Politechnika Łódzka
Instytut Informatyki

RAPORT Z PROJEKTU KOŃCOWEGO

STUDIÓW PODYPLOMOWYCH

NOWOCZESNE APLIKACJE BIZNESOWE JAVA EE

„System ewidencji kompetencji pracownika”

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Opiekun: dr inż. **Michał** Karbowańczyk
Słuchacz: Kamil Cecherz

Łódź 2015

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Spis treści

1. Cel i zakres projektu

1.1. Wstęp: środowisko pracy, założenia aplikacji.

Przestawiony raport opisuje poszczególne etapy tworzenia i konfiguracji aplikacji webowej z wykorzystaniem platformy *Java EE*. Celem mojego projektu stała się budowa programu wykorzystującego system zarządzania relacyjnymi bazami danych – posłużyłem się *MySQL* **[1]**.

Podczas analizy kolejnych stadiów mojej pracy skupimy uwagę na narzędziach, które sprawiają, że dzisiejsza praca praca programisty/web developera staje się przyjemniejsza, szybsza i bardziej efektywna. Obecnie „produkcja" aplikacji bez skonfigurowanego IDE, takiego jak *NetBeans*, czy serwera aplikacyjnego typu *Glassfish* jest praktycznie niemożliwa. Wysiłek związany z tworzeniem aplikacji w coraz większej mierze sprowadza się do aspektów konfiguracjno-administracynych. Implementacje każdej większej aplikacji poprzedza cały ciąg działań prowadzących do jej przystosowania, zarówno w środowisku produkcyjnym jak i po stronie korzystających z niej użytkowników.

Aplikacja będącą owocem mojej pracy ma za zadanie rozwiązać problem magazynowania, wyszukiwania i agregacji danych biznesowych pracowników firmy IT. Strona www z prostym i przyjemnym widokiem to możliwość rezygnacji z konieczności przechowywania wielu dokumentów CV o niejednoznacznej formie. Zaprojektowana przeze mnie aplikacja ułatwia organizację, korzysta z jednolitego szablonu, pozwala na szybką edycję danych, jest łatwa w obsłudze dla użytkowników nie znających się na szczegółach technicznych.

1.2. NetBeans – start !

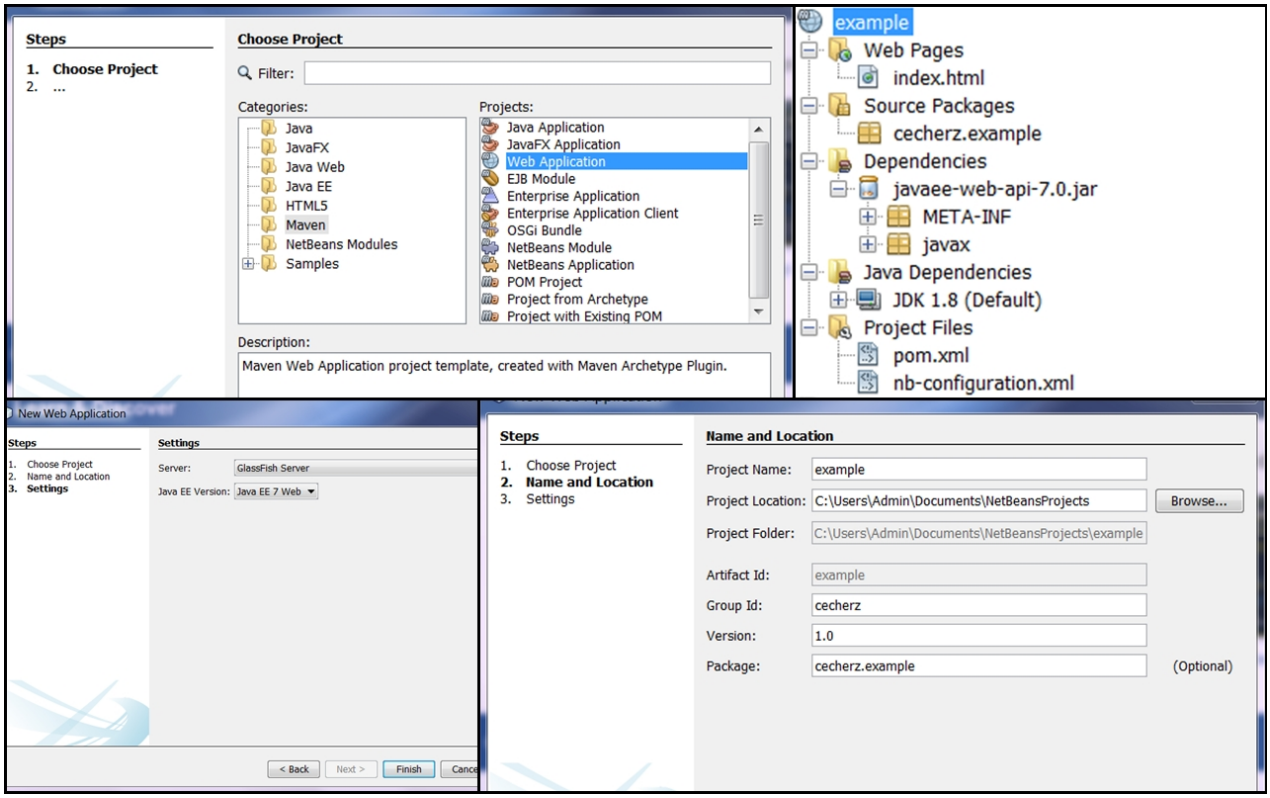
Tworzenie od zera. W tym przypadku aplikacji webowej jest to pojęcie względne, raczej nie buduje się jej linijka po linijce. Precyzyjniej jest powiedzieć, że jest ona składana z wielu gotowych plików klas wymagających edycji ręcznej lub automatycznej przy wsparciu środowiska. Nie sposób tu nie także wspomnieć o programowaniu aspektowym, na które otworzyła się *Java EE w wersji 5*, udostępniając programistom cały zestaw adnotacji pozwalających na rezygnację z ciężkim plików konfiguracyjnych. Nim jednak zagłębimy się w arkana platformy, zajmijmy się opisaniem budowy trzonu całego projektu.

a) Maven

Cały program którego budową się zająłem powstawał w *Apache Maven*, narzędzie to w które można zaopatrzyć jest środowisko programistyczne *NetBeans*, służy jako fundament

aplikacji webowych. Zrzuca ono z programisty ciężar porządkowania bibliotek, powiązań i aktualizacji repozytoriów. *Maven* pozwala na szybkie ładowanie plug-inów, umożliwia zarządzanie wersjami, testowanie programu. Za pomocą prostego pliku konfiguracyjnego – pom.xml, możemy sterować zależnościami rozszerzeń modułów i komponentów. **[2]**

Aby rozpocząć pracę z narzędziem *Maven*, należy w *NetBeans IDE* wybrać: *Menu główne / File / New Project /* folder *Maven* (jeśli nie dysponujemy wtyczką, należy ją zainstalować) */ Web Application.* Po wykonaniu powyższych instrukcji środowisko wyświetli nam kolejno dwa okna:



Rys. 1: Zrzuty ekranu / środowisko NetBeans 8.02 (prawy górny róg, widok projektu bezpośrednio po wygenerowaniu; pozostałe generowanie schematu aplikacji webowej)

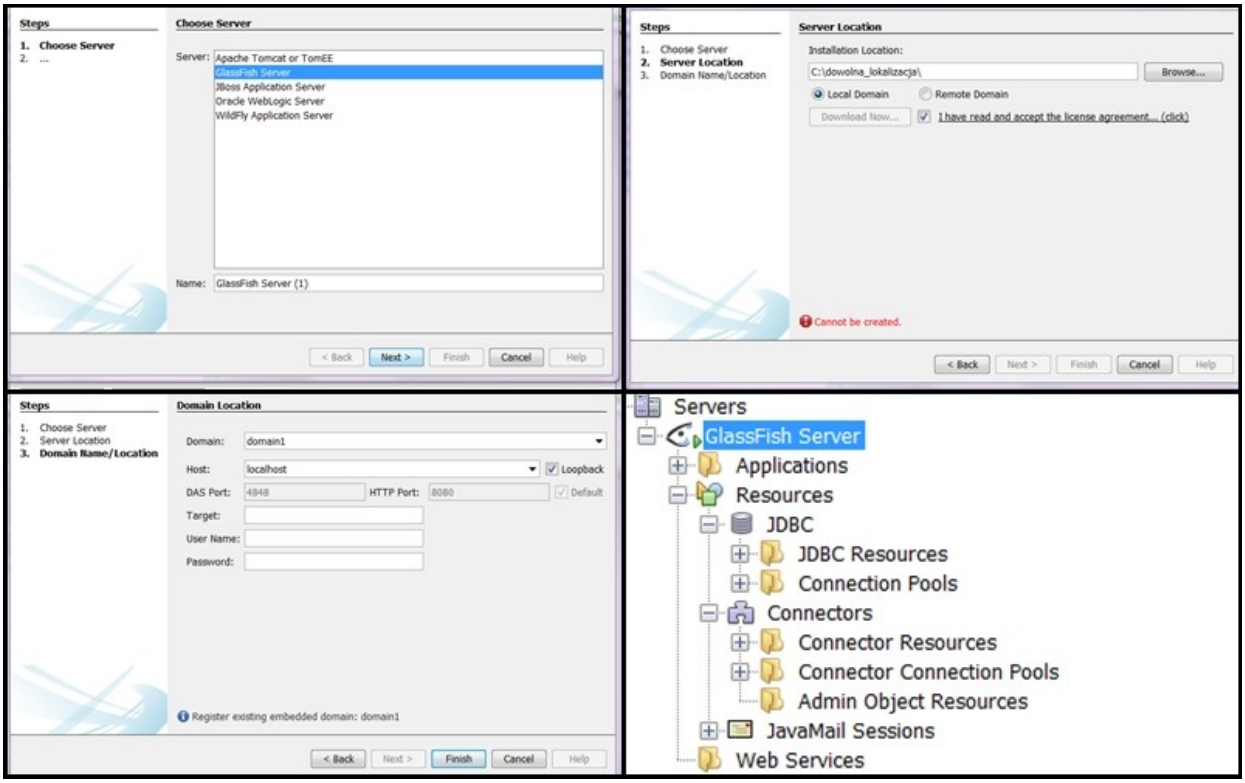
Znaczenie poszczególnych właściwości jest na tyle intuicyjne, że nie wymaga opisu. Uwagę należy zwrócić tylko na zakładkę *settings* gdzie znajduje się rozwijana lista *serwer.* Bez skonfigurowanego serwera dalsze etapy pracy nie będą możliwe!

b) GlassFish

Serwer GlassFish jest dedykowany dla platformy *Java EE*. Po pobraniu rozszerzonej wersji *NetBeans'a* **[3]**, mamy dostęp do niego w pakiecie instalacyjnym. Serwer jest niezbędny do obsługi bazy danych i do wdrażania aplikacji napisanych w Javie. Bez *GlassFish'a* nie zobaczymy pełnego efektu naszych działań, nie przetestujemy programu w przeglądarce.

Teoretycznie można pisać aplikacje nie komunikującego się bazami danych, stąd opisem wstępnej konfiguracji serwera zajmę się na początku.

Konfiguracja **nowego połączenia** rozpoczyna się w środowisku programistycznym. Z menu głównego wybieramy *Tolls / Servers / Add Server… / Choose Server: Glassfish.* Jeśli nie dysponujemy *GlassFish'em* możemy go pobrać do dowolnej lokalizacji, pozostawienie zaś domyślnej konfiguracji zapewni poprawne działanie w celach testowych. Wynikiem pracy programisty jest sprawna aplikacja nie skupiamy się na zaawansowanych aspektach konfiguracji serwera, pole do popisu ma tu administrator. Dostęp do panelu zarządzania *GlassFish* jest możliwy w zakładce: *Services / Servers / Glassfish – menu kontekstowe / View Admin Console* lub poprzez wpisanie w pasek adresu przeglądarki *http://localhost:8080/*.



Rys. 2: Zrzuty ekranu / środowisko NetBeans 8.02 (okna wstępnej konfiguracji połączenia z serwerem; widok zawartości GlassFish z poziomu IDE)

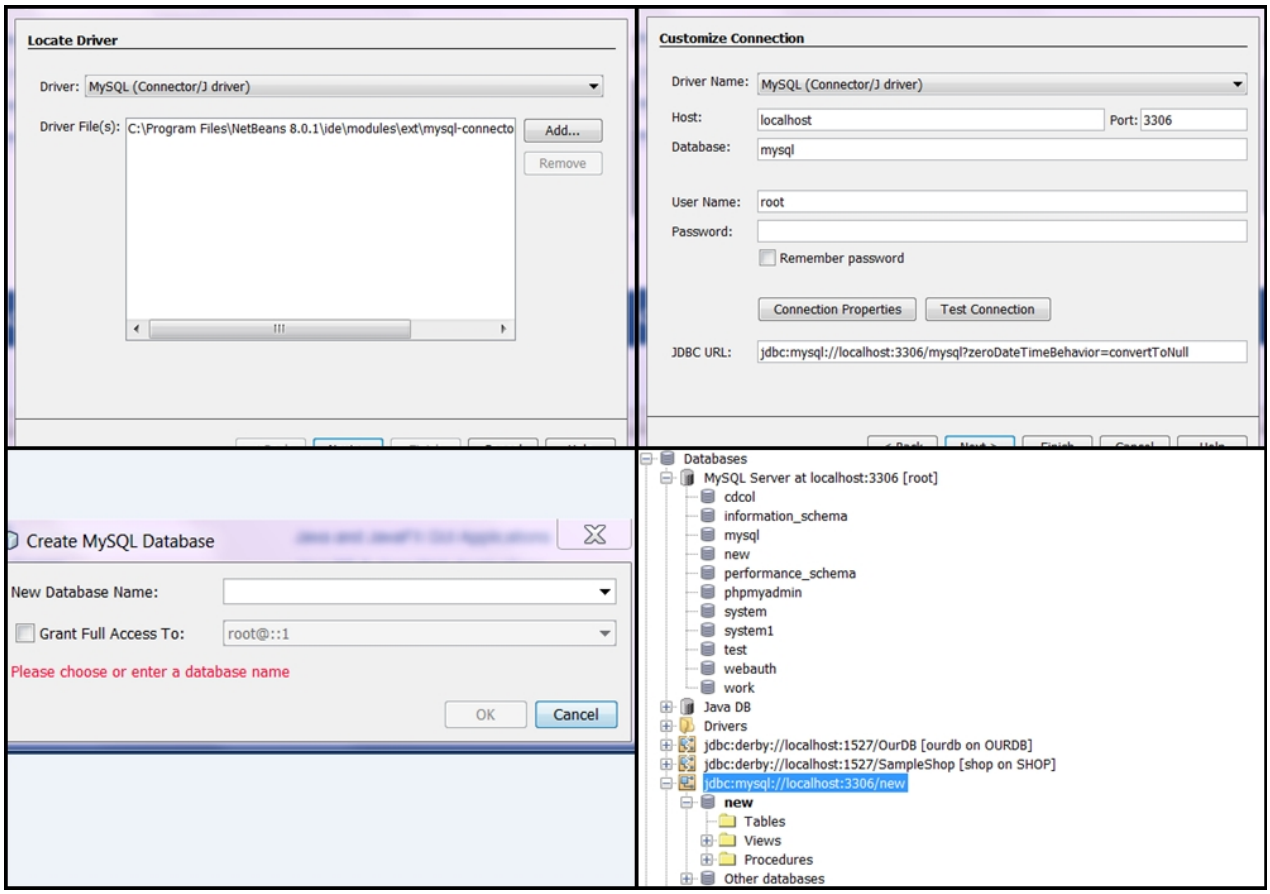
Po poprawnym skompilowaniu kodu w folderze *Applications* (rys.2 widok 4) powinien pojawić się program gotowy do uruchomienia w każdej przeglądarce www. Pozostałe foldery jak wskazuje nazwa *JDBC* przechowują ustawienia konfiguracyjne baz danych.

c) Czas na dane – MySQL

Mając gotowy szablon projektu i działający serwer można przystąpić do konfiguracji połączenia z bazą danych. Standardowo, czyli jeśli korzystam systemu *Java Derby* **[4]**, wspieranego przez środowisko programistyczne, nie wymaga to wiele trudu. Mój projekt zakłada jednak użycie innego systemu baz danych, co stwarza konieczność użycia dodatkowej biblioteki oraz narzędzi:

Konieczne jest dodanie zależności projektowych do biblioteki *mysql-connector-java* **[5]** w widoku projektu *Mavena***:** *Projects* – *Dependecies* lub, co wydaje się rozwiązaniem bardziej uniwersalnym, wgranie archiwum .jar do katalogów *NetBeans'a – / katalog instalacji IDE / de / modules / ext /*. Opcja druga zapewnia możliwość korzystania z *MySQL* za każdym razem, gdy tworzymy nowy program, bez uprzedniego dodawania zależności. Sama biblioteka nie wystarczy, potrzebujemy zewnętrznego programu do obsługi baz. Dostęp do zintegrowanego pakietu narzędzi jest możliwy po pobraniu i zainstalowaniu *XAMPP'A* (mowa tu o systemie zarządzania bazami jak i niezwykle przydatnym wizualnym programie do administracji danymi, *phpMyAdmin*) **[6]**. Wyposażeni w tak potężne IDE jak jest *NetBeans* jesteśmy zwolnieni z obowiązku korzystania z dodatkowych narzędzi do edycji zawartości baz danych, jednak wielu z pewnością doceni łatwość edycji tabel poza środowiskiem programistycznym. Należy jednak uważać by nie doprowadzić do rozbieżności pomiędzy modelem danych zawartym w klasach encyjnych, a tym w tabelach. Odrywanie się od *NetBeans'a* zalecam tylko w przypadku trudnych problemów. Ja z pomocą *phpMyAdmin* wprowadziłem domyślne kodowanie polskich znaków, tabele zaś generowałem bezpośrednio w środowisku programistycznym.

Dostęp do *MySQL* możemy skonfigurować bezpośrednio z poziomu *NetBeans'a*. W tym celu wybieramy zakładkę *Services - Databases* (menu kontekstowe: *New Connection…*). Po otwarciu okna wybieramy typ bazy i odpowiadający mu sterownik.



Rys. 3: Zrzuty ekranu / środowisko NetBeans 8.02 (tworzenie połączenia oraz nowej bazy danych MySQL)

Dane zapisane na w poszczególnych okienkach na Rys. 3 służą celom prezentacyjnym, w celach bezpieczeństwa nie należy tworzyć użytkownika o nazwie *root*, którego autoryzacja nie jest wymagana hasłem.

Po skonfigurowaniu dostępu do systemu bazodanowego, możemy wszystkim zarządzać. Zmiany w IDE są dokonywane na poziomie użytkownika o nieograniczonych uprawnieniach, trzeba więc zachować dużą ostrożność przy edycji tabel, tak by nie uszkodzić wcześniej utworzonych danych**.** Dobrze jest zacząć budowanie bazy aplikacji od klas encyjnych, nie zaś od tworzenia kolejnych tabel przy pomocy *SQL*.

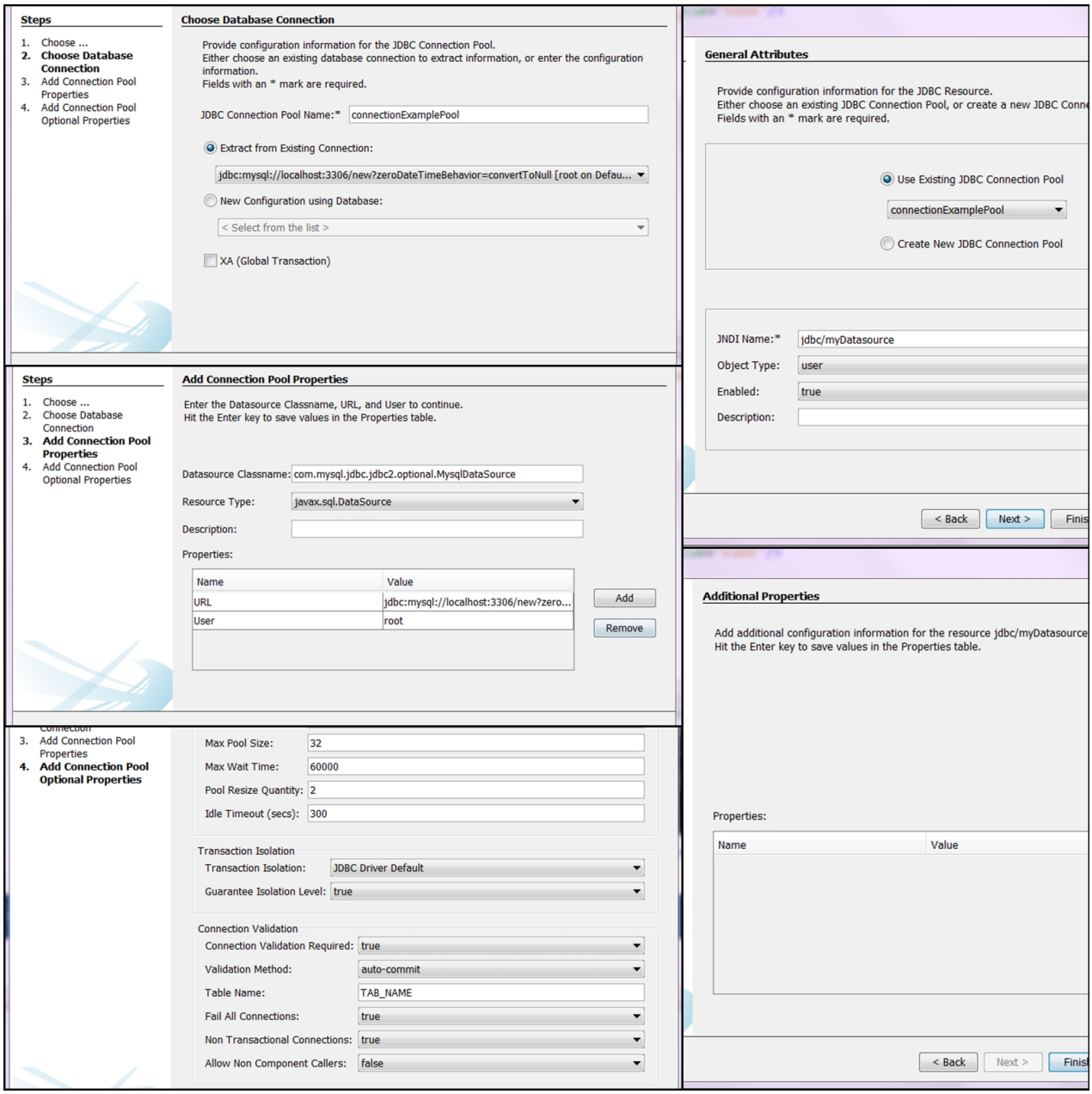
Nową bazę dodajemy zaznaczając serwer bazodanowy w panelu *Service* i wybierając z menu kontekstowego *Create Datebase…* By utworzona baza mogła „współistnieć” z kodem aplikacji musimy stworzyć *JDBC Connection Pool* oraz *JDBC Resource* (mylące polskie odpowiedniki nazw zostały celowo pominięte, angielskie odpowiadają tym bezpośrednio używanym w IDE). Określenia przekładają się na znaczniki w pliku *glassfish-resource.xml*, który odpowiada za komunikację z bazy z serwerem. Właściwości połączenia, do których

należą m.in. nazwa bazy, użytkownika, hosta czy hasło, generujemy bezpośrednio z poziomu środowiska. Osobno wybierając sekcje *JDBC Connection Pool* oraz *JDBC Resource*: *Menu główne / File / New File /* folder *Glassfish.* Tak może wyglądać najprostszy plik *glassfish-resource.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC + ...>
<resources>
  <jdbc-resource enabled="true" jndi-name="jdbc/myDatasource" object-
      type="user" pool-name="connectionExamplePool">
    <description/>
  </jdbc-resource>
<jdbc-connection-pool + ...>
<property name="URL" value="jdbc:mysql://localhost:3306/new"/>
    <property name="User" value="root"/>
    <property name="Password" value=""/>
  </jdbc-connection-pool>
</resources>
```

Plik tworzy się automatycznie, możemy do niego zajrzeć z panelu *Projects: Other Sources / setup /.*

GlassFish pozwala na skorzystanie z szeregu różnych właściwości, w zaprezentowanym pliku ukryłem część parametrów generowanych domyślnie. Dostęp do całego wachlarza ustawień można uzyskać przechodząc bezpośrednio do panelu serwera. Do działania aplikacji jednak wystarczy podstawowa konfiguracja.

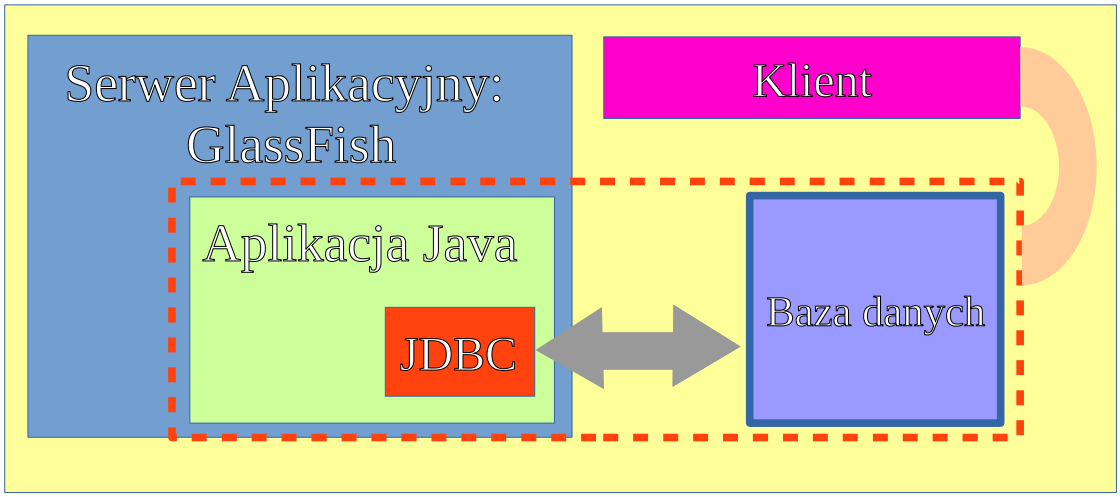


Rys. 4: Zrzuty ekranu / środowisko NetBeans 8.02 (po prawo konfiguracja JDBC Connection Pool; po lewo konfiguracja JDBC Resource)

Przedstawiony etap pozwala na stworzenie trzonu aplikacji, która będzie mogła być bezproblemowo zarządzana.

2. Założenia projektu

Spełnienie wymagań przedstawionych w rdz.1 pozwala na przejście do etapu tworzenia kodu; fazy projektu, która stanowi sedno pracy programisty.



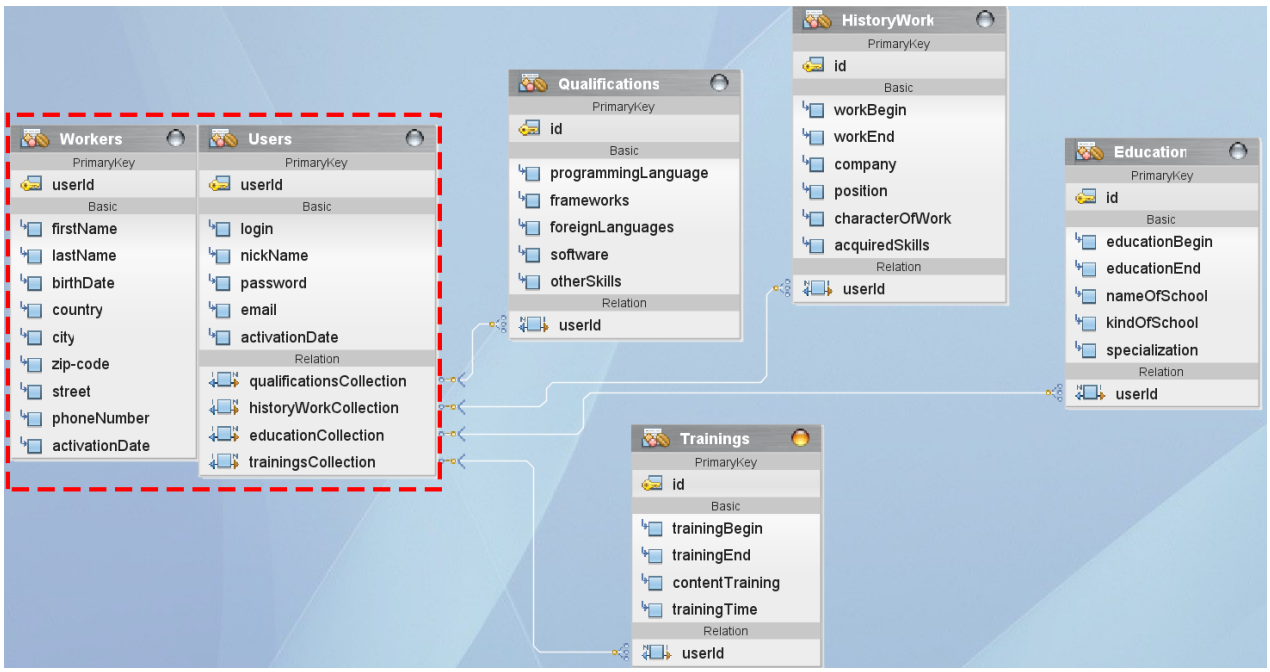
Rys. 5: Schemat otoczenia aplikacji (Serwer przechowuje aplikacje i pozwala jej na kontakt z dowolnym typem baz danych za pomocą interfejsu JDBC

Klientem naszej aplikacji będzie oczywiście przeglądarka www, wszystkie działania miały na celu udostępnienie programu jako witryny internetowej. Już w fazie produkcyjnej programista może podziwiać efekty swojej pracy oglądając wygenerowany xhtml.

2.1. Wprowadzenie do modelu danych.

System, który zaprojektowałem obsługuje bazę złożoną z sześciu tabel, dwie z nich można sklasyfikować jako narzędziowe, służą bowiem jako mechanizm obsługi użytkowników. Tabele odpowiedzialne za dane systemowe *Users* (role, hasła, loginy, identyfikatory, adresy e-mail) oraz podstawowe informacje identyfikacyjne *Workers* (imię, nazwisko, data aktywacji czy adres zamieszkania) mają w moim projekcie odwzorowanie w postaci jednej klasie encyjnej. Efekt uzyskałem dzięki adnotacji *@SecondaryTables* **[7]**. Tabela *Users* stanowi punkt węzłowy, zawiera pola typu *List,* z relacjami @ManyToOne **[8]**, które pozwalają na dostęp do danych zdefiniowanych w pozostałych czterech tabelach odpowiedzialnych za przechowywanie danych biznesowych aplikacji: *Education* (wykształcenie), *HistoryWork* (historia pracy), *Qualifications* (kwalifikacje), *Trainings* (odbyte szkolenia). Tak zaprojektowany model jest uzasadniony: przykładowo, użytkownik może mieć tylko jeden login ale nie można uniemożliwić mu wprowadzania kolejnych szkoleń odbywanych w

różnych miejscach i czasie. Przedstawione podejście jest konieczne, by aplikacja spełniała swoje wymogi, inaczej byłaby ona nieużytecznym, niedopracowanym narzędziem.

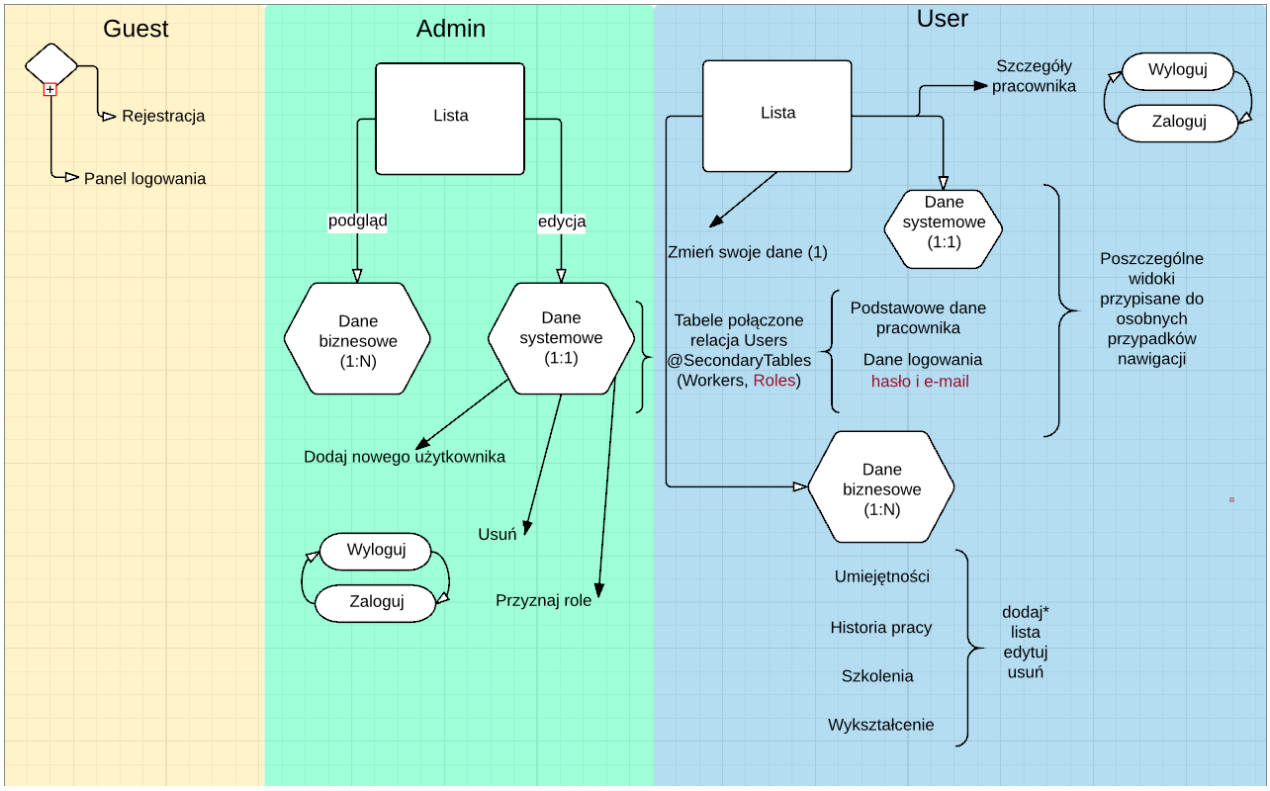


Rys. 6 Zrzuty ekranu / środowisko NetBeans 8.02 – narzędzie do generowania diagramów (model bazy danych).

Widok zaprezentowany na rys. 6 ma charakter pomocniczy, nazwy kolumn w bazie i odpowiadające im pola w klasach encyjnych są różne. Przedstawiłem na diagramie klasy encyjne tabel *Users* i *Workers* jako rozdzielne, by zobrazować zależności stworzone w aplikacji. W kodzie aplikacji tabela *Workers* nie istnieje jako osobna klasa encyjna, odwzorowanie jej pól znajduje tabeli pracowników się w klasie encyjnej *Users*.

2.2. Opis funkcjonalności oferowanej przez aplikacje.

a) Uprawnienia poszczególnych użytkowników.



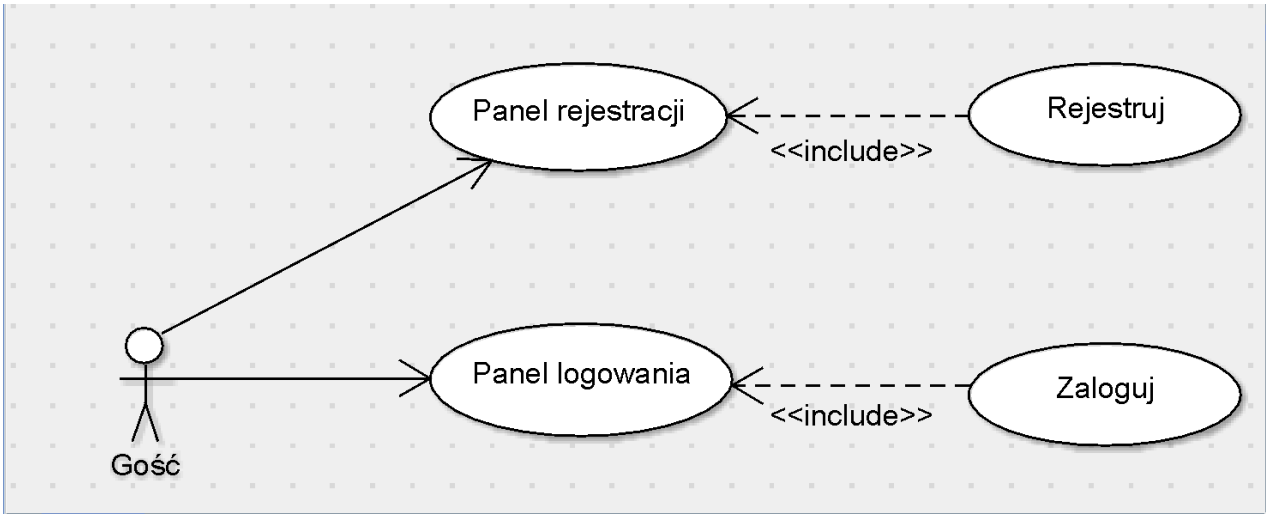
Rys. 7 *Listę* można rozumieć jako wykonanie polecenia SELECT na konkretnych tabelach. Konkretnie chodzi o pakiet *java.util.List* i o *javax.faces.model.DataModel*, który pozwala na wyeksponowanie danych z bazy w postaci obiektowej listy.

Przedstawiony model nie uszczegóławia funkcjonalności programu. Mam on na celu skondensowanie opisu związanego z mechanizmem zarządzania. Już tutaj uwidacznia się warstwowość aplikacji: dane z bazy muszą zostać odpowiednio spreparowane przez logikę, by można było je wyświetlić w widoku. Pojęcie *Listy* wymaga dokładnego zrozumienia, stanowi ono paradygmat. Nie można tutaj wszystkiego uprościć, gdyż dane trafiające do *Listy*, w naszym rozumieniu pewnego specyficznego zbioru, są poddawane innym operacjom na kolejnych poziomach aplikacji. Informacje nim zostaną wyświetlone są przetwarzane i filtrowane.

Aplikacja zakłada prostą i intuicyjną obsługę, stworzyłem tylko niezbędne przypadki użycia, można jednak zauważyć, że system łatwo rozszerzyć: wprowadzając nowe typy użytkowników, dodając kolejne widoki i akcje. Operacje użytkownika, jakie może wykonać w granicach aplikacji skupiają się wokół *Listy*. Pojęcie to bardzo ogólnie definiuje użyteczność systemu. Od niego dochodzimy do zagadnienia składowania danych, a co za tym idzie podziału ról, szybkiego wyszukiwania i edycji wielu powiązanych ze sobą kwerend. Jak

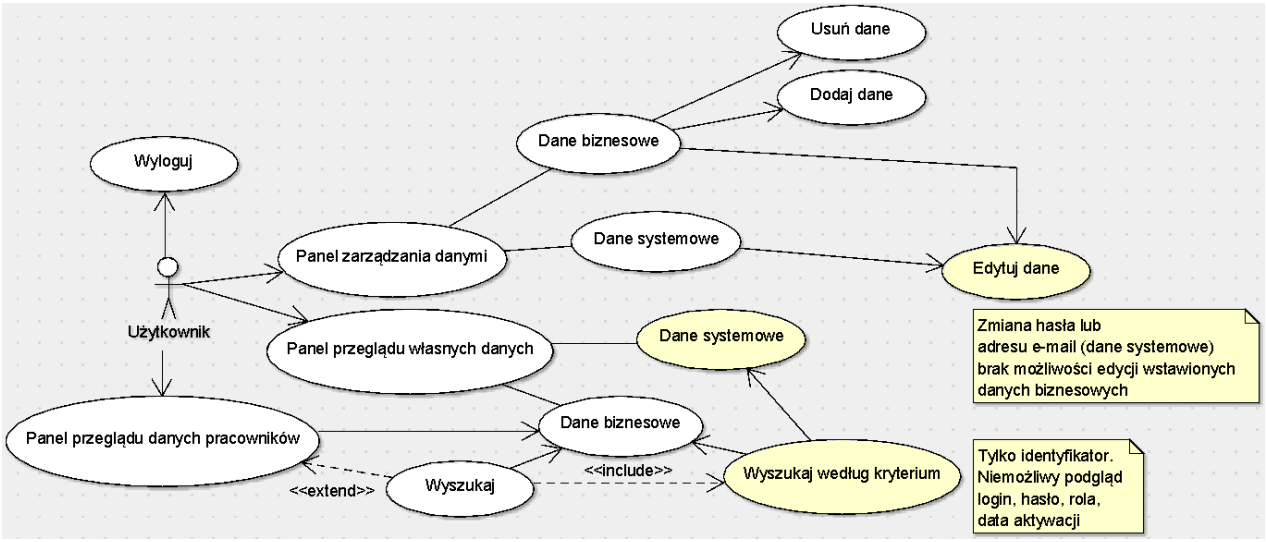
każdej szanującej się bibliotece czy archiwum, w komputerowej bazie danych powinien panować porządek. Za wszystko odpowiada umiejętne zarządzanie.

W przypadku tej konkretnej aplikacji: systemu ewidencji kompetencji pracownika, problem podziału na role częściowo rozwiązuje się sam. Nie potrzeba wielu poziomów dostępu, gdyż celem programu jest szybkie wprowadzanie informacji o własnych umiejętnościach oraz podgląd danych innych użytkowników.



Rys. 8 Diagram przypadków użycia użytkownik nieautoryzowany

Gość: lub osoba, która jeszcze nie uzyskała lub nie może uzyskać autoryzacji. Każdy kto uruchomia aplikację ma prawo obejrzeć stronę główną z której może przejść według woli, do panelu logowania, by się uwierzytelni, lub panelu rejestracji, by po ustaleniu hasła i loginu zostać pełnoprawnym użytkownikiem systemu. Mechanizm ten to standardowy sposób rozwiązujący problem bezpieczeństwa danych: dane logowania zna tylko osoba tworząca konto.

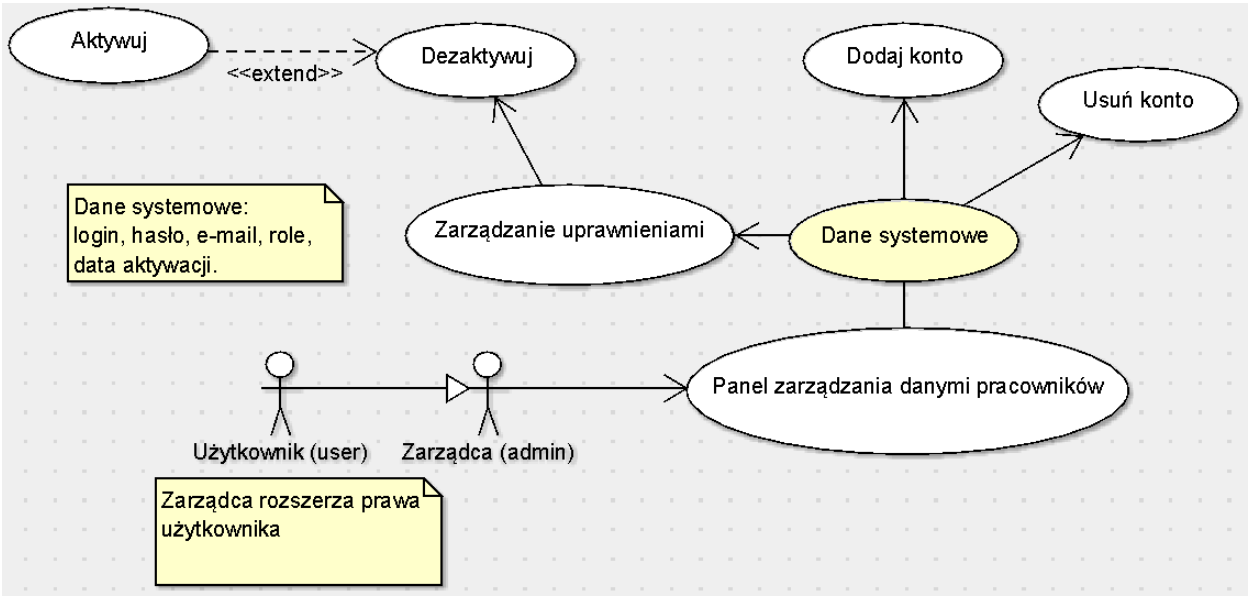


Rys. 9 Diagram przypadków użycia rola user

Użytkownik (user): rola ta jest przyznawana automatycznie po przejściu przez proces rejestracji. System zapisuje login, identyfikator, szyfruje hasło i ustala datę założenia konta. Użytkownik dysponuje stałym zestawem danych, których nie może modyfikować: login, data aktywacji, rola (są one potrzebne zarządcy do identyfikacji poszczególnych kont, tylko on może zmienić rolę). Danymi użytkownika, które on sam może edytować są: adres e-mail i hasło.

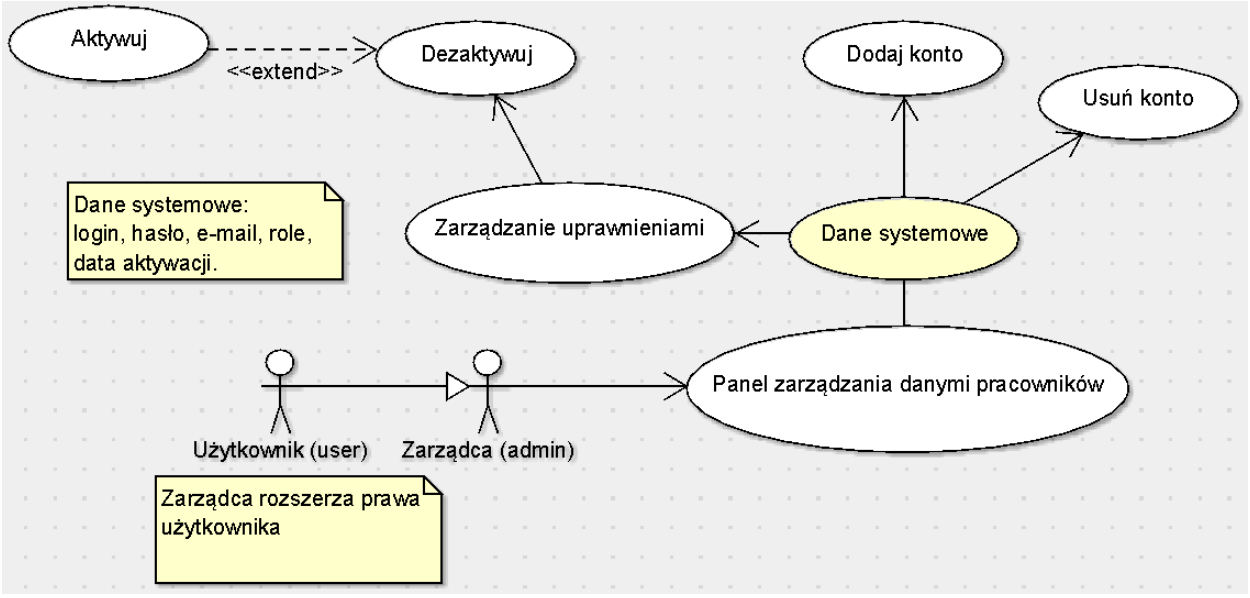
Tabele z danymi biznesowymi podlegają innym prawom, gdyż wszyscy pracownicy firmy mają do nich dostęp. Cała elastyczność systemu skupia się warstwowym podziale danych, prywatne mają ograniczony dostęp, biznesowe uznajemy za publiczne (nie ma konieczności ich dodawania od razu, wszystko zależy od zastosowania). Danych biznesowych, czyli ukończonych szkoleń, ukończonych szkół, historii pracy, kwalifikacji nie ma możliwości ukrycia – byłoby to rozwiązanie niepraktyczne. Można je jednak edytować, usuwać.

Użytkownik nie ma dostępu do żadnej z danych systemowych innych pracowników, które służą bezpieczeństwu konta. Jawnej prezentacji polegają jednak unikatowe identyfikatory, które możemy też nazywać ksywkami. Do edycji tych danych nie ma dostępu żaden z użytkowników. Pole to jest wypełniane raz podczas zakładania konta, służy wyłącznie lepszej orientacji.



Rys. 10 Diagram przypadków użycia rola admin

Zarządca (admin): czyli użytkownik o zwiększonych prawach. Admin ma, jako osoba uprawiona do administracji systemem, prawo przeglądać wszystkie dane pracownika. Nie ma jednak możliwości jawnego przejrzenia haseł, gdyż jest one zaszyfrowane. Zarządca swobodnie wprowadza dane systemowe: może stworzyć konto nowego użytkownika z poziomu autoryzowanego użytkownika jak i usunąć już istniejące. Tylko admin, może mianować kolejnego admina jak dezaktywować czasowo konto innego użytkownika. Zarządca nie jest uprawniony do edycji danych biznesowych użytkowników systemu.



Rys. 11 Diagram przypadków użycia rola admin

Tabela 1. Uprawnienia do korzystania z własnych danych wprowadzonych do systemu

	hasło	login	ksywka	e-mail	rola	data aktywacji	Dostęp do panelu rejestracji
gość	brak	brak	brak	brak	brak	brak	możliwy
nieaktywny użytkownik	posiada w bazie, użycie niemożliwe	posiada w bazie, użycie niemożliwe	posiada w bazie, użycie niemożliwe	istnieje w bazie, brak dostępu	istnieje w bazie	istnieje w bazie	niemożliwy
user	posiada w bazie, możliwa edycja i pogląd	posiada w bazie, edycja niemożliwa, możliwy podgląd	posiada w bazie, edycja niemożliwa, pogląd możliwy	posiada w bazie, edycja niemożliwa edycja podgląd	posiada w bazie, niemożliwa edycja i podgląd	posiada w bazie, niemożliwa edycja i pogląd	niemożliwy
admin	posiada w bazie, możliwa edycja i pogląd	posiada w bazie, edycja niemożliwa, możliwy podgląd	posiada w bazie, edycja niemożliwa, pogląd możliwy	posiada w bazie, edycja niemożliwa, możliwy podgląd	posiada w bazie, edycja niemożliwa, możliwy podgląd	posiada w bazie, edycja niemożliwa	możliwy

Tabela 1. Uprawnienia do korzystania z danych innych pracowników na poziomie systemu.

	hasło	login	ksywka	e-mail	rola	data aktywacji	Dostęp do panelu rejestracji
gość	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	możliwy
nieaktywny użytkownik	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	niemożliwy
user	nie posiada dostępu	nie posiada dostępu	posiada dostęp, edycja niemożliwa	nie posiada dostępu	nie posiada dostępu	nie posiada dostępu	niemożliwy
admin	nie posiada dostępu	posiada dostęp, edycja niemożliwa	posiada dostęp, edycja niemożliwa	posiada dostęp, edycja niemożliwa	posiada dostęp, edycja możliwa	posiada dostęp, edycja niemożliwa	możliwy

W systemie fizycznie nie istnieje *rola gościa*, obecna jest za to *rola inactive* czyli użytkownika, którego dostęp do systemu został czasowo lub trwale zablokowany. Po dezaktywacji pracownik będzie miał tylko możliwość wyświetlenia strony z komunikatem. Warto zauważyć, że model zakłada, co jest absolutnie koniecznie, brak dostępu do panelu rejestracji, po zalogowaniu, zwykłemu użytkownikowi. Gdy stało się inaczej, podział ról przestałby mieć sens.

Przedstawiony model dostępu jest realizacją jednego z wielu możliwych scenariuszy, nie można go jednak traktować jako szablonu, wszystko zależy od założeń projektu. Ja starałem się zapewnić korzystającemu z aplikacji maksimum bezpieczeństwa. Warto zauważyć, że podział ról, a co za tym idzie decyzja o dostępie do poszczególnych danych, nie jest kwestią trywialną. Mogą zastanawiać ograniczone możliwości administratora, zostaje w zasadzie tylko zmiana roli. Obecność roli admina, nie ważne jak „silnego", ułatwia w późniejszych wersjach aplikacji dodanie nowych funkcjonalności. Skupiłem się na stworzeniu klarownego systemu, jego rozbudowa jest drugoplanowa. Architektura aplikacji to zagadnienie wyjątkowo istotne, w moim mniemaniu pierworzędne.

b) Wyszukiwanie

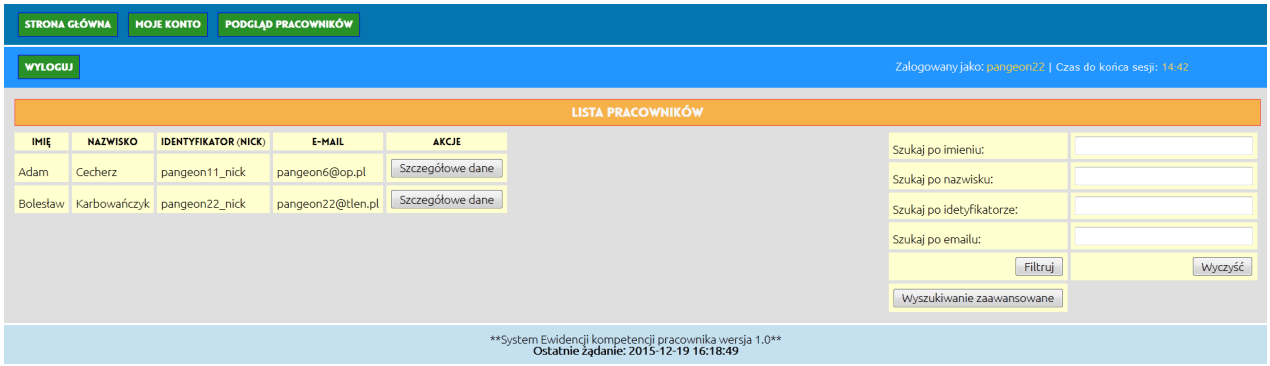
Wyszukiwanie odbywa się przy pomocy filtrowania, na podstawie różnych kryteriów. Jest to kolejna funkcjonalność stanowiąca fundament prezentowanej aplikacji. Dostęp do konkretnej grupy danych, użytkownik definiuje zakres zbioru, jest szybki. Zarządzanie informacją przy pomocy komputera jest bardziej efektywne. To banalne być może stwierdzenie leży u podstaw projektu. Magazynowaniem informacji zajmuje się baza danych, by jednak sprawnie móc z nich korzystać potrzebny jestem ułatwiający wydawanie prostych poleceń, który zarazem wyeksponuje wszystko w postaci dokumentów .xhtml. Łatwo jest wyobrazić sobie kilku użytkowników, w końcu to tylko model aplikacji prezentujący wybrane rozwiązania. Nie sposób jednak byłoby owej aplikacji wdrożyć jako narzędzia do obsługi przykładowo: tysiąca pracowników firmy N. W danej chwili może potrzebnych jeden lub więcej pracowników o specyficznych kwalifikacjach – i co wtedy?

Moja aplikacja realizuje wyszukiwanie poprzez zdefiniowany przeze mnie sposób: nie ma pewności czy dla wszystkich będzie on dobry i intuicyjny. W tym miejscu chce zaznaczyć, że nie tylko na barkach programisty leży decydowanie o funkcjonalności aplikacji, zwykle robi to cały zespół.

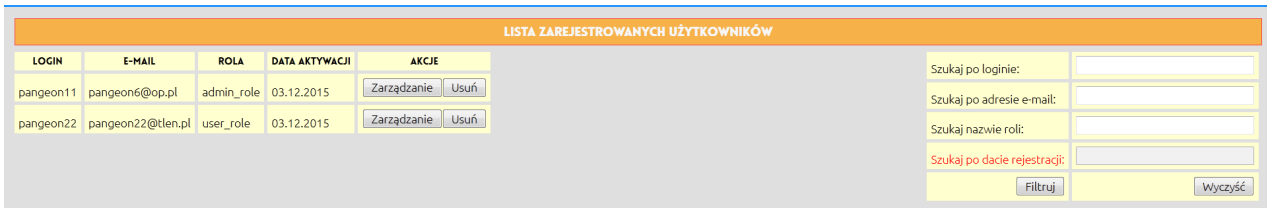
W moim programie o sposobie wyszukiwania decyduje poziom dostępu, a tym samym możliwości obejrzenia konkretnych danych. Tylko administratorzy będą mogli wyszukiwać użytkowników po loginie, adresie e-mail czy roli (dane dostępne w panelu administratora). Użytkownicy nie otrzymają możliwości wyszukiwania danych systemowych, gdyż te nie są im udostępnione, ale będą mogli skorzystać z:

- **Szybkiego wyszukiwania pracowników:** po imieniu, nazwisku, identyfikatorze

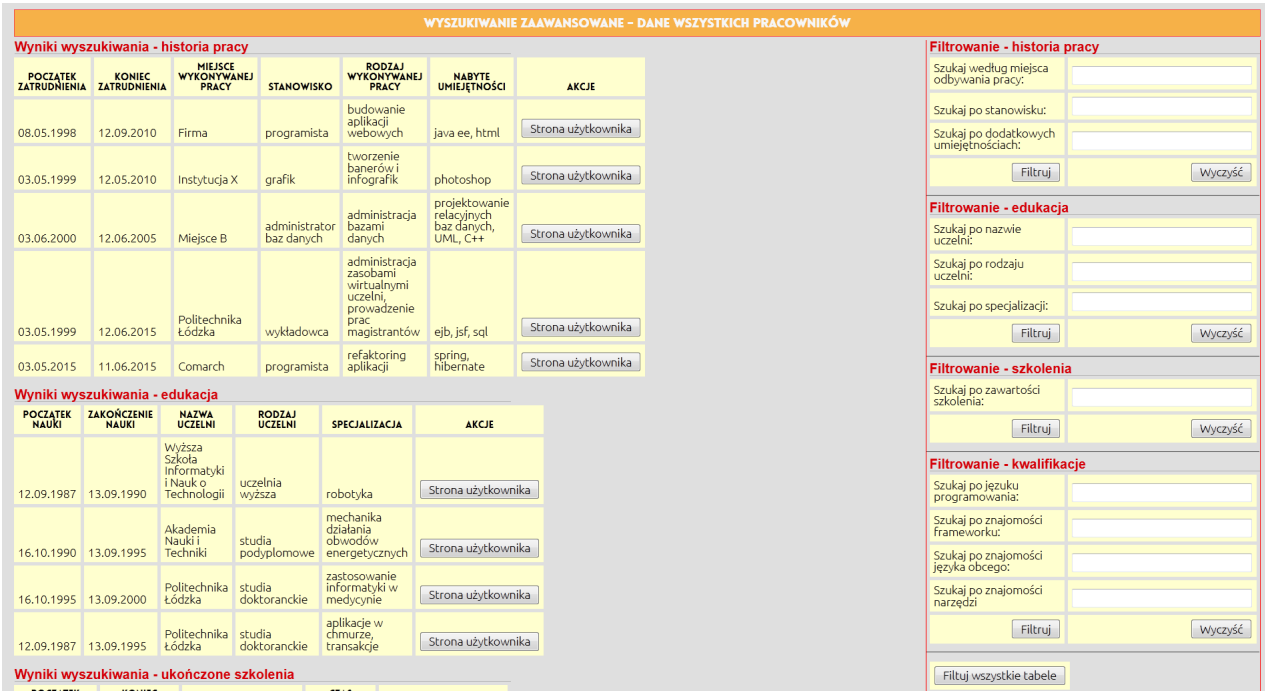
- **Zaawansowanego wyszukiwania według kwalifikacji:** w bazie dysponujemy całym zestawem danych biznesowych postaci kolekcji. Oferuje w mojej aplikacji możliwość filtrowania danych z tabel *historia pracy, edukacja, ukończone szkolenia, kwalifikacje* (nazwy tabel bezpośrednio w aplikacji są inne). Po odbiorze danych użytkownika z formularzy, które odwołują się tylko do wybranych kolumn z tabel, następuje filtrowanie (każdej z kolekcji osobno). Wyniki wyświetlone zostają w wielu tabelach osobno. Każdy wiersz odwołujący się do danych z kolekcji odsyła do widoku pełnych danych użytkownika.



Rys. 12 Zrzut ekranu z działającej aplikacji – szybkie wyszukiwanie



Rys. 13 Zrzut ekranu z działającej aplikacji – szybkie wyszukiwanie dla administratorów



Rys. 14 Zrzut ekranu z działającej aplikacji – zaawansowane wyszukiwanie

Aplikacja tworzy zapytania SQL z uwzględnieniem AND co oznacza, że każde kryterium musi być łączne, jeśli nastąpi inaczej zbiór wyszukanych obiektów będzie pusty. Wpisany ciąg znaków do poszczególnych pól jest interpretowany jako początkowy element składowy Stringa w bazie. Zrealizowane zostaje porównanie do wzorca, w SQL operator LIKE. Rozróżniane są duże i małe litery.

c) Bezpieczeństwo aplikacji.

Istotnym zagadnieniem, szczególnie podczas budowy dużych korporacyjnych aplikacji, jest bezpieczeństwo. Skala mojego projektu nie pozwala oddać złożoności problematyki, ochrona jest tematem na osobną rozprawę. Nad złożonymi systemami pracują zespoły złożone z dziesiątek programistów.

Na mój skromny, niemniej bezpieczny, system składa się:

- blokada w postaci konieczności autoryzacji: tylko zarejestrowani mają dostęp do funkcjonalności jakie oferuje aplikacja.
- szyfrowanie haseł: hasła podawane podczas rejestracji zostają przetworzone przez funkcje mieszającą *SHA-256***[9]** i zapisane w bazie danych. Nie istnieje funkcja, która pozwala na odwrócenie procesu, tj. otrzymania pierwotnej wiadomości z samego haszu. Widok wymaga od użytkownika hasła zgodnego ze wzorcem, o długości minimum 8 znaków.

- kontrola długości trwania sesji: użytkownik, który nie wykonał żądania przez 15 minut zostaje automatycznie wylogowany.

- odpowiedni podział dostępu do danych: standardowi użytkownicy nie widzą danych odpowiedzialnych za autoryzację, wewnętrzną identyfikację przez system; jako dane prywatne zostały też potraktowane adresy e-mail (w razie zagubienia hasła może zaistnieć konieczność wygenerowania nowego i przesłania go pocztą).

- przydzielanie ról wewnątrz systemu: administrator może wyłączyć konto prawnika bez konieczności usuwania go z systemu, aczkolwiek taka możliwość też istnieje.

- kontrola **HTTP**: plik *web.xml* definiuje jakie metody mogą być wykonane przez użytkowników systemu, w moim przypadku jest to **GET** oraz **POST**.

- kontrola wersji: dane systemowe użytkownika są zabezpieczone przed niewłaściwą edycją poprzez zastosowanie mechanizmu blokad optymistycznych. W czasie próby modyfikacji danych tabeli Users oraz Workers, bez uprzedniego ich odświeżenia, po zmianach przez inną otwartą sesje mającą dostęp do konta, użytkownik otrzyma komunikat o błędzie. *De facto* nie da zmieniać danych, które już uległy zmianie bez wczytania ich aktualnego stanu, co jest bardzo pożądane.

Bezpieczeństwem przedstawionej przeze mnie aplikacji zajmuje się serwer *Glassfish*, rozwiązanie to pozwala na szybkie skorzystanie z funkcjonalności ale wymaga od programisty współpracy z serwerem aplikacyjnym. Nie ma potrzeby tworzenia nowych klas obsługujących np. logowanie, wystarczy sprawna konfiguracja gotowych mechanizmów.

Budowa „ściany autoryzacyjnej" jest schematyczna, nie możemy złamać pewnych konwencji wypracowanych przez autorów narzędzi GlassFish inaczej nie wprowadzimy funkcjonalności, gdyż nic nie zadziała.

Zaczynamy od stworzenia formularza, o ściśle wyznaczonym nazewnictwie:

```
<form method="POST" action="j_security_check">
      Nazwa użytkownika:
      <input type="text" name="j_username" />
      Hasło:
      <input type="password" name="j_password" />
      <input type="submit" value="Zaloguj" />
</form>
```

W przedstawionym, możliwie najprostszym przykładzie formularza, nazwy pól muszą być zgodne z określonymi czyli **j_username** (pole odpowiedzialne za login) oraz **j_password** (hasło). Metodę POST w formularzu uzupełnia akcja **j_security_check –** nie może być inaczej! Sztywny formularz jest niewielką ceną za funkcjonalność i wygodę jaką oferuje *GlassFish.* **[10]**

W przypadku mojej aplikacji informacje na temat bezpieczeństwa znajdują się w bazie danych. Samo-zabezpieczenie użytkownika następuje poza jego kontrolą podczas gdy dokonuje rejestracji. Jeśli dane zostaną poprawnie zweryfikowane, system utworzy konto przypisując mu wprowadzone przez nas hasło (w postaci skrótu SHA-256), login, e-mail, identyfikator oraz aktualną datę (rozumianą jako data rejestracji, widoczną tylko adminom).

Zbudowałem mechanizm w pełni automatyczny, w oparciu o kryteria GlassFish'a. Warto prześledzić cały proces i dokonać opisu.

Zacznę od plików konfiguracyjnych, które w prosty sposób można dodawać i edytować z poziomu środowiska IDE.

- **Plik web.xml**, w kwestii bezpieczeństwa, przechowuje dane dotyczące: sposobu logowania (w moim przypadku będzie to ustawienie *web.xml/Security/Login Configuration/Form*), nazw poszczególnych ról, zasoby przydzielone rolom (ścieżki URL lub ścieżki plików klas Javy), ograniczenia dotyczące wykonania metod HTTP, informacje o szyfrowaniu protokołu. Tutaj decydujemy ile ról chcemy stworzyć w systemie i dostęp do których zasobów im umożliwić, nazwy są istotne, warto je zapisać, gdyż są konieczne w dalszych etapach konfiguracji. Podczas wyboru sposobu logowania podajemy stronę ze zdefiniowanym wcześniej formularzem (z akcją **j_security_check**), a jako stronę błędu można podać tę samą stronę, ale ze stosownym komunikatem o niepowodzeniu akcji. Rozwiązanie takie jest klarowne dla użytkownika aplikacji.

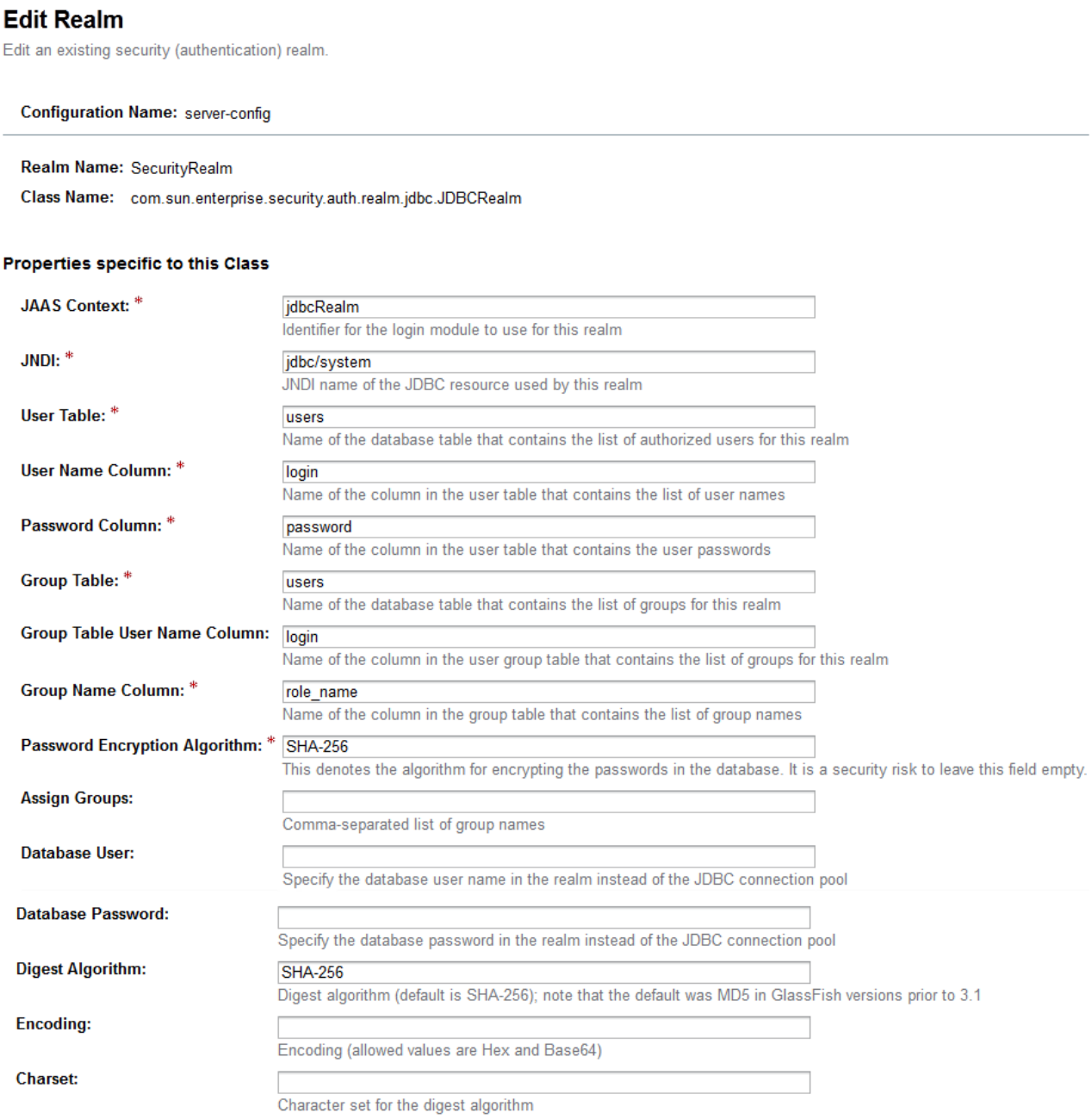
- **Plik glassfish.xml,** zawiera prostą konfigurację gdzie znajduje się mapowanie ról na poszczególne grupy.

Rozpisane konfiguracji w plikach .xml projektu jest stosunkowo proste i intuicyjne, trudniejsze może okazać się dostosowanie serwera do wybranych ustawień. By zdefiniować nowy *Realm* należy wybrać w panelu zarządzania serwerem

(http://localhost:4848/common/index.jsf) *Configurations/server- config/Security/Realms/New …*

Podczas tworzenia domeny bezpieczeństwa należy wyraźnie zaznaczyć gdzie będą przechowywane dane, w przypadku mojej aplikacji konieczny jest wybór klasy *com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm.*

Ten sposób tworzenia mechanizmu autoryzacji wymaga określenia specyficznych właściwości - założeniem autorów narzędzia jest, że aplikacja posiada osobną tabelę odpowiedzialną za przechowywanie danych autoryzacyjnych (hasło, login) oraz tabelę ról, która pozostaje w relacji z nazwami użytkowników. Mniej eleganckim, ale skutecznym rozwiązaniem, jest stworzenie kolumny z rolami w jednej tabeli z danymi autoryzacyjnymi - tak właśnie zrobiłem tworząc własną aplikacje. Kodowanie haseł nie jest konieczne, ale jeśli nie określimy go jako *none, GlassFish* zastosuje domyślne funkcje haszujące. Opis wszystkich aspektów konfiguracyjnych serwera można znaleźć w dokumentacji.**[11]**



Rys. 15 Konfiguracja domeny w panelu zarządzania GlassFish.

d) Opis warstwy widoku.

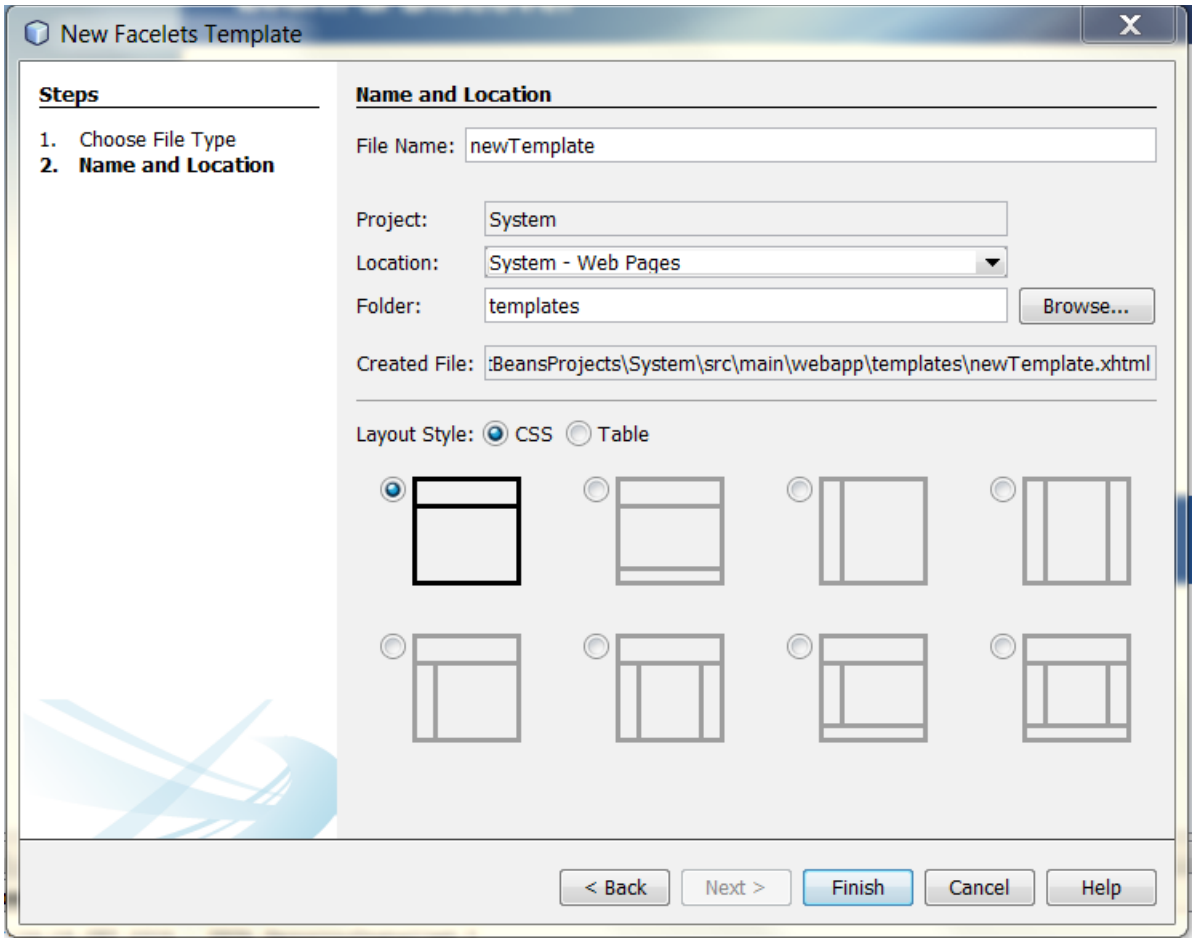
Obsługą kontrolera w mojej aplikacji zajmuje się framework JSF 2.2. **[12]**

JSF możemy rozumieć jako pewnego rodzaju paletę narzędzi, która pozwala na integrację warstwy widoku i silnika aplikacji. Programista otrzymuje zestaw znaczników o zdefiniowanej funkcji, których właściwości musi określić eksponując wyznaczony przez siebie zasób danych. Problem właściwej ekspozycji danych zajmuje tu miejsce kluczowe, to architekt aplikacji musi zdecydować o sposobie obsługi danych pobranych od jej użytkownika. Nie ma

wymogu budowania stosowania wzorców programistycznych, jeśli istnieje taka potrzeba dane z bazy można wyeksponować bezpośrednio w widoku, a kod Javy zapisać na stronach w postaci skrypletu. Praktyka taka jest wysoce odradzana, aplikacja z pozoru prostu prostsza, po rozbudowie może stać się bardzo trudna w utrzymaniu nawet przez jej własnego autora.

Podczas tworzenia systemu nie korzystałem ze stron JSP gdyż moim calem było zachowanie spójności i podziału aplikacji na warstwy. Rozwiązania takie jak JSP czy XUL są obecnie wypierane przez nowocześniejszy standard jakim jest JSF. Wszystkie strony do jakich użytkownicy w mojej aplikacji mają dostęp to .xhtml

Wykorzystałem dość wygodny system budowania szablonów wspomagany przez środowisko NetBeans. Statyczną część widoku, podlegającą wielokrotnemu powieleniu udostępniłem jako templatki: **<c:compisition>**, są one wykorzystywane opcjonalne w kolejnych stronach aplikacji: **<ui:define><ui:include></ui: udefine>**. Zasoby dodatkowe takie .css**,** pliki fontów czy skrypty .js zostały zamieszczone w osobnym katalogu, w wyznaczonych lokalizacjach. Krótko mówiąc, warto określić: co, gdzie i jak, na początku pracy ze stronami, by ich tworzenie nie było czynnością mozolną, a w pełni zautomatyzowaną i szybką.



Rys. 16 Zrzut z IDE NetBeans generowanie szablonu JSF.

O ile wprowadzanie w warstwie widoku bezpośrednio dostępu do danych z bazy jest rozwiązaniem niepraktycznym, o tyle blokady na polach formularza dotyczące walidacji wprowadzanych danych mogą okazać się przydatne, aczkolwiek nie ma jednej zasady, która określiłaby jak sprawdzać informacje przez użytkowników do wnętrza aplikacji. Jeśli chodzi o np. system logowania nie ma możliwości zdefiniowania na stronie formularza mechanizmu autoryzacji ale w łatwy sposób da się zrobić prostą walidację, w wyniku czego użytkownik dowie się, że pomylił się podczas wpisywania danych.

Wszystkie strony formularzy w tworzonej przeze mnie aplikacji zawierają przykłady walidacji danych wprowadzanych przez użytkownika. Co to daje? Przykładowo, jeśli użytkownik aplikacji nie spełni wymagań dotyczących liczby znaków określonych w kolumnie w bazie danych, może to być hasło o minimalnej liczbie znaków 8, otrzyma klarowny komunikat na zamiast nie mówiącej mu nic strony błędu. Wyjątek „zostanie zgłoszony” nim w ogóle wystąpi, taka strategia zapewnia wygodę, niemniej należy się wystrzegać pochopnej rezygnacja z obsługi wyjątku w aplikacji.

Walidacji danych dokonujemy osadzając w znacznikach JSF pól formularza dodatkowe znaczniki z odpowiednimi właściwościami:

```
<h:inputText id="reg_email" value="#{newUserPageBean.user.email}" >
      <c:validateLength minimum="7" maximum="40" />
      <c:validateRegex pattern="[\w\.-]*[a-zA-Z0-9_]@[\w\.-]*[a-zA-Z0-9]\.[a-
      zA-Z][a-zA-Z\.]*[a-zA-Z]" />
</h:inputText>
<h:message for="reg_email" />
```

W tym konkretnych przypadku mamy do czynienia ze sprawdzeniem zgodności wprowadzonego adresu e-mail: liczba znaków w podziale 7-40 i wzorzec, który uniemożliwia wpisanie np. *paczta@@poczta::prl* (adresu meilowego, który istnieć nie może).

Za wyświetlenie komunikatu błędu będzie odpowiadał znacznik **<h:message />.** Treści komunikatów są w języku angielskim i niektórym może nie odpowiadać ich forma, nie nadają się one do tworzenia przyjaznych użytkownikom aplikacji. Komunikaty są pobierane z pliku *jsf_mesages.properties* biblioteki *jsf-api.* Aby stworzyć własny zestaw komunikatów, wystarczy zrobić kopie pliku właściwości i przerobić jego treść, by odpowiadała naszym wymaganiom, np:

```
javax.faces.validator.RegexValidator.NOT_MATCHED=
[DEFAULT]Wartość niezgodna ze wzorcem. Wpisz poprawny adres-email.
```

Plikom odpowiedzialnym za komunikaty można nadawać wersje językowe, uzyskując w ten sposób internacjonalizację. W zależności od kraju, a tym samym ustawień przeglądarki użytkownik otrzyma komunikat w określonym języku.

2.3. Zmiana wymagań a re-konfiguracja systemu

W niniejszym rozdziale chcąc przedstawić możliwości aplikacji celowo nie unikałem wikłania się w aspekty konfiguracyjne. Nie musiałem ujawniać struktury klas i zawartego w nim kodu by pokazać jak dostosować program do własnych upodobań. Aplikacja może dysponować innym widokiem, generować inne odpowiedzi na reakcje użytkownika, posiadać nowe grupy ról itd.

Dzięki umiejętnemu rozdzieleniu warstw aplikacji stała się ona łatwiejsza w konserwacji. Chciało by się powiedzieć, nieco przewrotnie, że udało mi się nie zamknąć projektu: wprowadzanie kolejnych aktualizacji jest możliwe, co czyni program skalowalnym. To ważna cecha klient zawsze może bowiem zmienić swoje wymagania. Sama architektura aplikacji, podział odpowiedzialności poszczególnych warstw to kolejna ważna funkcjonalność.

3. Realizacja projektu

W tej części pracy postaram się opisać klasy *Javy* odpowiedzialne za kontrolę danych w aplikacji. Przedstawiłem etap przygotowań do budowania kodu aplikacji, zakładamy że zaczynamy dysponując bazą danych i wstępnie zaprojektowanym widokiem.

3.1. Obiektowy model danych – klasy encyjne

Klasy encyjne to „klasy lustrzane" do tabel w bazie. Każde pole tej klasy odpowiada kolumnie w konkretnej tabeli w bazie. Każda klasa z adnotacją *@Entity* to model obiektowy, operujemy na polach, odwzorowaniach kolumn będących obiektami. Dysponując plikami klas, możemy wygenerować strukturę bazy danych: każda jest „przepisem" na jedną tabelę.

Dużo wygodniej jest napisać klasę niż stworzyć tabelę przy pomocy SQL, klasy są bardziej elastyczne. Wszystko oczywiście musimy zaprogramować w postaci deklaratywnej, a więc określić identyfikator, nazwy i ograniczenia nałożone na kolumny, określić sposób generowania kluczy głównych itd. Pomijam tu ze względów objętościowych tematykę mapowania.

We własnych klasach przyjąłem zasadę co do adnotacji *@GeneratedValue* zawsze ustawiając jej właściwość na: *strategy = GenerationType.IDENTITY* czyli bez wykorzystania dodatkowych tabel.

Tabela *Workers* nie posiada żadnej relacji, a jednak manipulacja jej danymi jest możliwa. Dzieje się tak dzięki adnotacji *@SecondaryTables,* która „spina" ją z tabelą *Users*, jest to możliwe gdyż kolumny kluczy głównych tych tabel sobie odpowiadają. W modelu danych mamy sześć tabel, odpowiadających im klas encyjnych zaś pięć.

Opisałem już wcześniej, że wszystkie obiekty kolekcji z jakich korzystam spięte są relacjami. Wszystko widoczne jest w klasie Users:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "userId", fetch =
FetchType.EAGER)
    private List<Qualifications> qualificationsCollection = new
ArrayList<Qualifications>();
@OneToMany(cascade = CascadeType.ALL, mappedBy = "userId", fetch =
FetchType.EAGER)
    private List<HistoryWork> historyWorkCollection = new
ArrayList<HistoryWork>();
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "userId", fetch =
FetchType.EAGER)
    private List<Education> educationCollection = new ArrayList<Education>();
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "userId", fetch =
FetchType.EAGER)
```

```
private List<Trainings> trainingsCollection = new ArrayList<Trainings>();
```

Jak widać pola list tablicowych, zostają udekorowane dodatkowymi adnotacjami *@OneToMany* co oznacza, że aplikacja może zawierać wiele kolekcji określonego typu. Jednemu użytkownikowi przypiszemy np. wiele szkoleń. Rozwiązanie jest logiczne: jedna kolekcja to jeden zestaw pól o danym typie. Po przeciwnej stronie, czyli w klasie przechowującej pola kolekcji danego typu, znajdziemy:

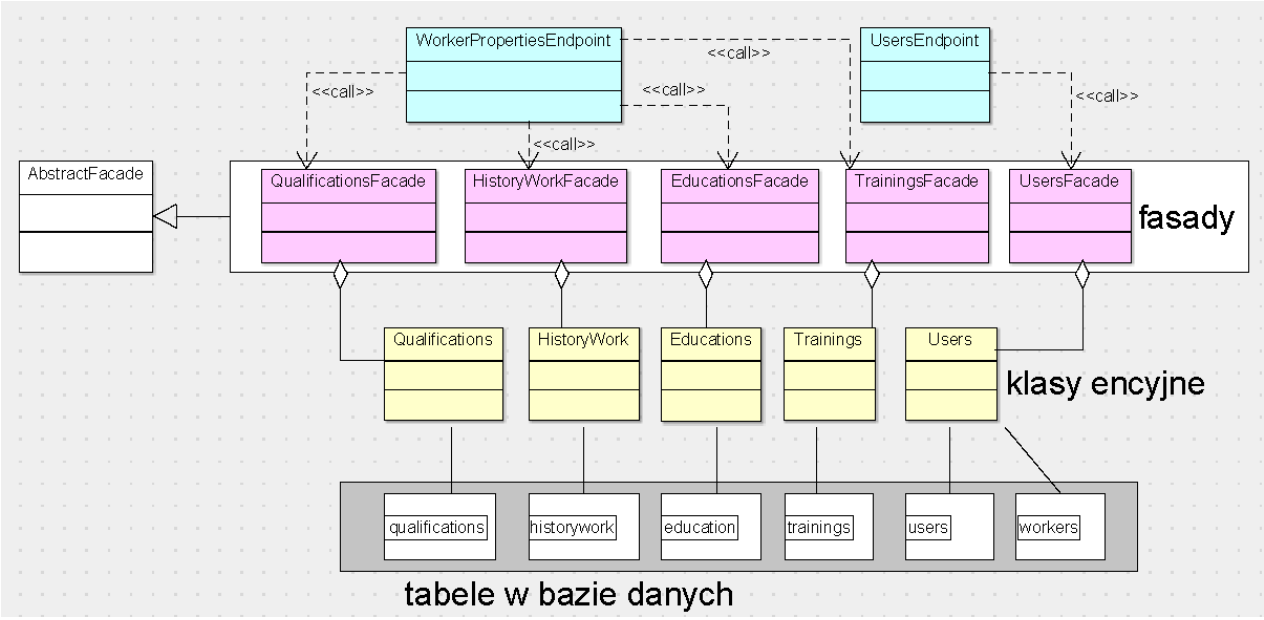
```
@JoinColumn(name = "user_id")
@ManyToOne(optional = false, fetch = FetchType.EAGER)
private Users userId;
```

Pole klucza głównego każdej z czterech tabel zostaje „połączone” z polem klucza głównym klasy encyjnej *Users*. Atrybut *name* w adnotacji *@JoinColumn* to nazwa kolumny z kluczami głównymi w tabeli Pracowników i Użytkowników (wspólna klasa encyjna).

W aplikacji wykorzystałem tworzone z generatora zapytania SQL *@NamedQueries* w klasach encyjnych. Skorzystałem z nich do budowy klas odpowiedzialnych za identyfikację użytkowników na poziomie aplikacji. Niemniej tworząc mechanizm wyszukiwania, użyłem *Criteria API.*

3.2. Logika aplikacji – podział zadań

a) Wzorzec fasada



Rys. 17 Uproszczony diagram klas.

Fasada w moim systemie pełni rolę „tamy”**[13]**, operacje wykonujemy na obiekcie

EntityMenager. Każda z klas encyjnych ma swojego zarządcę. Klasy fasad pozwalają na stworzenie „panelu operacyjnego”. Obiekt danego typu, odwołujący do konkretnej encji zyskuje w klasie fasady zestaw metod, które pozwalają na wyszukiwanie i zmianę jego stanu: *create(), edit(), remove(), find(),* itd. Budowa samych klas fasad jest bardzo prosta, mamy konstruktor i odwołanie do konkretnego kontekstu trwałości. Metody menagera encji zostają zdefiniowane w abstrakcyjnej klasie nadrzędnej o niezdefiniowanym typie.

```
@Stateless
public class UsersFacade extends AbstractFacade<Users>
{
    @PersistenceContext(unitName = "System.PU")
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager()
    {
        return em;
    }
    @Resource
    private SessionContext sctx;

    public UsersFacade()
    {
        super(Users.class);
    }
```

Fasada zastosowana przeze mnie w projekcie zwiększa przejrzystość kodu. W klasach tych stworzyłem m.in. metody *Criteria Api*[14] odpowiedzialne za wyszukiwanie. Każda z fasad o określonym typie, posiada metodę tworzącą zapytania SQL na podstawie wzorca:

```
    public List<Trainings> searchTrainingsProperties
(String contentTrainingPattern) {
        CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
        CriteriaQuery<Trainings> criteriaQuery =
criteriaBuilder.createQuery(Trainings.class);
        Root<Trainings> root = criteriaQuery.from(Trainings.class);
        criteriaQuery.select(root);

        Predicate predicate = criteriaBuilder.conjunction();
        if (null != contentTrainingPattern && !
(contentTrainingPattern.isEmpty() )) {
            predicate = criteriaBuilder.and(predicate,
criteriaBuilder.like(root.get(Trainings_.contentTraining),
contentTrainingPattern + '%'));
        }
        criteriaQuery = criteriaQuery.where(predicate);
        TypedQuery<Trainings> typedQuery = em.createQuery(criteriaQuery);
        return typedQuery.getResultList();
    }
```

Podział odpowiedzialności klas w aplikacji rysuje się jasno, operacje związane z manipulacją danymi: zapisem, odczytem, wyszukiwaniem, zostają zdefiniowane w warstwie fasad. Co łatwo zauważyć klasy encyjne odpowiadają tylko za strukturę danych.

b) Budowa aplikacji w oparciu o EJB

Metody *EntityMenagera* zostają określone w warstwie fasad, ale zaczynamy z nich korzystać dopiero w klasach endpoint. Endpointy specyfikują ogólne operacje zdefiniowane w fasadach, wprowadzają dla nich zadania charakterystyczne dla systemu. W zaprezentowanym poniżej przykładzie sposób obsługi i przechwycenia wyjątku zostaje określony w fasadzie metoda *edit()*, a kontekst działania metody *create()* zmienia się. Tworzymy użytkownika o ściśle określonych parametrach. Wykonane zostają te same metody ale wprowadzamy inne parametry, co uzasadnia ich ekspozycje w kolejnych warstwach aplikacji. Ta sama metoda w różnych klasach może służyć do czego innego.

klasa AbstractFacade

```
public void edit(T entity) throws AppException
    {
        getEntityManager().merge(entity);
    }
```

klasa UsersFacade

```
@Override
    public void edit(Users entity) throws AppException {
        try {
            super.edit(entity);
            em.flush();
        } catch (OptimisticLockException oe) {
            throw
RollBackOperationException.BuildExceptionOptimisticLock(entity, oe);
        }
    }
```

klasa UsersEndpoint

```
public void addUser(Users user) throws NoSuchAlgorithmException
    {
        Date currentTime = new Date();
        user.setActivationDate(currentTime);

        user.setRoleName("user_role");
        String passwordSHA26 = Sha.hash256(user.getPassword());
        user.setPassword(passwordSHA26);

        usersFacade.create(user);
    }
public void addUserBeforeEditing(Users addedUser) throws AppException
    {
        usersFacade.edit(addedUser);
    }
```

Wszystkie fasady to *komponenty bezstanowe*, których działanie jest realizowane w ramach technologii *Enterprise Java Beans***[15]**. Oznaczenie klas adnotacjami *@Stateless* zapewnia możliwość późniejszego ich wykorzystania. Warto dodać, że *komponenty bezstanowe* posiadają lepszą wydajność od stanowych, ponieważ serwer nie ma konieczności dbania o zachowanie ich stanu.

Wspomniane już klasy endpoint: czyli *UsersEndpoint* i *WorkerPropertiesEndpoint*, to klasy zdefiniowane jako *komponenty stanowe* zawierają one „wstrzyknięcia" - czyli zdeklarowane pola interfejsu komponentów. W dwóch wspomnianych klasach mamy także pola, z adnotacją @EJB, pochodzące z pakietu *app.authentication.beans - LoginEJB.*

Cechy charakterystyczne moich klas, to:

- **LoginEJB** – odpowiedzialny za identyfikację użytkowników oraz ich ról na poziomie systemu.
- **UsersEndpoint** – odpowiada za tworzenie, edytowanie i usuwanie użytkowników. Metody klasy odnoszą do obiektów typu Users, operujemy na pojedynczych użytkownikach.
- **WorkerPropertiesEndpoint –** odpowiada za realizację funkcjonalności związanej z dodawaniem i usuwaniem kolekcji danych biznesowych, operujemy na czterech typach danych (historia pracy, szkolenia, kwalifikacje, edukacja). Klasa nie zawiera metod umożliwiających edycję wstawionych rekordów, aplikacja nie przewiduje takiej funkcjonalności. Komponent zawiera wstrzyknięcia z fasad opisanych na Rys. 12. Dostęp do obiektów Users jest realizowany pośrednio poprzez LoginEJB

Przedstawione klasy endpoint „spinają" komponenty oznaczone jako *@Stateless.* Funkcjonalność fasad jak i klasy z pakietu *app.authentication.beans* jest realizowana w komponentach stanowych, utrzymywanych od początku do końca trwania sesji użytkownika.

3.3. Logika aplikacja – realizacja zadań

a) Klasy narzędziowe

Poprawne „złożenie" wszystkich zależności, czyli opisanie obsługi obiektów z klas encyjnych w kolejnych warstwach, pozwala na sprawne ich wykorzystanie podczas obsługi użytkowników.

Zdefiniowałem w aplikacji pakiet *app.web*, który odpowiada bezpośrednio za pobieranie i przekazanie danych, przesłanych przez korzystających z systemu, do „wyższych" warstw logiki. Metody endpointów zostały w wyeksponowane w klasach *app.web.users UsersController* (metody *UsersEndpoint*) i *app.web.workerproperties WorkerPropertiesController* (metody *WorkerPropertiesEndpoint).* Wspomniane klasy narzędziowe, wydawałoby się kolejne zbędne pliki obciążające aplikacje, pozwalają uprościć

tworzenie kolejnych klas odpowiedzialnych za łączenie z warstwą widoku. Nowością jaka pojawia się w kontrolerach jest metoda:

```
public String sendMessage(String resource, String key) {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        ResourceBundle resourceBundle = ResourceBundle.getBundle(resource);
        FacesMessage facesMessage = new
FacesMessage(resourceBundle.getString(key));
        facesContext.addMessage(null, facesMessage);
        return null;
    }
```

przydatna przy generowaniu komunikatów.

b) Obsługa danych z formularzy

Podczas budowy aplikacji starałem się nie zapominać o zasadzie „single responsibility principle". Do połączenia warstwy logiki z warstwą widoku służy aż 11 klas. Realizacja każdej funkcjonalności znajduje się w innej klasie. W pod-pakiecie *users* mamy osobne klasy służące do:

- wyświetlenia i wyszukiwania zwykłych użytkowników z listy,
- wyświetlenia i wyszukiwania administratorów z listy wraz ze zmianą ról,
- wyświetlania panelu tworzenia,
- wyświetlania strony zatwierdzenia
- wyświetlania panelu modyfikacji danych systemowych użytkownika

W pod-pakiecie *workerproperties* dysponujemy zaś klasami odpowiedzialnymi za dodawanie, usuwanie I wyszukiwanie danych osobno dla kolekcji poszczególnych typów:

- Education
- Qualification
- HistoryWork
- Trainings

- oraz klasę odpowiadającą za inicjację wyszukiwania dla wszystkich tabel z danymi biznesowymi jednocześnie

Taki podział odpowiedzialności w klasach nie musi odpowiadać każdemu, realizacja tej warstwy programu może wyglądać inaczej. Co warto jednak zauważyć zmiana kształtu aplikacji nie jest trudna właśnie dzięki umiejętnemu rozdzieleniu zadań w klasach w poprzednio opisanym etapie.

Klasy odpowiedzialne za nadzór nad stronami to „klasyczne ziarna JAVA EE" zawierają prywatne pola w postaci *String, Date, List*, służące do przechowywania tymczasowych

danych, oraz publiczne gettery i settery. Dostęp do pól z poziomu stron jest realizowany przy pomocy *Expression Language***[16]** przy wsparciu technologią *JSF*.

W kasach pośredniczących pomiędzy widokiem a logiką aplikacji zawarłem szereg dodatkowych metod odpowiedzialnych za wyświetlanie, pobieranie, weryfikację danych i przekierowania. Podaje przykłady. Poprawność haseł jest weryfikowana w widoku ale ich zgodność na poziomie beanów (hasło wymaga by wpisać je dwa razy). Podczas edycji danych program nie dopuści do wprowadzenia nieunikalnych danych, zamiast tego obsłuży wyjątek zwracając komunikat użytkownikowi:

```
public String modifyUser() throws NoSuchAlgorithmException, AppException {
   if (comparePassword()) {
       try {
             getUser().setPassword(Sha.hash256(newPassword));
             usersController.editUser();
             return "list";
           } catch (EJBException e) {
             usersController.sendMessage("i18n.messages", "not.unique");
             return null;
           }
        } else {
usersController.sendMessage("i18n.messages","passwords.not.matching");
return null;
        }
```

Osobno zwracam uwagę na metody udekorowane adnotacją *@PostConstruct* odpowiedzialne za generowanie poszczególnych na podstawie wzorców wyszukiwania podanych przez użytkowników. Deklaracja pozwala na wykonanie metod, po utworzeniu instancji komponentu:

```
private List<Trainings> listTrainings;
private DataModel<Trainings> myTrainingsCollection;
private String contentTrainingPattern = "";

@PostConstruct
private void initTrainingsList() {
listTrainings=
workerPropertiesController.searchTrainingsProperties(contentTrainingPattern);
myTrainingsCollection = new ListDataModel<>(listTrainings);
}

public DataModel<Trainings> getAllTrainingsList()
{
return (myTrainingsCollection = new
ListDataModel<>(workerPropertiesController.searchTrainingsProperties(contentTr
ainingPattern)));
}
```

Otrzymujemy rezultat wyszukiwania lub „nic” (fałsz) - czyli pełną listą użytkowników. Przypominam deklaracje metody wyszukującej w fasadzie:

```
if (null != contentTrainingPattern && !(contentTrainingPattern.isEmpty() )) {
      predicate = criteriaBuilder.and(predicate,
      criteriaBuilder.like(root.get(Trainings_.contentTraining),
      contentTrainingPattern + '%'));
}
```

Rozwiązanie to zapewnia spójność aplikacji, dane są wyświetlane poprawnie.

3.4. Instrukcja wdrożenia

Aby aplikacja poprawnie została wdrożona na lokalny serwer *GlassFish* musimy wykonać następujące czynności: skonfigurować połączenie z bazą danych, utworzyć odpowiednie struktury danych, zdefiniować przestrzeń uwierzytelnienia na *GlassFish* (Rys. 15) By zobaczyć program w akcji trzeba utworzyć, przy pomocy SQL, pierwsze konta z danymi inicjalnymi (dobrze by choć jedno konto pozostało w roli admin).

W mojej aplikacji gotowy plik *glassfish-resources* wygląda następująco (szczegóły opis tworzenia rozdziały 1.2 b.):

```
<resources>
  <jdbc-resource enabled="true" jndi-name="jdbc/system" object-type="user"
pool-name="Pool-System">
    <description/>
  </jdbc-resource>
  <jdbc-connection-pool ...>
    <property name="URL"
      value="jdbc:mysql://localhost:3306/system?
      zeroDateTimeBehavior=convertToNull"/>
    <property name="serverName" value="localhost"/>
    <property name="PortNumber" value="1527"/>
    <property name="DatabaseName" value="system"/>
    <property name="User" value="root"/>
    <property name="Password" value="admin"/>
  </jdbc-connection-pool>
</resources>
```

Hasło i nazwa użytkownika są u mnie definiują dostęp do wszystkich baz danych *MySQL*. Posługując się *XAMPP-em* należy, w razie edycji praw użytkowników, zmienić plik *config.inc.php* znajdujący się w *xampp/phpMyAdmin/*

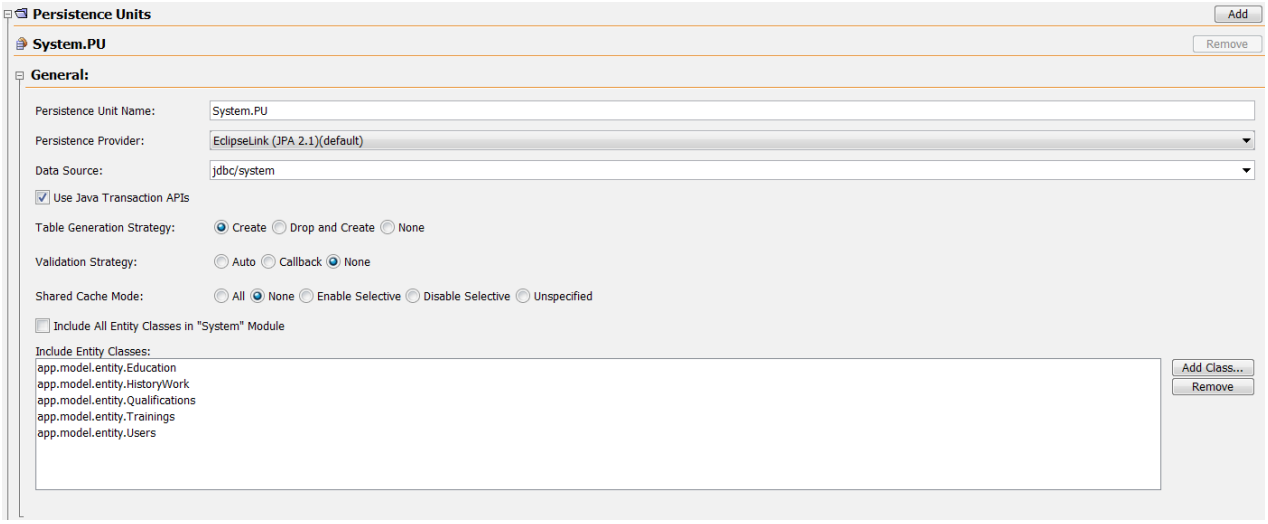
```
/* Authentication type and info */
$cfg['Servers'][$i]['auth_type'] = 'config';
$cfg['Servers'][$i]['user'] = 'root';
$cfg['Servers'][$i]['password'] = 'admin';
$cfg['Servers'][$i]['extension'] = 'mysqli';
$cfg['Servers'][$i]['AllowNoPassword'] = true;
$cfg['Lang'] = '';
```

W środowisku NetBeans też będzie trzeba wprowadzić te dane. Na podstawie zawartych w projekcie klas encyjnych istnieje możliwość wygenerowania tabel, jednak by wszystko przebiegło sprawnie warto ustalić odpowiedni sposób kodowania przed stworzeniem

struktury bazy, zapewni to możliwość obsługi polskich znaków. W pliku *my.ini (xampp/mysql/bin/)* należy wprowadzić następujące zmiany:

```
[mysqld]
character-set-server = utf8
collation-server = utf8_unicode_ci
[mysql]
default-character-set = utf8
[mysqldump]
default-character-set = utf8
```

By tabele mogły zostać poprawnie wygenerowane niezbędna jest zmiana w jednostce utrwalania *persistence.xml*



Rys. 18 - tryb edycji design pliku persistence.xml (NetBeans)

Zmiany dokonać należy w checkboxie *Table Generation Strategy: Create.* Opcja ta generuje wyłącznie strukturę danych, po utworzeniu szablonu danych dobrym zwyczajem jest ponowny wybór na None.

Końcowym etapem jest załadowanie przykładowych danych, umieściłem je w *System/src/main/java/app/model/sql/ExampleSQL.sql* (plik zawiera tylko operacje wstawiania danych, zadziała wyłącznie po poprawnym utworzeniu struktury danych). Chcąc wywołać plik należy go zaznaczyć i wcisnąć *Shift + F6.*

Hasła i loginy do kont są następujące:

login: pangeon11 - hasło: pangeon11
login: pangeon22 - hasło: pangeon22

Po wpisaniu danych w panelu logowania systemu użytkownik zostaje uwierzytelniony i zyska dostęp do wszystkich funkcjonalności.

Opisane czynności dotyczą środowiska *Windows*, na innych systemach zaprezentowane operacje mogą się różnić nieco od przedstawionych.

3.5. Podsumowanie

Udało mi zrealizować główne założenie projektu, aplikacja ma architekturę MVC zapewniającą jej łatwość w konserwacji i skalowalność. Komponenty mojego programu tworzą grupy, osobne segmenty. Każda z warstw aplikacji ma zdefiniowany zakres zadań. Starałem się trzymać zasady „single responsibility principle” - mówiącej, że nie powinno być więcej niż jednego powodu do edycji klasy.

Raport nie zawiera opisu klas odpowiedzialnych za wyjątki, gdyż nie zostały one w pełni wykorzystane w aplikacji. Kod programu posiada ogólne klasy wyjątków i interceptor – te fragmenty aplikacji wymagają jednak lepszego zespolenia z resztą logiki systemu. Widok zawiera tylko przykłady walidacji danych, ich realizacja nie jest jednak pełna, brak także internacjonalizacji dla całości aplikacji. Możliwość uzupełnienia pewnych luk została przewidziana, co sygnalizuje obecność plików właściwości.

4. Źródła

[1] - Włodzimierz Gajda – „**PHP Praktyczne Projekty**”, Gliwice, wyd. Helion, rok 2009, ISBN 978-83-246-0943-7

[2] - Balaji Varanasi , Sudha Belida – „**Introducing Maven**”, wyd. Apress, rok 2014, ISBN13: 978-1-484208-42-7 [ang.]

[3] Witryna IDE NetBeans: https://netbeans.org/downloads/ - „NetBeans 8.0.2 Java EE Installer for Windows/English (en)” (dostęp: 10.2015)

[4] Witryna The Apache DB Project: https://db.apache.org/derby/docs/10.4/devguide/cdevdvlp40653.html (dostęp 12.2015)

[5] Witryna MySQL: http://dev.mysql.com/downloads/connector/j/ (dostęp: 10.2015)

[6] Witryna Apachefriends: https://www.apachefriends.org/pl/index.html - „XAMPP for Windows 5.5.30 & 5.6.14” (dostęp: 10.2015)

[7], [8] - Bill Burke, Richard Monson-Haefel tłum. Mikołaj Szczepaniak, Krzysztof Ostrowski – "**Enterprise JavaBeans 3.0**", Gliwice, wyd. Helion, 2007, wydanie 5, ISBN 978-83-246-0726-6

[9] - Marcin Karbowski – „**Podstawy Kryptografii**”, wyd. Helion, rok 2007, wydanie 2, ISBN 978-83-246-1215-4

[10] - David R. Hoffelfinger, tłum. Tomasz Walczak – "**Java EE 6 Tworzenie aplikacji w NetBeans 7**", Gliwice, wyd. Helion, rok 2014, ISBN 978-83-246-8936-1

[11] Dokumentacja GlassFish: **https://glassfish.java.net/documentation.html** (dostęp: 12.2015)

[12] - Krzysztof Rychlicki-Kicior – "**Java EE6 Programowanie aplikacji WWW**", Gliwice, wyd. Helion, rok 2010, ISBN 978-83-246-2659-5

[13] - Robert A. Maksimchuk, Eric J. Nalburg, tłum. Andrzej Kosowski – **"UML dla zwykłych śmiertelników"**, Warszawa, wyd. Wydawnictwo Naukowe PWN, rok 2007, ISBN 978-83-01-15132-4

[14], [15], [16] - Arun Gupta - „**Java EE 7 Essentials**”, wyd O'Reilly, ISBN 978-1-449-37017-6 [ang.]