Politechnika Łódzka
Instytut Informatyki

RAPORT Z PROJEKTU KOŃCOWEGO

STUDIÓW PODYPLOMOWYCH

NOWOCZESNE APLIKACJE BIZNESOWE JAVA EE

APLIKACJA WEB UMOŻLIWIAJĄCA REJESTRACJĘ UCZESTNIKÓW BIEGÓW

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Opiekun: dr. inż Marcin Kwapisz
Słuchacz: inż. Piotr Biniek

Łódź, 09 października 2018 r.

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

Celem pracy jest zbudowanie systemu umożliwiającego utworzenie wydarzenia sportowego w postaci biegu, jak i rejestrację biegaczy do wybranego wydarzenia. System ma mieć możliwość rejestracji organizatorów i biegaczy oraz umożliwiać rejestrację biegaczy do wybranego biegu.

System będzie aplikacją wykonaną w technologii Java Enterprise Edition. Przechowywanie danych zaplanowano w bazie danych , a dostęp do niej zapewniony jest z poziomu przeglądarki internetowej.

System ma być aplikacją biznesową, w której interfejs użytkownika to dynamicznie generowane strony www, a dane systemowe mają być przechowywane w relacyjnej bazie danych. Aplikacja zostanie wykonana przy pomocy technologi Java Enterprise Edition, która jest platformą programistyczną języka Java. Środowiskiem programistycznym użytym w celu stworzenia systemu będzie program Netbeans IDE 8.2, a środowiskiem uruchomieniowym serwer Glassfish 4.1. Główne funkcjonalności systemu :

Zakres projektu obejmuje:

• opracowanie wymagań biznesowych dotyczących tworzonego systemu

• dobór technologii do wykonania systemu,

• zaprojektowanie architektury tworzonego systemu

• zaprojektowanie modelu danych aplikacji

• implementację systemu

• stworzenie instrukcji wdrożenia systemu

• podsumowanie

2 Założenia projektu

W rozdziale tym opisano, zastosowane narzędzia

2.1 Wersje zastosowanych technologii i narzędzi

Zastosowane technologie i narzędzia podczas realizacji projektu:

- Środowisko programistyczne NetBeans IDE w wersji 8.2

- Język programowania Java 8

- JDK wersji 1.8 update 151

- Maven w wersji 3.3.9 używane do budowania i zarządzania projektem

- Serwer aplikacyjny to Glassfish w wersji 4.1.

- Java Enterprise Edition 7.0 - platforma programistyczna umożliwiająca tworzenie aplikacji w architekturze kontener-komponent w której zrealizowano logikę biznesową

- Java Server Faces w wersji 2.2 obsługuje warstwę widoku. Dodatkowo w aplikacji wykorzystano elementy publicznej biblioteki Java PrimeFaces w wersji 6.2

- Java DB (Derby) odpowiedzialna za składowanie danych

- obiekty mapowane są z wykorzystaniem standardu Java Persistance API w wersji 2.1.

2.2 Wymagania funkcjonalne

W projekcie występują 3 poziomy dostępu:

- użytkownik niezalogowany (tzw. gość).

- biegacz

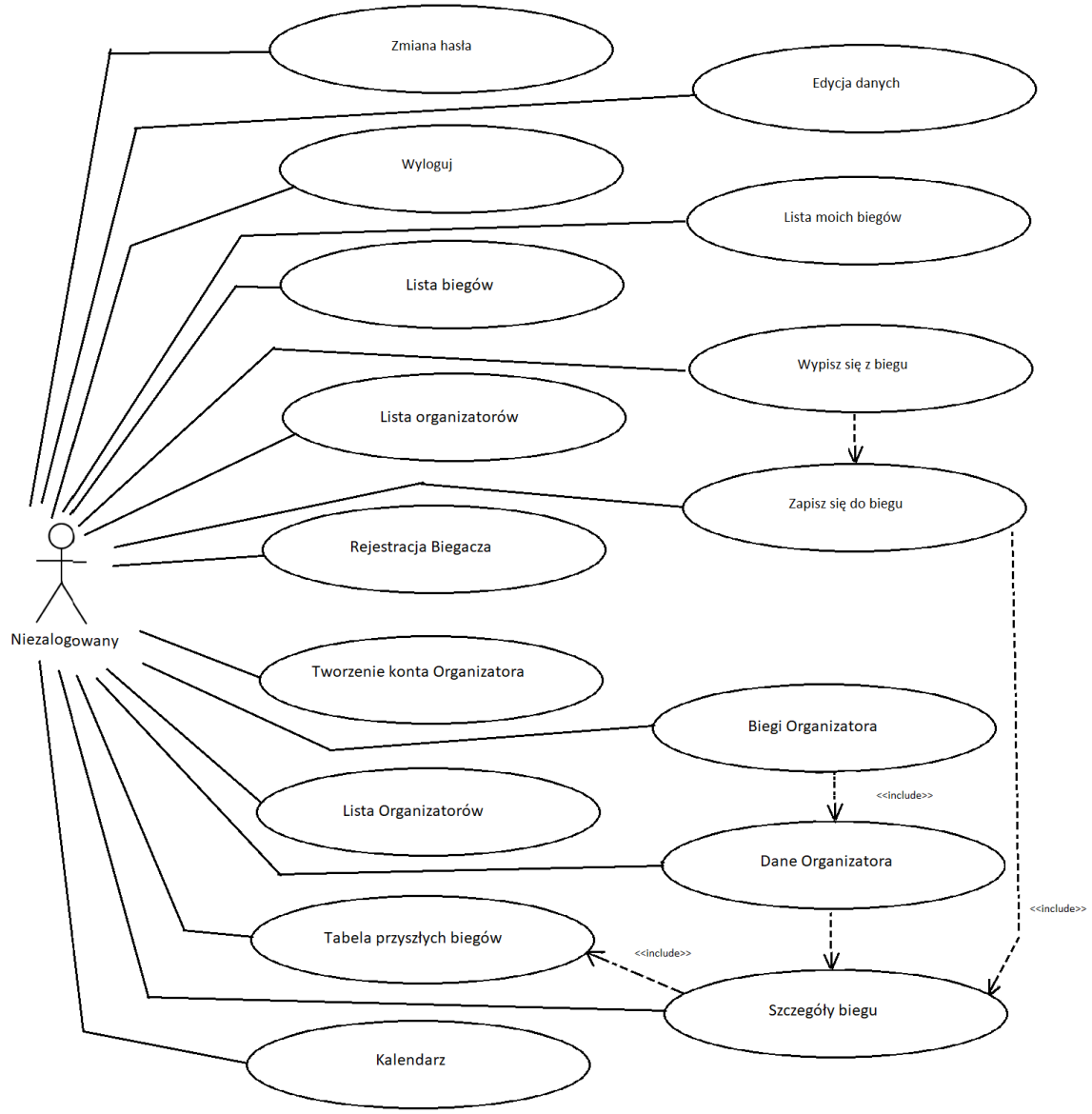
- organizator

- administrator

W założeniach projektowych przyjęto że użytkownik może posiadać tylko jedną rolę.

Możliwe przypadki użycia dla Biegacza i Użytkownika niezalogowanego pokazano na poniższych diagramach:

Rysunek 1. Diagram przypadków użycia dla niezalogowanego użytkownika.



Rysunek 2. Diagram przypadków użycia dla Biegacza.

2.3 Wymagania niefunkcjonalne

Dostęp do części funkcjonalności będzie osiągalny w przypadku uwierzytelnienia się poprzez podanie loginu i hasła.

System ma pozwalać na wylogowanie się uwierzytelnionego użytkownika.

System gromadzi logi systemowe.

Interfejs użytkownika musi być intuicyjny i oparty o dynamicznie generowane strony WWW.

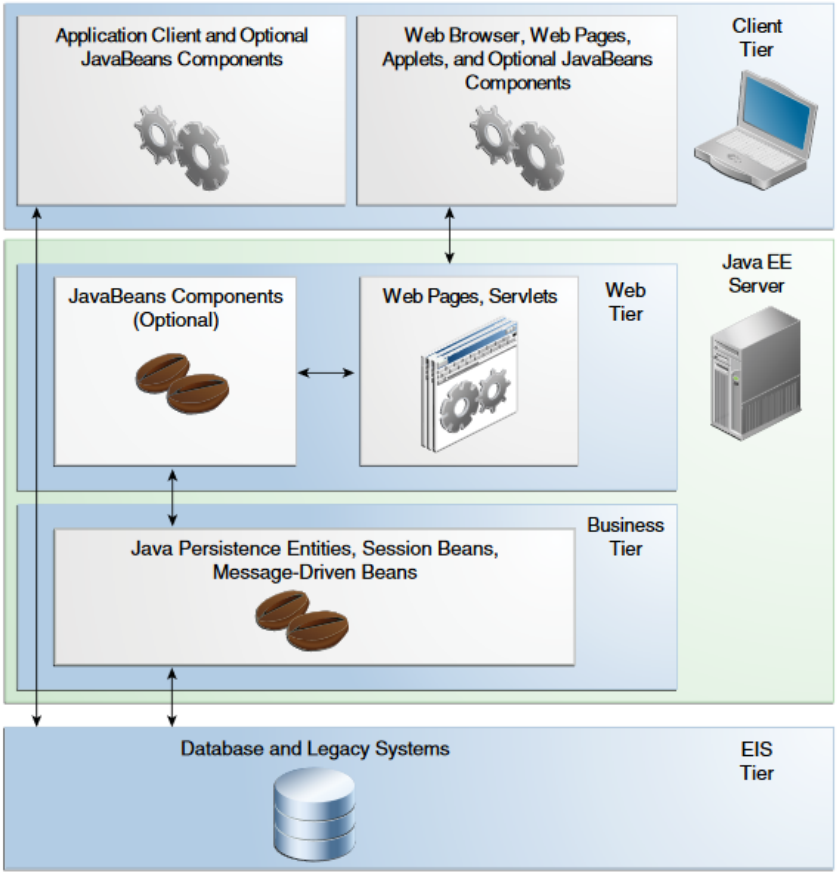
Do komunikacji zostanie wykorzystany protokół https.

System będzie wielodostępny.

System ma zapewnić bezpieczeństwo danych.

3 Realizacja projektu

Projekt został zrealizowany w oparciu o technologię Java EE 7 z podziałem na warstwy prezentacji, logiki biznesowej i składowania danych. Wygląda to następująco:



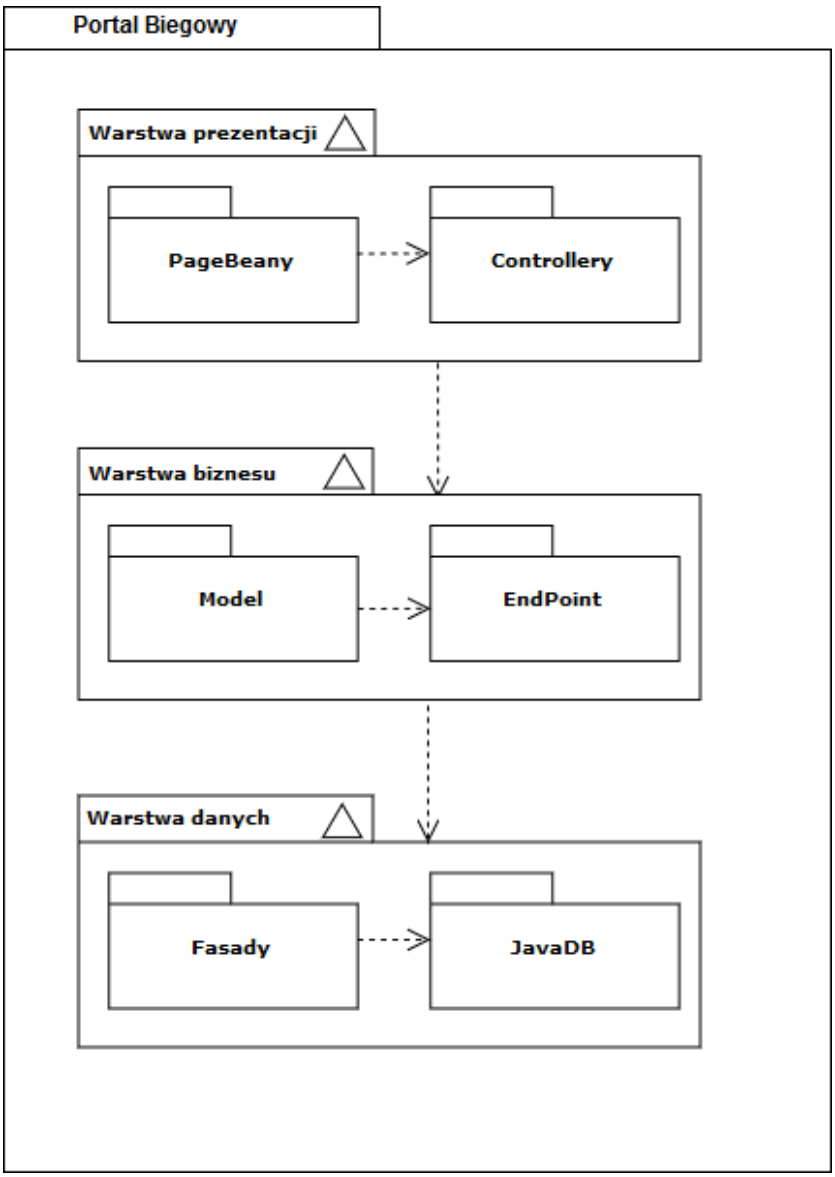
Rysunek 3. Podział na warstwy. Źródło: [1]

- warstwa prezentacji, działająca na serwerze GlassFish – strony internetowe oparte na technologii Java Server Faces z wykorzystaniem bibliotek PrimeFaces 6.2. oraz ziaren CDI. Warstwa ta umożliwia komunikację użytkownika z aplikacją. Dzięki niej możliwe jest przekazanie danych wprowadzonych przez użytkownika do dalszego przetworzenia w warstwie logiki biznesowej oraz odpowiednia nawigacja w aplikacji

- warstwa logiki biznesowej, działająca również na serwerze GlassFish – komponenty EJB stanowe i bezstanowe zawierające metody odpowiedzialne za powiązanie

rzeczywistych zdarzeń w systemie tj, utworzenie obiektów, zmianę ich stanu, ich utrwalenie, jak i sprawdzenie poprawności. W tym miejscu są zlokalizowane metody pobierające obiekty z bazy danych oraz je utrwalające

- warstwa składowania danych – relacyjna baza danych Java DB (Apache Derby) pozwalająca na składowanie danych w tabelach wraz z API czyli fasadami będącymi pośrednikami w zapisie danych.



Rysunek 4. Diagram UML obrazujący poszczególne warstwy modelu

3.1 Model danych

W skład modelu danych wchodzą przede wszystkim:

- klasy modelu których zadaniem jest odzwierciedlenie rzeczywistych obiektów i odwzorowanie ich zachowań,

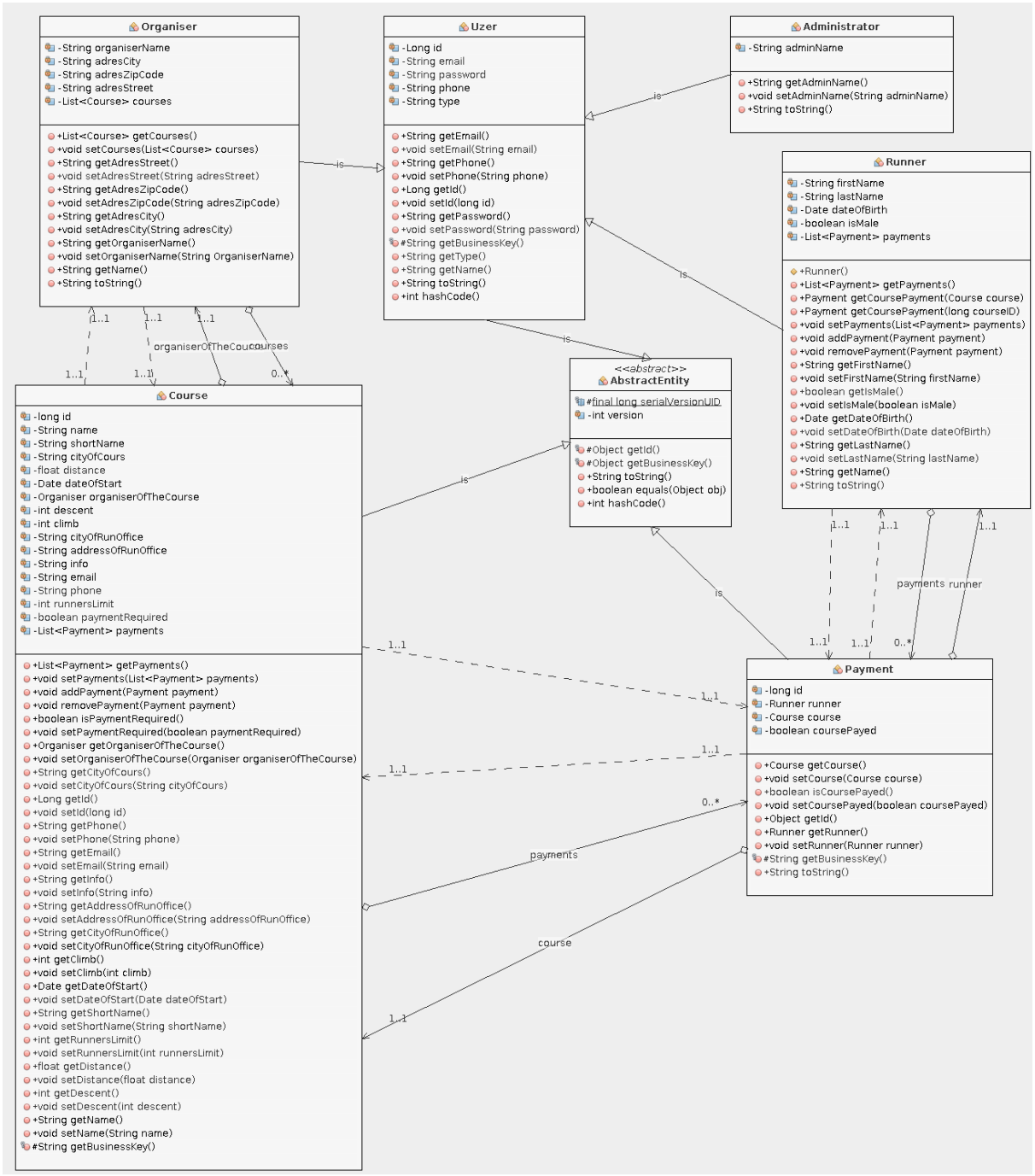
- klasy wchodzące w skład tzw EndPointów tj. obiektów, które są odpowiedzialne za operowanie na instancjach (rzeczywistych obiektach w systemie), a także są punktem styku pomiędzy modelem, a warstwą prezentacji oraz pośredniczą w komunikacji z warstwą danych.

*

3.1.1 Klasy modelu

Klasy modelu to klasy encyjne tj. takie, których instancja może być utrwalona w bazie danych. Mapowanie obiektowo-relacyjne zostało zaimplementowane z użyciem standardu JPA. W tym celu klasy posiadają adnotację @Entity określającą klasę jako klasę encyjną oraz adnotację @TableGenerator, definiujący generator klucza głównego dla poszczególnych rekordów tabeli, na które zamieniana jest encja.

Utrwalane klasy zawierają również adnotację @NamedQueries, która definiuje kwerendy służące pobieraniu z bazy danych oczekiwanych informacji. Samo JPA nie wymaga definiowania kwerend, gdyż może się posługiwać wyłącznie obiektami, to jednak kwerendy są wydajniejsze, niż przeglądanie kolekcji obiektów. Wybrane klasy są połączone ze sobą relacją dziedziczenia, extends lub relacjami opisanymi adnotacjami @ManyToOne w zależności od relacji między nimi. Poniżej diagram UML obiektów modelu oraz fragment kodu przykładowej klasy encyjnej:



Rysunek 5. Diagram klas modelu.

Tabela 1.Klasy encyjne w projekcie

Nazwa Encji	**Opis encji i rola**
Abstract Entity	Deklaruje ważniejsze metody abstrakcyjne, oraz pole wersji dla blokad optymistycznych, dziedziczą z niego wszystkie inne encje
Uzer	Jest obiektem definiującym podstawowe elementy każdego każdego użytkownika, takie jak mail, hasło, telefon i pole ID wymagane przez bazę danych.
Runner	Biegacz jest, osobą fizyczną, posiada imię i nazwisko i datę urodzenia, Biegacz może się zapisać do biegu.
Organiser	Organizator osoba prawna lub inny podmiot nie posiadający osobowości prawnej, a organizujący bieg.. Poza nazwą posiada również siedzibę. Tylko organizator może utworzyć bieg.
Administrator	Administrator – zarządca bazy może usunąć lub edytować dane organizatora, biegacza i biegu
Course	Bieg – wydarzenie sportowe, posiadające określone parametry, jak nazwa, dystans, liczbę uczestników itp.. Posiada pole Organizatora który go utworzył oraz listę „płatności”
Payment	Płatność jest obiektem łączącym wzajemnie bieg i biegacza, posiada dodatkowe pole określające czy dokonano płatności

```
@Entity
//opis oznaczający że jest to encja
@Table(name = "Course")
// określenie jak się ma nazywać tabela w bazie danych składująca obiekty
tej encji
@NamedQueries({
    @NamedQuery(name = "Course.findByName", query = "SELECT d FROM Course d
      WHERE d.name = :name"),
    @NamedQuery(name = "Course.findByShortName", query = "SELECT d FROM
      Course d WHERE d.shortName = :shortName"),
    @NamedQuery(name = "Course.findById", query = "SELECT d FROM Course d
      WHERE d.id = :id"),})


public class Course extends AbstractEntity implements Serializable {

    @Id
    @Column(name = "id", updatable = false)
    @TableGenerator(name = "CourseGen", table = "GENERATOR", pkColumnName =
"ENTITY_NAME", valueColumnName = "ID_RANGE", pkColumnValue = "Course",
initialValue = 7919, allocationSize = 7919)
    @GeneratedValue(strategy = GenerationType.TABLE,
            generator = "CourseGen")
    private long id;

    @NotNull
//@NotNull – używana przy konstrukcji tabeli w BD, oznaczająca że to pole
nie może być puste
    @Column(name = "name", unique = true, nullable = false, length = 256)
    private String name;
//ta adnotacja pozwala doprecyzować zarówno nazwę kolumny, jak i wprowadzić
ograniczenia nakładane na kolumny z danymi
    @NotNull
```

```
    @Column(name = "shortName", unique = false, nullable = false, length =
      20)
    private String shortName;

(…)
    @NotNull
    @Column(name = "dateOfStart", unique = false, nullable = false)
    @Temporal(javax.persistence.TemporalType.TIMESTAMP)
//mapowanie oznaczające pole czasowe
    private Date dateOfStart;



    @NotNull
    @JoinColumn(name = "ID_ORGANISER", referencedColumnName = "ID",
      nullable = false)
//oznaczenie łaczenia pola o nazwie ID_ORGANIZER z polem ID z tabeli
organizatorów (które jest tutaj polem,
    @ManyToOne
//łączenie wiele do jednego z organizatorem, po stronie omawianej klasy
jest jeden
    private Organiser organiserOfTheCourse = null;

    @Column(unique = false, nullable = true)
    private int descent;

    @Column(unique = false, nullable = true)
    private int climb;
      (...)
    @Column(unique = false, nullable = true)
    @Pattern(regexp = "^[_a-z0-9-]+(\\.[_a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-
      9-]+)*(\\.[a-z]{2,4})$", message =
      "{constraint.string.incorrectemail}")
// nakłada ograniczenie określone w wyrażeniu regularnym na pole tą
adnotacją  i słuzy do walidacji Beana, jeżeli warunek nie będzie spełniony
to pokazywany jest komunikat o treści/mapowaniu pokazanym w komentarzu.
    private String email;

      (...)
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "course")
 //oznaczenie relacji jeden do wielu, po stronie klasy jest wiele, w której
wszystkie operacje są  kaskadowane do obiektu powiązanego, jeżeli usuniemy
ten obiekt to obiekt powiązany też usuniemy
    private List<Payment> payments = new ArrayList<>();
```

Listing 1: Fragment klasy encyjnej Course bez metod i z pominięciem części pól i metod znajdującej się w pakiecie pl.java.biniek.model. Opis ważniejszych elementów czcionką italic w treści kodu po znakach //

W kodzie źródłowym innych encji znajdują się też dodatkowe adnotacje w postaci:

@MappedSuperclass – pozwala na dziedziczenie istotnych elementów z tej klasy, ale sama klasa nie ma swojego odwzorowania w bazie danych wykorzystano w głównej encji abstrakcyjnej z której dziedziczą wszystkie pozostałe obiekty.

@Version – pole wykorzystywane przy blokadach optymistycznych żeby stwierdzić czy dany obiekt nie został zmieniony między odczytem, a zapisem zmiany, wykorzystane we wszystkich encjach

@DiscriminatorValue("ADMIN") informacja dla JPA mówiąca jak mają być mapowane dziedziczenie w tabeli DB – w tym wypadku w polu oznaczającym Administratora, te adnotacje znalazły się w klasach Uzer, Administrator i Runner

@Temporal(javax.persistence.TemporalType.TIMESTAMP) mapowanie oznaczające pole czasowe

3.1.2 Klasy Endpointów

Endpointy dają dostęp warstwie widoku pośrednicząc w wymianie danych oraz łączą się z warstwą składowania danych bazodanową. W aplikacji wykorzystano trzy endpointy , co było możliwe dzięki skorzystaniu z polimorfizmu obiektów klasy Uzer i dziedziczących z niego klas Runner, Organiser i Administrator.

Poniżej zaprezentowano jeden z endopointów wraz z opisem funkcjonalności.

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
@Interceptors(LoggingInterceptorWithRepackingForEndPoint.class)public class
PaymentEndPoint implements Serializable {

    @EJB
    private PaymentFacade paymentFacade;

    @RolesAllowed({"Runner"})
    public void remove(Payment payment) throws BasicApplicationException {
        Date now = new Date();
        if (payment.getCourse().getDateOfStart().after(now)) {
            paymentFacade.remove(payment);
        } else {
            throw new AftertTimeException();
        }
    }

     @RolesAllowed({"Runner"})
    public synchronized void createPayment(Payment payment) throws
BasicApplicationException {
        Date now = new Date();
        if (payment.getCourse().getDateOfStart().after(now)) {
            paymentFacade.create(payment);
        } else {
            throw new AftertTimeException();
        }
    }

    @RolesAllowed({"Runner"})
    public void saveAfterEdit(Payment payment) throws
BasicApplicationException {
        Date now = new Date();
        if (payment.getCourse().getDateOfStart().after(now)) {
```

```
            paymentFacade.edit(payment);
        } else {
            throw new AftertTimeException();
        }
    }

...
```

Listing 2: Fragment klasy PaymentEndPoint z pominięciem części metod - pakiet pl.java.biniek.endpoints

Adnotacja @Stateless służy do oznaczania bezstanowych komponentów EJB. Komponenty takie nie posiadają stanu, w związku z czym mogą być bezproblemowo wielokrotnie wykorzystywane przez kontener.

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW) oznacza wymóg rozpoczęcia nowej transakcji, wszystkie wydarzenia po jej rozpoczęciu są rejestrowane w wypadku niepowodzenia takiej transakcji, wszystkie zmiany dokonane w obiektach zostaną cofnięte. Więcej w rozdziale 3.1.3.

@Interceptors(LoggingInterceptorWithRepackingForEndPoint..class) określa punkt wstrzyknięcia interoceptora, o którym będzie szerzej w rozdziale 3.4

@EJB – są to punkty wstrzyknięcia komponentów innych komponentów EJB w tym konkretnym wypadku fasady dostępu do bazy danych.

@RolesAllowed – jest informacją dla kontenera w jakiej roli musi być użytkownik „zalogowany” aby móc wykonać daną metodę, w przypadku nie posiadania tej określonej roli zwracany jest wyjątek javax.ejb.EJBAccessException.

3.1.3 Transakcje

W implementowanym systemie zastosowano transakcyjność. Transakcja to grupa zależnych od siebie operacji, które muszą zostać ukończone razem. Mechanizm ten ma za zadanie spełnić następujące warunki: atomowości, spójności, izolacji i trwałości.

W systemie użyto domyślnego mechanizmu transakcji CMT (ang. Container Management Transactions). Aby spełnić warunki transakcyjności w aplikacji wprowadzono adnotacje @TransactionAttribute na klasie komponentu z odpowiednim typem tych atrybutów:

- MANDATORY - dla tego atrybutu każde wywołanie metody wymaga aby metoda biznesowa komponentu EJB była realizowana w już istniejącej transakcji, ten atrybut wykorzystano w Fasadach

- REQUIRES_NEW – kontener w przypadku tak oznaczonej metody zawsze tworzy nową transakcję. Ewentualne trwające transakcje są zawieszane do czasu zakończenia

nowej. Ten atrybut wykorzystano w Endpointach, tak aby w metodach biznesowych zawsze zachodziła transakcyjność.

Inne możliwe poziomy transakcji to:

- REQUIRED – metoda zostanie zawsze wykonana w ramach transakcji. Jeżeli przychodzące żądanie nie niesie ze sobą transakcji wówczas tworzona jest nowa, a gdy transakcja istnieje, wykonanie metody następuje w jej ramach

- SUPPORTS – w przypadku gdy istnieje transakcja, wykonanie metody odbędzie się w jej kontekście, w przypadku braku transakcji metoda wykona się bez niej

- NOT_SUPPORTED – transakcja wykonuje się bez transakcji, Jeżeli transakcja istnieje to zostaje wstrzymana

- NEVER – podobnie jak NOT_SUPPORTED wykonuje się bez transakcji, ale w przypadku jej istnienia zgłaszany jest wyjątek.

Poziom izolacji transakcji bazodanowych został ustalony na READ COMMITED.

3.2 Warstwa składowania danych

Za składowanie danych odpowiada JavaDB oraz fasady pośredniczące w zapisie danych. Do wykonania modelu bazy i bazowych wersji fasad wykorzystano JavaPersistence API (Odwzorowuje ono architekturę systemu informatycznego w bazie danych tworząc tabele powiązane relacjami).

3.2.1 Fasady

Do pobierania i wprowadzania danych do bazy wykorzystano wzorzec fasady. Są to bezstanowe komponenty EJB, które pośredniczą w transakcjach bazodanowych ułatwiając i upraszczając komunikację z nią.

W aplikacji dla każdej klasy encyjnej stworzono osobną fasadę odpowiedzialną za wykonywanie zbioru podstawowych operacji CRUD (Create, Remove, Update, Delete). Każda z encji dziedziczy z abstrakcyjnej klasy fasady, która dostarcza podstawowe metody operacji na bazie. W tworzeniu fasad skorzystano z automatyki oferowanej przez środowisko programistyczne NetBeans, umożliwiającej tworzenie Fasad na podstawie encyjnych klas modelu. Dodatkowo w poszczególnych fasadach zdefiniowano dodatkowe metody wykorzystujące kwerendy JPQL.

```
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public abstract class AbstractFacade<T> {

    private Class<T> entityClass;
```

```
    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    protected abstract EntityManager getEntityManager();

    public void create(T entity) throws BasicApplicationException {
        getEntityManager().persist(entity);
         getEntityManager().flush();

    }

    public void edit(T entity) {
        getEntityManager().merge(entity);
         getEntityManager().flush();
    }

    public void remove(T entity)  {
        getEntityManager().remove(getEntityManager().merge(entity));
        getEntityManager().flush();
    }

    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }

    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }

    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0] + 1);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }

    public int count() {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
        cq.select(getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }

 }
```

Listing 3: Fragment klasy AbstractFacade z której dziedziczą wszystkie fasady - pakiet pl.java.biniek.facades

Wraz z generowaniem klas fasad został wygenerowany został deskryptor składowania persistence.xml, który definiuje parametry połączenia dla operacji bazodanowych.

3.3 Warstwa prezentacji

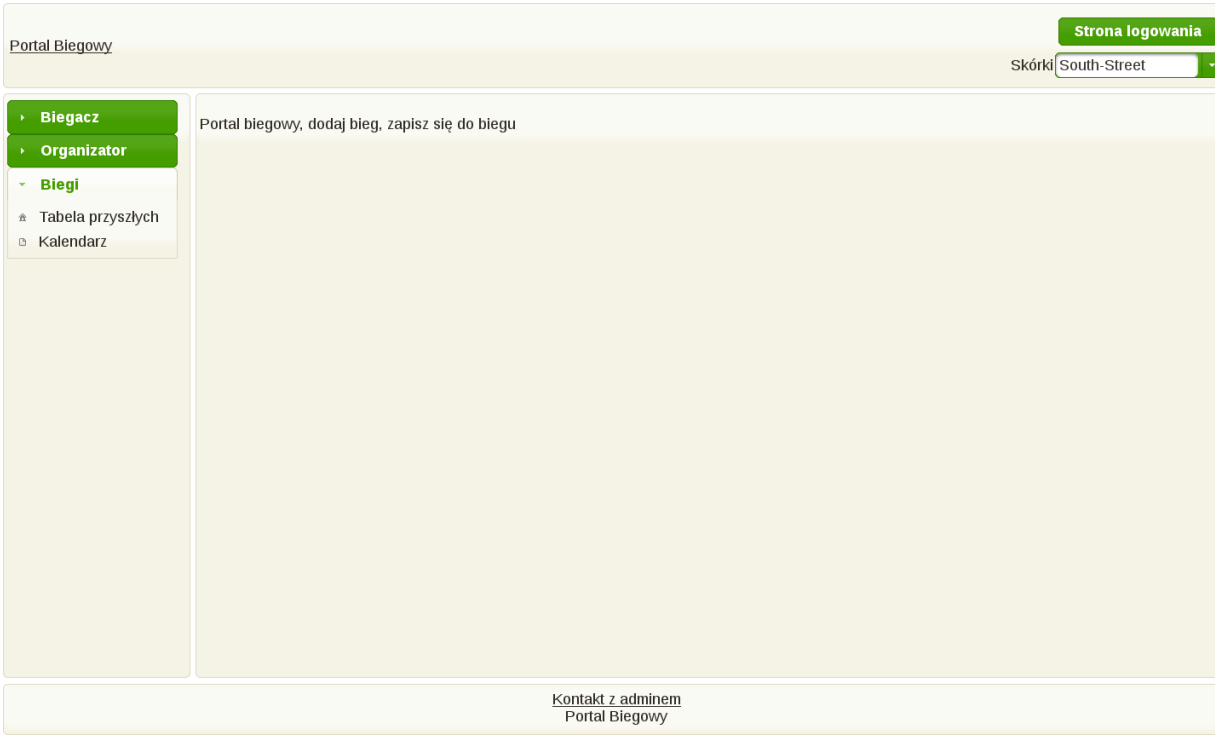
Warstwa prezentacji zbudowana jest w oparciu o ziarna CDI, framework JSF z wykorzystaniem elementów PrimeFaces.

Elementem zarządzającym widokiem są kontrolery które są ziarnami Javy, odpowiedzialne za przechowywanie pól oraz metod obsługujących akcje użytkownika. Widok został zbudowany z wykorzystaniem zestawu bibliotek elementów JSF. Kontrolerem jest servlet FacesServlet, którego ustawienia zdefiniowane zostały w pliku konfiguracyjnym faces-config.xml. Dodatkowo w projekcie użyto komponenty biblioteki Prime Faces.

Elementy, które mają być wyświetlone użytkownikowi są przechowywane w plikach xhtml i w przypadku otrzymania żądania od przeglądarki są renderowane na bieżąco. W powiązaniu z odpowiednimi ziarnami pozwalają na dynamiczne generowanie strony.

3.3.1 JSF i Prime Faces

Widok aplikacji został zaimplementowany z wykorzystaniem frameworku JSF.[2] „Java Server Faces (JSF) – framework, bazujący na języku Java, który upraszcza tworzenie interfejsu użytkownika do aplikacji Java EE”.[7]. Przy konstrukcji strony wykorzystano elementy biblioteki PrimeFeces w wersji 6.2.[1][5]



Rysunek 6. Wygląd aplikacji dla niezalogowanego użytkownika.

PrimeFaces to Open Sourcowe biblioteki, które ułatwiają developerowi przygotowanie projektu strony. Pozwalają one na skorzystanie zarówno z pojedynczych elementów np. masek na polach wejścia

```
  <p:inputMask value="#{organiserWizard.organiser.adresZipCode}" mask="99-
999" />
```

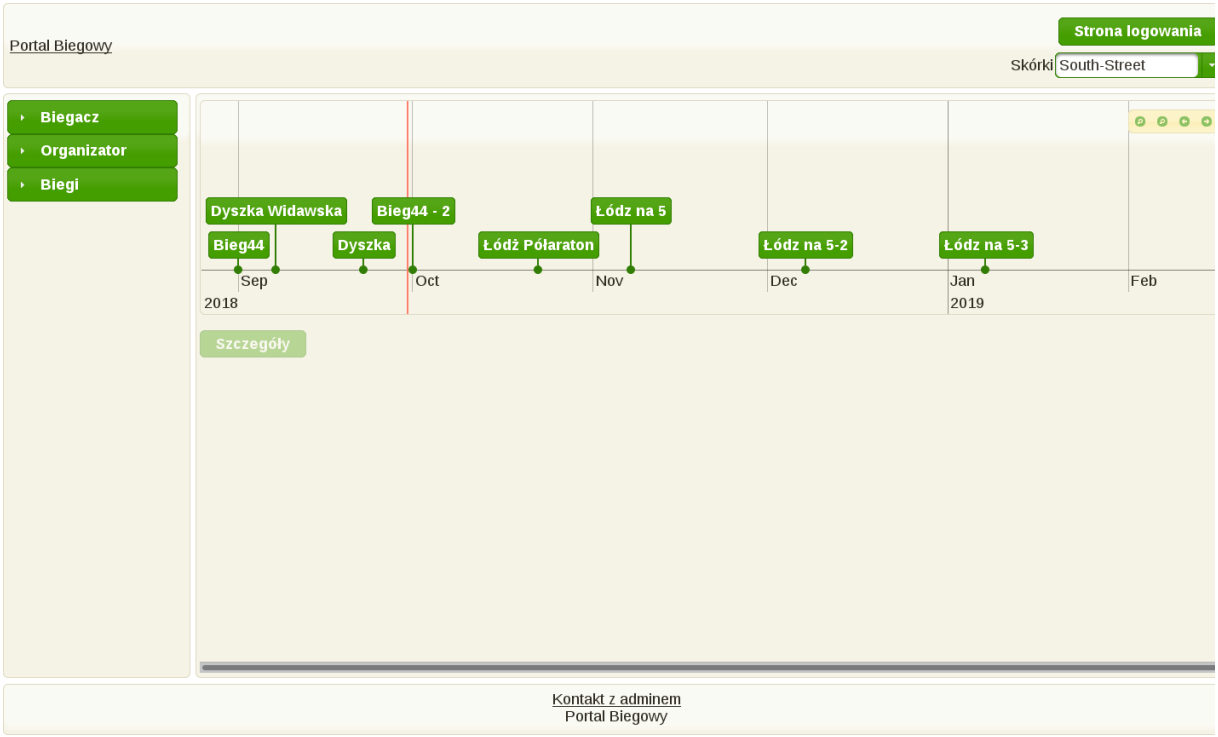
Listing 4: Maska na wejściu.

filtrów na pola tekstowe ograniczających możliwość wprowadzenia błędnych danych:

```
<p:inputText value="#{courseWizard.course.phone}">
   <p:keyFilter regEx="/[0-9-]/i"/>
</p:inputText>
```

Listing 5: Filtr klawiszy umożliwiający wprowadzenie w pole tekstowe jedynie cyfr, lub znaku '-'.

Prime Faces oferuje również kompletne elementy, jak kalendarze, elementy strony, formatki itp. Poniżej zaprezentowano elementy strony z wykorzystaniem elementu TimeLine oraz jego implementację po stronie xhtml.



Rysunek 7. Element TimeLine.

```
<p:timeline id="timeline" value="#{courseTimelineView.model}"
height="250px"
        selectable="#{courseTimelineView.selectable}"
        zoomable="#{courseTimelineView.zoomable}"
        moveable="#{courseTimelineView.moveable}"
        stackEvents="#{courseTimelineView.stackEvents}"
        axisOnTop="#{courseTimelineView.axisOnTop}"
        eventStyle="#{courseTimelineView.eventStyle}"
        showCurrentTime="#{courseTimelineView.showCurrentTime}"
        showNavigation="#{courseTimelineView.showNavigation}"
        start="#{courseTimelineView.date}"
        zoomMin="2000000000"
        animate="true">
        <p:ajax event="select" listener="#{courseTimelineView.onSelect}"
              update="info buttonDetails">
        </p:ajax>
</p:timeline>
```

Listing 6: Implementacja kalendarza typu TimeLine

3.3.2 Warstwa obsługi żądań – ziarna CDI.

Warstwę obsługi żądań zbudowano z wykorzystaniem ziaren CDI. Strony renderowane dla użytkownika oparte są na ziarnach o zasięgu widoku. Przyczyną dokonania takiego wyboru był fakt założenia, że w aplikacji ogólnodostępnej może zaistnieć przypadek używania aplikacji na wielu kartach tej samej przeglądarki. W przypadku ziaren o zasięgu żądania w trakcie testów obsługi zdarzały się przypadki, gdy strona zmieniała się niezgodnie z oczekiwaniami (np. przejście z zakładki widoku do edycji przez

administratora). Dla każdej strony zaimplementowano po jednym Ziarnie obsługującym go.

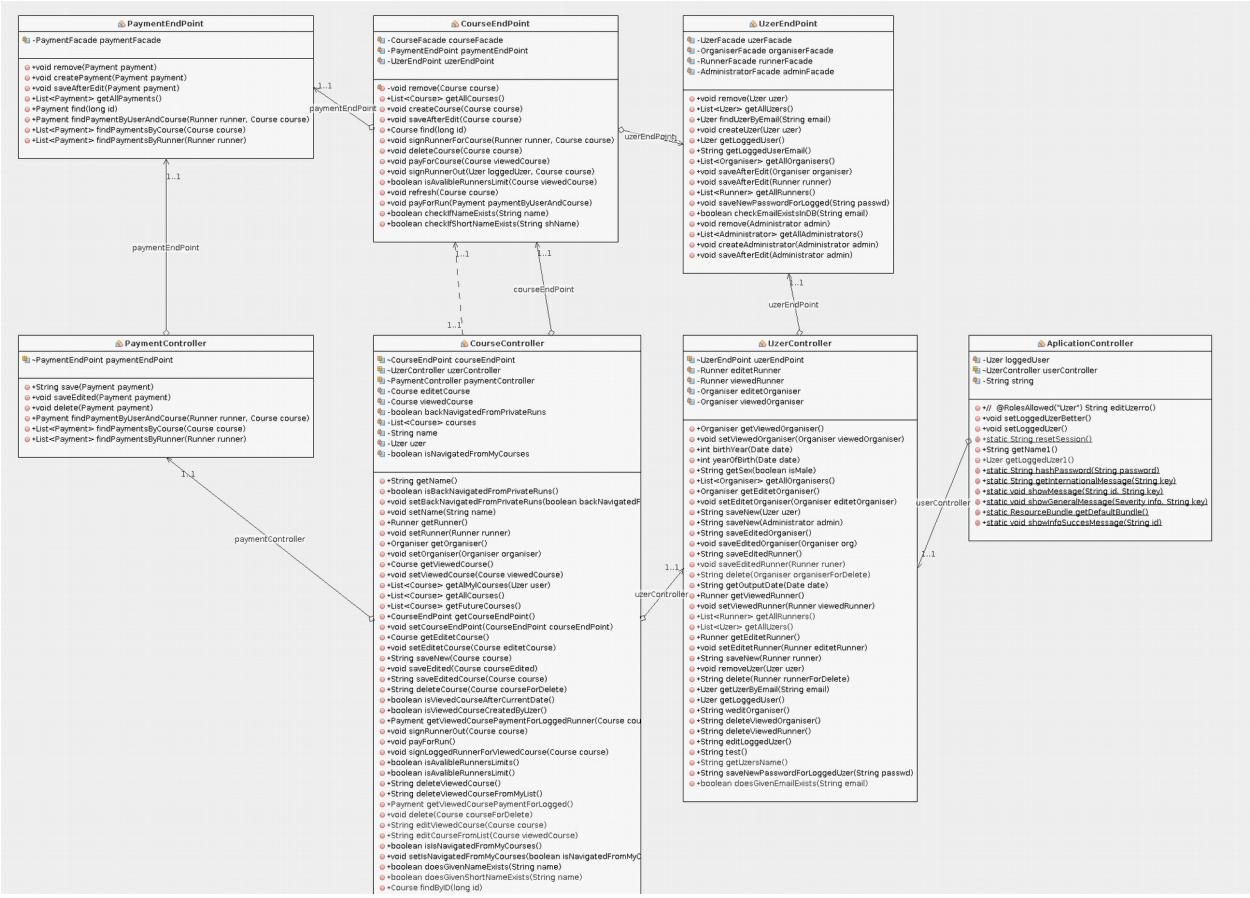
Część funkcjonalności systemu wymagała, aby dane zaprezentowane użytkownikowi były dostępne dla kolejnych żądań, dlatego też zdecydowano się na implementację 2 warstwy ziaren o zasięgu sesyjnym, zwanych kontrolerami, która odpowiedzialna będzie za:

- przechowywanie niezbędnych danych pomiędzy kolejnymi żądaniami

- nawigację po stronach kolejnych stronach, o ile żądanie przeglądarki wymaga komunikacji z warstwą biznesu (pobranie/aktualizację/zapis danych)

- za komunikację z warstwą logiki biznesowej i jest w stosunku do niej symetryczna tzn jednemu EndPointowi odpowiada jeden Kontroler, co pozwala zmniejszyć „powierzchnię” styku warstwy biznesowej i warstwy odpowiedzialnej za generowanie widoku.

Wzajemne relacje endpont-kontroler obrazuje poniższy diagram:



Rysunek 8. Diagram klas EndPont i Kontroler.

Aby ziarno było ziarnem CDI wymagane jest utworzenie pliku beans.xml – sam plik może być pusty.

Ziarna opisywane są następującymi adnotacjami:

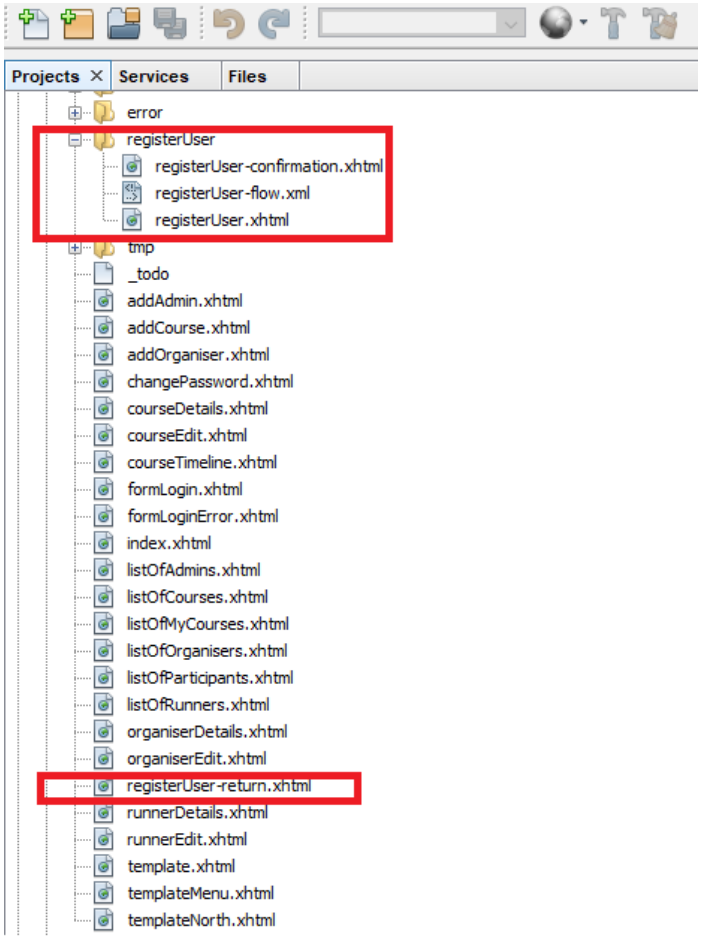
- @Named – adnotacja z pakietu javax.inject.Named adnotacja w klasie, wraz z adnotacją określająca zakres automatycznie rejestruje te klasy jako zasób z Java Server Faces. Ziarno, opisane tą adnotacją to komponent zarządzany przez CDI. @Named umożliwia dostęp do komponentu bean przy użyciu nazwy komponentu przy czym pierwsza litera jest mała. W adnotacji można użyć drugiego argumentu np @Named ("MojaNazwa") Dzięki temu JSF będzie odwoływać się do komponentu o nazwie wskazanej w cudzysłowie.
Co ciekawe jeżeli używamy standardowej nazwy ziarna to adnotacje @Named i @RequestScoped możemy zastąpić adnotacją @Model – taką adnotację nazywa się stereotypem.[4]

- @ViewScoped – adnotacja stanowiąca informację dla kontenera o czasie życia ziarna. Adnotacja informuje kontener o potrzebie stworzenia ziarna o zasięgu widoku, co oznacza, że dane ziarno istnieje dopóki istnieje dany widok, a kolejny request z tej samej przeglądarki, a innej karty nie powoduje jego usunięcia.

@RequestScoped – ziarno o długości życia zapytania z przeglądarki, każde kolejne zapytanie powoduje że dane ziarno przestaje istnieć, wykorzystano przy formularzu zmiany hasła.

- @FlowScoped – adnotacja oznacza ziarno o długości życia przepływu pomiędzy kolejnymi stronami. Pochodzi z pakietu javax.faces.flow. Aby móc używać tej adnotacji to:

a) strony powiązane z flowscoped muszą być zgromadzone w podkatalogu o takiej samej nazwie jak pierwsza strona, a także musi tam być pusty plik xml o tej samej nazwie, jak ma to miejsce w opisywanej pracy,



Rysunek 9. Konfiguracja FlowScoped w projekcie

Ostatni plik albo musi być już poza katalogiem i mieć w nazwie return, albo można dokonać odpowiedniej konfiguracji w pliku xml opisującym flow[3], lub musi być zarejestrowany w faces-config[4].

```
<flow-definition id="register">
        <flow-return id="endFlow">
                <from-outcome>/index</from-outcome>
        </flow-return>
</flow-definition>
```

Listing 7: Przykładowa rejestracja FlowScoped

- lub zgodnie z dokumentacją można skonfigurować przepływ programowo, tworząc adnotację na klasie @FlowDefinition.[4]

- @SessionScoped - ziarna mają zakres sesyjny, czyli istnieją do momentu istnienia danej sesji HTTP, która może być usunięta po określonym czasie lub poprzez wylogowanie się.

Innym popularnym zasięgiem jest @ConversationScoped – jest to zakres kontrolowany przez programistę i może istnieć przez wiele requestów, a jego maksymalny czas istnienia jest ograniczony długością sesji od której nie może być dłuższy.

W wykonanej implementacji metody ziaren w większości zwracają zwracają wartość String, która odpowiada nazwie strony która ma zostać wyświetlona. Przy nawigacji skorzystano z automatycznego mechanizmu przekierowywania do strony, o ile jej nazwa strony jest zgodna z wartością uzyskanego Stringu, co pozwoliło uniknąć żmudnej konfiguracji pliku faces-config.xml, w którym zawarte są min. zasady nawigacyjne.

Na listingu 7 przedstawiono PageBean odpowiedzialny za rejestrację biegacza.

```
@Named
@FlowScoped("registerUser")

public class RegistrationBeanUser implements Serializable {

    @Inject
    UzerController uzerController;

    private Runner runner = new Runner();

    public Runner getRunner() {
        return runner;
    }

    public void setRunner(Runner runner) {
        this.runner = runner;
    }

    public String save() throws BasicApplicationException {
        return uzerController.saveNew(runner);
    }

    public String next() {
        if (uzerController.doesGivenEmailExists(runner.getEmail())) {
            AplicationController.showMessage("registerRunner:email",
"messages.constraint.email");
            return null;
        }
        return "registerUser-confirmation";
    }
```

Listing 8: Implementacja RegistrationBeanUser lokalizacja pl.java.biniek.web.beans.runners

3.4 Obsługa wyjątków i logowanie aktywności

Zarówno logowanie aktywności jak i przechwytywanie wyjątków postanowiono w niniejszej implementacji rozwiązać za pomocą interceptorów. Rozważanym wcześniej rozwiązaniem było użycie w warstwie biznesu bloku try-catch albo interceptora, a w warstwie web użycie action-listenera.

Interceptory występują w od JavaEE w wersji 5, a wraz z 7 edycją javaEE stały się niezależnymi elementami. Iterceptory mogą odnosić się zarówno do cyklu życia ziaren

jak i ich metod biznesowych. Mogą znajdować się w klasach ziaren jak i w klasach bazowych tych ziaren, ale najczęściej wybieranym chyba rozwiązaniem jest implementacja ich w odrębnych klasach.

Jedną z metod wywoływanych przez interceptor jest metoda opisana adnotacją @AroundInvoke. Dopiero w metodzie opisanej tą adnotacją wywoływana jest właściwa metoda, którą uruchamiamy za pomocą „invocation.proceed()" (lub kolejny interceptor - jeżeli jest ich więcej). Wewnątrz metody interceptora mamy pełen dostęp do parametrów wywołania metody jak i do obiektów zwracanych przez nią. Te właściwości interceptora pozwalają zarówno otoczyć wykonywaną metodę elementami odpowiedzialnymi za przechwytywane wyjątków, zmienić sposób jej działania jak i zapowiedź jej wykonaniu.

W implementacji obsługi wyjątków wykorzystano te właściwości interceptora. Uznano, że opisywanie wszystkich metod, które mogą zwrócić wyjątek blokiem „try{ } cacth", po pierwsze nie jest funkcjonalne i może doprowadzić do pominięcia którejś z istotnych metod, poza tym znacznie zmniejsza czytelność kodu.

Aby dowiązać interceptor do ziaren EJB wymagane jest opisanie metod lub klas adnotacją @Interceptors(NazwaInterceptora.class).

Jako ze możliwość dowiązania interceptorów do ziaren CDI pojawiła się w późniejszej specyfikacji ich dołączanie przebiega nieco inaczej. Aby użyć interceptora na ziarnach CDI trzeba dowiązać interceptor poprzez adnotację wskazującą interfejs, w którym wyspecyfikowany jest nasz interceptor, a także dokonać jego wdrożenia w pliku beans.xml.[4]

Dokładna konstrukcja interceptora dla ziaren CDI została pokazana na poniższych przykładach zaczerpniętych z aplikacji.

```
@InterceptorBinding
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
public @interface BinderPageBean {
   }
```

Listing 9: Implementacja interfejsu wiążącego klasę/metodę z interceptorem lokalizacja pl.java.biniek.exception.interceptor.frontend;

```
@Interceptor
@BinderPageBean
public class InterceptorForPageBean implements Serializable {

    Object result;

    @AroundInvoke
    public Object interceptorMethodForPageBean(InvocationContext
      invocation) throws Exception {

        result = invocation.proceed();
        return result;
    }
```

```
}
```

Listing 10: Ważniejsze fragmenty interceptora dowiązywanego lokalizacja pl.java.biniek.exception.interceptor.frontend;

```
@Named
@RequestScoped
@BinderPageBean
public class RunnerDetailsBean implements Serializable {
…
}
```

Listing 11: Implementacja interfejsu wiążącego klasę/metodę z interceptorem lokalizacja pl.java.biniek.web.beans.runners

```
 <interceptors>
    <class>pl.java.biniek.exception.interceptor.frontend.InterceptorForCont
    rollers</class>

    <class>pl.java.biniek.exception.interceptor.frontend.InterceptorForPage
    Bean</class>

    <class>pl.java.biniek.exception.interceptor.frontend.InterceptorChangin
    gStringToNull</class>
     </interceptors>
```

Listing 12: Wdrożenie interceptora w pliku beans.xml

Dla każdej warstwy aplikacji skonstruowano niezależny interceptor:

Dodatkowo wszystkie interceptory wyposażono w elementy logowania aktywności. Pojawienia się wyjątków, normalna aktywność aplikacji, oraz biznesowa aktywność użytkowników jest logowana przez interceptory.

- dla warstwy fasad interceptor przepakowuje pojawiające się wybrane wyjątki bazodanowe, oraz inne wyjątki pojawiające się w tej warstwie, w wybrane wyjątki aplikacyjne

- dla warstwy endpointów interceptor również przepakowuje pojawiające się w tej warstwie wyjątki nieaplikacyjne w wyjątki aplikacyjne. Wszystkie wyjątki aplikacyjne zarówno przychodzące z warstwy fasad jak i wyjątki pojawiające się tej warstwie są przekazywane dalej do widoku

- dla warstwy kontrolerów interceptor wyłapuje wszystkie wyjątki, które pojawiły się we wszystkich 3 warstwach i zamienia je na odpowiednie reguły nawigacyjne i komunikaty które są przekazywane przeglądarce

- warstwa PageBeanów również posiada własny interceptor, który jest odpowiedzialny za wyłapywanie wyjątków, które pojawiły się w niej, lub pojawiły się w wyniku wykonania metod interceptora w warstwie kontrolerów.

Dodatkowo nad całą warstwą widoku zaimplementowany jest generalny ActionListener. Pierwotnie to on był rozpatrywany jako główny element obsługujący wyjątki aplikacyjne i inne, w warstwie widoku, a także miał być elementem logującym, ale interceptory wydają się być elementami o większych możliwościach, dającymi większą elastyczność.

ActionListener w obecnej wersji ma przypisaną rolę przede wszystkim wejściowego loggera, wskazującego jakiej interakcji dokonał użytkownik na stronie, a także ostatniego elementu logującego wyjątki, które mogły by się pojawić poza warstwą interreceptorów w wyniku wykonania ich metod. Użycie ActionListenera wymaga samodzielnego zaimpelmentowania metody abstrakcyjnej processAction(event) lub użycia którejś z implementacji z pakietów com.sun.faces.application.ActionListenerImpl lub org.apache.myfaces.application.ActionListenerImpl.

```
public class WebExceptionListener extends ActionListenerImpl implements
ActionListener {

    @Override
    public void processAction(ActionEvent event) throws
      AbortProcessingException {
        boolean excepfionOccured = false;
        StringBuilder sb = new StringBuilder();
        FacesContext fContext = FacesContext.getCurrentInstance();
        sb.append("WebListener called entering: " + event.toString() + ",
            element id " +
      event.getComponent().getClientId());//event.getPhaseId()
        sb.append(", SessionID : " +
      fContext.getExternalContext().getSessionId(false));
        sb.append(", by user: " +
      fContext.getExternalContext().getRemoteUser() + " contextname: " +
      fContext.getExternalContext().getContextName());
        sb.append(", element: " + ((HttpServletRequest)
      fContext.getExternalContext().getRequest()).getLocalName());

        try {
            super.processAction(event);
        } catch (Throwable ex) {
            Logger.getGlobal().log(Level.SEVERE, "UNCAUGCHT EXCEPTION IN
            FRONTEND " + this.getClass() + " Session ID"
                    + fContext.getExternalContext().getSessionId(false),
            ex);
            sb.append(" Session ID" +
            fContext.getExternalContext().getSessionId(false));
            excepfionOccured = true;


AplicationController.showGeneralMessage(FacesMessage.SEVERITY_ERROR,
"failure.page.message");

        } finally {
            if (excepfionOccured) {
                Logger.getGlobal().log(Level.SEVERE, sb.toString());
            } else {
                Logger.getGlobal().log(Level.INFO, sb.toString());
            }
```

```
        }
    }

}
```

Listing 13: Kod zródłowy WebActionListenera lokalizacja pl.java.biniek.exception.exlistenerandloggerfrontend

ActionListener wymaga dodatkowo rejestracji w piku faces-config.xml

```
    <application>
        <action-listener>

pl.java.biniek.exception.EXListenerAndLoggerFrontend.WebExceptionListener
        </action-listener>
</application>
```

Listing 14: Rejestracja listenera w pliku faces-config.xml;

W konfiguracji Interceptorów wykorzystano jeszcze jeden interceptor wiązany. Jest dowiązany do części metod pagebeanow w wyniku których działania metoda powinna zwrócić obiekt typu lista. W wyniku wygaśnięcia sesji lub innego naruszenia zasad dostępu może dojść do uruchomienia metody, która powinna zwrócić Listę, ale w wyniku pojawienia się wyjątku wynikającego z braku uprawnień zwrócona zostaje generalna zasada nawigacyjna czyli String. Zwrócenie innej wartości niż lista lub null powoduje powstanie wyjątku ClassCastException. Jednym z rozwiązań było zwracanie null zamiast wyjątku w przypadku naruszenia reguł dostępu do metody, ale w mojej ocenie nie było to rozwiązanie prawidłowe gdyż, mogło by ukryć np. próby włamania. Doskonale w takiej sytuacji sprawdza się Interceptor, który w tych kilku przypadkach przechwytuje konkretny wyjątek ClassCastException i zwraca null, dodatkowo logując ten fakt do dziennika. Zwrócić tu należy też uwagę na fakt że interceptor pozwala również na zamianę wartości i typu zwracanej wartości przez metodę.

Poniżej znajduje się jego listing:

```
@Interceptor
@BinderStringToNull
public class InterceptorChangingStringToNull implements Serializable {

    Object result;

    @AroundInvoke
    public Object interceptorMethodForDAO(InvocationContext invocation)
      throws Exception {
        FacesContext fc = FacesContext.getCurrentInstance();
        StringBuilder sb = new StringBuilder();
        try {
```

```
            result = invocation.proceed();
        } catch (ClassCastException ex) {
            Logger.getGlobal().log(Level.WARNING, "EXCEPTION  occured in
            InterceptorChangingStringToNull: " + ex.toString() + " SesionID
            " + fc.getExternalContext().getSessionId(false));
            sb.append(" occured exception " + ex);
            sb.append(" " + invocation.getTarget().getClass().getName() +
            "." + invocation.getMethod().getName());
            sb.append(", by user: " +
            fc.getExternalContext().getRemoteUser());
            sb.append(", SessionID:  " +
            fc.getExternalContext().getSessionId(false));
            Logger.getGlobal().log(Level.WARNING, sb.toString());
            result = null;
        } finally {
            return result;
        }
    }
}
```

Listing 15: Interceptor, zamieniający wybrane metody, lokalizacja pl.java.biniek.exception.interceptor.frontend.

3.5 Pozostałe elementy aplikacji

3.5.1 Uwierzytelnienie i bezpieczeństwo

Aby móc zalogować się w systemie użytkownik powinien się zalogować, czyli uwierzytelnić. Wykorzystano standardowy formularz uwierzytelnienia zawierający element J_SECURITY_CHECK. Listing elementów strony logowania załączono poniżej :

```
<form method="post" action="j_security_check">
  <h:panelGrid id = "panel"  columns="2" border = "1"
               cellpadding = "10" cellspacing = "1">
      <f:facet name = "header">
          <h:outputText value = " $
            {msg['account.login']}"/>
      </f:facet>
      <p> ${msg['account.login']}: <input type="text"
      name= "j_username" /> </p>
      <p> ${msg['account.password']}: <input
      type="password" name= "j_password" /> </p>
      <f:facet name = "footer">
          <h:panelGroup style = "display:block; text-
      align:center">
      <input type="submit" value="$
      {msg['action.login']}" />
          </h:panelGroup>
      </f:facet>
  </h:panelGrid>
</form>
```

Listing 16: Elementy strony logowania.

W deskryptorze wdrożenia web.xml zdefiniowano ograniczenia dostępu:

- wymagania dotyczące transportu danych (wykorzystanie protokołu HTTPS),

- role uprawnione do w aplikacji.

W deskryptorze można tam również określić uprawnienia konkretnych ról do określonych katalogów zawierających formularze xhtml,

Dodatkowo w EndPointach zmapowano zasady dostępu do metod zależnie od poziomu dostępu poprzez adnotację @RolesAllowed, w której określa się jakie role mają możliwość wykonania metody. W przypadku próby dostępu bez uprawnień zwrócony zostaje wyjątek.

W przypadku metod zwracających widoki indywidualne zaimplementowano już w warstwie widoku metody weryfikujące czy dany użytkownik ma możliwość dostępu do fanej formatki – np. edycji danych, tak aby nie doszło do przypadkowego ujawnienia danych które powinny być chronione, stosowne metody zwracają zależnie od sytuacji, wyjątki NullUserException lub WrongUzerException.

Przy zapisie danych w endpointach następuje również weryfikacja czy dany użytkownik ma prawo do tego.

Loginem użytkownika jest jego email, a hasło jest przechowywane w bazie danych w polu password w postaci zaszyfrowanej algorytmem SHA-256.

3.5.2 Internacjonalizacja

W implementacji interfejsu użytkownika wykorzystano internacjonalizację zgodną ze standardem i18n. W systemie widnieją 2 języki, polski i angielski. Język jest wybierany przez system na podstawie ustawień przeglądarki internetowej użytkownika oraz zawartości pliku messages.properties w którą wchodzi wartość klucza i komunikatu. Aby skorzystać z internacjonalizacji w aplikacji konieczna była dodatkowa konfiguracja w pliku deskryptora web.xml.

```
<context-param>
    <param-name>resourceBundle.path</param-name>
    <param-value>i18n.messages</param-value>
</context-param>
```

Listing 17: Konfiguracja elementów językowych web.xml

Powyższe parametry wskazują gdzie kontener powinien szukać wartości dla kluczy. Do nazw kolejnych wersji językowych dodawane są sufiksy determinujące język np. „_en”. Pliki językowe są umieszczane w domyślnej lokalizacji tj „\src\main\resources\i18n\ messages”.

Dodatkowym elementem konfiguracyjnym są wpisy dotyczące przestrzeni nazw plików i wspieranych lokalizacji zgodnie z zapisami pliku faces_config.xml

```
<locale-config>
```

```
        <default-locale>pl</default-locale>
        <supported-locale>pl</supported-locale>
        <supported-locale>en</supported-locale>
    </locale-config>
    <resource-bundle>
        <base-name>i18n.messages</base-name>
        <var>msg</var>
    </resource-bundle>
    <message-bundle>
        i18n.jsf_messages
    </message-bundle>
```

Listing 18: Konfiguracja elementów językowych faces-config.xml

Plik określa wspierane lokalizacje oraz przestrzeń nazw plików dla wykorzystywanych przez system do komunikacji z użytkownikiem oraz komunikatów generowanych przez JSF.

Zgodnie z konfiguracją domyślna lokalizacja to PL wspierane są lokalizacje pl i en, plików należy szukać w katalogu i18n w pliku messages. Parametrem wywołania jest „msg" i dokonuje się go poprzez następujące wywołanie {msg["nazwa.klucza"]}. Natomiast klucz do standardowych komunikatów JSF znajduje się w pliku o nazwie jsf_messages.

3.5.3 Serwis Mailowy

Serwis zbudowano w oparciu o Java Mail API, który dostarcza niezależny od platformy zerwis mailowy. API to nie jest dostarczane wraz z Java EE 7, ale stało się częścią standardu zgodnie ze specyfikacją Java EE 8. [6]

Aby móc z niego skozystać trzeba dodać do pliku pom.xml następującą zależność:

```
<dependency>
      <groupId>com.sun.mail</groupId>
      <artifactId>javax.mail</artifactId>
      <version>1.6.2</version>
  </dependency>
```

Listing 19: Zalezność java Mail API w pliku pom.xml

Serwis automatycznie generuje maila z konta biegowyportal@gmail.com do biegaczy zapisanych do biegu gdy parametry biegu ulegną zmianie, a także w sytuacji gdy biegacz zapisze się do biegu. Jego funkcjonalność można w dowolnej chwili rozszerzyć.

3.5.4 Skórki

W projekcie dodatkowo zaimplementowano funkcjonalność zmiany motywu strony. Motywy są jednym z podelementów PrimeFaces, natomiast nie są częścią udostępnianą wraz z PrimeFaces. Zmiana skorki odbywa się za pomocą elementu <p:themeSwitcher> który w polu value przyjmuje String, który jest nazwą odpowiedniej skórki.

By móc z nich korzystać trzeba dodać link do zewnętrznego repozytorium. Dokładną konfigurację wskazano poniżej:

```
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>all-themes</artifactId>
    <version>1.0.10</version>
</dependency>

    <repository>
        <id>prime-repo</id>
        <name>Prime Repo</name>
        <url>http://repository.primefaces.org</url>
    </repository>
```

Listing 20: Zależność i wskazanie zewnętrznego repozytorium w pom.xml

```
<context-param>
  <param-name>primefaces.THEME</param-name>
  <param-value>#{themeSwitcherBean.theme}</param-value>
  </context-param>
```

Listing 21: Wpis w pliku web.xml pozwalający na dynamiczną zmianę skórek.

Z względu za fakt że repozytorium nie posiada odpowiedniego certyfikatu ssl występują pewne różnice w konfiguracji systemu. W przypadku systemu linux udało się projekt zbudować żądając od mavena aby korzystał z serwerów bez ważnego certyfikatu ssl, a w przypadku Windows trzeba było pobrać odpowiednie pliki samodzielnie i je zainstalować . Fakt ten będzie uwzględniony w instrukcji wdrożeniowej.

Wartość wybranego motywu jest przechowywana w ziarnie o zasięgu sesji. Przy dalszej rozbudowie aplikacji należało by uwzględnić fakt zmiany skórek w profilu zalogowanego użytkownika w bazie danych tak aby po zalogowaniu się użytkownik, widział ostatnio wybierany motyw.

4 Instrukcja wdrożeniowa

Poniższy rozdział przedstawia najprostszy sposób wdrożenia aplikacji i jej uruchomienia.

Wymagane oprogramowanie to:

- Java JDK w wersji 8, które dostępne jest na stronie pod adresem https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

- środowisko NetBeans w wersji 8.2 dla Java EE, które dostępne jest na stronie pod adresem https://netbeans.org/downloads/index.html

1) instalujemy pakiet Java JDK,

2) instalujemy środowiska NetBeans IDE 8.2 –

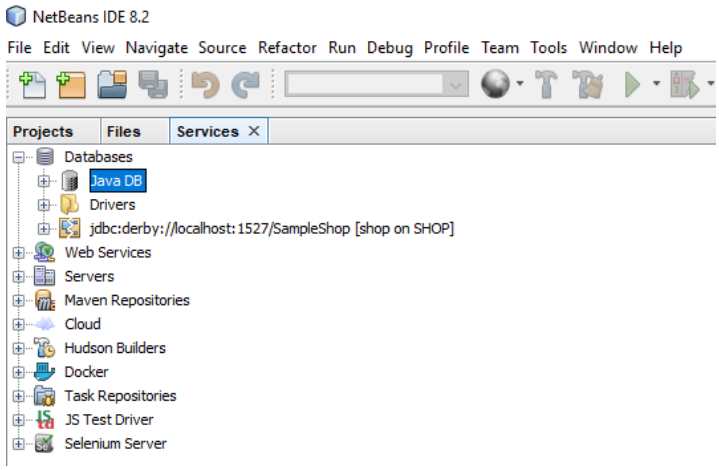
instalacja tego środowiska spowoduje instalację dodatkowych pakietów niezbędnych przy uruchomieniu aplikacji tj: Java DB, GlassFish 4.1, Maven.

3) Uruchamiamy aplikację NetBeans

4) Aplikacja jest udostępniana jako paczka ZIP, którą należy zaimportować. Dokonujemy tego za pomocą komendy File > Import Project > From ZIP

5) Konfiguracja bazy danych za pomocą NetBeans

a) W oknie Services należy rozwinąć gałąź drzewa Databases > Java DB



Rysunek 10. Widok Services i bazy danych.

b) Z menu kontekstowego wybrać Create Database.

c) W oknie dialogowym należy wprowadzić dane zgodnie z zawartością pliku wdrożeniowego glassfish-resources.xml

```
<property name="URL" value="jdbc:derby://localhost:1527/piotrDB2"/>
<property name="serverName" value="localhost"/>
<property name="PortNumber" value="1527"/>
<property name="DatabaseName" value="piotrDB2"/>
<property name="User" value="piotr"/>
<property name="Password" value="piotr"/>
```

Listing 22: Konfiguracja bazy danych

W tym miejscu oczywiście należało by użyć własnego hasła, należy je ustawić zarówno w bazie jak i w w/w pliku.

6) Budowanie projektu. Ze względu na pewne różnice pomiędzy systemem windows i linux wdrożenie jest odmienne, choć w systemie linux można zastosować metodę z systemu windows.

a) Windows

należy pobrać ze strony https://mvnrepository.com/artifact/org.primefaces.themes/all-themes/1.0.10 plik jar zawierający biblioteki do obslugi skórek, następnie je zainstalować wydając komendę: `mvn install:install-file -Dfile=pelnaŚcieżkaDostepuDoPliku\all-themes-1.0.10.jar -DgroupId=org.primefaces.themes -DartifactId=all-themes -Dversion=1.0.10 -Dpackaging=jar`

b) linux

należy wejść do katalogu do którego zaimportowaliśmy projekt i w którym znajduje się plik pom.xml i uruchomić komendę: `mvn clean install -DskipTests -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true -Dmaven.wagon.http.ssl.ignore.validity.dates=true -X`

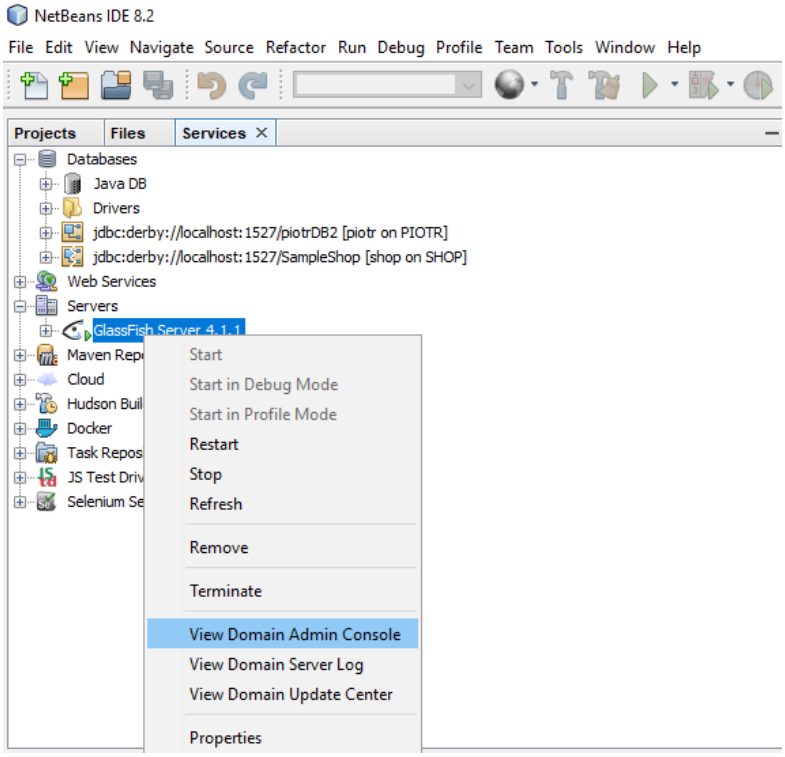
Jeżeli komenda nie będzie się chciała uruchomić to znaczy że nie jest zarejestrowana ściezka dostęlu do aplikacji maven. Aby ją zarejestrować należy w linii polecen wpisać

```
MAVEN_HOME="/ścieżka dostępu do aplikacji maven/"
export MAVEN_HOME
PATH=$PATH:$MAVEN_HOME/bin
```

można też wykorzystać instalację dla systemu Windows.

7) W tym momencie należy uruchomić projekt, aby zostały utworzone odpowiednie tabele w bazie danych, poprzez wybranie Run → Run project lub wciśniecie F6 na klawiaturze.

8) Ponownie wybieramy services i uruchamiamy konsolę administracyjną serwera GlassFish zgodnie z załączoną grafiką

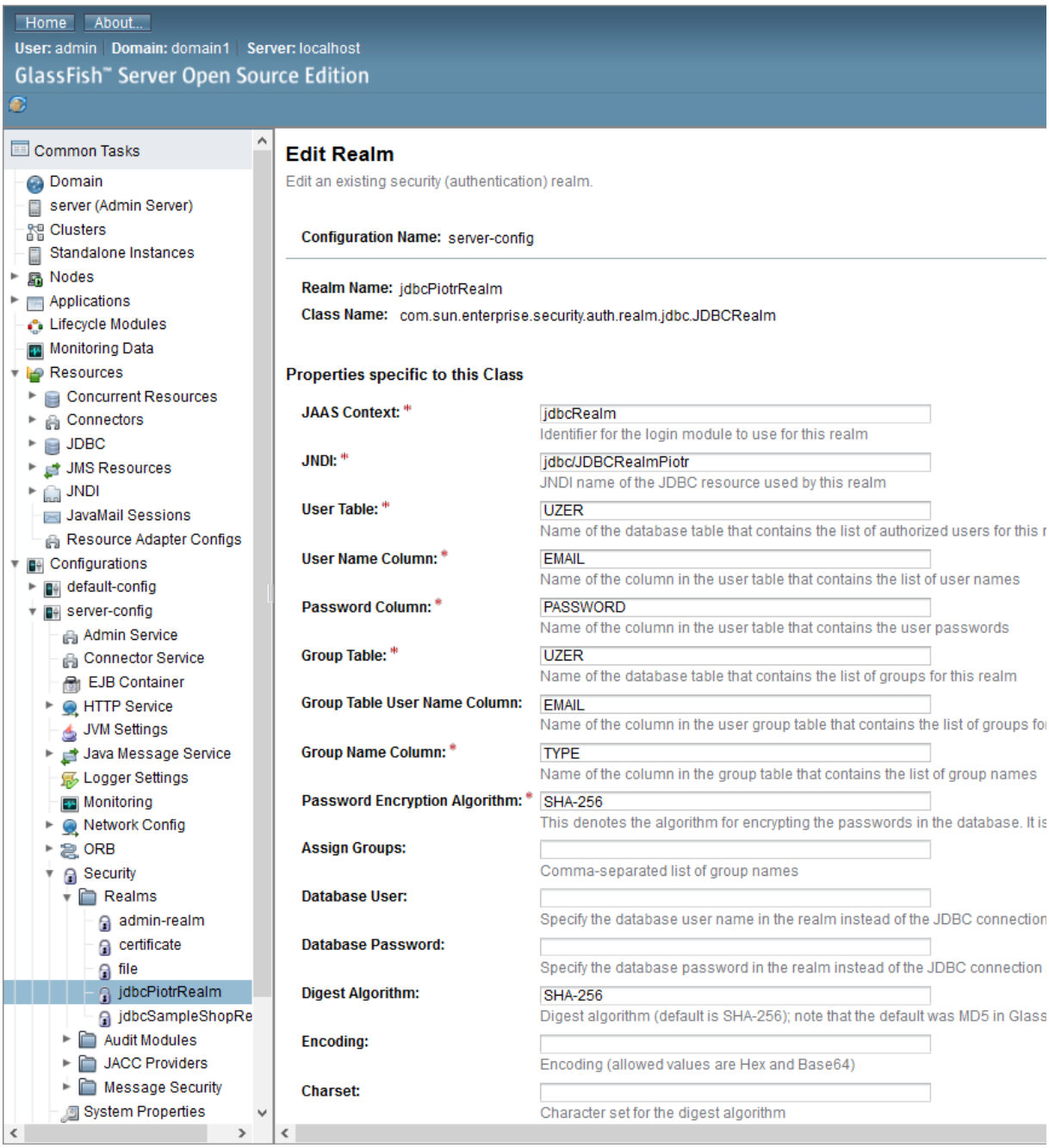


Rysunek 11. Uruchomienie konsoli administracyjnej serwera.

9) Należy skonfigurować parametry uwierzytelnienia „Security Realms” . W tym celu należy z menu konsoli administracyjnej wybrać kolejno:

Configuration > Server-config > Security > Realms i wcisnąć przycisk New..

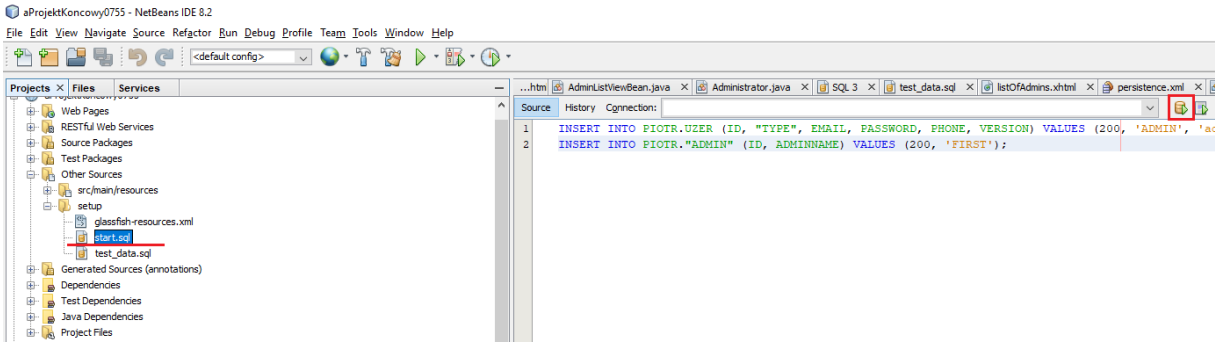
Poniżej przedstawiono konfigurację Realm



Rysunek 12. Konfiguracja Realm.

10) Wczytanie danych początkowych

Na pulpicie roboczym zawierającym projekt należy rozwinąć ścieżkę Other sources → setup → start.sql – zawierający skrypt SQL. Skrypt zawiera kwerendę tworzącą 1 administratora o loginie adm1@adm1.pl i haśle 111111



Rysunek 13. Tworzenie pierwszego administratora

11) Od momentu wczytania do bazy danych powyższego skryptu możliwe jest uwierzytelnienie w aplikacji.

5 Podsumowanie

Celem pracy było stworzenie aplikacji służącej do rejestracji biegaczy do biegów. W projekcie wykorzystano technologię, którą dostarczyła JEE w wersji 7. Wybrana technologia pozwala na znaczne skrócenie implementacji całego systemu i umożliwia skupienie się na tworzeniu logiki biznesowej oraz strony wizualnej całego projektu. Pozostała praca jest wykonywana automatycznie przez środowisko JEE. Zarówno połączenie z bazą danych, wzajemne relacje pomiędzy poszczególnymi warstwami, weryfikacja poprawności logowania, dopuszczenie do wykonania metod, dbanie o sesje i wiele innych czynności o które normalnie musiał by dbać programista, przejmuje na siebie logika środowiska. Dodatkowa funkcjonalność udostępniana przez środowisko to Java Persistence API które umożliwia wykonywanie operacji bazodanowych w ogóle bez znajomości kwerend SQL. Oczywiście ich znajomość znacznie zwiększa wydajność aplikacji umożliwiając bezpośrednie pobranie informacji z bazy, ale nie jest wymagane, a operacje bazodanowe można przeprowadzić bez znajomości SQL. Należy bardzo pozytywnie ocenić możliwości wsparcia, jakiego udziela środowisko JEE nawet początkującemu programiście.

Dzięki separacji poszczególnych warstw poprzez użycie fasad (fasady i endpointy) uzyskana została znaczna elastyczność implementacji. Z kolei delegacja komunikacji z systemem zarządzania bazami danych do serwera aplikacji uniezależniła aplikację od konkretnego producenta systemu zarządzania bazami danych.

System został zrealizowany zgodnie z założeniami. Złożony jest z relacyjnej bazy danych Java DB, logiki biznesowej opartej na ziarnach EJB oraz interfejsu wyświetlanego przez przeglądarkę renderowanego w oparciu o pliki xhtml i ziarna CDI. W aplikacji występują cztery poziomy dostępu zgodnie z założeniami. Do stworzenia aplikacji wykorzystałem zintegrowane środowisko developerskie NetBeans.

Aplikacja stanowi dobrą bazę do dalszej rozbudowy i rozwijania. Na kolejnych etapach planowane est wdrożenie systemu przypomnienia o nadchodzących wydarzeniach w postaci SMS, w dalszych planach w przypadku zainteresowania możliwe jest dołączenie systemu płatności tak aby faktycznie można było zapłacić za bieg, gdyż w chwili obecnej jest to tylko mock.

6 Źródła

Bibliografia

1: Çağatay Çivici, PrimeFaces User Guide 6.2,
2. David R. Heffelfinge: Java EE 7 Development with NetBeans 8, Wydanie , Packt Publishing, 2015
3: Dennis Kriechel, Java EE Tutorial #16.1 - JSF Faces Flow,
4: Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Kim Haase, William Markito, Java Platform, Enterprise EditionThe Java EE Tutorial Release 7 E39031-01, 2014
5. Mert Çalışkan, Oleg Varaksin: PrimeFaces Cookbook Second Edition, Wydanie , Packt Publishin, 2015
6: , Java EE 8 ,
7: , Wikipedia wg stanu na dzień 28-09-2018,

7 Rysunki

Indeks ilustracji

8 Listingi

Spis Listingów

Tabele

Indeks tabel