NEWSPAPER: A DIGITAL VIEW INTO EVENTS AROUND YOU

TEAM 1

Jayasurya Pinaki

Aswin Prasad

Rajeev Sebastian

## Database Project Miniworld

This is an online interactive news site where newsreaders can interact with news articles posted by editors. Editors can view and analyze statistics that are based on the newsreader's interaction with the article they have posted.

## Technologies used

Database Engine: MySQL on Amazon RDS, MongoDB on Atlas

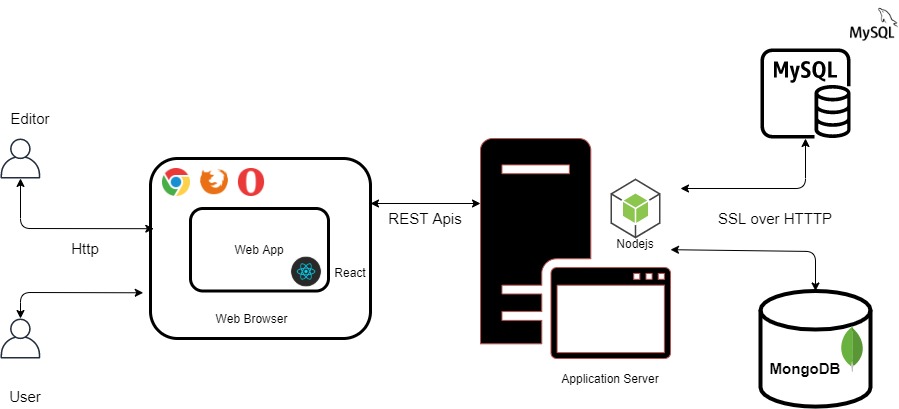
DB application: Workbench 8.04, Compass

Frameworks: MERN stack (MongoDB/MySQL, Express, React, NodeJS)

Languages: PL/SQL, Javascript, SQL

DB access technology: Mongoose

## Overall architecture diagram:



## 

## 

## Contributions:

|  |  |  |
| --- | --- | --- |
| Aswin Prasad | Jayasurya Pinaki | Rajeev Sebastian |
| 1. Found use cases for views 2. Found use cases for NoSQL 3. Implemented logging 4. Worked on view article, like and comment | 1. Found use cases for stored procedures 2. Used transactions 3. User roles and encrypting password 4. Implemented subscribe to category and view notifications | 1. Normalized the schema 2. Worked on analytics 3. Implemented view activity service 4. Worked on editor roles (Create and edit an article) |

### Aswin Prasad

Proposal:

* + Came up with the idea of newspaper editorial website
  + Defined the miniword for this project

Entity Relationship Diagram:

* + Defined the data requirements
  + Found out the need for age to be a derived attribute.
  + Defined the correct multiplicity for each relationship

Implementation:

* + Authored the APIs for liking an article, commenting on an article, subscribing to a category and viewing an article
  + Made use of REST standards for the API
  + Created APIs to get complex analytics like The number of articles read during a time period in the day and newspaper categories read by newsreaders based on age group
  + Created the structure for mongoDB by identifying the use cases and need for NoSQL in this application.
  + Setup RDS on AWS for the application to use.
  + Created tables as required by the application
  + Found use cases for views and implemented them
  + Created UI component to view an article, like and comment on it
  + Created UI component to view analytics using graphs
  + Implemented logging feature for the application to capture all activities in the application
  + Carried out to end to end testing
  + Data was not fetched properly in the DOB dropdown in the update profile page. Found that during the end to end testing and fixed the issues in react.

Report:

* + Use case for mongoDB and its justification.
  + Use case for Views and its justification.
  + Logging and its need.

### Jayasurya Pinaki

Proposal:

* + Defined the purpose of the system
  + Came up with scenarios for the application

Entity Relationship Diagram:

* + Wrote the data requirements for our Miniworld.
  + Designed the skeletal structure for the ERD with their attributes and primary keys

Implementation:

* + Authored the APIs for creation, viewing and updating of user/editor details and login while encrypting password.
  + Made use of REST standards for the API
  + Created APIs to get complex analytics of Newspaper categories read by newsreaders based on their location and also notification service.
  + Found use cases for stored procedures and implemented them which reduced the complexity of code.
  + Defined the data types needed for each DB object
  + Created a theme for all UI components
  + Created UI components to login, create and update user/editor
  + Created Reusable UI components to get headlines for each category or all along with viewing activity of an user
  + Carried out to end to end testing
  + Since we were using both SQL and NoSQL, the timestamp stored in both were different (since they are different servers). Found this issue and fixed the problem.

Report:

* + Specified the need for each DB object
  + Explained the stored procedures used by the application
  + Analyzed the project and wrote down the post mortem report

### Rajeev Sebastian

Proposal:

* + Wrote about how the application can be used to show analytics to the user or the editor.
  + Explained the roles for the user and editor

Entity Relationship Diagram:

* + Recognized that the article entity is a weak entity.
  + Identified the key attributes like email need to be underlined
  + Showed the team members the need for different relationships for the comment, read, like use cases.
  + Wrote the data requirements for our Miniworld.

Schema Diagram:

* + Generated the schema diagram from the above ERD

Implementation:

* + Authored the APIs for creation of article, retrieval of article, headlines of all articles or belonging to a category, updating the article
  + Made use of REST standards for the API
  + Created APIs to get complex analytics like most read article and most read category
  + Identified error in DB creation and corrected it for the views table as it did not include read\_time in the primary key
  + Identified how a editor can repeatedly access his article and can artificially inflate the read count -Modified the API to handle these cases
  + Created Reusable UI components to get the user notifications
  + Created Reusable UI components to get the user activity and history
  + Made use of unique fonts to emphasize different notifications types
  + Wrote Test cases and test plan
  + Carried out to end to end testing
  + Analyzed the various parameters during transactions and their effect on the concurrency problems

Report:

* + Generated skeletal body of the report.
  + Wrote about the normalization and Schema diagram.
  + Test plan and execution
  + Drew the architecture Diagram.

## Final design of database:

### Entities

#### Objects

#### Article: It has a text-based headline and a body that is posted by an editor.

#### Category: Each article is tagged to one or more categories. This is a grouping of similarly

* 1. themed articles (Sports, Politics, Business, Science, etc).

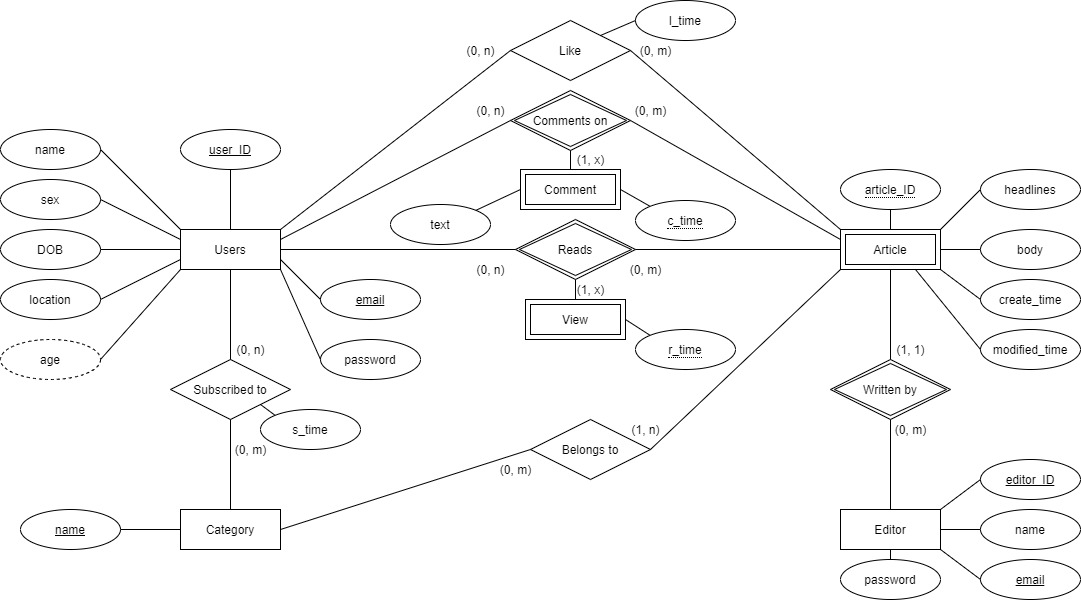
#### Actors

* 1. Newsreader: View, read and leave feedback on an article.
  2. Editor: Post article and view analytics based on all the articles posted by them.

### Functionality and Operations

* Encrypt password for all users
* Unregistered users can view headlines only.
* News readers: They are the ones to read the articles.
  + View or update profile information (Name, Age, Location, etc)
  + View all the article headlines posted by every editor
  + View article headline based on the category it has been tagged to.
  + View/Read individual article
  + Like an article only once
  + Comment on articles multiple times.
  + Subscribe to a category of news.
  + View notifications when there is a new article posted or edited in the category they have subscribed to.
  + View notifications on new comments being added to a post that the user has previously commented on.
  + See his activity history (Articles viewed, liked, commented and categories subscribed to).
* Editors: Creation of articles and review reach of them.
  + Each editor has a name
  + Post articles by tagging each article to one or more categories (Text only).
  + Update article (Text only).
  + View graphical analytics based on newsreaders activity such as
    - Most frequently read articles.
    - The most frequently read category.
    - Most liked articles.
    - The number of articles read during a time of the day.
    - Newspaper categories read by newsreaders based on age group
    - Newspaper categories read by newsreaders based on their location
  + Actionable actions based on analytics
    - Suggest articles to be written based on what the users are reading.
    - Show popularity/ ranking of the editor based on the number of users reading their article.

### Entity Relationship Diagram (ERD):

****

The entities and their relationships in the ERD are

* Users – Who will read like comment on articles
* Editor – Who will write, publish articles and look at analytics
* Article (Weak Entity) – Articles that are written are always associated with an editor and belong to a category.
* Comment – In order to capture the condition that the same user can comment on the same article multiple times it was necessary to include this weak entity in a three-way relationship between Users and Article.
* View – Similarly in order that reads were also made according to a user to an article we introduced a weak entity view.
* Category – It shows the broad area that the article belongs to.

### Schema Diagram

From the ER diagram we can create the schema diagram following the 7-step process we learned in class.

**A screenshot of a cell phone

Description automatically generated**

### 

### Functional Dependencies and Normalization

The following are the various relations from the schema diagram and the FD associated with them

**Users (user\_ID, name, sex, DOB, location, email, password)**

PK: (user\_ID)

FDs: user\_ID → name

user\_ID → sex

user\_ID → DOB

user\_ID → location

user\_ID → password

email → password

The above table is in 1NF because it has a primary key, no attribute is multi-valued and all the non-prime attributes are functionally dependent on the primary key user\_ID

It is in 2NF as there is no composite PK

Also, since there is no transitive dependency the relation is in 3NF

**Category (name)**

PK: (name)

FD: name → name

Trivial to know that the relation is in 3NF

**Editor (editor\_ID, name, email, password)**

PK: (editor\_ID)

FDs: editor\_ID → name

editor\_ID → email

editor\_ID → password

email → password

The above table is in 1NF because it has a primary key, no attribute is multi-valued, and all the non-prime attributes are functionally dependent on the primary key editor\_ID

It is in 2NF as there is no composite PK

Also, since there is no transitive dependency the relation is in 3NF

**Article (editor\_ID, article\_ID, headlines, body, create\_time, modified\_time)**

PK: (editor\_ID, article\_ID)

FDs: editor\_ID, article\_ID → headlines

editor\_ID, article\_ID → body

editor\_ID, article\_ID → create\_time

editor\_ID, article\_ID → modified\_time

The above table is in 1NF because it has a primary key, no attribute is multi-valued, and all the non-prime attributes are functionally dependent on the primary key editor\_ID, article\_ID

It is in 2NF because even though there is composite PK all non-prime attributes are fully functionally dependent on the PK

Also, since there is no transitive dependency the relation is in 3NF

**Comment (user\_ID, editor\_ID, article\_ID, text, c\_time)**

PK: (user\_ID, editor\_ID, article\_ID, c\_time)

FD: user\_ID, editor\_ID, article\_ID, c\_time → text

Easy to see that this table is in 3NF

**View (user\_ID, editor\_ID, article\_ID, r\_time)**

PK: (user\_ID, editor\_ID, article\_ID, r\_time)

Since the whole table is a primary key it is already in 3NF

**Subscribed\_to (user\_ID, name, s\_time)**

PK: (user\_ID, name)

FD: user\_ID, name → s\_time

This relation is in 3NF

**Belongs\_to (article\_ID, editor\_ID, name)**

PK: (article\_ID, editor\_ID, name)

Already in 3NF since each attribute is a part of the primary key.

**Like (user\_ID, article\_ID, editor\_ID, l\_time)**

PK: (user\_ID, name, editor\_ID)

FD: user\_ID, editor\_ID, article\_ID → l\_time

We can see that this relation is in 3NF.

### Denormalization

In order to provide a framework to store different type comments easily such as gif, images, sounds we have stored all comments in mongoDB as well. Though this causes redundancy we can avoid joins and get all the comments for a given article with just a simple read.

### Description of Database Tables

**article**: This table stores articles information posted by all editors

* *editor\_id* and *article\_id* is the primary key because article id is unique for each editor
* *headlines* hold the headline for each article
* *body* holds the body of the article
* *create\_time* is the time at which the article has been created
* *modified\_time* is the time at which the article has been modified

|  |  |
| --- | --- |
| editor\_id | int(11) |
| article\_id | int(11) |
| headlines | varchar(200) |
| body | varchar(4000) |
| create\_time | datetime |
| modified\_time | datetime |

**belongs\_to**: This table stores articles information posted by all editors

* *Editor\_id, article\_id* and *name* is the primary key because each article can belong to n number of categories

|  |  |
| --- | --- |
| editor\_id | int(11) |
| article\_id | int(11) |
| name | varchar(50) |

**category**: This table stores all the possible category names

* *Name* is the primary key which is also the name of the category

|  |  |
| --- | --- |
| name | varchar(50) |

**comments**: This table contains the information of all comments done by every user on all articles

* *User\_id, article\_id and editor\_id* is the primary key because each user can comment on n articles
* *Text* holds the comment user has done on text format
* *c\_time* holds the time at which the user has commented on the article

|  |  |
| --- | --- |
| user\_id | int(11) |
| article\_id | int(11) |
| editor\_id | int(11) |
| text | varchar(250) |
| c\_time | datetime |

**editor:** This table stores basic account information for the editor

* *Editor\_id* is the primary key
* *Name* is the name of the editor
* *Email* is the email of the editor which will be used for login which is also unique
* *Password* is encrypted password of the editor to login

|  |  |
| --- | --- |
| editor\_id | int(11) |
| name | varchar(50) |
| email | varchar(50) |
| password | varchar(100) |

**likes:** This table contains the information of all likes done by every user on all articles

* User\_id, article\_id and editor\_id is the primary key because each user can like n articles
* l\_time holds the time at which the user has liked an article

|  |  |
| --- | --- |
| user\_id | int(11) |
| article\_id | int(11) |
| editor\_id | int(11) |
| l\_time | datetime |

**subscribed\_to:** This table holds information related to all the categories a user has subscribed to

* *User\_id* and name is the primary key
* *S\_time* is the time at which a user subscribed to the category

|  |  |
| --- | --- |
| user\_id | int(11) |
| name | varchar(50) |
| s\_time | datetime |

**user:** This table stores basic account information for the user

* *User\_id* is the unique id which identifies every user
* *Name* is the name of the user
* *Email* is the email of the user which will be used for login which is also unique
* *Password* is encrypted password of the user to login
* *Sex* is the gender of the user where they can enter M or F
* *DOB* is the date of birth of the user
* *Location* is 2 digit state code of where the user is present

|  |  |
| --- | --- |
| user\_id | int(11) |
| email | varchar(50) |
| password | varchar(100) |
| name | varchar(50) |
| sex | char(1) |
| DOB | date |
| location | varchar(2) |

**views:** This table contains the information of all likes done by every user on all articles

* *User\_id, article\_id* and *editor\_id* is the primary key because each user can view n articles
* *l\_time* holds the time at which the user has viewed an article

|  |  |
| --- | --- |
| user\_id | int(11) |
| editor\_id | int(11) |
| article\_id | int(11) |
| r\_time | datetime |

## Stored procedures

This stored procedure checks if the user has already liked an article and returns true or false

**CREATE** **PROCEDURE** hasUserLikedTheArticle(“ userId **INT**(11), **IN** articleId **INT**(11), **IN** editorId **INT**(11))

**IF** **EXISTS** (**SELECT** \* **FROM** likes **WHERE** user\_id=userId and article\_id=articleId and editor\_id=editorId) **THEN**

**SELECT** true;

**ELSE**

**SELECT** false;

**END** **IF**;

**END** $$

This stored procedure checks if the category name is valid or invalid as all articles needs to be part of at least one category

**CREATE** **PROCEDURE** isValidCategoryName(**IN** categoryName **VARCHAR**(50))

**IF** **EXISTS** (**SELECT** \* **FROM** category **WHERE** name = categoryName) **THEN**

**SELECT** true;

**ELSE**

**SELECT** false;

**END** **IF**;

**END** $$

This stored procedure checks if the email **ID** exists for an user and returns true or false

**CREATE** **PROCEDURE** doesEmailExistForUser(**IN** emailId **VARCHAR**(50))

**IF** **EXISTS** (**SELECT** \* from user where email = emailId) **THEN**

**SELECT** true;

**ELSE**

**SELECT** false;

**END** **IF**;

**END** $$

This stored procedure checks if the email already for an editor whose id is other than requested editor id

**CREATE** **PROCEDURE** checkDuplicateEmailForEditor(**IN** emailId **VARCHAR**(50), editorId **INT**(11))

**IF** **EXISTS** (**SELECT** \* from editor where email = emailId and editor\_id != editorId) **THEN**

**SELECT** true;

**ELSE**

**SELECT** false;

**END** **IF**;

**END** $$

This stored procedure checks if the email **ID** exists for an editor and returns true or false

**CREATE** **PROCEDURE** doesEmailExistForEditor(**IN** emailId **VARCHAR**(50))

**IF** **EXISTS** (**SELECT** \* from editor where email = emailId) **THEN**

**SELECT** true;

**ELSE**

**SELECT** false;

**END** **IF**;

**END** $$

This stored procedure checks if the email already for an user whose id is other than requested user id

**CREATE** **PROCEDURE** checkDuplicateEmailForUser(**IN** emailId **VARCHAR**(50), userId **INT**(11))

**IF** **EXISTS** (**SELECT** \* from user where email = emailId and user\_id != userId) **THEN**

**SELECT** true;

**ELSE**

**SELECT** false;

**END** **IF**;

**END** $$

This stored procedure updates user information depending on if the user has updated password or not

**CREATE** **PROCEDURE** updateUserInformation(**IN** emailId varchar(50), **IN** userPassword varchar(100), **IN** username varchar(50), **IN** userSex char(1), **IN** dateOfBirth date, **IN** userLocation varchar(2), **IN** userId int(11))

**IF** password is **NULL** **THEN**

**UPDATE** user **SET** email = emailId, name = username, sex = userSex, **DOB** = dateOfBirth, location = userLocation **WHERE** user\_id = userId;

**ELSE**

**UPDATE** user **SET** email = emailId, password = userpassword, name = username, sex = userSex, **DOB** = dateOfBirth, location = userLocation **WHERE** user\_id = userId;

**END** **IF**;

**END** $$

This stored procedure updates editor information depending on if the editor has updated password or not

**CREATE** **PROCEDURE** updateEditorInformation(**IN** emailId varchar(50), **IN** username varchar(50), **IN** editorId int(11), **IN** userPassword varchar(100))

**IF** password is **NULL** **THEN**

**UPDATE** editor **SET** email = emailId, name = username **WHERE** editor\_id = editorId;

**ELSE**

**UPDATE** editor **SET** email = emailId, password = userPassword, name = username **WHERE** editor\_id = editorId;

**END** **IF**;

**END** $$

## User login password (Encrypted / Hashed)

**import passwordHash from 'password-hash'**

**var EncryptPassword = (password) => {**

**return passwordHash.generate(password);**

**}**

**var validatePassword = (password, hashedPassword) => {**

**if (passwordHash.verify(password, hashedPassword)){**

**return true;**

**}**

**return false;**

**}**

We are following the process of storing hashed passwords in Database. With the increasing hacking going on these days. If DB is somehow compromised. User’s password should not be let out. As the user may have reused those passwords in some other sites also. The hacker can login in to the user account and can do some malignant things.

We are using the “password-hash” npm package, which simplifies the process of password hashing for us. It uses a salt that we have to define in the application. The password can be decrypted only with that salt.

This way the security of the user is ensured.

## Multi statement database transactions.

exports.commentOnArticle = async (req, res) => {

logger.info('Inside ' + req.originalUrl + ' Body ' + JSON.stringify(req.body));

let commentData = req.body;

await SQLConnection.beginTransaction()

let mongoConnection = await Article.startSession();

mongoConnection.startTransaction();

try {

let query = SQLQueries.commentOnArticle(req.body.user\_id, req.body.article\_id, req.body.editor\_id, req.body.text)

await SQLHelper(query);

/\*

Appending the comment to the article document, incrementing the commentCount.

\*/

let condition = {

articleId: commentData.article\_id,

editorId: commentData.editor\_id

};

let comment = {

userId: commentData.user\_id,

text: commentData.text,

commentTime: getTime.getTime()

}

//console.log(comment);

let r = await Article.findOneAndUpdate(condition, { $push: { comments: comment }, $inc: { commentCount: 1 } }, { new: true });

//console.log(r);

await mongoConnection.commitTransaction();

await mongoConnection.endSession();

await SQLConnection.commit()

logger.info('Returning from ' + req.originalUrl + constants.STATUS\_CODE.CREATED\_SUCCESSFULLY\_STATUS + JSON.stringify(commentData));

return res

.status(constants.STATUS\_CODE.CREATED\_SUCCESSFULLY\_STATUS)

.send(commentData);

} catch (error) {

console.log(`Error while getting user profile details ${error}`);

await mongoConnection.abortTransaction();

await mongoConnection.endSession();

await SQLConnection.rollback()

logger.info('Error in ' + req.originalUrl + constants.STATUS\_CODE.INTERNAL\_SERVER\_ERROR\_STATUS + error.message);

return res

.status(constants.STATUS\_CODE.INTERNAL\_SERVER\_ERROR\_STATUS)

.send(error.message);

}

}

Here if you see, there are multiple modifying transactions in a single API. what if the last modifying statement fails. We have to manually rollback the changes. Instead of that, we can use the Transaction feature that the MySQL database provides. To begin a transaction we have to call “SQL.beginTransaction()”. To mark success we can call “SQL.commitTransaction()” and to abort a transaction we can call “SQL.rollback()”.

rollback is called during the failure scenarios. The special thing is that we have combined the MySQl and MongoDB transactions.

## Indexes

We have made use of indexes on primary key as they are the search parameters on 95 % of our queries. This improves the performance to a great degree. Also since we have made use of MongoDB we can get the auxiliary data like comments like information using simple reads supported by the document model.

## Concurrency Control

Since InnoDB makes use of 2 phase locking protocol we have made use of REPEATABLE READ as our default transaction control mechanism to avoid the Dirty read and Non repeatable read problems that can occur. Suppose an update to an account fails then while fetching the user account details, we must ensure consistency that was present in the previous state i.e. before the update transaction failed. Also we don’t want our analytics to be tainted by inconsistent writes (like commenting on the article when all its comments are being accessed).

exports.moduleName = async (req, res) => {

await SQLConnection.beginTransaction()

let mongoConnection = await collection.startSession();

mongoConnection.startTransaction();

try {

// MySQL operations

// MongoDB operations

await mongoConnection.commitTransaction();

await mongoConnection.endSession();

await SQLConnection.commit();

} catch (error) {

await mongoConnection.abortTransaction();

await mongoConnection.endSession();

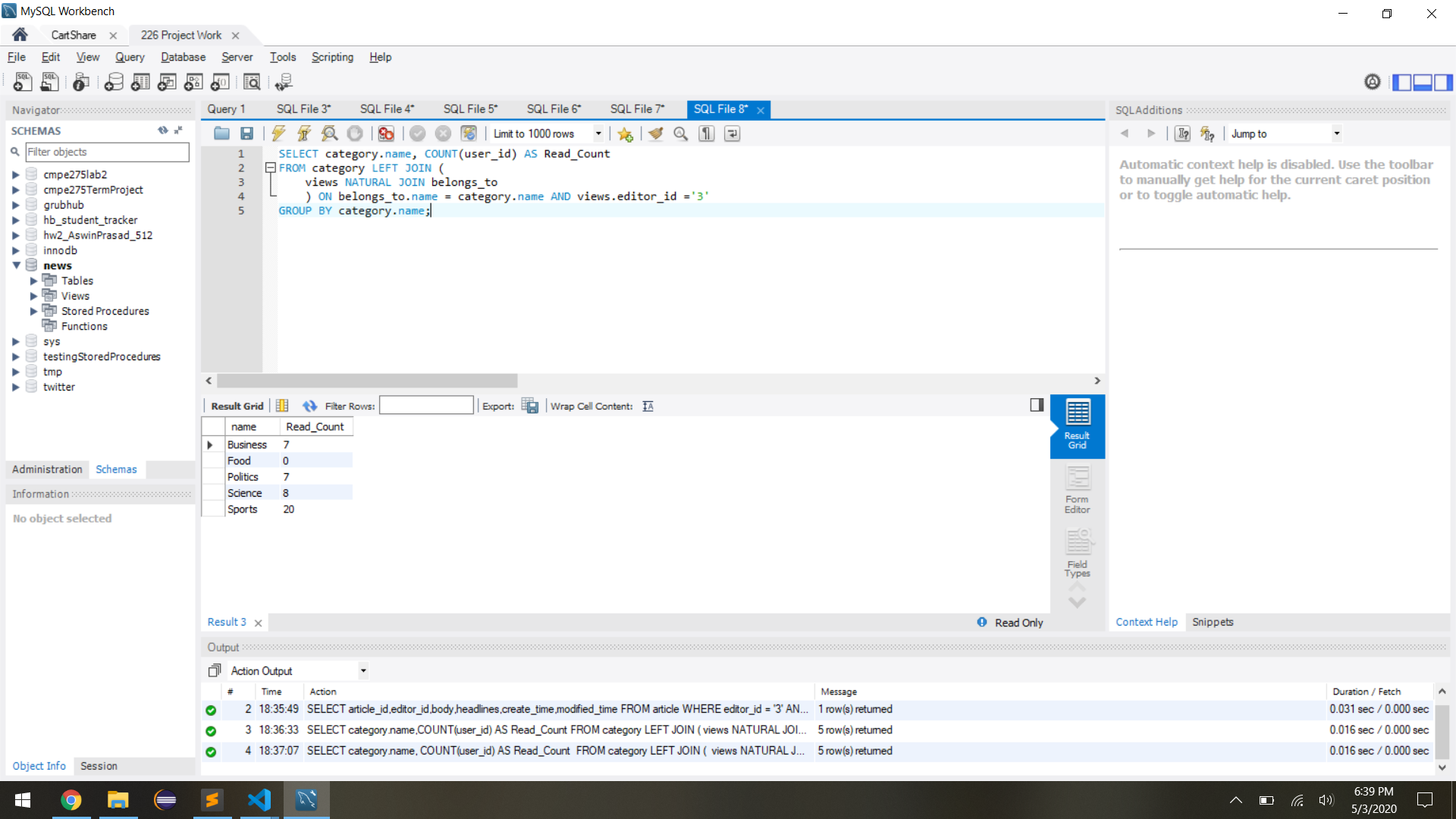
await SQLConnection.rollback()

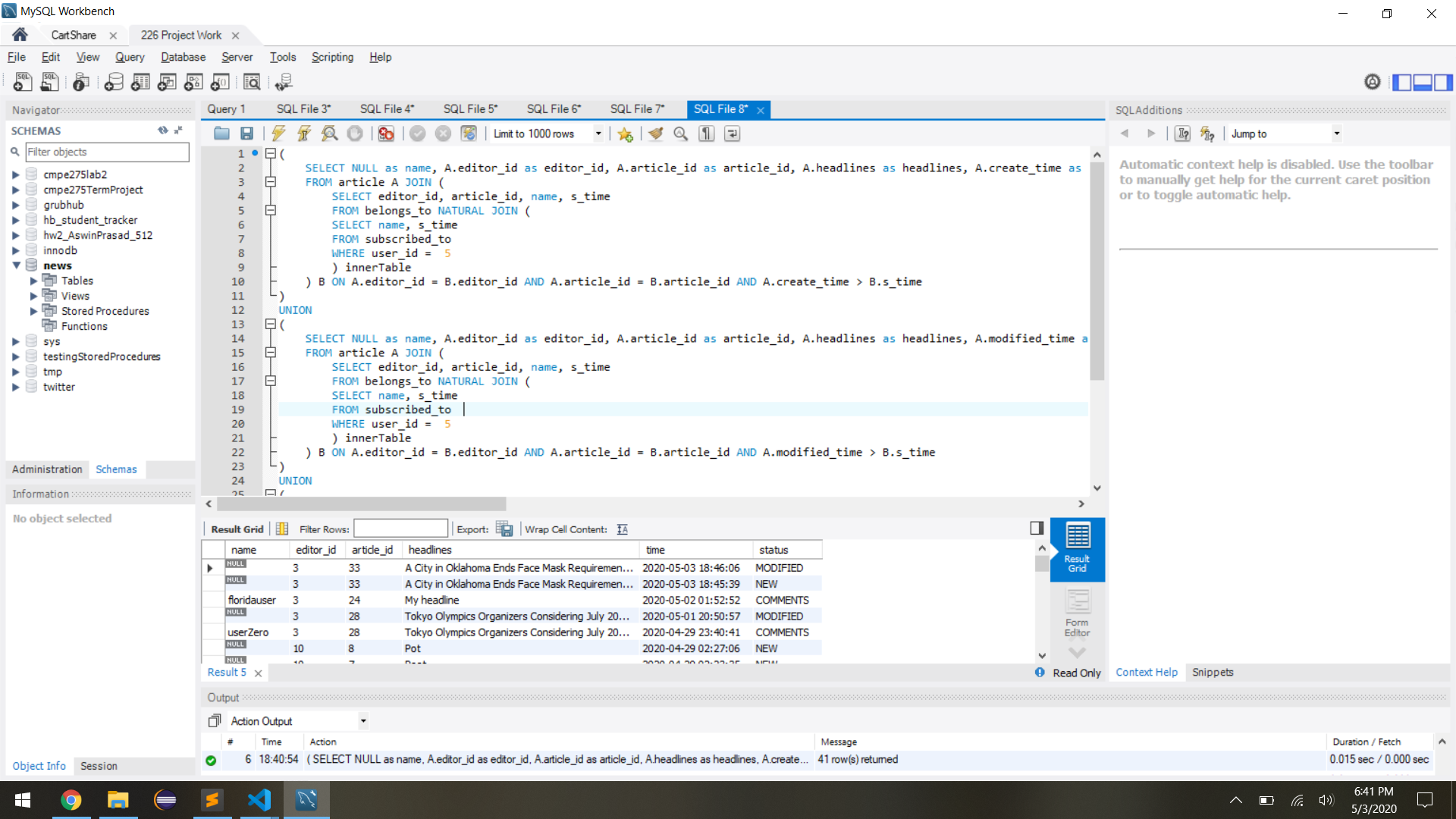
}

}

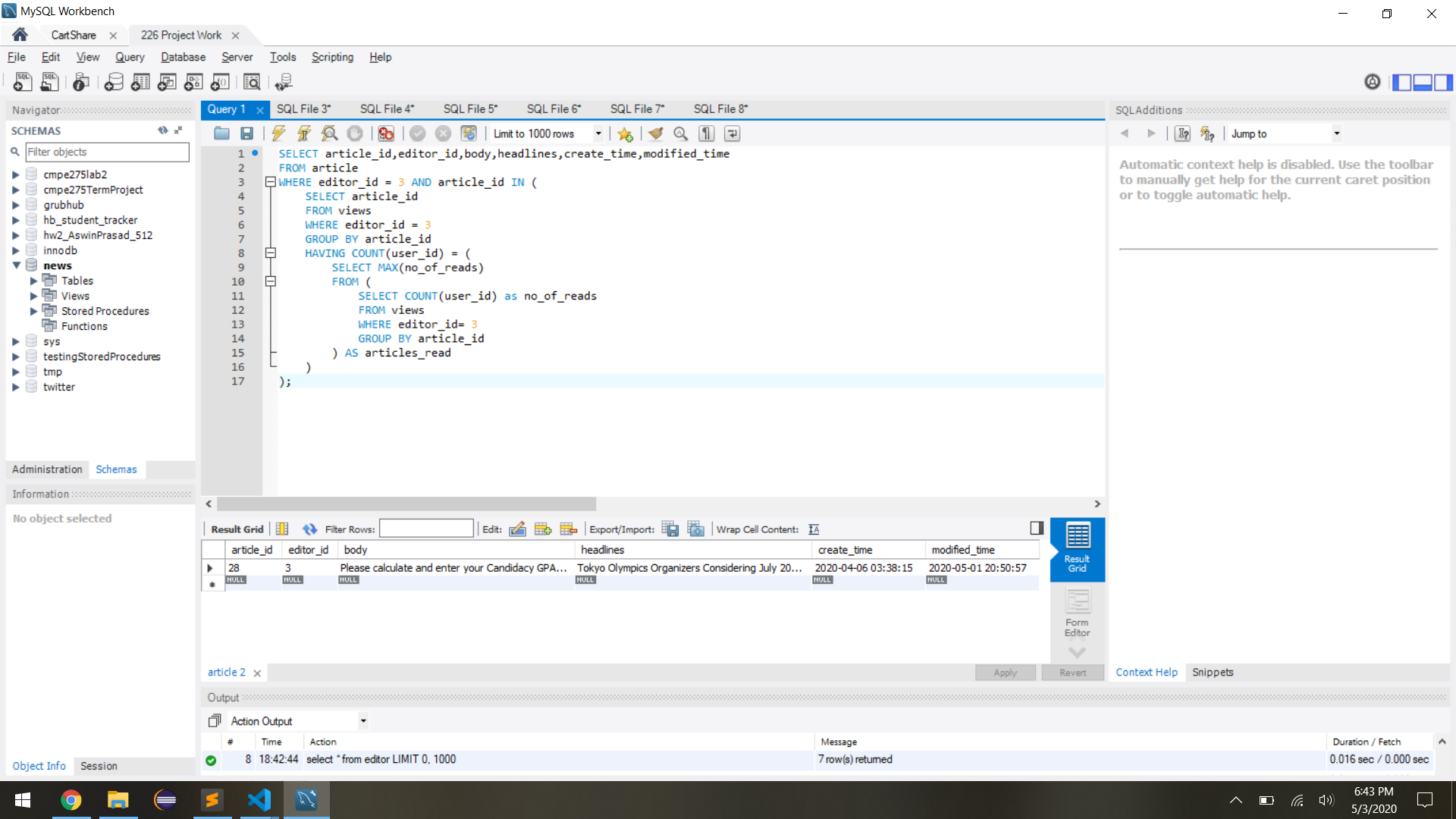
## 

## Screenshots of Manual Execution of Queries

This query will retrieve the total of number of times every article of a particular editor in a category has been read 

This query will retrieve name, editor\_id, article\_id, headlines, time and status for all newly posted articles in the category a user has subscribed to, and the same information for all articles that has been modified in the category a user has subscribed to, and finally same information for any comments that has been done on an article where the user has already commented. All of these records are sorted by timestamp in descending order i.e Latest notification first 

This query will retrieve details of the article which has been read the most number of times by any user in the system



This query will retrieve the total of number of times every article of a particular editor in a category has been read

## 

## 

## 

## 

## 

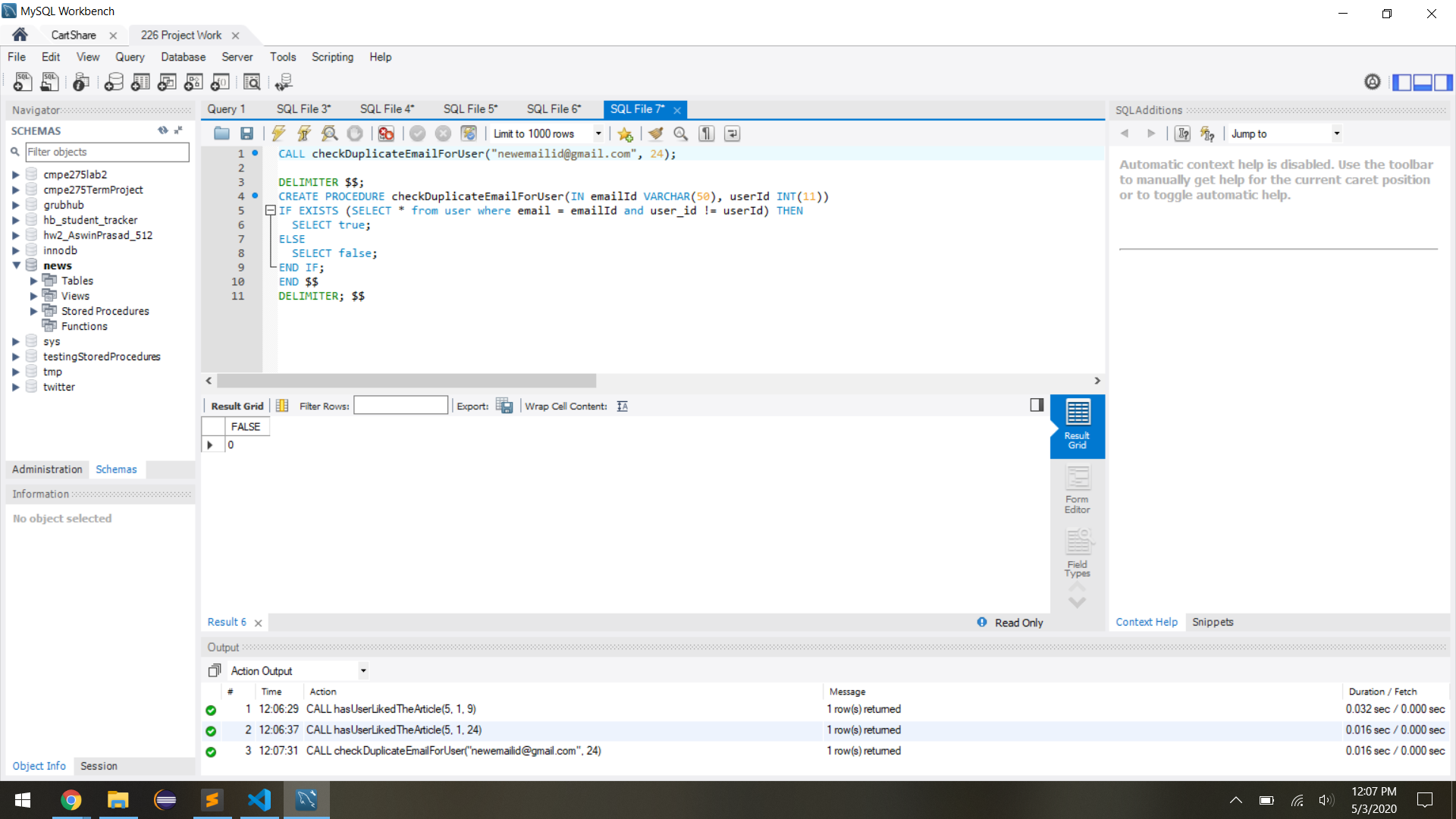
## 

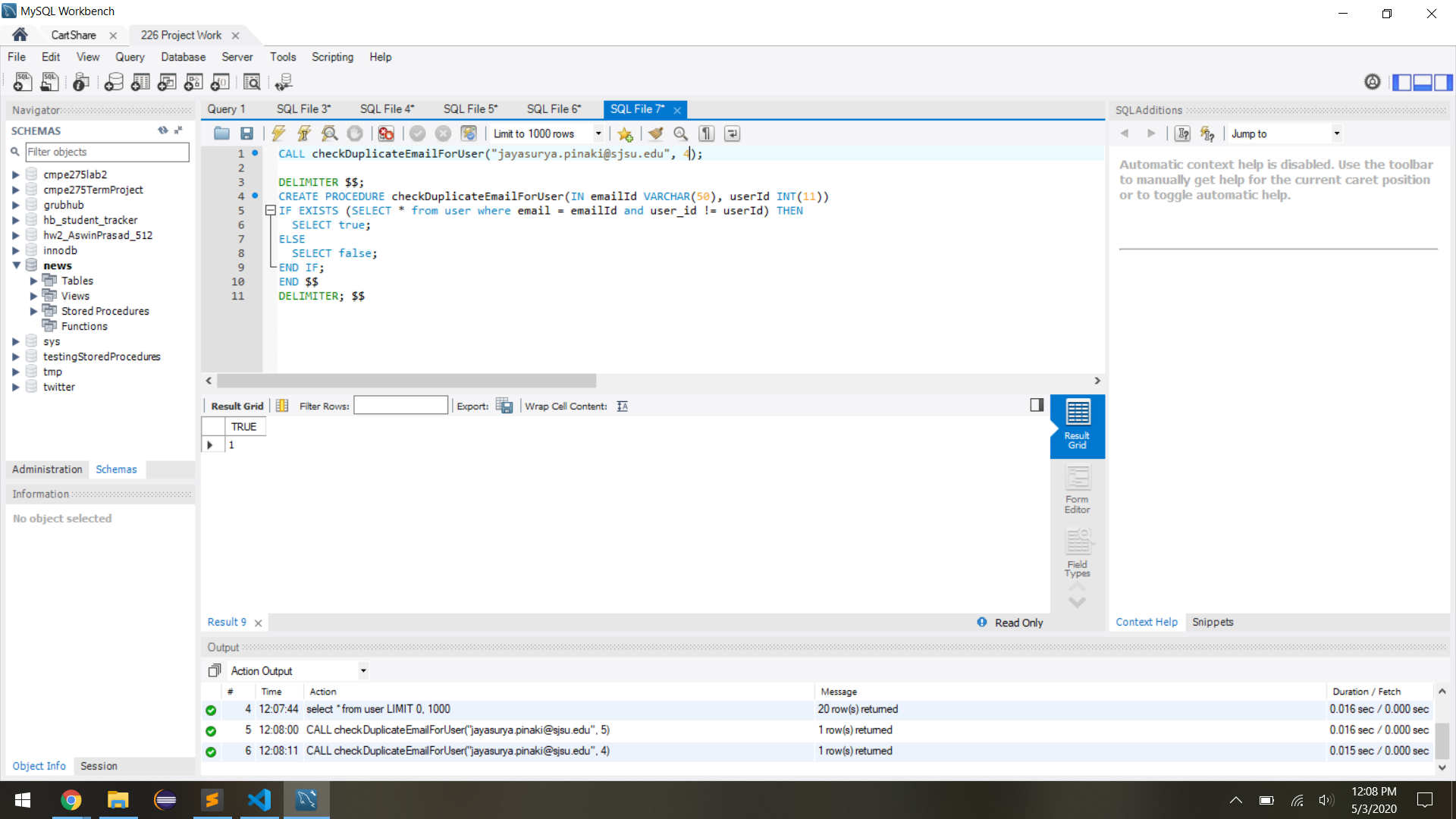
## 

## 

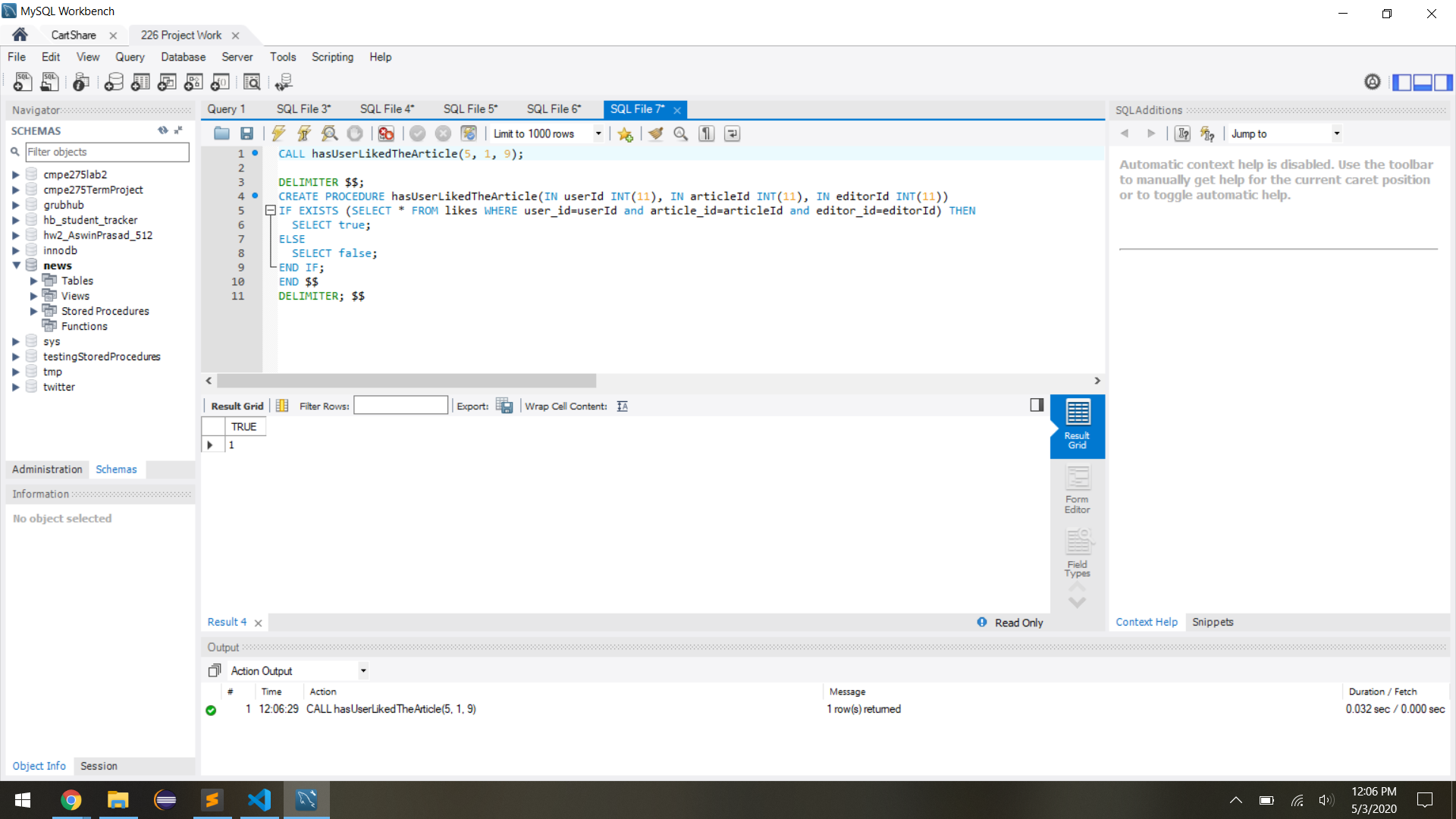
## Screenshots of Stored Procedures

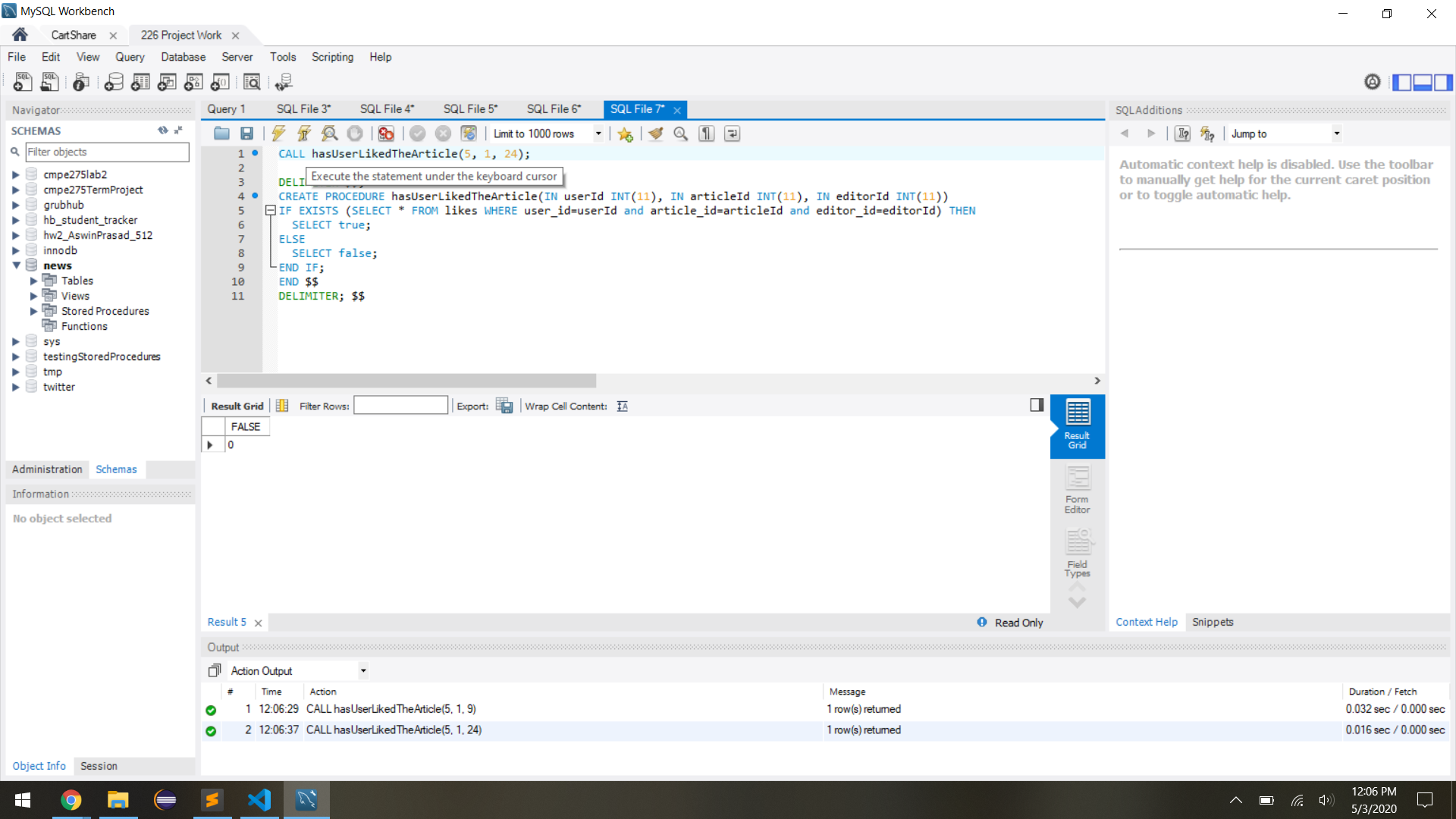
This stored procedure updates user information depending on if the user has updated password or not





This stored procedure checks if the category name provided is valid and returns true or false





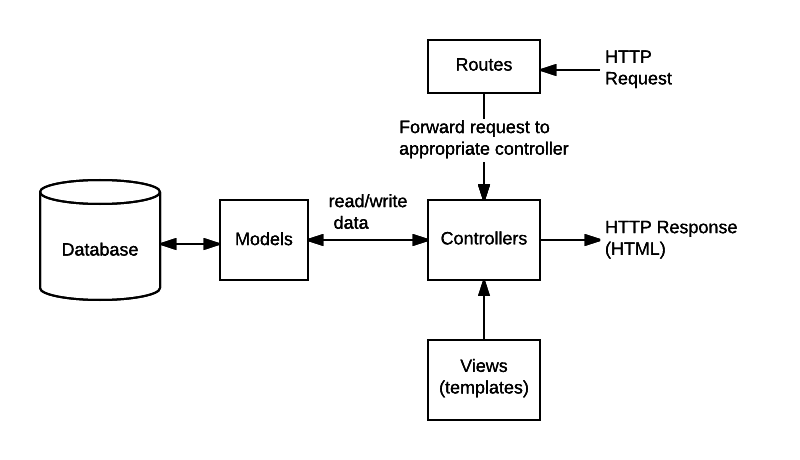
## Final Design of Database application

We have made use of the MVC architecture to create a NodeJS application.

Express is used for the server creation and request handling.

Mongoose is use to connect to MongoDB

mysql library to connect to MySQL Database



## Application Log files Text based

Logging is an important part in any enterprise level application. This gives the admins the knowledge of where things have gone wrong and get back to the core of the problem. Security is also greatly improved when maintaining the logs to audit.

We have used the log4js package in NPM to log requests, database calls, and responses.

Here is a sample of our logs.

[2020-04-29T16:39:33.558] [INFO] log - Starting the News Paper application! - This is logged during the logger creation in config folder once.

[2020-04-29T16:39:38.778] [INFO] log - Inside /article/headlines?type=all

[2020-04-29T16:40:20.314] [INFO] log - Inside /article/view/3/28/30

[2020-04-29T16:40:20.314] [INFO] log - Executing Query SELECT name, headlines, body, create\_time, modified\_time FROM article NATURAL JOIN editor WHERE article\_id ='28' AND editor\_id ='3'

[2020-04-29T16:40:20.316] [INFO] log - Inside /users/subscribedCategories/30

[2020-04-29T16:40:20.316] [INFO] log - Executing Query SELECT name FROM subscribed\_to WHERE user\_id = 30;

[2020-04-29T16:40:20.353] [INFO] log - Returning from /users/subscribedCategories/30200[]

[2020-04-29T16:40:20.459] [INFO] log - Executing Query SELECT name FROM belongs\_to WHERE article\_id ='28' AND editor\_id ='3'

[2020-04-29T16:40:20.474] [INFO] log - Executing Query INSERT INTO views (user\_id,editor\_id,article\_id,r\_time) VALUES (30,3,28, NOW())

[2020-04-29T16:40:20.656] [INFO] log - Inside /article/likes/3/28/30

[2020-04-29T16:40:20.656] [INFO] log - Executing Query CALL hasUserLikedTheArticle(30, 28, 3)

[2020-04-29T16:40:20.676] [INFO] log - Returning from /article/likes/3/28/30200{"status":true}

[2020-04-29T16:40:41.000] [INFO] log - Inside /users/comment/ Body {"article\_id":"28","editor\_id":"3","user\_id":"30","text":"Hey Comment."}

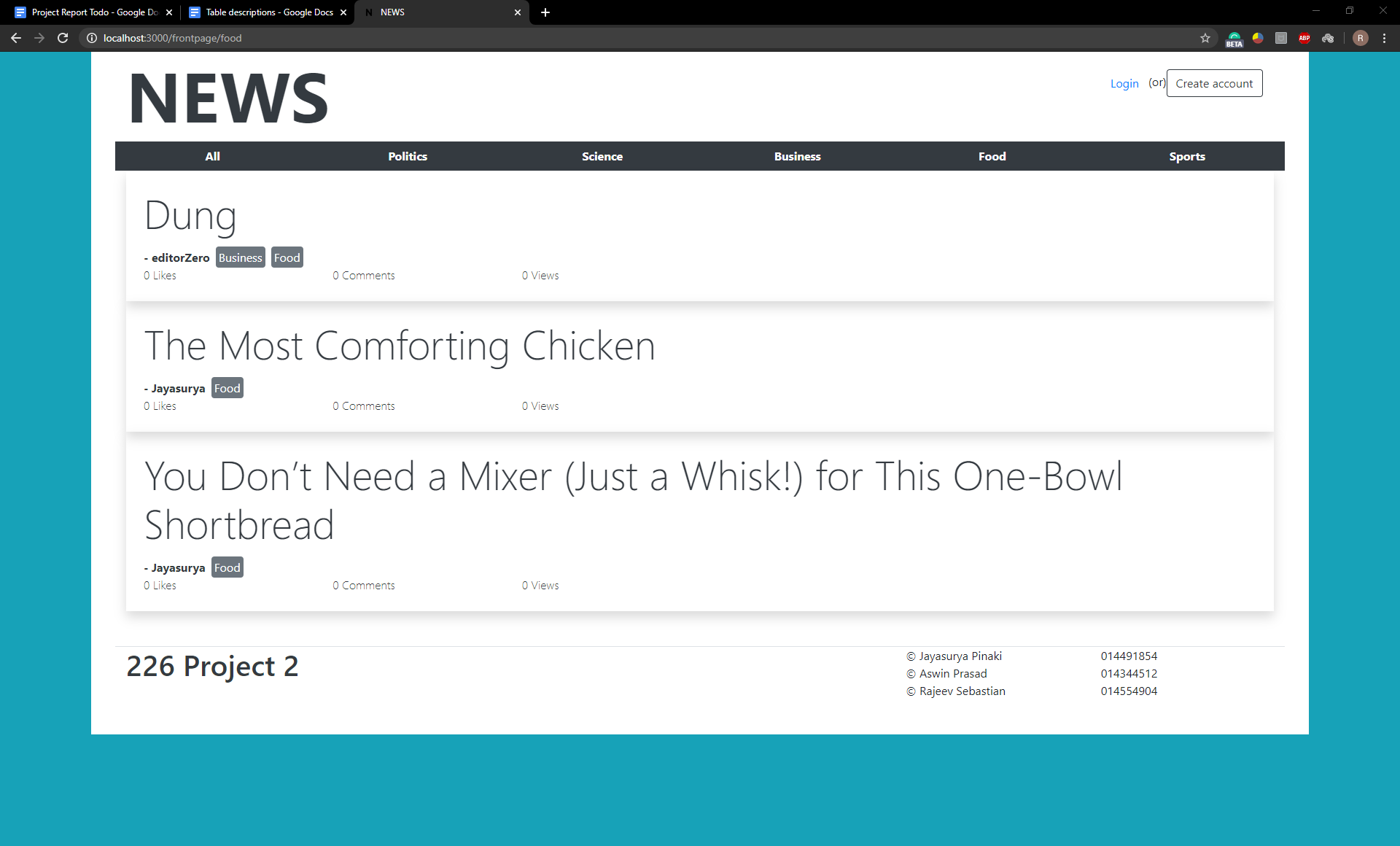
[2020-04-29T16:40:41.001] [INFO] log - Executing Query INSERT INTO comments (user\_id, article\_id, editor\_id, text, c\_time) VALUES ( 30 , 28 , 3 , 'Hey Comment.' , NOW() );

[2020-04-29T16:40:41.169] [INFO] log - Returning from /users/comment/201{"article\_id":"28","editor\_id":"3","user\_id":"30","text":"Hey Comment."}

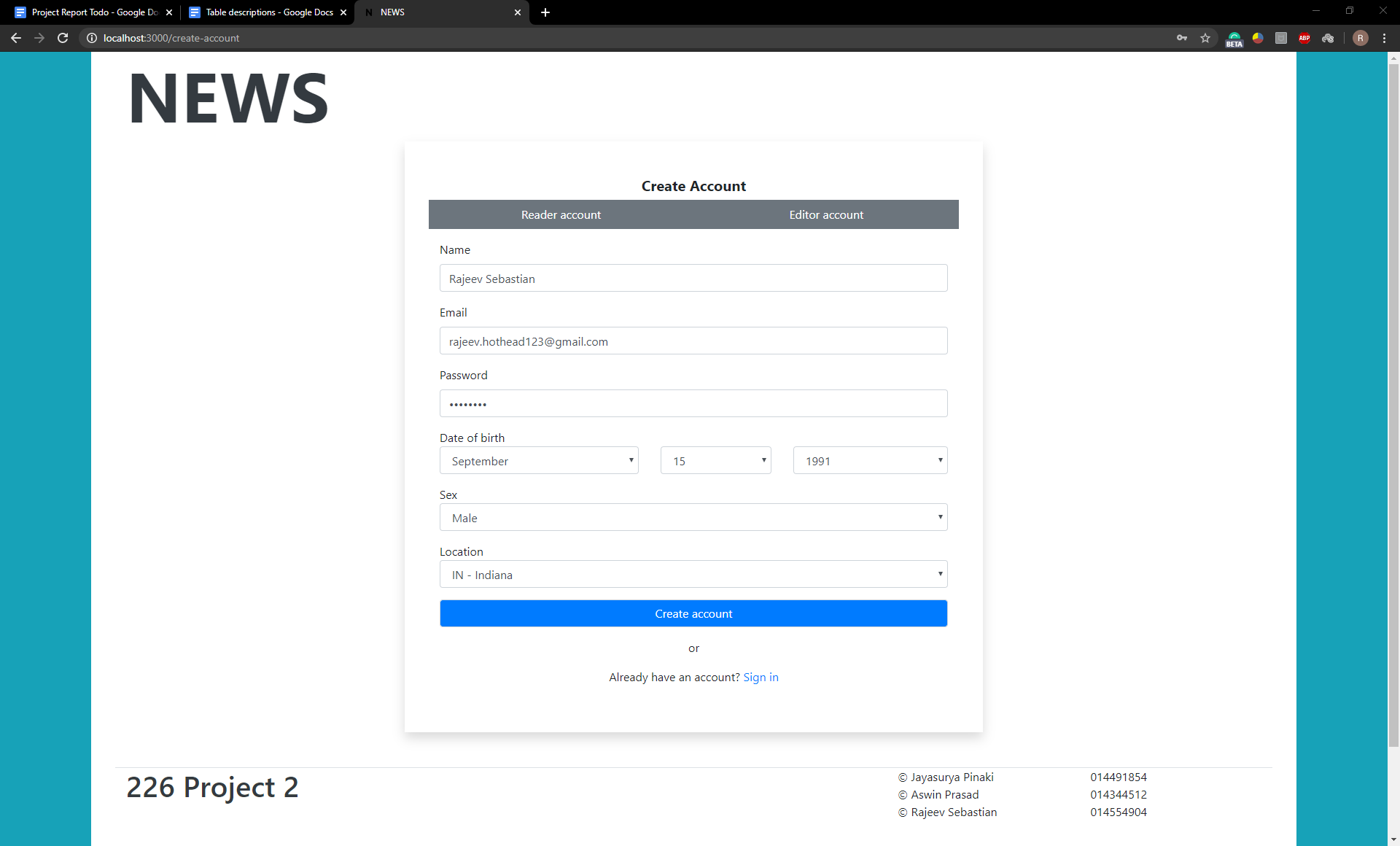
[2020-04-29T16:40:41.179] [INFO] log - Inside /article/view/3/28/30

## Screenshots of App execution

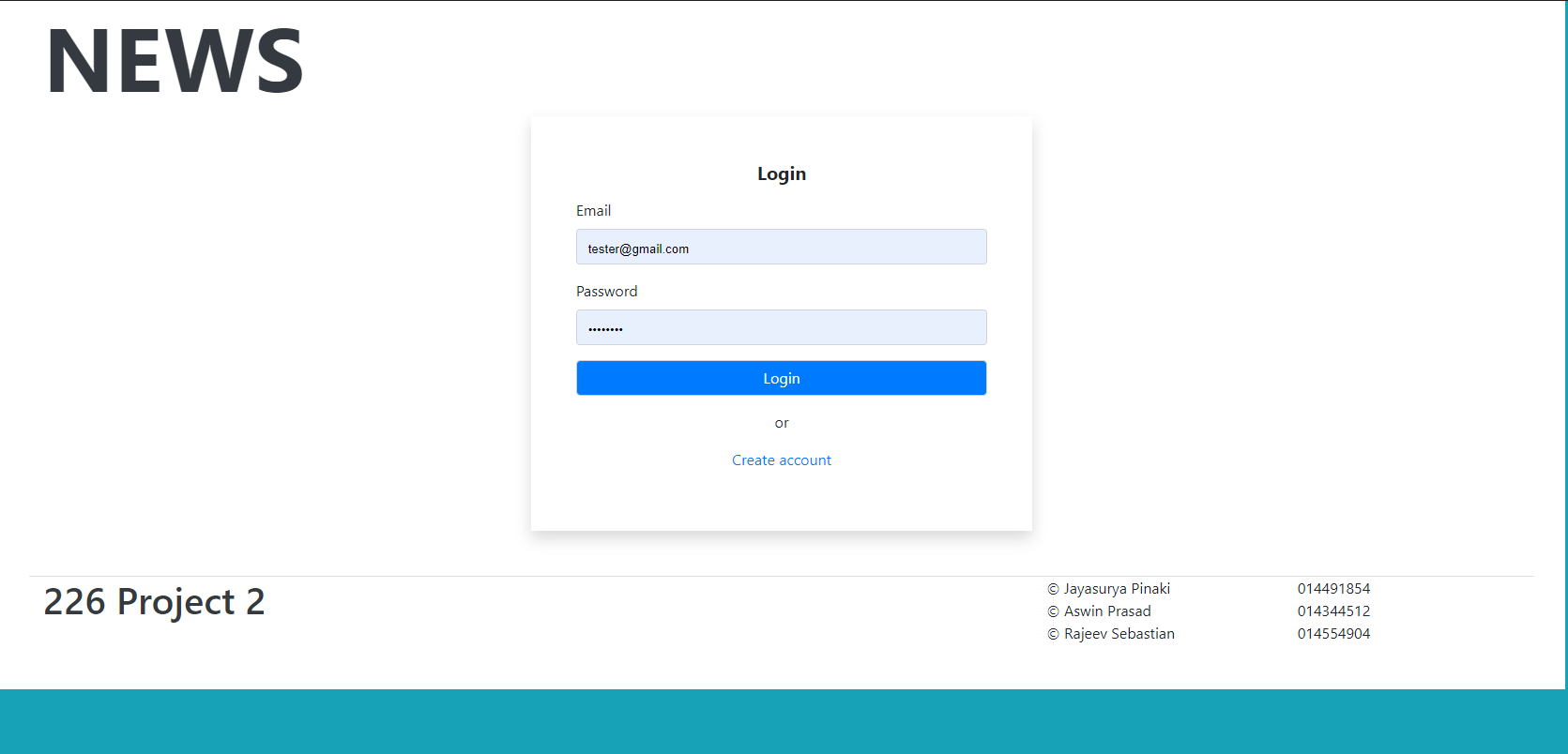
1. Landing Page



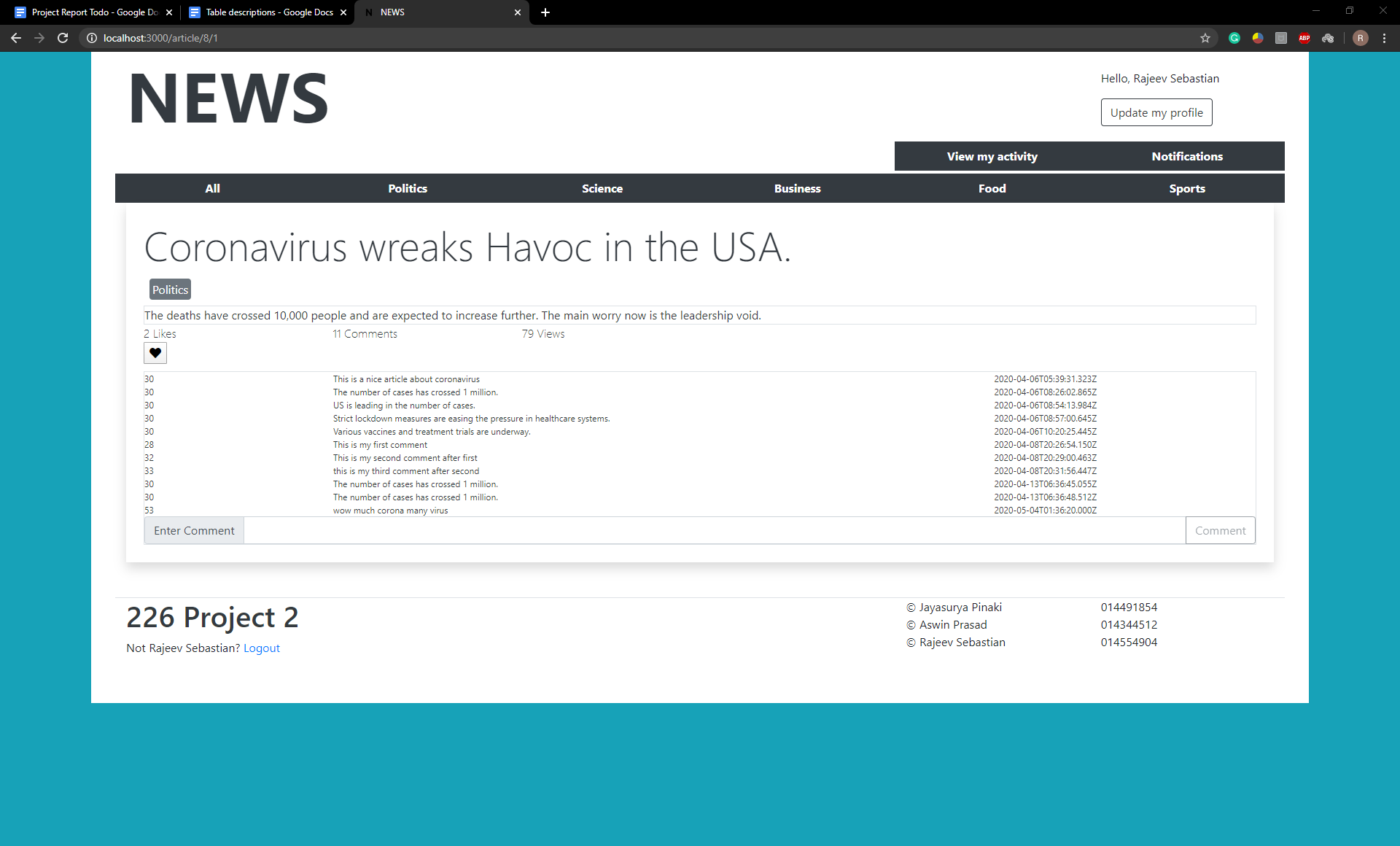
1. Register Page



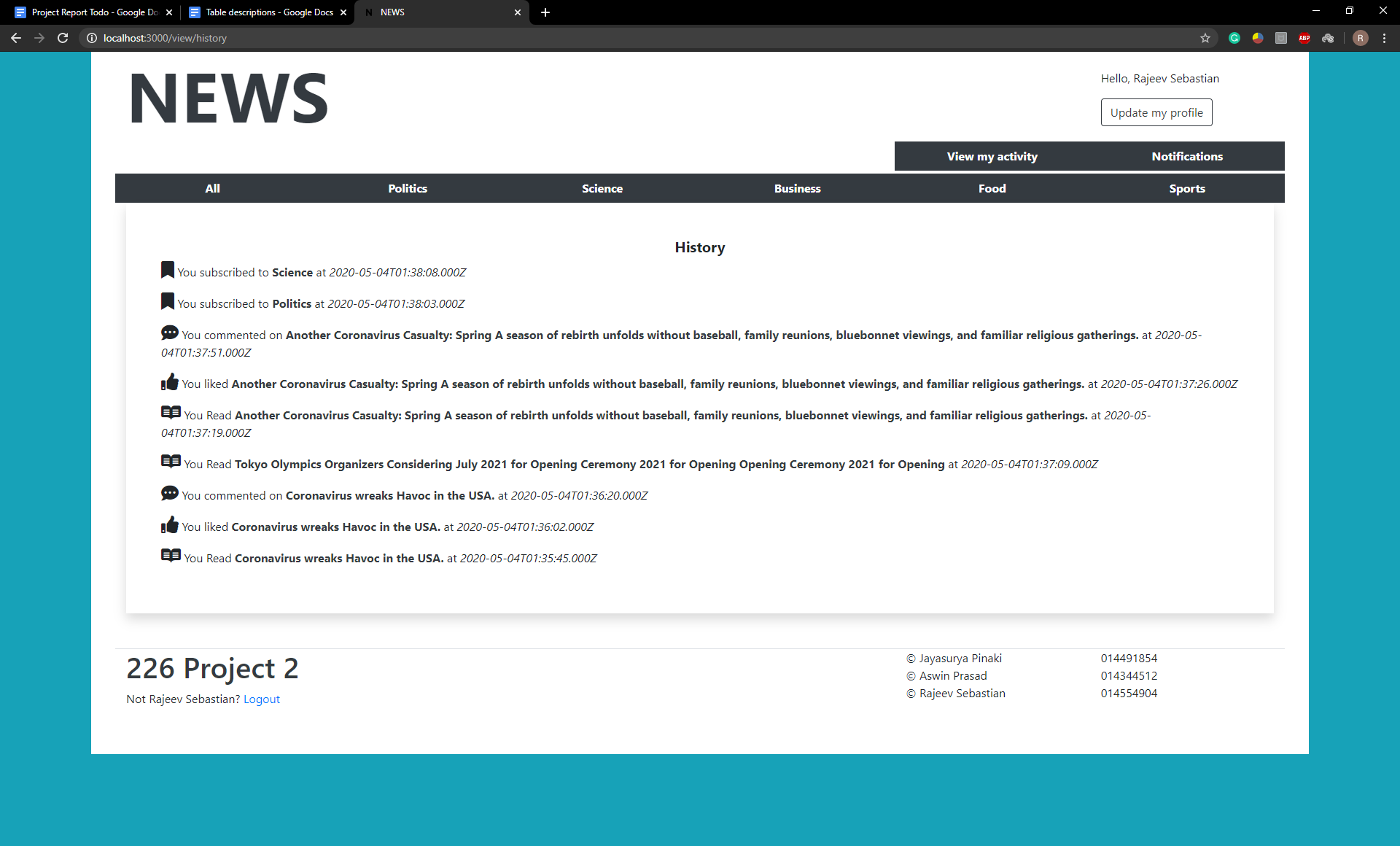
1. Login Page



1. Reading Liking Commenting on a article



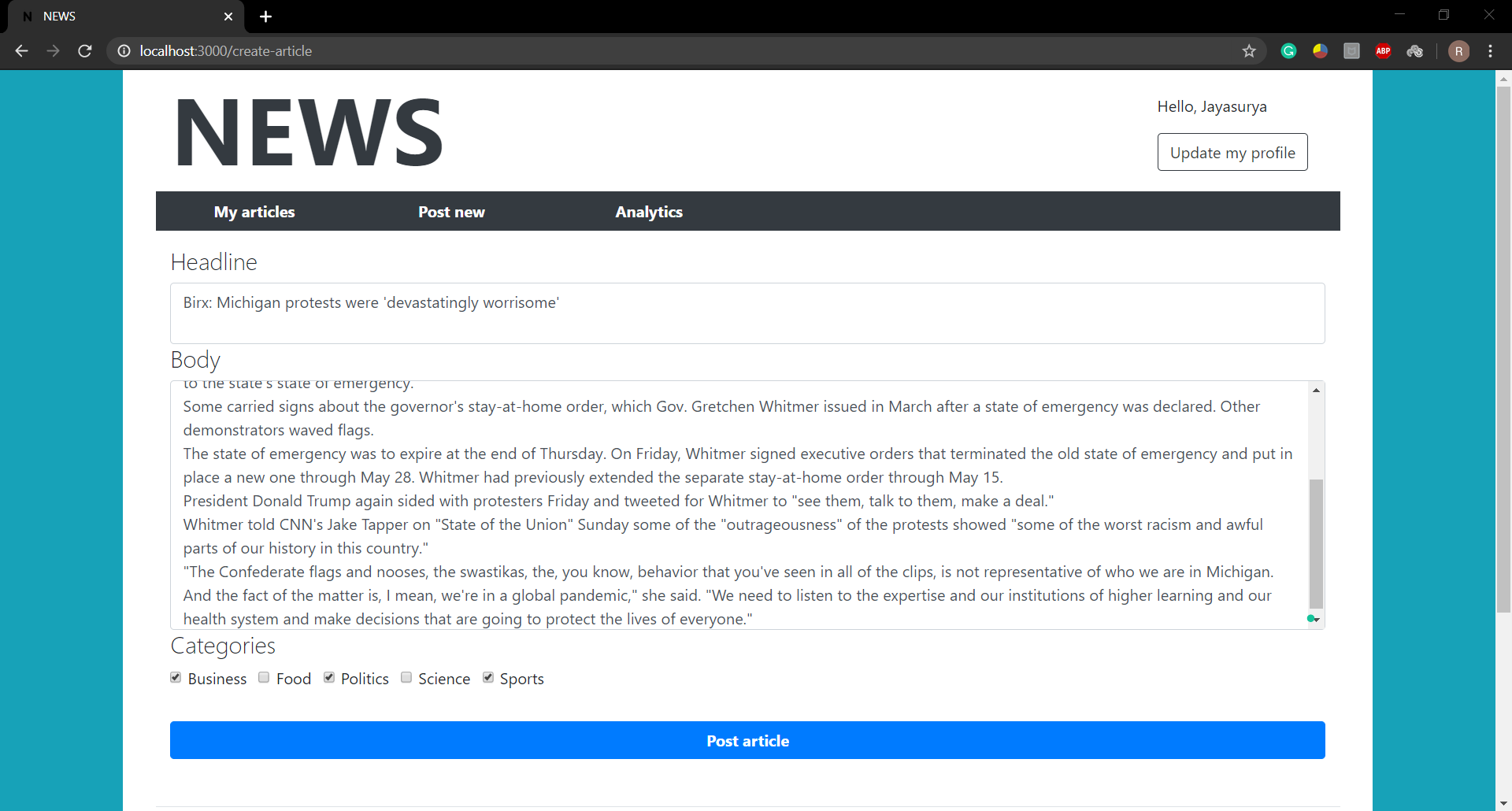
5)User History



6) User Notification

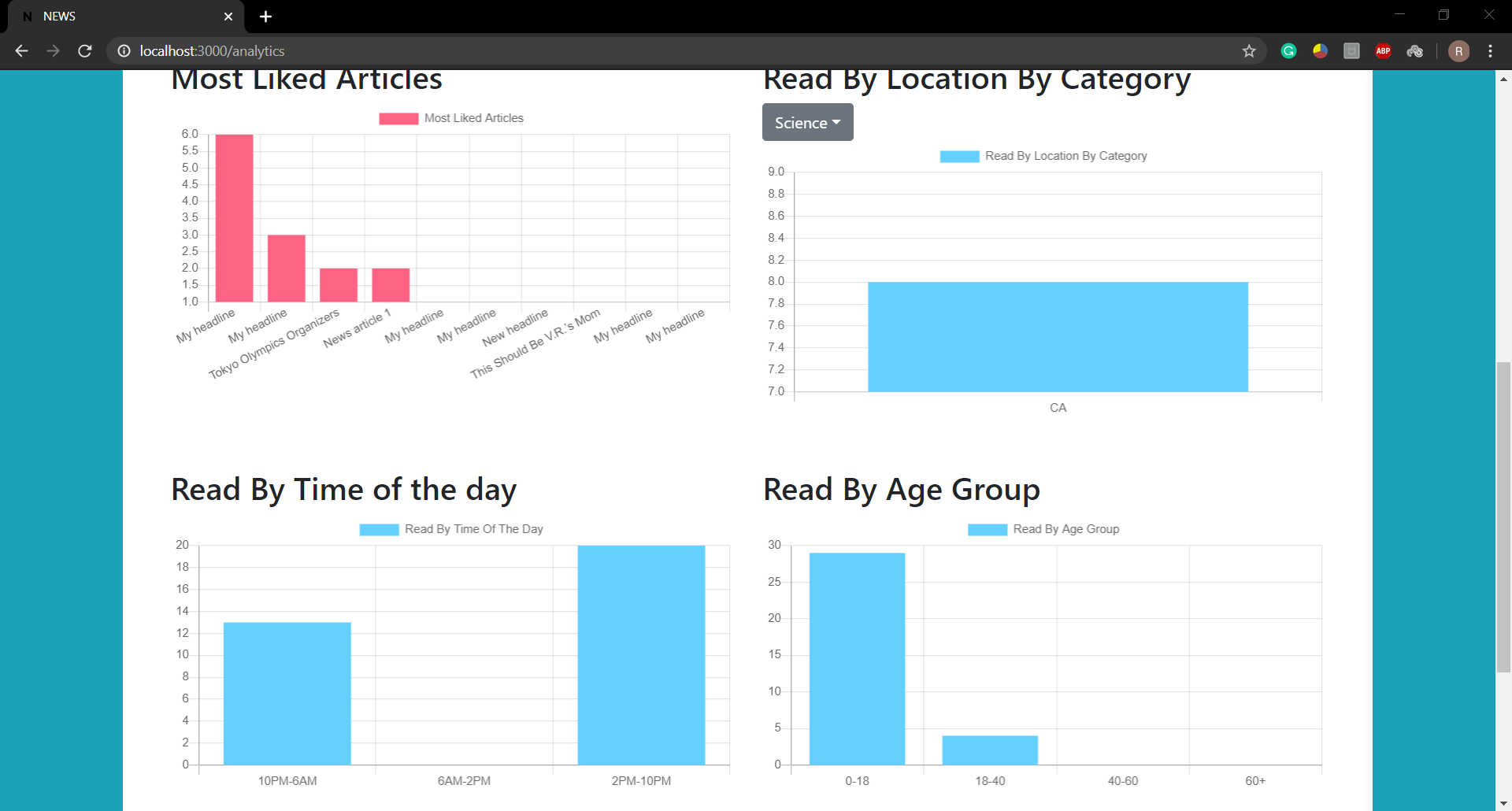


7) Editor posting new article



8) Editor Analytics





## Major design decisions, trade offs

We have decided to include mongoDB in our project after careful consideration. As MySQL is a known, stable database with various functionalities. Most of the scenarios in our application are fine working in the MySQL world but some requirements put more pressure on the underlying system to retrieve data. One of the examples is with the article, there can be 100’s of articles in our system. For each article there can be 1000’s of comments for each article, so comments are known to grow exponentially. In this case it can overwhelm the database by its sheer size. So we decided to start storing comments in a MongoDB document. We also maintained the counts to retrieve the comment count alone in some scenarios. This way we were able to reduce the number of joins required to retrieve the comments.

We created an Article document in mongoDB, which stores the article’s likeCount, commentCount and Comments.

There are trade offs though with the approach of using MongoDB,

* We have to be careful about the constraints while modifying the data as mongoDB won't place any restrictions on those processes.
* Joins are not possible with MondoDB.
* We have noted that for a comparable process, mongoDB is a tad slower than MySQL.
* In many cases, using mongoDB increases the read performance but it is achieved in the loss of memory. As redundant data is being stored.

## Unique Designs

We are proud of the way we have created transactions which involve both a SQL Database(SQL DB) and a NoSQL DB (MongoDB) being updated in the same transaction like whether a user has liked an article or not. The mongoDB update only works when the mysql transaction commits successfully.

Making use of MongoDB has created a reusable framework to insert comments of different types like gifs,jpeg,sounds etc.This will help in making the web site more feature rich and user friendly.

## Test cases and test plan execution

1. **Title:** Login Page – Authenticate successfully on login

**Description:** A registered user should be able to successfully login at gmail.com.

The user can be either a reader or an editor.

On login he will be redirected to the home page which shows all articles for the reader and only those articles written by that editor if the logged in user is an editor.

**Precondition:** The user must already be registered with an email address and password.

**Assumption:** A supported browser is being used.

1. **Title:** Signup Page – Register successfully to the news website

**Description:** An incoming reader must provide his name email address password state where he is living and date of birth to register into the site.

An editor must only provide his email address and password.

**Precondition:** None

**Assumption:** A supported browser is being used.

1. **Title:** Home Page – View all the articles in the website

**Description:** A user logged in or not must be able to view the headlines of all articles in the website.

If he is logged in, then clicking on the headline will redirect to the entire article.

If not, then he will be redirected to the login/signup page.

There is a tabbed view for the various categories and clicking on these tabs will only display headlines of that category

**Precondition:** None

**Assumption:** A supported browser is being used.

1. **Title:** Article Page – View the specific article in the website

**Description:** The reader will be able to go through the article

He may like it and as well as comment on it multiple times

**Precondition:** A user must be successfully logged in and clicked on the headline in

the home page.

**Assumption:** A supported browser is being used.

1. **Title:** Home Page – Subscribe to the specific category in the website

**Description:** On the upper right-hand corner he will be able to click on the

subscribe to button which will make him follow that category.

He will receive notifications whenever an article of that category has been posted

**Precondition:** A user must be successfully logged in and must be in his home page and must be a reader

**Assumption:** A supported browser is being used.

1. **Title:** Notifications Page – View past notifications

**Description:** On the upper right-hand corner he will be able to click on the

notifications tab which will make him redirect to the notifications page.

He will receive notifications whenever an article of that category has been posted.

He will also receive notifications when another user comments on an article he previously commented on.

**Precondition:** A user must be successfully logged in and must be in his home page and must be a reader

**Assumption:** A supported browser is being used.

1. **Title:** Activity History Page – View activity

**Description:** On the upper right-hand corner he will be able to click on the

activity tab which will make him redirect to the activity page.

Here he can see the time he subscribed to a category

Also, when he has read a article and can revisit it.

Information of his comment history

Like history of the articles he has liked.

**Precondition:** A user must be successfully logged in and must be in his home page and must be a reader

**Assumption:** A supported browser is being used.

1. **Title:** Update Account Page – Update account details

**Description:** On the upper right-hand corner he will be able to click on the

update account button which will make him redirect to his account.

Here he will be able to update account details such as name location

and email only if there is no existing user with that email address.

**Precondition:** A user must be successfully logged in and must be in his home page and must be a reader or an editor

**Assumption:** A supported browser is being used.

1. **Title:** Home Page – View all his articles in the website

**Description:** An editor logged in must be able to view the headlines of all his articles on the website.

Clicking on the article will allow him to make edits to the article.

It will update the modified time in the article.

Changes will be reflected to all the readers

**Precondition:** Editor must be logged in

**Assumption:** A supported browser is being used.

1. **Title:** Analytics Page – View analytics of the website

**Description:** An editor logged in must be able to view the analytics of all his articles on the website.

Clicking on the analytics will allow him to see how he is doing.

Here he can see various graphs showing the following

Most read articles.

Most read category

User age

Time of the day articles are read

Where the user is located

**Precondition:** Editor must be logged in

**Assumption:** A supported browser is being used.

## Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Time(hrs)** | **Month Week No** | **Passed / Failed** |
| 1 | 0.5 | March 1st | Passed |
| 2 | 1 | March 2nd | Passed |
| 3 | 0.5 | March 3rd | Passed |
| 4 | 1 | April 1st | Passed |
| 5 | 0.5 | April 1st | Passed |
| 6 | 1.25 | April 2nd | Failed |
| 7 | 1.5 | April 2nd | Passed |
| 8 | 1 | March 3rd | Passed |
| 9 | 1 | April 2nd | Passed |
| 10 | 2.5 | April 2nd | Failed |

## Project post mortem

### Issues uncovered

* Getting activity history and notifications turned out to be more difficult than expected because of the number of tables the data was spread across.
* Analytics needed to be generated frequently for the editor as in real time analytics
* View table needed to be modified to include the time into the primary key
* The article ID could not be set to auto increment. Must be handled in the application code.
* Time generated by MongoDB and the MySQL servers were different. Needed to synchronize them in the application code.

### Implement something differently

* Follow an editor instead of following category or maybe both
* Different type of graphs used for the analytics

### Potential improvements

* Comment with image, gifs.
* Post articles with images and videos
* Report article for content.
* Moderator role who will be able to delete hurtful comments
* Automatically generate proposed category for a given article