Autowired
    public FigureSubjectEditPresenter(
        FigureSubjectEditView pojoEditView,
        AbstractServiceInterface<FigureSubject> pojoServ)
    {
        setView(pojoEditView);
        setPojoService(pojoServ);
    }
}
```

Jak widać, powyższy kod implementuje się bardzo szybko. Dodatkową prostotę autor osiągnął dzięki zastosowaniu kontenera IoC Springa, który wstrzykuje zależności bezpośrednio do konstruktora, za pomocą adnotacji `@Autowired`. Warto zwrócić uwagę, że każda klasa konkretna definiuje swój odrębny interfejs widoku, który jednak standardowo posiada takie same metody jak wszystkie inne w tej grupie widoków (okno edycji).

Klasa przedstawiona na listingu jest prezydentem okna edycji obiektu, który nie posiada ani kolekcji obiektów podrzędnych (związek Jeden-Do-Wielu oraz Wiele-Do-Wielu) ani obiektów pochodzących ze słownika. Jeżeli by tak było, konieczne byłoby zaimplementowanie dodatkowych metod:

- `getDataProvider()` - metoda zwracająca obiekt implementujący interfejs `DataProvider`, używany w kontekście słowników do wypełniania wartości
- `getDictionaryPresenter()` - metoda zwracająca, na podstawie argumentu, prezydent używany przez widok powiązany z danym słownikiem
- `getActiveFieldPresenter()` - metoda zwracająca, na podstawie argumentu, prezydent używany przez widok powiązany z jednym z obiektów kolekcji (powinien to być Prezydent Listy w przypadku związku Wiele-Do-Wielu oraz prezydent okna edycji obiektu podrzędnego w przypadku związku Jeden-Do-Wielu, np. specyfikacja rysunku).
- `fillValueInOneToManyRel()` - metoda, która wzbogaca obiekt zwrócony z potomnego prezydenta, o referencje do obiektu prezentowanego

Kod prezydenta listy obiektów jest niewiele bardziej skomplikowany:

```
@Component
@Scope("session")
public class FigureSubjectListPresenter
    extends PojoListPresenter<FigureSubject>
{
    public interface FigureSubjectListView
        extends PojoListView<FigureSubject>
    {
    }

    @Autowired
    public FigureSubjectListPresenter(
        AbstractServiceInterface<FigureSubject> pojoServ,
        FigureSubjectListView pojoListView,
        FigureSubjectEditPresenter photoSubjectEditPres)
    {
        super(FigureSubject.class);
        setPojoService(pojoServ);
        setPojoListView(pojoListView);
        setPojoEditPresenter(photoSubjectEditPres);
    }
}
```

```
@Override
protected Criterion getCriterion()
{
    return null;
}

@Override
protected FigureSubject getEmptyObject()
{
    return new FigureSubject();
}
}
```

Ten prezwenter także zawiera definicje interfejsu widoku, z którym się komunikuje oraz także w tym przypadku użyty został kontener Springa wstrzykujący zależności bezpośrednio do konstruktora.

Dodatkowo pojawiły się dwie proste metody:

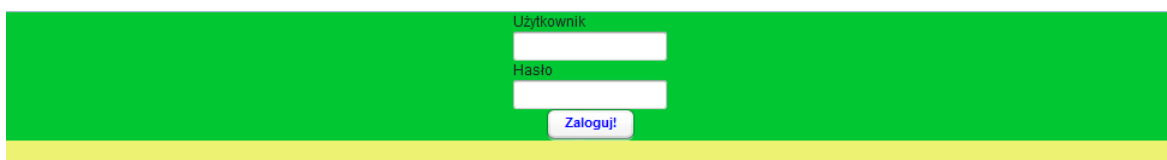
- `getCriterion()` - zwracająca kryterium służące do wyboru listy elementów prezentowanych
- `getEmptyObject()` - tworząca nowy obiekt, zgodnie ze wzorcem fabryka

Jak widać na przykładach powyżej, cała implementacja reakcji prezwenterów na zdarzenia przychodzące z interfejsu graficznego użytkownika jest zaimplementowana w klasach abstrakcyjnych, natomiast w klasach konkretnych zostało jedynie ustawienie wartości konkretnych pól, a także zaimplementowanie odpowiednich abstrakcyjnych metod, które są użyte w klasach abstrakcyjnych.

5. Prezentacja aplikacji

W niniejszym rozdziale zostanie zaprezentowany od strony użytkowej system, który powstał w ramach niniejszej pracy.

W momencie rozpoczęcia nowej sesji, użytkownikowi pokazuje się ekran logowania:



Rysunek 5.1. Ekran logowania

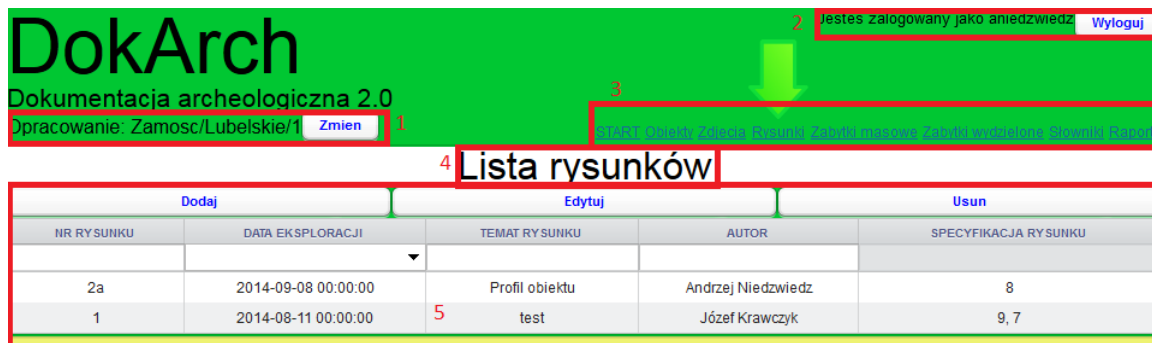
Następnie, po zalogowaniu, użytkownikowi jest wyświetlane okno wyboru opracowania (na poniższym rysunku). Ekran ten jest widoczny na samym początku sesji, ale może też być wyświetlony w dowolnym momencie użytkowania systemu, poprzez kliknięcie przycisku "Zmień" w sekcji "Opracowanie", w górnej belce strony. Z poziomu wyświetlonego ekranu użytkownik może dodawać, edytować bądź usuwać (tylko te, dla których nie zostały dodane żadne dane) opracowania. W ten sposób jest spełniony przypadek użycia UC1. - Zarządzaj listą opracowań.



MIEJSCOWOSC	GMINA	WOJEWODZTWO	NR OBSZARU AZP	NUMER OBSZ
Zamosc	m. Zamosc	Lubelskie	22-400	

Rysunek 5.2. Ekran logowania

Standardowy ekran aplikacji wyświetlający listę elementów wygląda jak na poniższym rysunku.

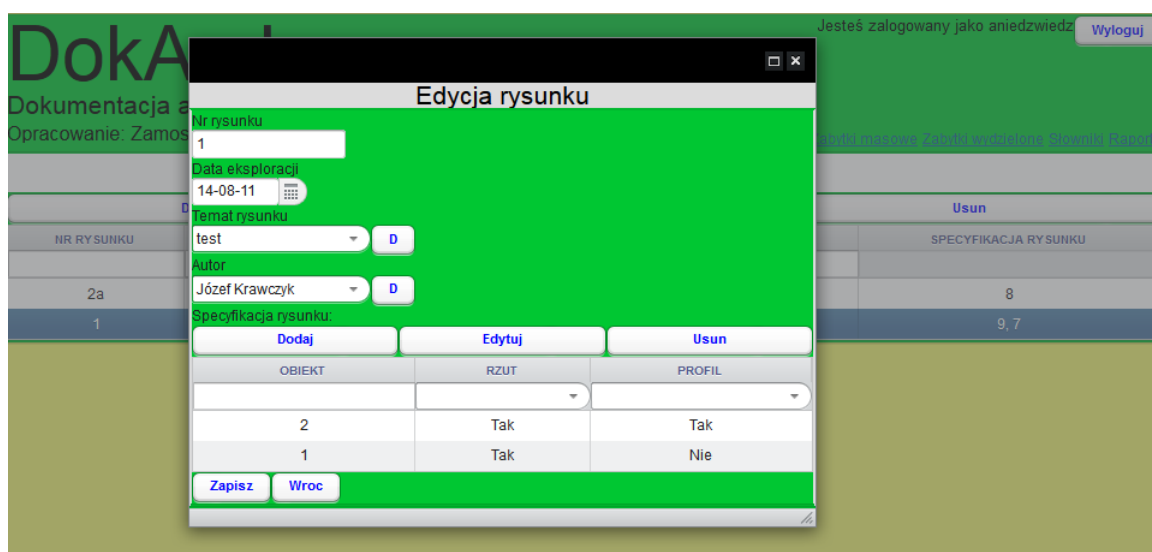


Rysunek 5.3. Przykładowa podstrona systemu

Na załączonym obrazku widać, że strona aplikacji składa się z:

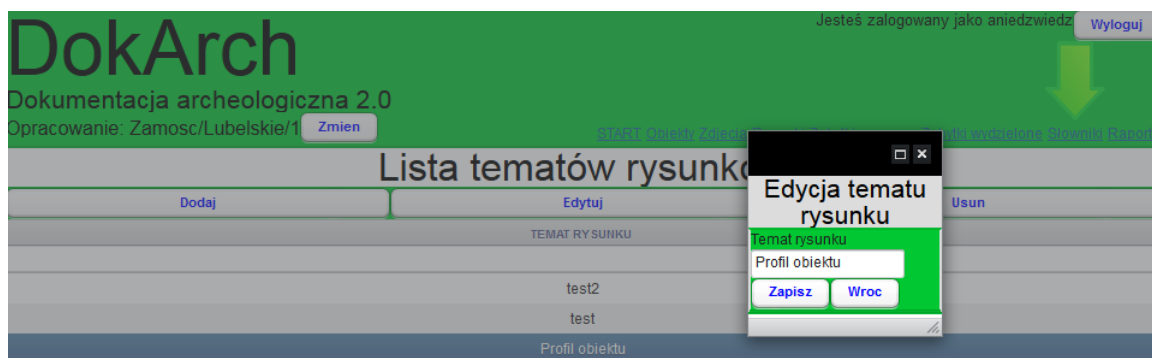
- górnej belki zawierającej w sobie informacje o bieżącym opracowaniu (1 na rysunku), informacje o zalogowanym użytkowniku (2) oraz menu (3)
- belki tytułowej (4)
- głównej treści strony (5)

Powyższy rysunek demonstruje także przykładowy widok listujący wprowadzone dane (w tym przypadku rysunki). Dzięki addonowi FilterTable, możliwe jest filtrowanie widocznych elementów po ich zawartości w konkretnych polach. Standardowa funkcjonalność Vaadina pozwala także sortować wiersze wg wartości w kolumnie - w tym celu wystarczy kliknąć nagłówek kolumny.



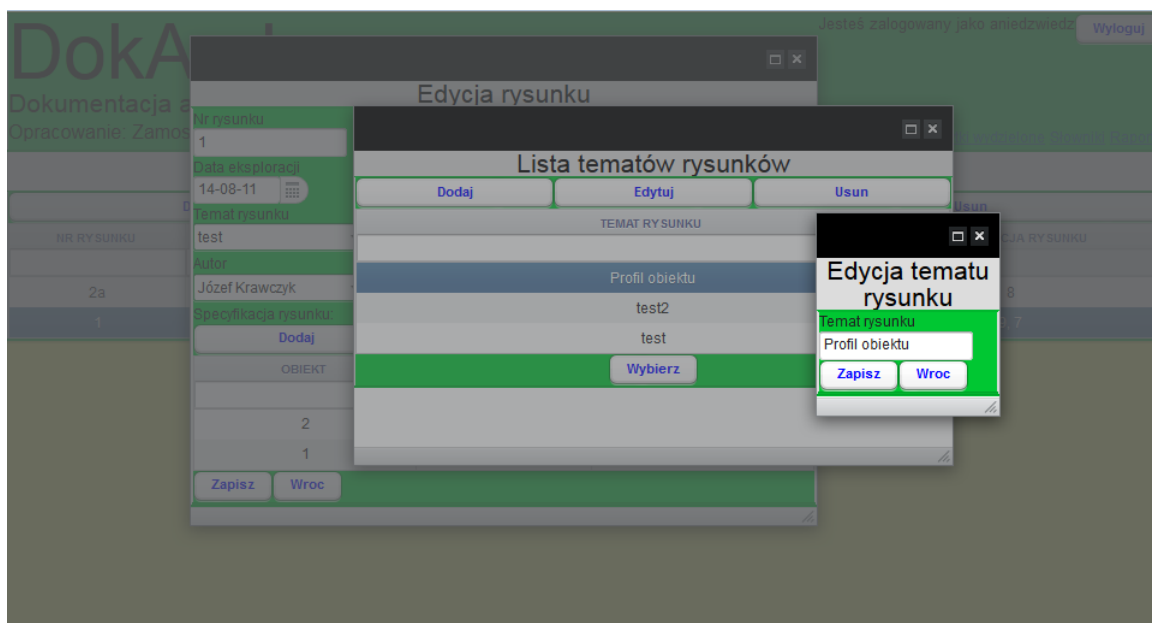
Rysunek 5.4. Przykładowa edycja obiektu dokumentacyjnego

Rysunek znajdujący się nad tym tekstem demonstruje ekran edycji wiersza (w tym przypadku wiersz odzwierciedla ewidencjonowany rysunek). Ten sam ekran (tylko z niewypełnionymi wartościami) jest wyświetlany użytkownikowi po otrzymaniu informacji o wciśnięciu przez użytkownika przycisku dodaj. Lista wysunków wraz z oknem edycji rysunków wypełniają przypadek użycia "UC5. Zarządzaj rysunkami".



Rysunek 5.5. Przykładowa edycja słownika

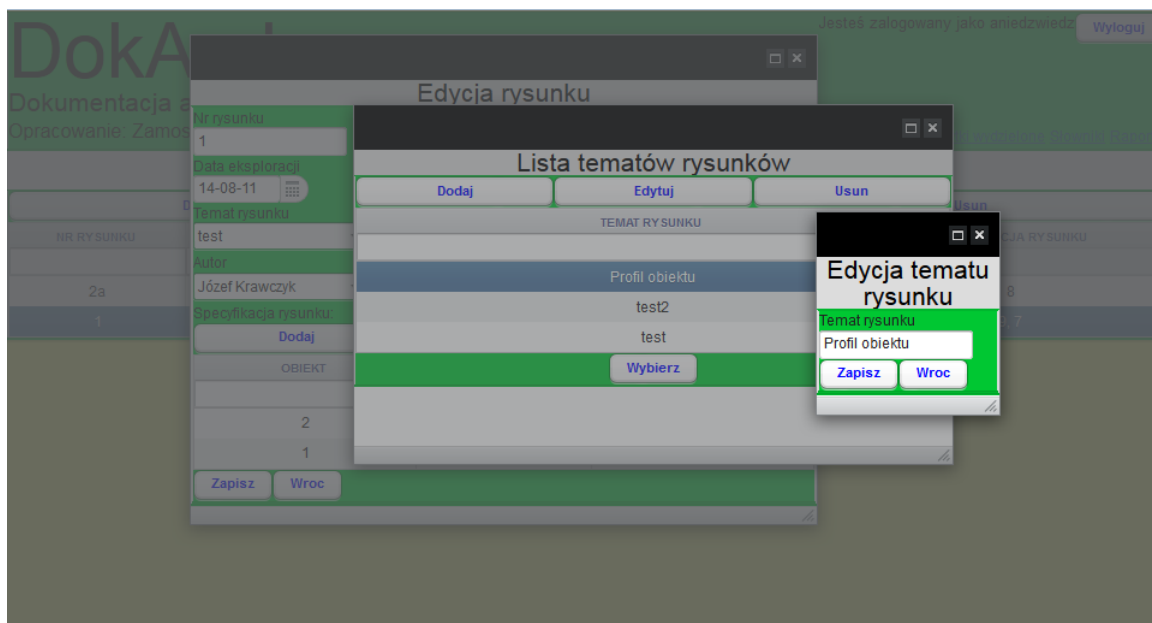
Rysunek powyżej przedstawia standardowy sposób edycji słownika (wybranie w menu w górnej belce Słowniką następnie wybranie słownika "Tematy rysunków"). Jest to jeden ze sposobów modyfikacji wartości słownika.



Rysunek 5.6. Przykładowa edycja słownika w trakcie edycji obiektu go używającego

Powyżej został przedstawiony ekran modyfikacji słownika w "locie", czyli w trakcie edycji wiersza, który zawiera wartość ze słownika. Możliwe jest dynamiczne wprowadzanie wartości bez konieczności zamykania okna edycji.

Wyświetlenie listy wartości oraz ich modyfikacja może nastąpić na dwa sposoby. Pierwszy sposób to wybranie przycisku D przy polu słownikowym. Wtedy otwiera się okno z tabelką widoczne poniżej:



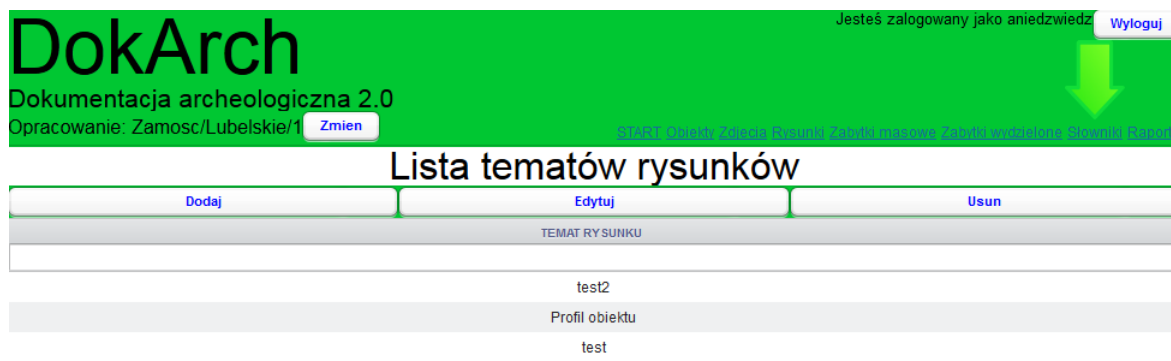
Rysunek 5.7. Lista tematów rysunków zmieniana w "locie"

Drugim sposobem jest kliknięcie w link w menu strony w nazwie Słowniki". Link ten przeniesie użytkownika do strony widocznej poniżej:



Rysunek 5.8. Lista słowników

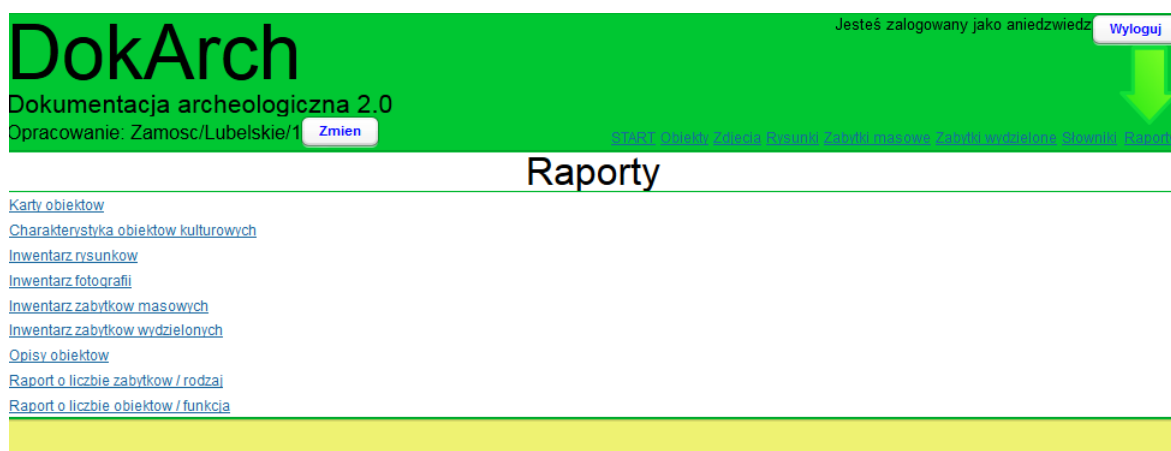
Następnie użytkownik wybiera interesujący go słownik i klika w link prowadzący do niego. Przykładowy słownik po wciśnięciu linka:



Rysunek 5.9. Lista tematów rysunków

W ten sposób pokazano, że system ma zaimplementowany przypadek użycia UC9. Zarządzaj tematami rysunków.

System potrafi także generować raporty. Aby tego dokonać wystarczy kliknąć link w menu "Raporty", i wybrać interesujący raport z listy:



Rysunek 5.10. Ekran raportów

Przykładowy raport wygenerowany przez system:

Inwentarz fotografii					
Numer zdjęcia	Data eksploracji	Autor	Temat zdjęcia	Ary	Nr obiektu
1	08.09.14	Andrzej Niedzwiedz	temat	2a, 1	1
2	08.09.14	Józef Krawczyk	Zdjęcie profilu	1	2

Rysunek 5.11. Przykładowy raport

6. Dodatkowe produkty pracy

W trakcie tworzenia niniejszej pracy powstało także kilka produktów ubocznych, które zdaniem autora mają szansę zostać popularnymi wtyczkami do szkieletu aplikacji Vaadin oraz o których warto wspomnieć. Każda z tych wtyczek używa adnotacji zaszytych bezpośrednio w klasie obiektu, na których operują.

6.1. Przykładowa klasa modelu danych

Implementacja systemu opierała się przede wszystkim przy użyciu komponentów wymienionych w tym rozdziale. Każdy z nich korzysta ze stworzonych od nowa adnotacji, którymi opatrzone są definicje klas:

- `ColumnHeader` - definiuje nagłówek oraz kolejność przedstawiania właściwości obiektu w komponencie `CRUDTable`
- `EditField` - opisuje sposób wyświetlania pola edycji odpowiedzialnego za wypełnienie pola z którym jest związane (kolejność wyświetlenia oraz opis, a także dodatkowa informacja używana w implementacji - obiekt klasy będący typem generycznym zbioru, będącego opisywanym polem.
- `ForeignKeyLabel` - umożliwia konfigurację sposobu wyświetlania klasy obiektu w komórce tabeli (`CRUDTable`) jak i polu słownikowym (`ForeignKey`).

Poniżej została przedstawiona uproszczona standardowa klasa zawierająca konfigurację z użyciem adnotacji używana przez wymienione w tym rozdziale komponenty.

```
@Entity
@Table(name = "figure_specifications")
@ForeignKeyLabel(pattern = "id")
public class FigureSpecification
{
    @Id
    @GeneratedValue
    @Column(name = "figure_specification_id")
    private Long id;
```

```
@Column(name = "figure_specification_profile")
@ColumnHeader(value = "Profil", order = 3)
@EditField(label = "Profil", order = 3)
@NotNull
private Boolean profile = false;

@Column(name = "figure_specification_projection")
@ColumnHeader(value = "Rzut", order = 2)
@EditField(label = "Rzut", order = 2)
@NotNull
private Boolean projection = false;

@JoinColumn(name = "figure_id")
@NotNull(message = "Rysunek nie może być pusty")
@ManyToOne(optional = false)
private Figure figure;

@JoinColumn(name = "object_id")
@ColumnHeader(value = "Obiekt", order = 1)
@EditField(label = "Obiekt", order = 1)
@NotNull(message = "Obiekt nie może być pusty")
@ManyToOne(optional = false)
private ArchObject archObject;

...
(gettery i settery)
}
```

Jak widać, w powyższej klasie zostały użyte trzy typy adnotacji:

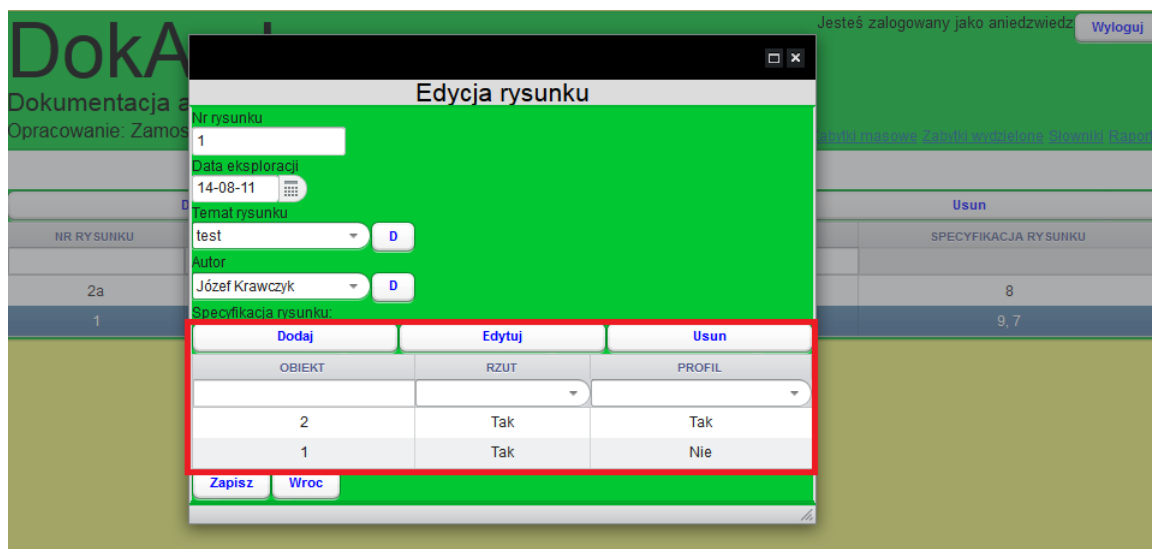
- `javax.persistence` - odpowiedzialne za mapowanie relacyjno-obiektowe wewnątrz systemu
- `javax.validation` - odpowiedzialne za walidację wprowadzonych wartości
- dedykowane dla stworzonych komponentów

Jak dalej zostanie opisane w tym rozdziale, każdy z wyżej wymienionych rodzajów jest wykorzystywany przez niżej opisane komponenty.

Warto zwrócić także uwagę na adnotację `@FormFieldLabel`, zawierającą definicje sposobu wyświetlania obiektu POJO w poniższych komponentach. Składnia wyrażenia zawartego w adnotacji pozwala wyświetlać zarówno pojedyncze pole (wartość `nazwa_pola` lub `"$nazwa_pola$"`) lub kombinacje jednego bądź większej ilości pól ze statycznym ciągiem znaków (np. `"Pan: $imie$ $nazwisko$"`, nazwy zmiennych zawierają się między znakami `"$"`). Jedynym warunkiem poprawnego wyświetlenia napisu odpowiadającego wartościom obiektu jest konieczność implementowania przez użyte pola metody `toString()`.

6.2. CRUDTable

CRUDTable to komponent, który wyświetla listę obiektów wykorzystując wtyczkę FilterTable opartą na komponencie CustomTable wzbogaconą o standardowe przyciski generujące akcje związane z przetwarzaniem danych tabelarycznych, tzn. dodawania, edytowania i usuwania wierszy. Dodatkowo, zaimplementowana jest funkcjonalność z użyciem wtyczki ContextMenu, która zawiera domyślnie także akcje związane z przetwarzaniem zbioru danych.



Rysunek 6.1. Komponent CRUDTable (zaznaczony na czerwono)

Nic nie stoi na przeszkodzie, aby rozszerzać funkcjonalność komponentu CRUDTable. Istnieją metody pozwalające modyfikować listę przycisków znajdujących się nad tabelą, można także dodawać pozycje widoczne w menu kontekstowym.

Dzięki wykorzystaniu komponentu `FilterTable` istnieje możliwość filtrowania wierszy ze względu na wartości przechowywane w konkretnych polach. Istnieje możliwość zawężenia wyświetlanych rekordów zarówno po wartościach liczbowych jak i wartościach tekstowych, a nawet takich, które odwołują się do innych obiektów dziedziny problemu (czyli będących kluczem obcym).

Najważniejszą jednak funkcjonalnością jest sterowanie wyglądem treści przedstawianej przez komponent za pomocą adnotacji. Dopisując adnotację `@ColumnHeader` do pola obiektu POJO definiuje się tak naprawdę wygląd tabelki wyświetlającej dany obiekt. Atrybut `value` definiuje nam wartość widoczną w nagłówku kolumny natomiast parametr `order` decyduje o kolejności wyświetlanych kolumn (są one wyświetlane w kolejności rosnącej). Uwaga! To programista jest odpowiedzialny za poprawne wypełnienie wartości `order`, w przypadku zdublowania wartości wyświetlona zostanie tylko jedna kolumna.

Warto także dodać, że w kolumnie w sposób sensowny są także wyświetlane inne obiekty niekoniecznie będące typami prostymi w Javie. Aby wyświetlić poprawnie obiekt, wystarczy do deklaracji klasy dodać adnotację `ForeignFieldLabel`, która opisuje w jaki sposób wyświetlić wartość w danej komórce tabeli.

Obiekt, który zawiera w sobie komponent `CRUDTable` powinien implementować interfejs `CRUDTableListener`, ponieważ sam komponent nie podejmuje żadnych akcji w wyniku zdarzeń wygenerowanych przez użytkownika, a jedynie przekazuje je dalej do nasłuchiwanego. Obiekt ten jest także odpowiedzialny za wypełnienie komponentu wartościami.

6.3. ForeignField

Kolejnym komponentem, który wg autora może być przydatny w innych projektach jest `ForeignField`, który implementuje kontrolkę odpowiedzialną za wprowadzenie wartości ze słownika. Kontrolka używa komponentu ze standardowego zestawu komponentów Vaadina - `ComboBox` oraz przycisku.

OBIEKT	RZUT	PROFIL
2	Tak	Tak
1	Tak	Nie

Rysunek 6.2. Komponent ForeignField (zaznaczony na czerwono)

Użycie kontrolki sprowadza się do zdefiniowania wartości wyświetlanej jako wartość obiektu w liście rozwijanej (adnotacja `@ForeignFieldLabel`) oraz ustawienie listy możliwych wartości. Uwaga! Kontrolka sama w sobie nie podejmuje żadnych akcji po naciśnięciu przycisku. Obiekt, który zawiera w sobie komponent powinien implementować interfejs `DictionaryFieldListener` i tam zdefiniować zachowanie będące reakcją na naciśnięcie przycisku "D".

6.4. DefaultForm

DefaultForm jest komponentem, który na podstawie adnotacji `@EditField` zawartych w klasie obiektu tworzy formularz złożony z domyślnych komponentów Vaadina oraz dwóch poprzednio wymienionych: `CRUDTable` oraz `ForeignField`.

Rysunek 6.3. Komponent DefaultForm (zaznaczony na czerwono)

Adnotacja `@EditField` zawiera w sobie informacje na temat kolejności wyświetlania komponentów do edycji kolejnych wartości obiektu (wartość `order`) oraz wartość etykiety (`label`), która powinna być wyświetlona przy danym polu. Uwaga! To programista jest odpowiedzialny za poprawne wypełnienie wartości `order`, w przypadku zdublowania wartości wyświetlona zostanie tylko jedno pole edycji.

Komponent `DefaultForm` deleguje zadanie tworzenia kontrolki do obiektu-fabryki implementującego interfejs `ExtendedFieldGroupFieldFactory` (rozszerzającego interfejs `Vaadin FieldGroupFieldFactory`). W zależności od typu pola obiektu tworzona jest inna kontrolka. Dla standardowych typów takich jak `Integer` lub `String`, kontrolki generowane są przez domyślną fabrykę `Vaadin - FieldGroupFieldFactory`. W przypadku wystąpienia przy polu adnotacji `javax.persistence` odpowiedzialnych za relacje `Wiele-Do-*` lub `Jeden-Do-Wielu` generowana jest odpowiednia kontrolka. Jeżeli nad polem znajduje się adnotacja `@ManyToOne`, generowana jest kontrolka typu `ForeignField`, natomiast dla adnotacji `@OneToMany` oraz `@ManyToMany`, generowany jest komponent `CRUDTable`.

Aby wypełnić słowniki, należy dostarczyć do komponentu `DefaultForm` obiekt, który implementuje interfejs `DataProvider`, ponieważ komponent sam w sobie nie potrafi wypełnić ich wartościami.

DefaultForm nie podejmuje także samodzielnie żadnych akcji, a jedynie oddelegowuje je do zarejestrowanych listenerów. Komponent przechwytyuje także zdarzenia wygenerowane w zawartych komponentach ForeignField i CRUDTable, czyli jest ich nasłuchiwcem, jedyne co robi z tymi zdarzeniami to przekazuje je dalej, dlatego istotne jest, aby obiekt zawierający w sobie komponent DefaultForm implementował interfejs FormFieldListener. Dodatkowo, aby przechwytywać zdarzenia związane z przyciskami *Źapisz* oraz *Wróć* należy zarejestrować nasłuchiwcę implementujący interfejs FormListener.

Ważną cechą komponentu DefaultForm, o której należy wspomnieć na koniec, jest walidacja wprowadzonych wartości na podstawie adnotacji javax.validation. Wszystkie standardowe ograniczenia na wartości pól, zgodne z JSR-303 są sprawdzane przed wysłaniem akcji *żapisz* do obiektu nasłuchującego. W przypadku niepowodzenia walidacji, zdarzenie nie jest przekazywane dalej.

6.5. Szkielet aplikacji przetwarzania danych biznesowych

W trakcie tworzenia systemu wspierającego prowadzenie ewidencji zabytków archeologicznych powstał także szkielet aplikacji, który został opisany w rozdziale 5.

7. Podsumowanie

W trakcie realizacji niniejszej pracy zapoznano się z technologiami używanymi powszechnie w przemyśle informatycznym przez firmy produkujące oprogramowanie. Dzięki użyciu najpopularniejszych technologii zapewniona była możliwość znalezienia rozwiązania problemów napotkanych w trakcie pisania pracy w internecie.

Aplikacja, która stała się produktem pracy zostanie przekazana do eksploatacji firmie "JN-Profil- badania archeologiczne i historyczne"i będzie dla niej wyraźną pomocą w generowaniu dokumentacji archeologicznej. Przyspieszy to czas jej tworzenia, co wpłynie na zmniejszenie kosztów.

Ponieważ nie udało się zrealizować wszystkich zakładanych raportów, a jedynie część z nich, dalsza przyszłość rozwoju będzie pod znakiem implementacji brakujących elementów.

Pomimo tego, że dziedzina problemu raczej nie powinna ulegać zmianie, to istnieje prawdopodobieństwo, że dokumentacja archeologiczna może wymagać dokumentów, których wygenerowanie nie zostało zapewnione przez system stworzony w ramach niniejszej pracy inżynierskiej. Jeżeli zajdzie potrzeba, system jest w łatwy sposób rozszerzalny i jego rozwój jest jak najbardziej planowany w przyszłości.

Dodatkowymi produktami powstałymi w trakcie pisania niniejszej pracy są komponenty CRUDTable, ForeignField, DefaultForm oraz szkielet aplikacji. Wszystkie komponenty w niedługiej przyszłości zostaną zgłoszone do twórców szkieletu aplikacji jako kandydat na wtyczkę ułatwiającą pracę programistom, którzy wybrali pracę z Vaadinem.

Bibliografia

- [1] Book of Vaadin, <https://vaadin.com/download/book-of-vaadin/vaadin-7/pdf/book-of-vaadin.pdf>, dostęp 29.05.2014.
- [2] Dokumentacja modułu Spring Security, <http://docs.spring.io/springsecurity/site/docs/3.2.5.RELEASE/reference/htmlsingle/>, dostęp 25.07.2014.
- [3] Dokumentacja szkieletu Spring, <http://docs.spring.io/spring/docs/4.0.1.BUILD-SNAPSHOT/spring-framework-reference/pdf/spring-framework-reference.pdf>, dostęp 28.06.2014.
- [4] Oficjalny poradnik narzędzia JasperReports, <https://community.jaspersoft.com/wiki/jasperreports-library-tutorial>, dostęp 18.07.2014.
- [5] Oficjalne poradnik dla szkieletu Spring, <https://spring.io/guides>, dostęp 01.07.2014.
- [6] Joshua Bloch. *Java. Efektywne programowanie. Wydanie II*. Helion, 2009.
- [7] Gavin King Christian Bauer. *Java Persistence with Hibernate*. Manning Publications Co, 2006.
- [8] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*. Wydawnictwa Naukowo Techniczne, Kwiecień 2005.
- [9] Martin Fowler. *Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe*. Helion, 2005.
- [10] Craig Larman. *UML i wzorce projektowe. Analiza i projektowanie obiektowe oraz iteracyjny model wytwarzania aplikacji. Wydanie III*. Helion, 2011.
- [11] John Brant William Opdyke Don Roberts Martin Fowler, Kent Beck. *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*. Helion, 2011.
- [12] Krzysztof Sacha. *Inżynieria oprogramowania*. Wydawnictwo Naukowe PWN, 2010.
- [13] Craig Walls. *Spring w akcji. Wydanie III*. Helion, 2013.