COMP5347 Web Application Development
Group Assignment Technical Report
Group: 6

Tutor: Johan Alibasa
Liang-Chun Yu (480317999)
Ruotao Lin (480509930)
Jinvara Vesvijak (470500387)

1. Abstract

In this assignment, our task is to build a small data analytic web application to analyze the individual and overall articles of Wikipedia and their authors.

2. Introduction

The main page of the application is where the user can see and implement the analytics functionalities. But the user has to Sign-up or Login before being able to access the main page. This is done by typing http://localhost:3000 in the browser, after successfully logged-in, the page is redirected to http://localhost:3000/main where the main page is accessed. The functions are divided into overall article analytics, individual article analytics, and author analytics.

In the overall article analytics part, we separately show titles of the two articles with the highest and lowest number of revisions. The user is allowed to change the number of articles shown. We also individually show the title of the articles edited by the largest and smallest group of registered users. Lastly, we show the top 2 articles with the longest history and an article with the shortest history, by calculating the duration between now and its creation time. A bar chart and pie chart is provided, showing the distribution of revisions by year and user type, and the revision number distribution by user type. The visual analytics gives our application users a clearer view of the analysis.

For the individual article analytics part, we provide a drop-down list to select the available article from the dataset and a search bar to allow end users to type the title of interest. Once the article is selected, we check if the history of the article is up to date with the boundary of 24 hours. We query MediaWiki API to pull all possible new revisions and show the number of newly pulled revisions on the page as well as the title, total number of revisions, top 5 regular users of the chosen article. For the visual analytics, we show the same two charts from the overall analytics part but specified to just the chosen article, and a year filter function is added. In addition, we show a bar chart of revision numbers by year made by one or more of the top 5 users for this article.

Finally, for the author analytics, we allow end users to display articles of a specific author by typing the author name in a free text format. The name of the author and the number of revisions is shown. End users are also allowed to see the timestamps of the article if by selecting the article title from the drop-down list.

3. Overall MVC structure

We applied the Model View Controller (MVC) structure for our architecture, setting 5 sub-folders in our app folder, routes, controllers, models, import, and views. Below we show the overview of our architecture:

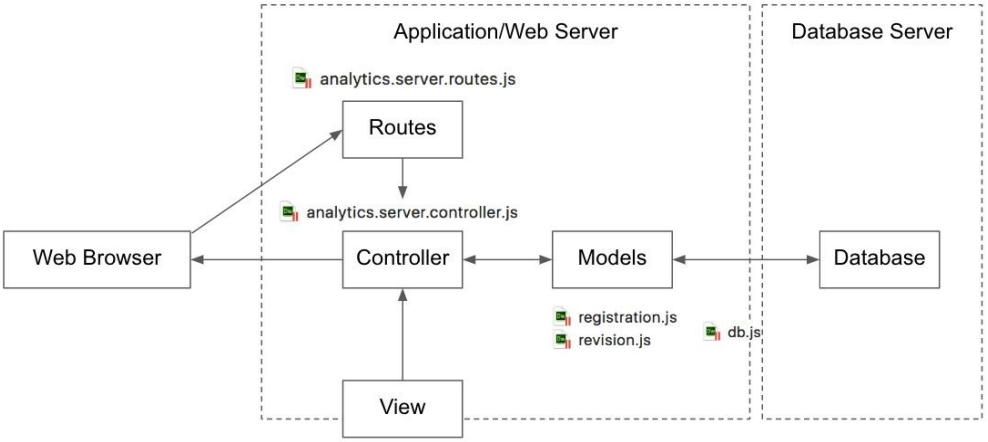

Figure 1: Architecture of MVC structure.

In the **routes** folder:

- *anlytics.server.routes.js* is used to map between the controller and HTTP requests.

In the **controllers** folder:

- *analytics.server.controller.js* controls the logic part of the application, calling functions from other models, returning results back to the browser. The controller is invoked by the routes.

In the **models** folder:

- *db.js* connects with the database.
- *revision.js* contains the functions to process data analytics tasks.
- *registration.js* deals with log-in and registration tasks.

In the **import** folder:

- *importJSON.js* imports the JSON dataset into the database.
- *update.js* updates the 4 administrator types as well as the bot users type.

In the **views** folder, the ejs files in the folder call a js file in the **public/js** folder as shown below:

- *index.ejs -> index.js* the page to log-in, register as a user.
- *main.ejs -> main.js* display all page components including the sidebar, the overall analytic results, and input forms for users to query the article or author of their interest for individual analytic and author analytic results.
- *highLowRevResult.ejs* display the articles with the highest and lowest number of revisions by a number of articles.
- *individualArticleResult.ejs -> individual.js* display the individual analytics results.
- *finalBarChartTop5.ejs -> articletop5.js* visualizes bar chart of revision number distributed by year made by one or a few of the top 5 users for the article.
- *authourAnalyticsResult.ejs -> author.js* displays the author analytics results.
- *timestampResult.ejs* shows the timestamp of the selected author and article.

The static files are in the public folder, containing the JS and CSS files. The main file for our application, *sever.js* is in the root folder.

4. Front end structure

Our application consists of two pages, the Log-in/Register page, and the Main page. First of all, the style of our application is in the *public/css/style.css* file, to make the font, color, structure reasonable and good-looking.

In the Log-in/Register page, users will be shown with a log-in form, or else users can choose to register a new account. Validation of new-users or checking if the logged-in information is identical to any of our users stored in the database. This is done by showing *index.ejs* file and extracting the script from *index.js*. After the user is successfully logged in, the page is redirected to the main page with url http://localhost:3000/main/.

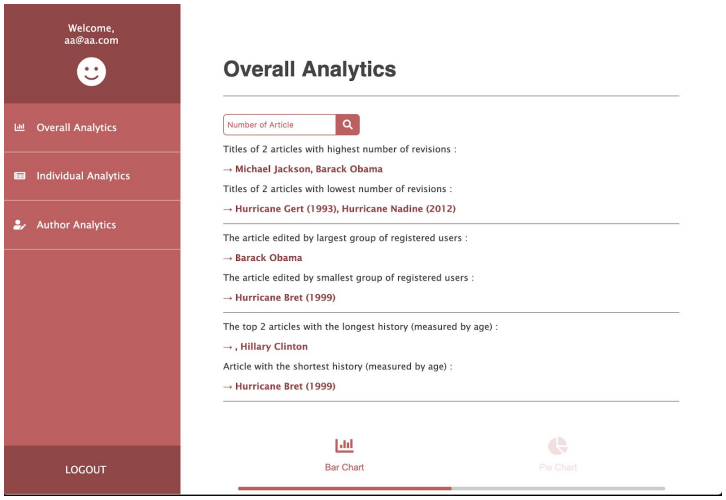

Figure 2: Display of the main page.

The main page is where all the analytics is shown, by showing the *main.ejs* file. The main page is split to the sidebar (id=”contentLeft”), and the analytics (id=”contentRight”), shown in figure 3. For the sidebar, we have Overall Analytics, Individual Analytics, Author Analytics buttons, as well as the Logout button. By default, we show the overall analytics, but when users click on other analytics buttons, the display of contentRight is changed to the selected area, defined in the *main.js* file.

```
//Display Overall section
$('#overallMenu').on('click', function(e){
  $('#overallSection').css("display", "block");
  $('#individualSection').css("display", "none");
  $('#authorSection').css("display", "none");
});

//Display Individual section
$('#individualMenu').on('click', function(e){
  $('#overallSection').css("display", "none");
  $('#individualSection').css("display", "block");
  $('#authorSection').css("display", "none");
});

//Display Author section
$('#authorMenu').on('click', function(e){
  $('#overallSection').css("display", "none");
  $('#individualSection').css("display", "none");
  $('#authorSection').css("display", "block");
});
```

Figure 3: Display of each analytic part from main.js.

When showing the default overall analytics, the top 2 highest and lowest revisions, the result is parsed to main.ejs from the results of functions written in the controller. User can choose the number of the top revisions they want to see, from the main.js, we use jquery AJAX to update our page asynchronously. A get request is sent to the routes, then calling functions in the controller. The result is then written to highLowRevResult.ejs, then parsed into main.ejs from the callback function of the initial get request.

```
//When submitted author's name, respond with that author's data analytics result
$('#selectAuthorSubmit').on('click', function(e){
  var author = $('#selectAuthor').val();
  var encodedAuthor = encodeURIComponent(author);
  console.log(author);
  $.get("/main/author?user=" + encodedAuthor, function(result) {
    //console.log(result);
    $('#authorerror').html("");
    $('#authorAnalyticsResult').html(result);
  });
});
```

Figure 4: Sample ajax request from main.js

The process is similar when getting the analytics results for the individual and author analytics, sending a get request to the routes, calling the function in controllers, writing results to an extended ejs file, then by the callback function of the get request, the result is parsed back to the main.ejs file. Figure 4 shows the code when the user clicks the button to select a specific article or searches a specific author in the main.js file.

The visualization of the analysis is from the Google Visualization API, to draw bar charts and pie charts of our results.

5. Server-side structure

Framework of MVC

This project is using MVC architectural. Model View Controller(MVC) is a useful pattern in Web application framework design. The MVC pattern is a proven, effective way for the generation of organized modular applications [1]. There are three different modules when using MVC pattern: model, view, controller, which associate with each other to implement all functions that underlying classes. By using MVC, the programmers can reduce the syntax error of SQL in command, and can easily control the users' events and render all data to the browser. MVC helps to reduce the complexity of program architectural and increase the flexibility of writing code.

Routing

the routing in MVC architectural is a mapper to associate work with front-end and backend system. it will make easier to handle the request and send the response to users(figure 5).

```
router.get('/', controller.showForm);
router.post('/main', controller.loginRegister);
router.get('/main', controller.showMain);
router.get('/main/getHighLowRev', controller.getHighLowRev);
router.get('/main/article', controller.showIndividualResult);
router.get('/main/article/getBar', controller.getIndividualBarChartTop5);
router.get('/main/author', controller.showAuthorResult);
router.get('/main/author/showTimestamp', controller.showTimestampResult);
router.get('/logout', controller.logout);
```

Figure 5: Routing code

Different requests will implement different functions that wrote in the Controller. For example, when the server is on, use request: localhost:3000/, the “showForm” function that set up in the controller will be activated and return the response to the client side.

Controller

In the controller, all functions that user may use are written down in this section. At first, before the user tries to login the main page, the controller will detect if there is a valid session in the user’s browser. If the session is still valid, the user can access the main page directly without going through login/registration page. The registration.js file contains the models to query or insert data into the database collection “registration”. In the controller, the function “loginRegister” will check if the user’s login/registration information matches with any entity in the database. If a condition is not met such as registering email is already existed, the “getIndex” function will show the login/registration page with an error message. If all conditions match, then the session will be stored and “getMain” function will render the main page. For the overall analytics section, all related querying functions are in the “getMain” function. In this function, all results will be rendered to the main.ejs file to display the query results and the charts.

```
function parseAllResult(allResult, res, count, viewfile){
  console.log(count);
  if(count < 10){return;}
  else{res.render(viewfile, {allResult: allResult});}
}
```

Figure 6: parseAllResult function code

The “parseAllResult” function is used to check if all functions in “getMain” have been executed before rendering the results to main.ejs(figure 6). For the individual article analytics, when the user chooses one article title in the list, an ajax request will be sent and the “getIndividualResults” function will render all the results to “individualArticleResults.ejs” file and show on the main page. The similar flows are applied to other ajax requests contained in this project.

Models

In the models, all queries related to data analytics are written in revisions.js. And queries related to user login/registration are written in registration.js. Mongoose module is used to connect and communicate with the database(figure 7).

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/testeddie', { useNewUrlParser: true }, function () {
  console.log('mongodb connected to testeddie!');
});

module.exports = mongoose;
```

Figure 7: Code for connecting with database

In order to make a query or insert new data, a schema and document need to be defined before reading/writing on MongoDB (figure 8).

```
var RevisionSchema = new mongoose.Schema({
  title: String,
  timestamp:String,
  user:String,
  anon:String,
  usertype:String,
  registered:Boolean,
  admintype:String
},{
  versionKey: false
});
```

```
var RegistrationSchema = new mongoose.Schema({
  email: String,
  password:String,
  firstname:String,
  lastname:String
},{
  versionKey: false
});
```

Figure 8: Code of the schemas

6. Pre-work: Import data into MongoDB and Update

To quickly import all the JSON file into the database, the file called “importJson.js” needs to be executed to run the mongoose command to import all JSON files into the “revisions” collection. Different types of users in the original JSON files need to be distinguished in order to get the correct result when doing the query. The “update.js” file needs to be executed after the import to update and insert usertype, registered, and admintype into the database. Below is one example that shows the updates to be made for all admin users (figure 9). Instruction on pre-work required before running the server is in “READ ME.txt” located in the root folder.

```
var Revision = mongoose.model('Revision', RevisionSchema, 'revisions');

//Set usertype for all admin users
Revision.update(
  {'user':{'$in':allAdmin, '$nin':bot}},
  {$set:{usertype:'admin',registered:true}},
  {multi: true, upsert:true},
  function(err,result){
  if (err){console.log("ERROR")}
  else{
    console.log('admin usertype has been updated/inserted');
    console.log(result);
  }}
);
```

Figure 9: Example of a model for updating Admin users

Reference

1. Hofmeister C, Nord R.L, Soni D (2000) Applied Software Architecture, Addison-Wesley.