

UNIVERSITY OF PISA

Large Scale and Multi-Structured Databases

*Year 2019/20*

**OrgaBET**

*An aggregator website for sport bets, powered by MongoDB*

*GERARDO ALVARO*

*SILVIA NOIRJEAN*

*RICCARDO POLINI*

*GIULIO SILVESTRI*

**INDEX**

|  |  |  |
| --- | --- | --- |
| 1. INTRODUCTION...…………………………………………………………. | | 1 |
| 2. REQUIREMENTS ANALYSIS…...………………………………………… | | 2 |
|  | 2.1. Application Actors.....…………………………………………… | 2 |
|  | 2.2. Functional and Non-Functional Requirements……………….. | 2 |
| 3. UML DIAGRAMS……...….....……………………………………………… | | 3 |
|  | 3.1. Use-Case Diagram…..…………………………………………… | 3 |
|  | 3.2. Class Diagram…………………...……………………………….. | 4 |
| 4. ENTITIES AND DATABASE ORGANIZATION…................................... | | 5 |
|  | 4.1. Main Entities.…………………………………………………….. | 5 |
|  | 4.2. E-R Diagram of the DB….……...……………………………….. | 5 |
| 5. SOFTWARE ARCHITECTURE……………………………………………. | | 6 |
|  | 5.1. Repository Structure…………………………………………….. | 6 |
| 6. INSTRUCTION MANUAL.………………………………………………... | | 7 |
|  | |  |
|  |  |  |
|  |  |  |
| 8. CONCLUSIONS...…………………………………………………............... | |  |

1. **INTRODUCTION**

*OrgaBet*, acronym of Organize your Bets, is a web-application that acts as an aggregator for sports betting odds, suggesting to the user which bookmakers offer the best odds for the desired events.

When a visitor lands on the homepage, he is prompted to log-in (sign up first if he is not registered) in order to use the functionalities of the application.

Once logged in, the user is presented with a homepage containing a default match list of the day. The application will have a side panel in which the sport and competition may be selected by the user in order to display all the playable matches for the current date, with the respective average odds. If the user wishes to bet on a certain result, he/she will select the desired odd and the event will be added to “My Coupon”.

“My Coupon” is a recap of all selected events and will be displayed in a side panel. The list will show the total odds for each available bookmaker (sorted by convenience) so that the user may choose the one to bet on. The coupon may be saved by the user: if this happens, the list is saved in an archive accessible through the personal profile.

Some sports statistics are also available in a specific panel, accessible to users. Here one can browse various analytics about Teams and Players of different sports and divisions.

1. **REQUIREMENTS ANALYSIS**

**2.1. Application Actors**

The actors of the application are the *User* and the *Admin.*

The first one is the main user of the application.

The second, has the same functionalities as the regular user but also has additional responsibilities available only to an administrator of the website.

**2.2. Functional and Non-Functional Requirements**

The *functional* requirements of this application, divided with respect to the two actors, are as follows:

* The application is available only to registered users.
* The application acts as an aggregator for sports betting odds.
* For each sport, the application presents all the events of the day for the available competitions in the available nations.
* For each event, the application displays the average odds of the available bookmakers.
* For each sport and division, the application presents statistics and analytics relative to the previous years.
* A User selects a result to bet on and this is added to “My Coupon”.
* A User may remove a selected result from “My Coupon”.
* A User may clear “My Coupon”.
* For each element added to “My Coupon”, the application shows the most convenient bookmakers for betting on the selected events, if any.
* A User may save the “My Coupon” in the personal profile.
* A User can browse the saved coupons in his profile
* A User may delete his/her profile.
* A User may edit his/her personal profile.
* A User may remove a previously saved coupon from the personal archive.
* An Admin can view the profiles of registered users, including their coupon archive.
* An Admin can ban users who violate the Terms & Agreements.
* An Admin can add matches to the collection.

The *non-functional* requirements of the application are:

* The application’s interface must be user-friendly.
* The application must have a low response time.
* The application will store information in a non-relational Database (MongoDB).
* The application must guarantee data availability.
* The application must ensure prevention against server crashes thanks to replicas.
* The application, thanks to sharding, allows the possibility to store more data and handle more load without requiring more powerful machines.
* The application must be easily scalable.
* The application must be reliable: no system crashes, exceptions are handled etc.
* Admins of the application periodically monitor the behaviour of Users in order to guarantee that they comply to the Terms & Agreements.

1. **UML DIAGRAMS**

**3.1. Use-Case Diagram**

See the linked PDF file for a high-resolution version of the diagram.

**Immagine che contiene testo, mappa

Descrizione generata automaticamente**

*Figure 1: Use-Case Diagram*

In the above figure is reported the Use-Case diagram in which we can see: the actors of the application, *Admin* and *User*; their respective action lists and some notes to specify when an action is available only to the admin or only to the user (if it is not already obvious from the diagram).

**3.2. Class Analysis Diagram**

In *Figure 2* are reported the main entities of the application and the relationships among them. *User* and *Admin* are generalized into *Person*.

**Immagine che contiene cielo, testo, mappa

Descrizione generata automaticamente**

*Figure 2: Class Diagram*

**­­­­**Each user can create one or more coupons, each of which will be composed of one or more odds and will generate a list of *BookmakerTot* which represent the total multiplier for each available bookmaker.

Odds are characterized by a bookmaker, a type of bet and a unique quote. All matches are made of two players with the relative score and other fields such as the sport, the league, the timestamp and the outcome of the game.

1. **ENTITIES AND DATABASE ORGANIZATION**

**4.1. Data Model**

|  |
| --- |
| **<User>**  {  username:  password:  email:  name:  surname:  roles:[]  banned:  coupons: [{  id  date:  bookmakerTot: [{  bookmaker:  quoteTot:  }]  bets: [{  matchId:  homeTeam:  awayTeam:  result:  avgOdd:  quotes: [{  bookmaker:  odd:  }]  }]  }]  } |

The roles are provided by a collection used by Spring Security to authenticate users and restrict access to some functionalities (admin only ones).

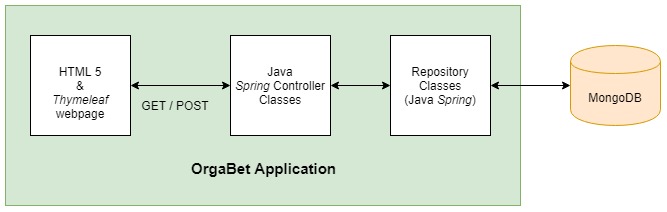
|  |
| --- |
| **<Match>**  {  sport:  division:  date:  time:  homeTeam:  awayTeam:  fullTimeHomeScore:  fullTimeAwayScore:  fullTimeResult:  odds: [{  type:  quotes: [{  bookmaker:  odd:  }]  }]    //other sport-dependant data in our dataset  } |

1. **SOFTWARE ARCHITECTURE**

*OrgaBet* is a web application programmed with the use of the S*pring Framework*, an application framework for the Java platform.

*Spring* uses a Model-View-Controller (MVC) paradigm where the *view* layer is handled by *Thymeleaf*, a template engine fully integrated with *Spring*. *Thymeleaf* allows the programmer to extend the functionalities of HTML5 tags by adding new types of tags and options fully accessible by the associated java-written controller class.

The back-end is composed of a *MongoDB* document database which is used by *Spring Data*, anative MongoDB driver for the *Spring Framework*.

**

*Figure 3: Software architecture of OrgaBet*

**5.1. Repository Structure**

*OrgaBet* is a *Maven* project. The project repository is organized as follows:

* **./ :** contains the *makefile* and the *POM* file used to generate the maven dependencies and build the project.
* **src/main/java/com/example/Orgabet:** contains all the source files of the application.
  + **/controller:** contains the controller classes.
  + **/repository:** contains the *MongoDB* repository classes, used to store/retrieve data from the database.
  + **/dto:** contains the Data Transfer Object classes.
  + **/models:** contains the Java classes that reflect the entities of the application.
  + **/config:** contains configuration classes used to implement functionalities of the *Spring Security* extension framework like password encoding and customized Authentication handler.
  + **/services:** contains functions used to build authorities on registration and authenticate users.
* **src/main/resources:** contains the subdirectories *static* and *templates* which hold all the HTML, CSS and image files used in the application.

1. **INSTRUCTION MANUAL**

Installation

After downloading *BibliOS* on a UNIX-like system, import the Database schema **bibliosDB.sql** in a MySQL Server application. After this operation, open the terminal inside the project folder.

If you do not already have *maven* installed, make sure you have a working internet connection and run the following command:

* **sudo apt install maven**

*BibliOS* requires **java** (version 11 or higher) and **javafx** (version 11 or higher) to run, install these if you do not already have them.

After installing all the necessary packages, you can run *BibliOS* by executing the following command:

* **make**

1. **AGGREGATIONS AND INDEXES**

**7.1. Aggregations**

In *OrgaBet* there are two pages which show to the user the main informations he can exploit to create his coupon: the first one contains the matches on which he can bet with the relative odds, divided by sport and division; the second one contains the statistics computed for the last three years about all the team (or tennis player), divided by sport and division. Both times these informations are computed using MongoDB Aggregation pipelines.

|  |
| --- |
| **<Compute Odds Function>**  @Override  public List<AvgDTO> computeAverageOdds(String id) {  MatchOperation filterMatch = Aggregation.match(new Criteria("id").is(id));  UnwindOperation unw = Aggregation.unwind("odds");  UnwindOperation unw2 = Aggregation.unwind("odds.quotes");    GroupOperation grp = Aggregation.group("odds.type").avg("odds.quotes.odd").as("avg");    Aggregation aggr = Aggregation.newAggregation(filterMatch, unw, unw2, grp);  List<AvgDTO>res=mongoTemplate.aggregate(aggr,Match.class,AvgDTO.class).getMappedResults();    return res;  } |
|  |

In the Matches page there is a function called to compute the average odds for every bet type and every match. The following code describes the aggregation:

The function computeAverageOdds filters the match collection using the match id in order to find a single match (Aggregation.match(new Criteria("id").is(id))). Then it used the undwind operations to deconstruct the array field “odds” and “odds.quotes” in the input document to output a document for each element, each output document has the value of the array replaced by a single element (Aggregation.unwind("odds")). The group operation (Aggregation.group("odds .type").avg("odds.quotes.odd").as("avg")) group the output documents from the previous aggregation, according to the bet type ("odds.type"), and compute the average odd for each bet type.

Finally the aggregation pipeline is created using an Aggregation object (Aggregation aggr = Aggregation.newAggregation(filterMatch, unw, unw2, grp)), it is applied on the match collection to get the desired result (List<AvgDTO>res = mongoTemplate.aggregate(aggr,Match.class,AvgDTO.class).getMappedResults()) and the function returns it.

In the Statistic page the application shows the team statistics, regarding the winning percentange and the average odds about the different bet type in home or away matches.

The function computeTeamHome receives in input a team name and the number of home matches regarding the team in a certain year (received in input) and returns a data transfer object (a StatsDTO object), previously created to store all the statistic we are interested in. A StatsDTO object contains: the team name, the winning, drawing and losing percentage, the percentage of over/under matches played and the average odds for each bet type.

|  |
| --- |
| **<Home Statistic Function>**  @Override  public StatsDTO computeTeamHome(String division, String team, Double totHome, String sport, String date) {  …  MatchOperation filterDiv = Aggregation.match(new Criteria("division").is(division));  MatchOperation filterTeam = Aggregation.match(new Criteria("homeTeam").is(team));  MatchOperation filterHomeWin = Aggregation.match(new Criteria("fullTimeResult").is("H"));  MatchOperation filterDate = Aggregation.match(new Criteria("date").regex(date+"$"));    GroupOperation grpHW = Aggregation.group("homeTeam").count().as("count");  Aggregation aggr;  if(sport.equals("Football"))  aggr = Aggregation.newAggregation(filterDiv, filterTeam, filterDate, filterHomeWin, grpHW);  else aggr = Aggregation.newAggregation(filterTeam, filterDate, filterHomeWin, grpHW);    List<countDTO>res=mongoTemplate.aggregate(aggr,Match.class,countDTO.class).getMappedResults();    try {  homeWin = ((res.get(0).getCount())/totHome) \* 100;  } catch(Exception e){}    …  ProjectionOperation prjHO=Aggregation.project("id").and("fullTimeHomeScore").plus("fullTimeAwayScore").as("totScore");  MatchOperation filterHomeOver = Aggregation.match(new Criteria("totScore").gt(2));  GroupOperation grpHO = Aggregation.group().count().as("count");    Aggregation aggr4;  if(sport.equals("Football"))  aggr4=Aggregation.newAggregation(filterDiv,filterTeam,filterDate,prjHO,filterHomeOver,grpHO);    List<countDTO>res4=mongoTemplate.aggregate(aggr4,Match.class,countDTO.class).getMappedResults();    try {  homeOver = ((res4.get(0).getCount())/totHome) \* 100;  } catch(Exception e){}    …    stats = new StatsDTO(team, homeWin, homeDraw, homeLost, homeOver, homeUnder, res5);    return stats;  } |
|  |

In the previous code we report an extract of the computeTeamHome, in particular the aggregation to compute the winning percentage and the one to compute the percentage of over matches. In each of these aggregation we want to filter the match collection basing on division and year (filterDiv, filterDate).

In the first aggregation examined, the function filters the collection basing on the homeTeam (Aggregation.match(new Criteria("homeTeam").is(team))) and then, exploiting a group operation, it calculates the number of match won by the team we are interested in (Aggregation.group("homeTeam").count().as("count")).The aggregation pipeline is created and applied on the match collection and, finally, the function compute the winning percentage (homeWin = ((res.get(0).getCount())/totHome) \* 100;).

In the other aggregation reported, the function exploit a ProjectionOperation to calculate the total points scored in a match:

Aggregation.project("id").and("fullTimeHomeScore").plus("fullTimeAwayScore").as("totScore"));

So it can filters the resulting collection selecting only the matches in wich the total score is more than 2.5 (Aggregation.match(new Criteria("totScore").gt(2))) and then counts the number of these matches (Aggregation.group().count().as("count")). Finally, like in the previous aggregation, the function compute the percentage of over matches (homeOver = ((res4.get(0).getCount())/totHome) \* 100;).

1. **CONCLUSIONS**

The proposed application is only provided with the main features requested for this task however, it would be possible to implement other functionalities that a realistic context would otherwise require such as: …