COMENIUS UNIVERSITY, BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

PORTAL FOR BETTING AND TIPPING

BACHELOR THESIS

2013
František Gerčak

COMENIUS UNIVERSITY, BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

PORTAL FOR BETTING AND TIPPING

BACHELOR THESIS

Study programme:	Computer Science
Study field:	2508 Computer Science, Informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Tomáš Kulich, PhD.

Bratislava, 2013

František Gerčak

70986281

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	František Gerčak
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Portal for Betting and Tipping

Aim: Create a portal that allows user to tip and bet virtual money on the outcome of sport events, resolve the outcome of the events with respect to betting odds and keep user's account consistent with the changes.

Supervisor:	RNDr. Tomáš Kulich, PhD.
Department:	FMFI.KI - Department of Computer Science
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	17.10.2012
Approved:	24.10.2012

doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

....

Student Supervisor

70986281

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	František Gerčak
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický

Názov:	Portál pre tipovanie a stávkovanie
Cieľ:	Vytvoriť portál na tipovanie a stávkovanie svojich virtuálnych peňazí na výsledok športových udalostí, vyhodnocovanie javov a držanie konzistencie užívateľovho účtu

Vedúci:	RNDr. Tomáš Kulich, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	17.10.2012

Dátum schválenia: 24.10.2012

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent

....
vedúci práce

Acknowledgement

I would like to thank my supervisor RNDr. Tomáš Kulich, PhD. for his help and advices and my family and friends for support.

Abstract

Target of my project was to describe functionality of web portal destined to online betting and tiping, where you can see odds by multiple bookmakers. I described the application's design and used technologies. In more detail, I focused on project specification, database model and internal implementation of interesting parts of application.

KEYWORDS: web application, portal, betting and tiping, MVC, analyses

Abstrakt

Cieľom mojej práce bolo popísať funkcionalitu webového portálu určeného na online stávkovanie a tipovanie, kde je možné pozrieť si kurzy od viacerých stávkových kancelárií. Opisoval som návrh aplikácie a použité technológie. Detailnejšie som sa zameral na špecifikáciu projektu, databázový model a vnútornú implementáciu zaujímavých častí aplikácie.

KĽÚČOVÉ SLOVÁ: webová aplikácia, portál, stávkovanie a tipovanie, MVC, analýzy

Contents

List of Figures

List of Tables

Introduction

In this modern age, almost everyone tried to bet on some sport's result. Many portals of bookmakers offer you to tip some sport's results and bet some money on your tip. But there is no place for slovak tippers to learn how to bet. This is, what we want to offer to slovak users.

Our application is collecting odds from multiple bookmakers and users can choose any of them with the most advantageous odds. User can place his ticket, and create an analyse, why he has chosen this team to win. Other users can watch analyses of all users (except V.I.P. users) and learn to bet in more logical way. You are spending just some virtual money. This idea comes from my supervisor, who received an offer from a man from bookmakers surroundings to create a portal similar to OddsPortal.com for slovak users. [Odd]

My participation on creating this application, was more than seventy percent overall. I was creating the most of the server-side programming and database structure of this project. I made neither a design of application nor javascript part of code. This project was created by Test-Driven Development (TDD). In this paper I am presenting the interesting application's features implemented by me.

First chapter will include fully-qualified specification of the project. I will comment all the features, which application will have. I will specify the requirements for user, what roles and actions he can provide in our application. This chapter will also include specification of user lists, analyses and tickets. I will mention what functionality should the wallet part of application have and what administrator can do in his special administration interface.

In second chapter, I will describe the application's logic. All we need to code behind the user viewport in the application. It is background application's behaviour which helps us to organize application better. I will write about authorization and authentication processes, component system which is handling some kind of events. In this chapter you will find also our-developped translating system using database backend.

Database structure model will be introduced in the third chapter. All the important database tables representing user objects and some their relations are the parts, which I will describe in this chapter. Also some application structure, which we made as XML Parser of bookmaker results and odds, and SessionSection object to separate session for different use. At the end, there will be mentioned mainly used server-side technologies.

Chapter 1

Specification of the Project

1.1 Project's description

The project is about betting and tipping on sport's results as I have presented in Introduction. This portal is for people, who want to tip and win. Users spend their virtual money on sport events and matches, also parts of matches, which are automatically updated from bookmaker's xml url, which can be editable by admin in administration interface.
Main page of the application consists of blog articles, main navigation and sport navigation for matches. Web consists of administration interface and user interface. Navigation is implemented in hierarchical way - sport-region-league. Users can see matches from any level of this navigation. This does really mean, that user can watch matches from any sport, or all matches of the currently set sport and region, or a league. User can choose the language of webpage.

Application brings registered users periodical income of credits (monthly income of credits ‘A‘ and daily income of credits ‘B‘), which they can use for betting. This credit is set to zero every month, because of the competition between all users, including the new registered ones. When a new user registers himself, he is rewarded by ‘C‘ amount of credits. All this values ‘A‘,‘B‘,‘C‘ can be changed in administration interface.

1.2 Users

Our project differs users:

- **non-registered user**
  1. can watch all the blog articles, matches with odds and user analyses
  2. can change the language of the webpage

3. can see list of the users by ROI or cash

- **registered user**
  1. can see all the data that non-registered user can not
  2. can choose default bookmaker
  3. can create own ticket
  4. can see all the tickets, which he had placed
  5. can change registration data
  6. can edit registered user's info
- **VIP user**
  1. can do all actions the registered user can do
  2. can see analyses of VIP users
- **admin**
  1. has permission to access administration interface
  2. can create/edit/delete analyses, matches, leagues, bookmakers, user's transactions
  3. can create/edit/delete blog articles
  4. can cancel every match
  5. can add VIP status to registered user
  6. can edit registered user's info

1.3 List of Users

Every registered user has his own wallet consisting of transactions, which he has made - placing ticket, earning registration bonus, won ticket's reward, ... Admin can send users some amount of money. This will be described in next section.

List of users by Cash

Users are competing in list of users - how much cash they have got at the end of the month - you can filter any month, whole year or overall score.

List of users by ROI

ROI(Return on Investment) is computable as:

$$\text{ROI} = \frac{\text{right number of analyses}}{\text{total number of analyses}}$$

Statistics about right and total number of analyses are stored in a database and you can also filter through the history of statistics for any month, whole year or overall. You can filter users with at least A number of analyses or users analysing events of specified sport. You can filter also users with admittance, who have got at least B

1.4 Tickets

Registered user can create some tickets, which are evaluated later via cron technology. Tickets have got some status:

draft

There could be only one draft ticket per user. This is the ticket, which you can see, when you are adding new bets into ticket.

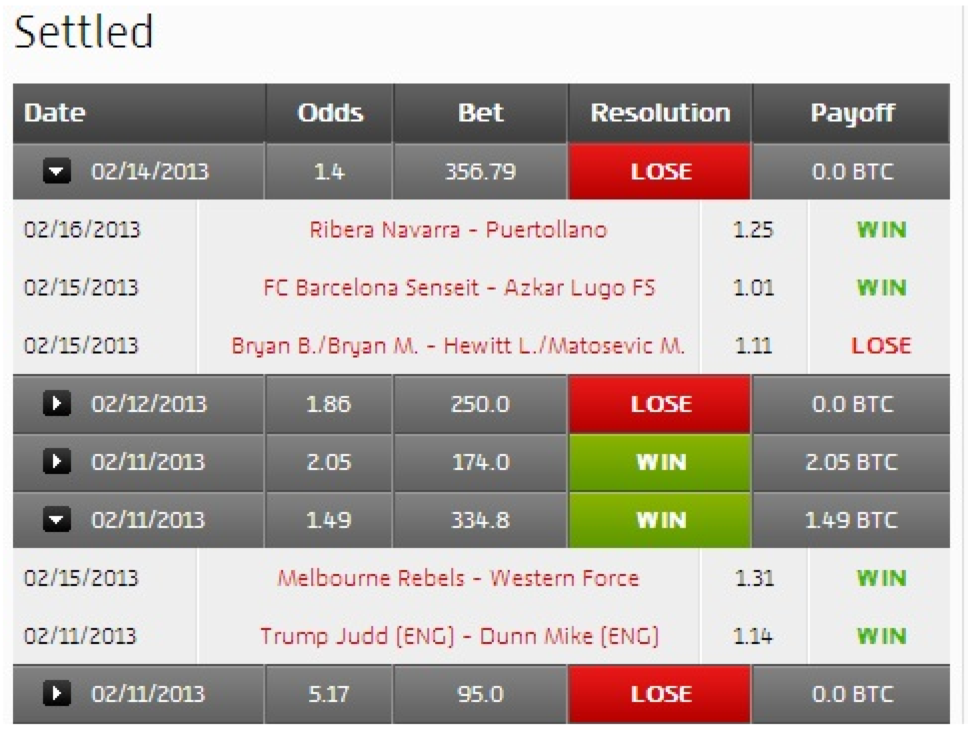
placed

After you have got all your bets, you can place ticket. Ticket is in status placed until it is evaluated.

settled

When the match results are published, ticket will be evaluated and you know how much money you had earned.

There is a place for list of tickets, which are settled, and other list of non-settled tickets, which are not in draft. Both of the lists contains information about matches you had bet on and actual result - WIN or LOSE for each match. Waiting tickets, which were not evaluated have got also potential revenue information as it is presented in figure 1.1.


Settled

Date	Odds	Bet	Resolution	Payoff
▼ 02/14/2013	1.4	356.79	LOSE	0.0 BTC
02/16/2013	Ribera Navarra - Puertollano		1.25	WIN
02/15/2013	FC Barcelona Senseit - Azkar Lugo FS		1.01	WIN
02/15/2013	Bryan B./Bryan M. - Hewitt L./Matosevic M.		1.11	LOSE
▶ 02/12/2013	1.86	250.0	LOSE	0.0 BTC
▶ 02/11/2013	2.05	174.0	WIN	2.05 BTC
▼ 02/11/2013	1.49	334.8	WIN	1.49 BTC
02/15/2013	Melbourne Rebels - Western Force		1.31	WIN
02/11/2013	Trump Judd (ENG) - Dunn Mike (ENG)		1.14	WIN
▶ 02/11/2013	5.17	95.0	LOSE	0.0 BTC

Figure 1.1: List of tickets

1.5 Administration Interface

In our application, there is an administration interface for users with administration privilegies, where you can easily change almost all the data presented in webpage through web forms. Administration menu is sorted in ascending order.

Banner Administration

Banners presented on website should be editable by inserting HTML code. They are multilingual, so there are so many edit boxes as languages. There could be nine banners together. You can set them active or not-active. Active banners are presented in website, non-active banners are disabled and users can not see them. Banners can be only editable and you can not create them, or delete them. The only way how to remove the banner out of the website is to deactivate it or put empty HTML piece of code into edit-boxes for all languages.

Blog Administration

Only admin users can add new articles into blog, edit or delete them. Editing articles is easy because of using markdown language. Python has an library to work with markdown language, which I had used. User just click the button and user-friendly language will appear in edit box.

Example:

```
[example link][1]
[1]: http://example.com/
```

This will create a link to website http://example.com with a link description "example link".

Bookmaker Administration
Consist editing name of the bookmaker, URL and URL to XML with offered odds, which will be processed and saved into database.

Mailer
Admin can send e-mails to all users throughout one web form.

General settings
Administration of general settings for whole application - number of credits per day, number of credits per month, registration's reward, maximum winning count of BTC (Bet and Tip Coins), maximum number of tickets, which can be placed in one day. Also there you must set an url to xml with results of matches.

General contents
Long texts accesible on website as contact, privacy policy or terms of use are editable here.

Translations Administration
Adding translation for a webpage is easy. Download excel file with all words presented in website, translate them and upload back. If you change language, you will have a translated application.

User settings
Editing user's information, adding VIP status to user or administration of user's wallet. Administrator see user's wallet, but he can edit its transactions, add some amount of money or delete transactions. Available part of this wallet page is also listing in history of transaction throughout web form, in which you fill dates "Transaction from date" and "Transaction to date". This will filter transactions transfered in selected time interval.

Other's database entities administration

Competitor, Languages, League, Matches, Region, Sports, Country entities i will describe in Database model of the application. Editation is creating by using select-box, where you can choose for which region belongs concrete sport, also you can edit sport's name.

1.6 Other application's features

1.6.1 Analyses

Analyses are comments of users about non-started matches. Every registered user can add analyse before the match starts, consisting of text, adding his tip and probability of outcome. More right analyses users have got, they are earning higher ROI number. This ROI number helps him to get better position on the List of users, so they are more popular as a good tipper. The main reason for creating these analyses is to learn people to think about their choice and teach them to argument as well as possible, because all non-VIP analyses are public. So everyone, including non-registered users, can see analyses comments, tips and probability of anyone else.

1.6.2 Wallet

Everyone, who wants to be in the top of the List of users, must have his transaction under control. It is necessary to see all the transactions together and see how much amount of money the user paid for the lost tickets in selected date interval. In wallet is presented history of all of the transactions linked with user's account. Kinds of transactions:

Daily bonus

Every day is added to all accounts some amount of money. Amount is editable in administration iterface - General Settings.

Monthly bonus

Similarly like "Daily bonus", Monthly bonus differs only in period of time, when the bonuses are assigned to users.

Ticket win/lose

When the new outcomes of matches are processed, ticket is recalculated. If we know results of all matches presented in the ticket, and all of the ticket bets are calculated, then we can evaluate ticket and positive or negative transaction will appear on user's account.

Administrator's bonus

Administrator of the system can send user any amount of money. He can also send to user negative amount of money - this does mean, that administrator grabbed user's money.

Users can not help each other by sending their money. This feature is forbidden and it can not be provided.

Chapter 2

Application's Logic

2.1 Authentication and Authorization

Every page with user accounts, which you can see on the Internet, is using authentication and authorization process. The reason is, that the account of user contains his private data. It is not possible to change data of the other user on current website. Every user must by unambigiously identified and recognized by application before showing up him his own data. Non-administrator user should not have access to page with the data of other users.

2.1.1 Authentication

Authentication process lets application know, whether the client is, who he claims to be. *Who is the user?* - is an important question in authentication process. Webserver somehow determines (for example using database), if the user account exists. In private and public computer networks (including the Internet), authentication is commonly done by using logon passwords, the otherway could be fingerprints or any other way of recognization identity of the currently logging user. Knowledge of the password is assumed to guarantee, that the user is authentic. [Rou07]

Authentication process on our website is provided by creating object Authenticator. Authenticator takes 2 arguments:

1. database object
2. request object

Authenticator has got also public functions login and logout. If we want to log in, we must yield our email address and password. This will be sent to authenticator object's login function. It can throw an exception by logging in, when the user does not exist, the user's password is incorrect or account was not activated. Otherwise, the user will be logged - authenticated. Login and logout function are using remember and forgot method from pyramid.security package. These functions return suitable headers for remembering/forgetting login information. Login component takes this new headers from authenticator, if no error occured, and redirects you to mainpage website with new headers. [Doc13]

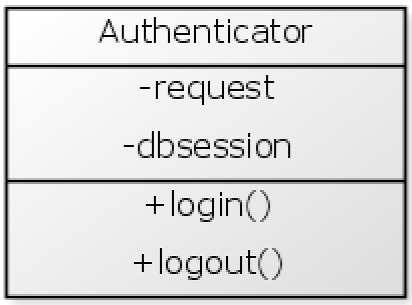


Figure 2.1: UML: Class Authenticator

2.1.2 Authorization

Authorization process is not process of recognition the person. It is the process, which tells us, whether the user has permission to access object, or if the user can do an operation. Authorization is important mechanism, because of protecting secured resources. In pyramid, it is based on some principals objects, action and permission. This way we are building an ACL - Access Control List object. ACL object must have `__acl__` attribute set. `__acl__` is a list of triplet. First item of this triplet tells, which relation do you have with this permission (Deny, Allow). Second part is a principal, it is telling, who has this permission set. Pyramid knows also some special principals as Authenticated, or Everyone. Permission is the last part of this triplet.

For example:

```
__acl__ = [
    (Allow, 'administrators', All),
    (Deny, 'banned', All),
    (Allow, Authenticated, 'view'),
    (Allow, 'editors', 'edit'),
]
```

This part of code tells, that users with 'administrators' groupset can do all the actions in website and access all the pages. Authenticated - actually logged users, can access pages, which have set permissions just to view, and editors can visit pages with 'edit' permissions. This ACL also presents that, everyone who has actually set 'banned' group, can do nothing with website, because this user has no permissions now. The permissions of pages are set to view functions with decorator `@view_config`.

```
@view_config(route_name='add_page',
    renderer='templates/edit.pt', permission='edit')
```

```
class RootFactory:
    __acl__ = [
        (Allow, 'admin', All),
        (Allow, 'editors', 'edit'),
    ]

    def __init__(self, request):
        self.request = request
```

This code is for simple factory RootFactory,which creates context objects with `__acl__` attribute. ACL declares to allow admin all actions and editors can just edit pages. Now I need only set this factory, where I am adding a path to project.

```
config.add_route('route_name', '/some/path/', factory=RootFactory)
```

2.2 Components

Important thing in software engeneering is to separate modules. Each part of code that has different functionality has its own module. This allows us to create wide application without repetation of code. It is useful when I am using for example login component and I want to repair something in the code, I must change only one login module now, and everything is allright. Also if I want to add some functionality to this component, I have to change only one modul - login controller (MVC design pattern). I can create components, which can render self-one by using decorator:

```
@template_config(renderer_name='betandtip:templates/component.mako')
```

Component should not be created by using its constructor. It is good practice to use its factory, which creates all necessary objects, which we need to create concrete component. Every component has got its own factory. Factory creates objects, which passes to component's constructor.

2.2.1 Component Request

Component request is abstraction and modification of pyramid request object.

When creating component, we need to create component's factory with parameters:

1. request factory

2. database session

Request Factory is creating new object - component request. Using its `create_request` function, which takes arguments - fully qualified name of parent component, component's name and default values. Created component request has attribute GET similarly to pyramid request object. Our GET attribute is a dictionary containing the GET variables related to the component. Component request's GET attribute contains only variables attending to him. This way, we have provided seperated GET variables for every component. Default parameter is used for filling in the default values for parameters which are not in GET dictionary.

Example:
We have a component with name "Child", which is subcomponent of component called "Parent", and its property "prop", pyramid request.GET would contain key Parent.Child.prop, but our GET contains key prop.

Component request has got public method called `route_path`. This method has arguments :

query
: Dictionary consisting of parameters needed to be added to URL. Setting a value of attribute "a" of subcomponent "child", of current component should be:
`"{'a': value}"`

anchor
: Anchor is simply added to end of the URL.

event
: Dictionary consisting of 'name' string and 'parameters' dictionary, containing parameters for event. It should be also only a string with name of event, that needs to be added to URL.
Example: Triggering event called "event_name" with parameters "x" and "y" with values "value1" and "value2" respectively should be represented as:
`"{'name': event_name, 'parameters': {'x': value1, 'y': value2}}"` or event can be simply a string with event name.

These parameters are repaired to a special form, which is unique and all parameters could be appended to url by pyramid request, where we append to all query parameters prefix "Parent.Child.". Event has its own name "event.name", and we can see a component which evokes this event in GET parameter event.component. All event parameters have got names event.parameter.(property). These are added to query dictionary and send to pyramid request's method `current_route_path`.

2.2.2 Components system

We had a problem how to handle POST/GET operations generated by web-forms. Components should have a possibility to handle events that they are producing. It is the most important for components with forms. They can produce a lot of events - for example: creating a draft ticket, updating user info, registering user, sending some amount of money to user, logging user on, etc... We thought about an uniformity in this case and adding this kind of functionality to components. All of the components have one object called ComponentEventManager with public functions listen and trigger.

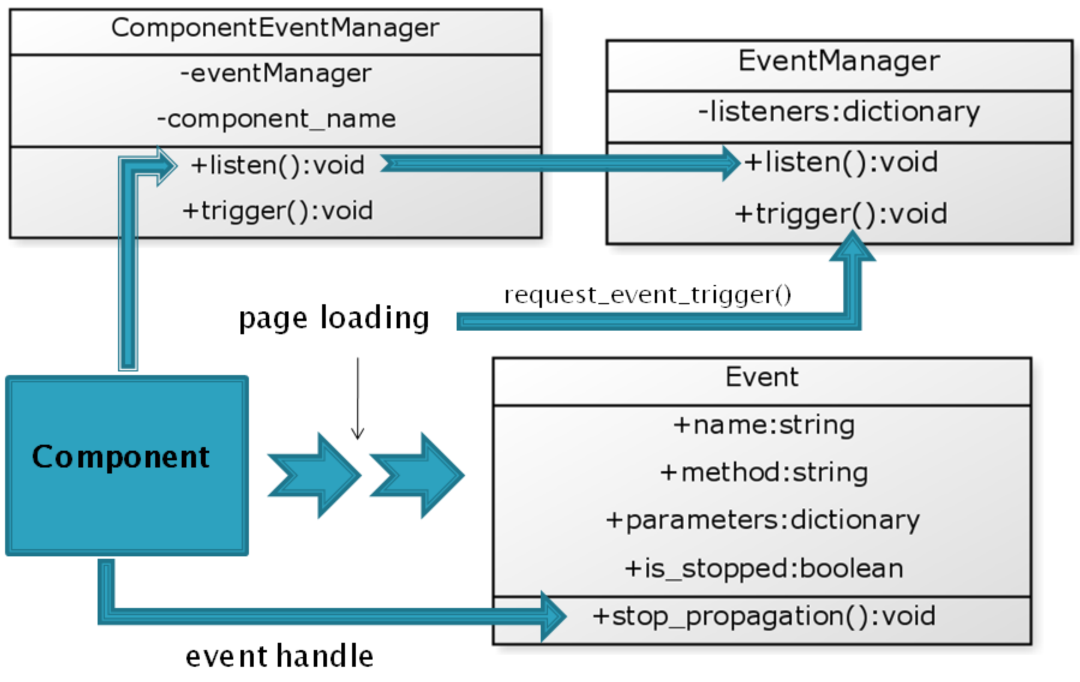


Figure 2.2: Components system

We have built a component system, where component can register to ComponentEventManager object handler function for some event. When the event will be triggered, all the handle functions component registered for event will be invoked to handle this event, until event is stopped or no other handle functions are present. Event is a special object with name, method and parameters. Event is created by EventManager's `trigger` function. This function is invoked by function `request_event_trigger`. It is called in views. Main role of that function is to trigger an event based on request. It has got attributes - EventManager and request. Informations are encoded in request's GET and POST. Method of event is received from request's method field.

request.GET parameters:

- .event.name - fully qualified event name
- .event.component - specify fully qualified component
- .event.parameter.{name} - Additional parameters in GET method, where "name" is concrete parameter's name.
  All the other parameters is set in request.POST

`request_event_trigger` invokes EventManager's trigger function. EventManager creates an Event object based on parameters set to its trigger function and look component in listeners dictionary. If there is any handler for this kind of event, it evokes all the handlers with created event as a parameter, until event is stopped. If there is no handler for event, EventManager tries to triggers event on parent component.

Component can also trigger event throughout ComponentEventManager too. Every component has its own manager to do these operations. ComponentEventManager has EventManager as its attribute, and use it to trigger some event. EventManager saves the handle function for the component and specified event into listeners dictionary.

Example Login Component: based on figure 2.2 (p.14)
We have got a login component in our application. In initialization, login component register its private functions _login and _logout as a listeners for 'log-in-user' and 'log-out-user' events. When user tries to log in with his email and password, page will be reloaded and in the view we evoke `request_event_trigger` function, which finds in request.POST special parameters as described above. `request_event_trigger` evokes EventManager's `trigger` function. This function creates an object Event with all necessary information as event name, event method and event parameters. This object passes to handler function _login as a parameter. _login looks into event.parameters and can use authenticator object to provide login operation. If everything is OK, login component provides redirection to a main page, otherwise the errors will be shown.

2.2.3 CRUD Component

Many of our components are used for editing or creating database objects. We can create match, league, sport, transactions, general settings, ... We have implemented a CRUD component. It is a shortcut for create, read, update and delete. Our component have got these arguments:

1. request - component request object
2. eventManager - component event manager
3. dbsession - database session object
4. form - smallform form
5. Model - class representing the model we want to operate with
6. model ID - id of model for operations update/delete. Possible value is None, if we are creating new object.
7. allowed operations - list of allowed operations [create,update,delete]

All our components used for working with database objects should inherit from CRUD component. It has functions like create, update, delete, set_values and validate. In initialization, parent CRUD component registers for the current component all the handlers for allowed operations. Since these operations are for security reasons, we could not allow user to delete object using special path. This way, we can allow only admin to delete or edit items. Creating function is used for creating new object. Object will be created only if process of validating function was successful. If not, new object will not be created. Smallform has its own validator, we are using it to validate some kinds of values (integer, string, range, input length,..). Also some of these values cannot be checked by a form, so we must do validation on our own. We are overriding validate function if it is necessary. We can override all of these functions in our component. When object is created, then `set_values` function will be evoked. If success, CRUD component adds a new message to flashes queue.

2.3 Internacionalization and Localization

Internationalization (i18n) is the act how to adapt application/software to different languages, to regional differences or different meanings of a phrase or of a word. By creating software we should think about its possibility to display given texts (system messages, errors, menu, flash messages, buttons labels, ..) in more than one language or cultural context. It is also about user's accessibility to our application. This is one of way how to attract potentional user/member. Users stays on website more time, if they understand the page's content, what the page is presenting. Localization (l10n) is a concrete way to serve user translated and understandable application. We are adapting application for specific countries, regions and languages. We are adding special locale for handling special situations - such as words bending, different meanings, adding suffixes to some words in plural forms.

2.3.1 Database backend

As described in first chapter, the page should be multilingual. You should be able to change page's language, items should be editable easily. Pyramid framework has built translation system. There is used `gettext` API. Actual pyramid's translation system is based on directory scheme, in which we can find translations. Translation directory includes languages subdirectories. Translation files are compiled, and it was too huge, robust and slow for us. We should have easy access for user how to change texts presented on website. We considered, it should be better to create our own translation system based on database, which should be faster as the system in Pyramid. We took some inspiration from Pyramid web framework how to create such a system for application translating. It was important feature to implement it to our application.

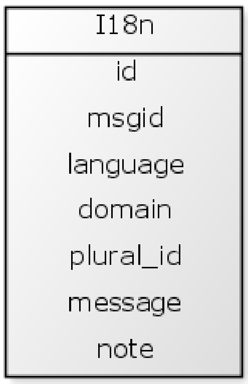


Figure 2.3: I18n

We want to create a database based translation, so we need a database object representing translation of current phrase. First, we need some key, because we must know, what we are going to translate. This keyword is using as argument when we are calling translating function. It is always the same. It will not change in the future. We need language column for considering the current language, domain is used to find-out the right meaning of the phrase. Phrases should have different meanings in different situations. For example, java is a coffee and also the programming language. Plural ID column will be used for considering plural form of the phrase. For example, in english we have got three plural IDs (You have not got any tickets. You have got 1 ticket. You have got 47 tickets). Message column is concrete translation for some msgid, plural ID, language and a domain. Note is additional field, if it is necessary.

All the translations are saved in the backend's private field - `translations` dictionary. Translations are always available in the memory, while application is running. This is, because we need to get translations really fast, without accessing database, which is a longer operation than index to associative hashmap. The key, we are indexing into this dictionary is a tuple consisting of msgid on the first place, language on the second one, plural ID on the third and domain as a last item of tuple.

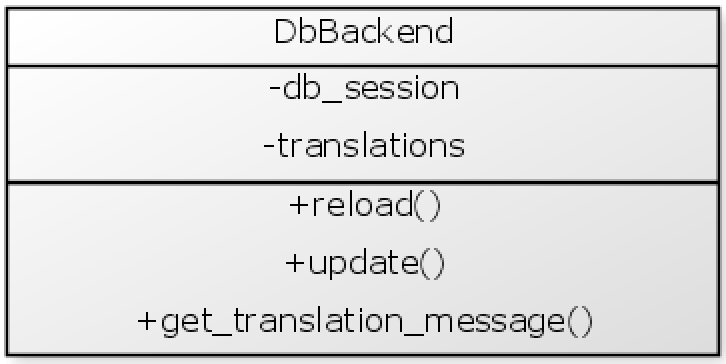


Figure 2.4: Database Backend

Database backend stores the translations in the dictionary and we can ask for some translation via its function `get_translation_message`. When no translation string was found, adds string to database and returns message ID, else returns translation of the message. It has also a function `reload`. After we call this function, all the translations in the dictionary will be removed and the dictionary will be filled by translations from database. It is useful after the server was restarted. We can also use `update` function of this backend, which takes arguments the tuple - key for the dictionary, and the value as a new replacement.

2.3.2 Localizer

Localizer is abstraction using database backend and customize translation process. Localizer is created by LocalizerFactory when the application starts. LocalizerFactory takes database session argument and the default language. In initialization it creates also Database Backend object. LocalizerFactory has got some handlers for different languages stored in the private field - `handlers` dictionary, which returns current plural ID for some number `n`.

Example of english plural handler:

```
def en_handler(self, count):
    if count == 0: return 0
    if count == 1: return 1
    if count > 1:  return 2
```

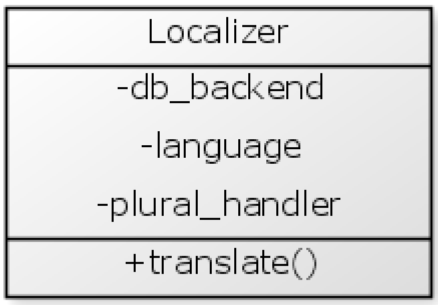


Figure 2.5: Localizer

LocalizerFactory has public function `get_localizer`, which returns Localizer as shown in figure 2.5. Localizer provides translating of some message. Its `translate` function takes arguments:

1. msgid - message ID of the currently translating string.
2. mapping - dictionary consisting of concrete values
3. language - language, I want translate to.
4. n - number of objects
5. domain - in consequence of context, there should be more options how to translate a string. Example: 'java' - as a programming-language or a coffee

With this way of translating we solved problem with very nearby strings. There would be too much messages as "John bet on match Slovakia - Germany 30 BTCs.". But thanks to mapping as an argument of translate function, we can translate strings "{user} bet on match {first_competitor} - {second_compentitor} {n} BTC". We translate this string with parameters mapping - {'user': 'John', 'first_competitor': 'Slovakia', 'second_competitor': 'Germany', 'n': '30'}, parameter n=30. After localizer asked its backend for translation, it provides also mapping with dictionary `mapping` from parameters.

2.3.3 Excel Model

The last component of translating system is Excel model. I have used openpyxl python's library for working with excel files. This helps us to generate list of translations generated in excel columns. This is user-friendly model of translating. Excel model can set translations from the excel files to database using Database Backend's update function. Also it can export translations to excel file - we can choose all the items or only non-translated items from the database table i18n. User can download the excel file with non-translated messages, translate them, and upload back to server by special form in administration section.

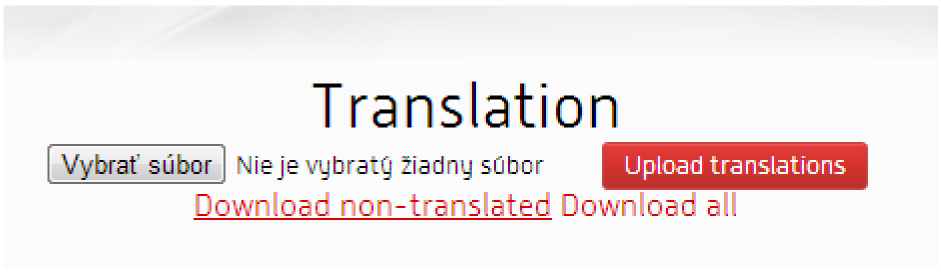


Figure 2.6: Administration: Translation

Chapter 3

Application's Architecture and Technologies

3.1 Database model

Database model includes complex application data structure. Its items are joined with some kind of relationships as One-To-Many, Many-To-One, One-To-One. Entities in database models are used in application. We realize an application analyze and got some database structure. We created entities for user base information `user`, user `member` entity, statistics entities, analyses, matches, sports, tickets, generalsettings for application settings, generalcontents, etc...

3.1.1 Users and account entities

We created two main entities for holding user information - information used while logging on - database entity `user` containing email address, hashed password via python's library bcrypt and One-To-One relationship to `member`. We wanted to seperate this two entities, because `member` entity is used when the logged-on user browses the web application, and we do not need user entity this time. `user` is used only for authentication and `member` is the entity, application is all the time using.

`user` entity contains columns ID, email, hybrid property password, private property _password, member One-To-One reference (it is a foreign key to `member` entity), and Many-To-Many list of user roles. `role` is a database entity containing ID and name of the role. This list is used in authorization process, its elements are references to Role objects. We need also 'active' column to determine, whether the user's account is activated. We created also validation of correctness actually added email address. In sqlalchemy framework, we can use `@validates('column_name')` decoration for the function which is providing validation. We created also associative table because of Many-To-Many relationships and declared indexes for fast database joining.

```
association_table = Table('association', Base.metadata,
    Column('user_id', Integer, ForeignKey('user.id'), index=True),
    Column('role_id', Integer, ForeignKey('role.id'), index=True)
)
```

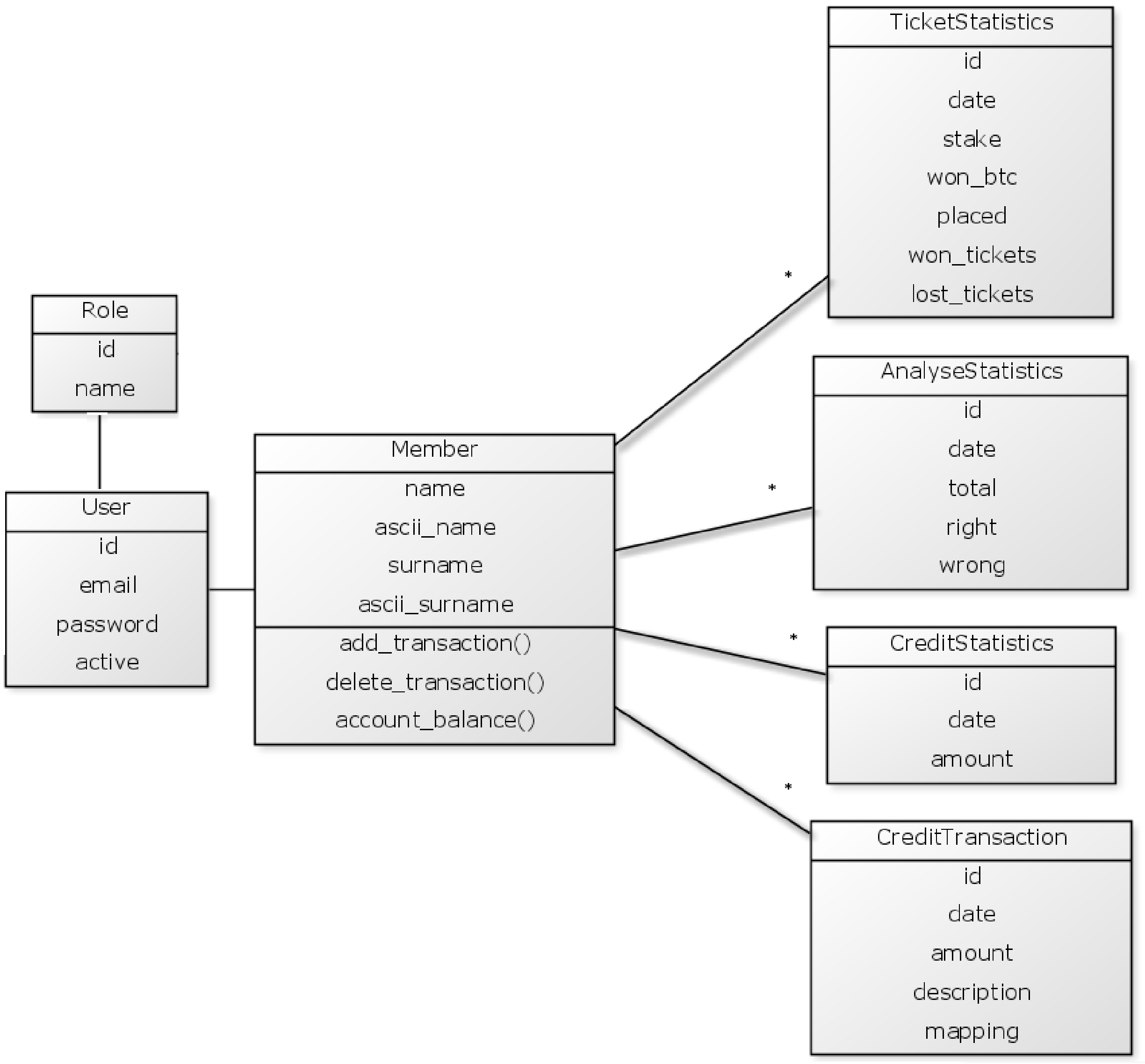


Figure 3.1: User database structure

We are holding a hashed password in database, so we needed to create a hybrid property, when we want to compare hashed value stored in database with text password in the python code directly. Hybrid value representing a hashed representation of the password. Class must inherit from Comparator class. It is used for comparing string instances with hashed values. In the class we are storing a string as a field hash_password. We provide operating with argument other as follows:

```
op(hashpw(other, self.hash_password), self.hash_password)
```

`member` database entity is used by application all the time. It contains all necessary account data, which we need to display and operate with. We decided to include these columns: reference to user database entity, name and surname, name and surname in ASCII, Many-To-One reference to `country` entity. We have also foreign keys to country and user entity. We added transactions list containing references to credit_transaction entity. Application uses also credit_statistics and analyse_statistics entities. These entities I will describe in next subsection.

3.1.2 Credit transactions & statistics

To manage user account actions we provided an database entity `credit_transaction`. We need to hold information about time the transaction was performed, amount which is send from account to place a ticket, and amount of money if the ticket was evaluated as correct. We need to send system information about this turnover to user. While system messages are translated, we need to use mapping on this message, so we need next column with mapping dictionary.

Member should be able to control his transactions. We added to this entity follow functions:

- add_transaction
  Takes an argument 'transaction'. It appends transaction to the list of credit transactions and update credit statistics information.
- delete_transaction
  Deletes the transaction in its list of transactions and updates credit statistics.
- account_balance
  Takes an argument date. If 'date' is None, the current month account balance will be returned. We search the list of credit statistics for selected month and return the amount of money.

Credit statistics contain for each month only one record with date - first day of the current month and year. This is because we must display user transactions history and account balance for some concrete month or year. Similarly we store information about analyses and tickets in entities `analyse_statistics` and `ticket_statistics`. From analyse statistics we are computing member's ROI number. We store total revenue, number of right, wrong and total number of analyses. It is necessary to hold total number of analyses too, because some analyses should not be evaluated yet. Ticket statistics cotain important information about tickets - total stake, total won BTC, number of placed tickets and number of won and lost tickets. Some tickets could also be non-evaluated. Figure 3.1 shows a structure of these entities.

3.1.3 Results and Odds

The important part of application structure is to analyse right database structure of results and odds, because there will be too much records in this database tables. We decided to create result database with all useful information. Result entity contains references to other entities ids as a foreign keys. IDs are indexes to those entities. Result contains:

- match
  Match for which we have got this result.
- result type
  Result type its a new entity reference, which contains name and short name of type of bet. This should be Asian Handicap (AH), European Handicap (EH), 1X2,...
- parameter
  Additional parameter (default is 0.0). It is needed for Over/Under or Handicaps.
- choice
  Choice of the outcome (integer) - differs on the result type

```
def render_choice(type_name, choice):
    if type_name == 'over/under':
        choices = {-1: 'Under', 1: 'Over'}
    elif type_name == 'double_chance':
        choices = {
            1: 'Home or Draw',
            2: 'Away or Draw',
            3: 'Home or Away'
        }
    else:
        choices = {1: 'Home', 2: 'Away', 0: 'Draw'}
    return choices[choice]
```

- part of match
  This field tells us on which part we tip the current result. Default value for full-time is zero.

For the database optimalization we needed to create index for this database table. Index contains columns match, result type, parameter, choice and part of match. In sqlalchemy we are using following syntax:

```
Index('ix_result_match_type_parameter_choice_part_of_match',
    Result.match_id,
    Result.type_id,
    Result.parameter,
    Result.choice,
    Result.part_of_match,
)
```

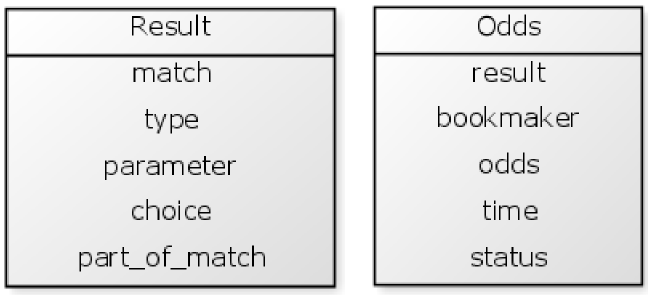


Figure 3.2: Odds and Results

Database table odds contains odds (float number) from bookmaker object. Bookmaker contains some basic information as its name, xml url, url to registration on bookmaker's website. Odds has also got reference to result object on which odds was created. We are storing time of last odds update and the status of odds - actual(default), old, canceled. We used enumaration for the status.

3.1.4 Matches, sports, regions, ...

We created database structure in the way that matches belong to leagues, leagues belong to regions and regions belong to sports. Leagues are the smallest unit. Each match must be a part of some league. All this database entities except matches has its own images. Image is a database entity which is refering to its variant. We need it to have an image stored as thumbnail and a full-sized image. ImageVariant entity contains file name, from which it reads the binary content. All of the entities has its own name.

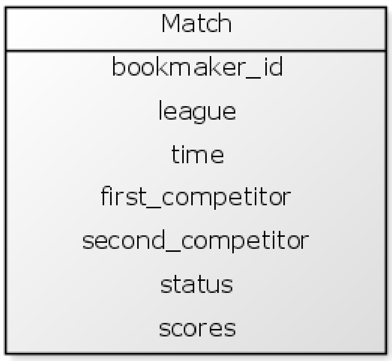


Figure 3.3: Matches database structure

Match contains also references to first and second competitors. They are items of Competitor entity. This entity concretes the name of the competitor. Scores is the One-To-Many reference to `score` entity storing information about number of goals for the first and second competitors for some part of match. Status is enumaration of values - available, ended, canceled. Default value is 'available'. Bookmaker_id is unique identifier of match used by primary bookmaker. This is important for results management.

3.1.5 Analyses, Articles and Tickets

Analyse database entity store information about user analysing a match. It stores reference to member which has created the analyse, time when analyse was submitted, time of evaluation, member's message and probability of outcome, elected odds and result. We set foreign keys on member, odds and result ids as indexes. We store also analyse resolution and revenue. Other entity we used in application development is `article`. It has got some date of publication, title, excerpt and text.

Ticket database structure is complicated. Every ticket must have at least one ticket bet. Ticket bet is a seperate database entity. Columns of `ticket_bet` table are:

- ticket - the ticket object to which ticket bet belongs
- odds - reference to odds object we are betting on
- state - should be win, lose, half_win, half_lose or refund in the case of canceled match

- final odds - final resolution odds for the bet after evaluating (refund, half_win, ...)

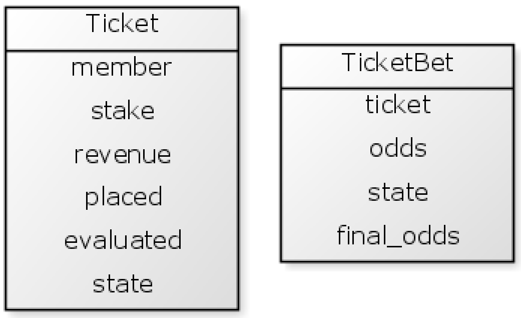


Figure 3.4: Ticket Database Structure

`ticket` columns evaluated and placed are datetime values. State is enumeration of values draft, filled and settled. 'settled' value tells us, that ticket was placed and all the ticket bets refering this ticket was evaluated, while ticket with 'filled' state is placed, but it has at least one non-evaluated bet refering the ticket. When ticket has set state 'draft', it represents that ticket is already creating and filling with bets.

3.1.6 General entities

Application has got two general entities. `general_settings` and `general_content`. `general_settings` is the database table with settings for whole application. It has some setting's name and a value. With this settings we can set values:

- credits per day
- credits per month
- winnig maximum
- registration bonus
- maximum placed tickets per day

We can not add a new settings only change them. It is the same with `general_content`. It stores longer texts used on website as privacy policy, terms of use, etc ...

3.1.7 Database optimalization

Because of the slow database queries we had to provide database optimalization. We had to create indexes on database entities. It's an optimalization on postgress side. This access is several times faster than the queries to database entities without indexes. The most required was indexes for entity Result. We had to add indexes as I described above in database structure of Result entity. Also all foreign keys, which we had defined are also indexes to other database entity. Match was extended with index consisting of `first_competitor` and `second_competitor`.

The table shown below represents duration of execution SQL query without index on result database table on the same package of data. This number I retrieved using postgres function called ‘explain‘.

Result without index	Result with index
0.4711s	0.2053s

Table 3.1: Result optimalization

We had to use indexes on Result table as it is shown above in database model section. We picked these indexes, because we needed to perform more queries which are using left outer joins on result table on these columns. These columns are frequently compared in our queries and used for sorting in `order by` clause. We've got queries similar like in this example with some concrete values instead of questionmarks.

```
SELECT * FROM odds
    JOIN result ON result.id = odds.result.id
    JOIN match ON match.id = result.match.id
    LEFT OUTER JOIN result ON result.id = odds.result_id
    LEFT OUTER JOIN match ON match.id = result.match_id
    LEFT OUTER JOIN score ON match.id = score.match_id
    LEFT OUTER JOIN result_type ON result_type.id = result.type_id
WHERE odds.bookmaker_id = ? AND odds.status = ?
      AND result.id IN (?, ..., ?)
ORDER BY match.time ASC, match.league_id ASC,
         result.choice ASC, score.part ASC
```

We also found out, that sqlalchemy's construction for queries should be also optimized. We loaded items only in lazy mode. This is a default in sqlalchemy for all inter-object relationships. That means, that the attributes of queries was not loaded. We had only references to them. If we asks for them, the sqlalchemy created a new query, new database connection. While we needed to work with entity attributes widespread, it was really unefficient because of many new database connections for database objects.

We solved it, when we found out that sqlalchemy supports options, which we can set for loading. In the sqlalchemy queries optimalization we used an option which brought us to load inter-object relationships in the same time as the parent in a single SQL query. This type of query building uses 'left outer join' for creating queries. This option is `joinedload`, or `joinedload_all`. Other option `subqueryload` we used, is kind of eager loading. It emits additional SQL statement for each collection we need to load. [que]

After all optimalizations have been done, we noticed the application is running at least 30-times faster than before.

3.2 XML Parser

We are getting odds from bookmaker in XML file. We need to create offer and result XML parser to create/update sports, regions, odds, results, etc ...

3.2.1 OfferParser

We created class for parsing offers OfferParser. It is parser of offer feed from bookmaker. Parses the offer xml and creates in database corresponding sport, region, leauge, competitor, match, odd and result objects.

During the parsing process, parser needs to identify matching objects (sports, regions, leagues, matches)in XML file with existing objects in database. Due to lack of common unique identifier for database and various xml files from different bookmakers, we need to do matching heuristically based on the objects human friendly names. To resolve some ambiguities (Pressburg is just another name for Bratislava), we use the identity map that defines synonyms for each name. Matching criteria for each of the objects are described below. Two objects match when:

- Sports - Their names are synonyms.
- Regions - They belong to the same sport and their names are synonyms.
- Leagues - They belong to the same region and their names are synonyms.
- Competitors - Their names are synonyms.

- Matches - They belong to the same league, they have same competitors and the same date.

Parser has got dictionary with tuples market codes and action. To do regexps more readable we are mapping on the string regexp dictionary with regexps of integers and floats matching. Action returns us dictionary with keys ‘market_type‘ (asian_handicap, 1x2, double_chance, ...), ‘parameter‘, and ‘part of match‘ that can be read from market code.

Example of tuple consisting market code pattern and action for over_under:

```
('goal{integer}p{integer}$', lambda groups: {
                                    'market_type': 'over_under',
                                    'part_of_match': int(groups[1]),
                                    'parameter': int(groups[0])+0.5,
                                    }),
```

We created common function `_parse_common` for parsing elements like sports, regions and leauges. The function assumes the following structure of parsed document:

```
<elements>
    <element id=103>
        <parentID>45</parentID>
        <name>Lorem</name>
    </element>
    ...
    <element id=808>
        <parentID>47</parentID>
        <name>Dolor sit amet</name>
    </element>
</elements>
```

Parsed elements are looked up in database and stored in the ‘element_map‘ under the id found in XML. The lookup is done with respect to element <name> tag value and <‘parent_id_tag‘> tag value. Name matching is done using ‘self._identity‘ identity map. Parent matching is done using ‘parent_map‘. If the element is not found in the database, it is created using provided factory method “create_element“ and inserted into the database.

Arguments of the `_parse_common`:

- element_map
  dictionary mapping XML ids to element objects from database
- element_class
  class used to query database for an element
- container_tag
  tagname containing elements, it is "elements" in our example.
- create_element
  factory method for creating elements. It is invoked with two named parameters: name of the element as "name" and element parent as value of ‘parent_reference‘.
- parent_map
  dictionary mapping XML ids to parents objects from database.
- parent_reference
  name of field in ‘element_class‘ used to store the parent reference
- parent_id_tag
  name of the XML tag containing the parent (XML) id. In our example it is "parentID"

Example of `_parse_regions` function using `_parse_common`, that finds all regions in the XML file and store the information about them to the ‘self._regions‘ dictionary. If the region does not match any existing region in database, creates it.

```
self._parse_common(
    element_map=self._regions,
    element_class=Region,
    container_tag='regions',
    create_element=lambda name, sport:
                      Region(name=name, sport=sport, image=Image()),
    parent_map=self._sports,
    parent_reference='sport',
    parent_id_tag='sportId',
)
```

The only thing we must do, is to parse events - create matches and selections or update them. We are looking for <events> element in XML tree and parse events. Each event has got markets. `_parsing_selections` will create new Odds objects even if there are already some, unless they possess the same odds rate. If the rate differs, new Odds object is created and marked as "actual", all previous Odds are marked as "old".

3.2.2 ResultParser

Parser of result feed from bookmaker. Parses the result XML and creates in database corresponding score objects. During the parsing process, parser uses ‘Match.bookmaker_id‘ to identify matching events. The ‘Match.bookmaker_id‘ is id of event as used by bookmaker in XML file.

Parse scores for events in XML file and create/update match scores. If the scores are created/updated, (re)evaluate tickets and analyses for appropriate matches.

The events has following structure in XML file:

```
<events>
    <event id="4538">
        <leagueCupID>684</leagueCupID>
        <name>Bratislava vs. Helsinki</name>
        <startDateTime>2012-09-26T20:00:00</startDateTime>
        <score>3:3 (0:1, 3:2, 0:0) (0:1 $OT$)</score>
    </event>
    ...
    <event id="4638">
        <leagueCupID>684</leagueCupID>
        <name>Budapest vs. Wien</name>
        <startDateTime>2012-09-27T20:00:00</startDateTime>
        <score>0:3 (0:1, 0:2)</score>
    </event>
</events>
```

Score consists of several partial scores (brackets [] symbolize optional parts of score string):
"‘total_score‘ [(‘part1_score‘, ‘part2_score‘, ..., ‘partn_score‘)] [(‘overtime_score‘ OT)]"
Each partial score has a form: "“home_points“:“away_points“" where points is integer. Total score does not include overtime.

We created function for creating regexps for matching score, parts, overtime and scores. This function is used to match values in right element in xml tree.

```
def _create_result_regexps():
    """Create regular expressions used to match scores by ResultParser"""
```

```
regexps = {}

regexps['score'] = '(-?[0-9]+):(-?[0-9]+)'
regexps['parts'] = "{score}(?:, {score})*".format(**regexps)
regexps['overtime'] = "{score} \$OT\$".format(**regexps)
regexps['scores'] = \
    "(?P<regular>{score})\s?(\((?P<parts>{parts})\))?\s
        ?(\((?P<overtime>{overtime})\))?".\
        format(**regexps)

return {key: re.compile(value) for key, value in regexps.items()}
```

3.3 Session section

Session Section object represents a part of session that has got own timeout. Session section behaves like an associative array or a dictionary. To reproduce this behaviour we needed to add some functions that are evoked, when we index to such a structure or we are adding new value:

- `__setitem__` - it is evoked when assigning value to dictionary
- `__getitem__` - this function is used when indexing to the dictionary
- `__delitem__`
  deleting an item from dictionary (**del** dictionary[item])
- `__contains__`
  looking for some key value in dictionary's keys (item **in** dictionary)
- `__len__` - number of dictionary items (**len**(dictionary))

These functions are working with private field session_section, which we got from SessionSectionFactory. Factory class holds dictionary of used sections. If we asks `get_session_section` for some name and timeout, factory looks out the dictionary, if this session section exists, returns it, else returns newly created section from pyramid session object. All the functions that are modifying indirectly session content, must call `session.change` that is passed into Session Section by initialization. Than we can access last_accessed field to get know if timeout expired. As a consequence of fact we need to have timeouted sessions, factory do section expiration before getting a session section from a factory, if timeout expired.

3.4 Ticket Manager

Ticket Manager creates us friendly interface to work with tickets. In the background it provides all necessary controls and throws useful kinds of self created exceptions. It contains functions:

- `add_bet(ticket, odds)`
  Raises exceptions when ticket is not in draft, odds are not in state actual, match status is not available or match has already started, or we want to bet on the same match. Else it creates new TicketBet object on elected odds.

- `remove_bet(ticket_bet)`
  Controls if the ticket is in draft state, if yes, the ticket_bet will be deleted.

- `delete(ticket)`
  Ticket must be in draft state. Then all of the ticket_bets and ticket will be removed.

- `place(ticket)`
  We need to carry out several controls to place ticket, else useful exception is raised. Ticket must be in draft state, all the odds on bets must be actual. All matches must be available and cannot start before placing ticket. Member must have enough money and the stake must be positive float number. Member can not exceed the maximum placed tickets per day stored in general settings. After all controls was sucessful the transaction is created and ticket state is changed to the 'filled' state.

3.5 Revenue calculator

Calculates an actual revenue and return revenue and resolution state. Possible resolution states are 'win', 'lose', 'half win', 'half lose' and 'refund' in case of canceled match. For each result type we have calculate functions in result_types dictionary. All the functions take parameters first score, second score, tip, param and odds. This is the way how we are calculating odds for these supported result_types.

- 1x2
  winning condition: `(_cmp(first, second) - tip) % 3 == 0)`

- asian_handicap
  Parameter number (without quarters) tell us, how many goals is handicap of first team. There are some types, how we calculate final odds: 0, 0.25, 0.5, 0.75 (tip = 1)
  win - if my team win after goals recalculation

lose - if my team lose after goals recalculation
0 - refund
0.25 - you deal your stake between 0 and 0.5 (+-0.25)
0.75 - you deal your stake between 0.5 and 1 (+-0.25)
For dealing stake if you are refunded in dealing tips and you won one of them your resolution is 'half win' and you won `(1+odds)/2` odds, else your resolution is 'half lose' with odds `(1+0)/2`

- european_handicap
  it is same as 1x2 after recalculation of goals.

- double_chance
  We can tip on first team win, on the second team win, or one of the teams win. If a deal fall, then user is refunded with `1` odds.

- draw_no_bet
  If a deal falls, then user is refunded, else it is same as 1x2.

- over_under
  winnig condition: `_cmp(first + second, param) == tip`

3.6 Server side technologies

In this section, I will describe the most used technologies, we were using by application developping.

Python - Pyramid

We decided to use a Python language for writing the application. It is a strong language, we can override many methods how should some functions behave. Python has got also many libraries we can use in the application.
Pyramid web framework is:

- fast
- documented
- tested
- flexible
- simple

MAKO

MAKO is a python library. It is using as a templating system with non-XML syntax. We used it because of its performance and easy use. Its code is compile into python modules. Mako is extremely efficient, it uses standard python controls as conditions and loops (for, while, if, ...). We can define blocks and inherit it in other templates. This way we can create 'controls' for templating base page content as header and footer. [Doc]

Example:

```
% for member in members:
    <tr class="${'even' if loop.index % 2 else 'odd'}">
        <td>${member.name} ${member.surname}</td>
        <td>${member.account_balance()}</td>
    </tr>
% endfor
```

Mako helps us to create templates flexible to page content.

SQLAlchemy

SQLAlchemy is toolkit and object relational mapper. It is a toolkit for Python programming language. It is really important, when we are developing some website to do effective operations with database and have the control about all database tables, and that is what SQLAlchemy is good at. Why not we use MySQL? The reason is that we can do our database tables represented as objects with attributes. We can also relate one object to another quite easy with foreign keys to another table. We have something like query-builder for it, where we can write more clearly code. [SQLa]

```
database.query(User).\
    filter(User.name == 'John').\
    filter(User.city == 'Bratislava').\
    order_by(asc(User.surname)).\
    all()
```

The piece of code, which I showed you above, represents idea of getting all users with name 'John' which are from 'Bratislava' from database table `User`. This query will return as a result a list of these users ordered ascending by surname. [SQLb]

www.yuml.me

Web applications, where I have been creating UML diagrams images for this thesis.

TDD - UnitTest & Mock

While we were trying to write application in Tested Driven Development (TDD) way, using UnitTest (PyUnit) was familiar choice. Using PyUnit was simple. We used also Mock from mock library for creating dummy object with our-defined behaviour. We needed to prepare also a testing database. We created two test classes. First test class is BaseTestClass, which inherits from unittest.TestCase. We made it to have non-required `setUp` and `tearDown` functions as it is in unittest.TestCase class. Also we have created a test-class that needs testing database. This class inherits from BaseTest-Class. At the setup it prepairs connection to the database and creates database session object. After that it calls setUp function of the superclass. It is done similarly with tearDown function with the difference. tearDown calls child's tear_down method and rollbacks the transaction.

To test factories we had to create also our own mechanism. We were mocking classes with special class that takes as argument class we want to mock. Result of a call such mocked class is mocked object with attributes `_original_class` (name of the original class) and `_init_parameters` (parameters used, when creating new object of original class).

```
class ClassMock():

    def __call__(self, *args, **kwargs):
        """Create mocked constructor that returns
           Mock object when invoked."""
        instance = Mock()
        instance._original_class = self._original_class
        instance._init_parameters = inspect.getcallargs(
            self._original_class.__init__, self, *args, **kwargs
            )
        del instance._init_parameters['self']
        return instance


    def __init__(self, original_class):
        """Initialization of the class."""
        self._original_class = (original_class
            if not isinstance(original_class, ClassMock)
            else original_class._original_class
            )
```

Conclusion

In this thesis, I wrote fully-qualified project specification, containing the feature it includes. I described it in detail. Then, I described the application security using `pyramids.security` package, from which I used Authentication and Authorization objects. I described both of these objects functionality and how to use that. Next I focused on implementation of Component system. This system brought us easier event handling, handling form parameters and triggering our event. Thanks to component system we separated modules. We created also CRUD component, which helps us to create, update or delete objects. It also uses component system when handling forms and triggers 'successful' events.

We described database structure and indexes that must be created to be the application faster. We had some problems with database optimalization but thanks to indexes and database architecture it is really fast. Non-trivial was parsing huge XML files from bookmakers. We used `etree` python library and regexps. We separated XML Parser to OfferParser and ResultParser.

At the last, we described revenue calculating and using of session section with timeout to expiration, which is used i.e. for password recovery. Thesis ends with important technologies we used by developping this application.

We accomplished the application planning and constructing successfuly, with all the features, database entities and also application's background behaviour. Application's background behaviour should be also used in other projects because it is seperated from the application and gave us lot of programming experiences and skill. I learnt how to develop such a huge application with many features and many database records.

The application can be improved by SEO (search engine optimalization) and test usability of the application in the future. From the functionality extensions I can image the application for Android or iOS. Possible way should be connecting application with social networks and sharing analayses and tickets. Also there could be email invitations from registered users.

I hope, that this thesis will help someone to plan some other application. I think, reader should find here some good and useful instructions or tips to construct robust application.

Bibliography

[Doc] Mako templates for python. http://www.makotemplates.org/.

[Doc13] pyramid.security. http://docs.pylonsproject.org/projects/pyramid/en/1.0-branch/api/security.html, January 2013.

[Odd] Oddsportal. http://www.oddsportal.com.

[que] Using loader strategies: Lazy loading, eager loading. http://docs.sqlalchemy.org/en/rel_0_7/orm/loading.html.

[Rou07] Margaret Rouse. Authentication. http://searchsecurity.techtarget.com/definition/authentication, June 2007.

[SQLa] The python sql toolkit and object relational mapper. http://www.sqlalchemy.org/.

[SQLb] A step-by-step sqlalchemy tutorial. http://www.rmunn.com/sqlalchemy-tutorial/tutorial.html.