

W6 PRACTICE


My SQL

 *At the end of his practice, you should be able to...*


- ✓ Establish a **MySQL connection** on the back-end app
- ✓ Implement a **repository** layer using **MySQL queries**
- ✓ **Test the endpoints** (REST API client + front-end app)
- ✓ **Extends the project** to handle **4 tables** in the database

 *How to start?*

- ✓ Download **start code** from related MS Team assignment
- ✓ Run `npm install` on both front and back projects
- ✓ Run `npm run dev` on both front and back projects to run the client and the server

 *How to submit?*

- ✓ Submit your **code** on MS Team assignment

 *Are you lost?*

To review MySQL queries syntax

https://www.w3schools.com/mysql/mysql_sql.asp

To install My SQL Server (if needed)

<https://dev.mysql.com/doc/refman/8.4/en/windows-installation.html>

<https://dev.mysql.com/downloads/>

To connect Node back end to MySQL

https://www.w3schools.com/nodejs/nodejs_mysql.asp

<https://sidorares.github.io/node-mysql2/docs/documentation>

EXERCISE 1 – MySQL Manipulation

Before starting !

You should have a MySQL Server running. Check it out with bellow command:

```
mysql -u root -p
```

You should see MySQL monitor run properly:

```
C:\Users\PC>mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13
Server version: 9.3.0 MySQL Community Server - GPL
```

If not, you need to install and configure MySQL server properly.

<https://dev.mysql.com/doc/refman/8.4/en/windows-installation.html>

Q1 - Create the database and the table of articles

- Open the terminal and launch MySQL monitor:

```
mysql -u root -p
```

- Create a new database (e.g. **week6Db**) using the command line
- Create a new table (articles) with the columns below:

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
title	varchar(255)	YES		NULL	
content	text	YES		NULL	
journalist	varchar(100)	YES		NULL	
category	varchar(50)	YES		NULL	

Q2 - Review My SQL queries

- Complete the bellow table with the appropriate MySQL query

Use case	My SQL Query
Get all articles	SELECT * FROM articles
Get articles written by the journalist 'RONAN'	SELECT * FROM articles WHERE journalist = 'RONAN';
Add an article	INSERT INTO articles (title, content, journalist, category) value ('Tech Update', 'AI is changing news.', 'RONAN',

	'Tech');
Delete all articles whose title starts with "R"	DELETE FROM articles WHERE title LIKE 'R%';

EXERCISE 2 – MySQL on Backend

For this exercise, you start with a start frontend and a backend code.

The goal for this exercise is to replace the provided mock repository with a MySQL repository.

Q1 - Run Frontend & Backend

Open a dedicated terminal to run the server:

```
cd back
npm i
npm run dev
```

Open a dedicated terminal to run the client:

```
cd front
npm i
npm run dev
```

Open the browser and check the front end is correctly **connected with the back end** :



The project already works as we provide fake data (mock repository).

Let's understand in detail the back and front ends.

Q2 - Look at ArticleForm

How does the component know whether to create a new article or update an existing one?

The component uses the `isEdit` prop to determine the mode:

- If `isEdit` is false, it creates a **new article** using `createArticle(formData)`.
- If `isEdit` is true, it updates an **existing article** using `updateArticle(id, formData)`.

So, the `isEdit` prop controls the logic inside `handleSubmit` to either **create** or **update** based on the context.

Why is the `useParams` hook used in this component? What value does it provide when `isEdit` is true?

`useParams()` is used to extract the `id` from the URL (e.g., `/articles/edit/5` gives `id = 5`).

When `isEdit` is true, the `id` is used to:

- **Fetch the article data** from the backend via `getArticleById(id)`.
- **Update the specific article** via `updateArticle(id, formData)`.

So, `useParams` provides the necessary article ID when editing an existing article.

Explain what happens inside the `useEffect` hook. When does it run, and what is its purpose?

The `useEffect` hook runs **once on component mount** (because of the empty dependency array `[]`). Its purpose:

- When in **edit mode** (`isEdit` is true) and an `id` is present, it calls `fetchArticle(id)` to load the existing article data and populate the form (`formData`).
- This ensures that the form is **pre-filled with the current article values** for editing.

***mount means** the moment when a component is first added to the DOM (i.e., it appears on the screen for the first time).

***populate means** Filling the form with the article's current values.

Ex: You're **populating** the form with data from the article — meaning you're putting the article's title, content, journalist, etc., into the input fields so users can see and edit it.

Q3 - Look at the ArticleList

How are the three promise states (loading, success, and error) handled in the `fetchArticles` function?

The `fetchArticles` function uses `try-catch-finally` to handle the three states of a promise:

1. Loading State:

```
setIsLoading(true);
```

```
setError("");
```

This sets the loading state to true and clears any previous error before the async operation starts.

2. Success State:

```
const data = await getArticles();
```

```
setArticles(data);
```

If the request is successful, it stores the retrieved articles in state using `setArticles`.

3. Error State:

```
catch (err) {
```

```
  setError("Failed to load articles. Please try again.");
```

}
If an error happens (e.g. network issue), it sets an error message.
4. Final State (cleanup):
finally {
setIsLoading(false);
}
No matter what, <code>isLoading</code> is set to <code>false</code> when the operation ends.

What is the role of the `ArticleCard` component, how does it communicate with the parent `ArticleList`?

The `ArticleCard` component is a reusable UI component that **displays a single article** (title, journalist, and action buttons like Edit, Delete, and View).

It communicates with the parent (`ArticleList`) through callback props:

- `onEdit`, `onDelete`, and `onView` are functions passed down from `ArticleList`.
- When the user clicks a button in `ArticleCard`, it calls one of these functions with the article's ID:

`onEdit(article.id)`

`onDelete(article.id)`

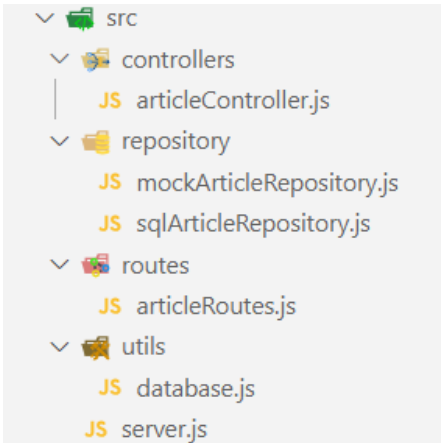
`onView(article.id)`

So the parent stays in control of the logic (e.g., navigation or API calls), while `ArticleCard` handles the UI.

BACK-END

Q4 - Why 3 layers ?

The backend is composed of the below 3 layers : routes, controllers and repository :



Describe the **responsibility** of each **layer** by completing the table below:

LAYER	RESPONSABILITIES
Routes	Define the API endpoints and map them to the corresponding controller functions.
Controller	Handle incoming HTTP requests, process inputs, and call the appropriate repository or service logic. It acts as a bridge between routes and business logic.
Repository	Interact directly with the database; responsible for querying, inserting, updating, and deleting data. Abstracts the data access layer.

Q5 - Implement the database connection

Here are the files you need to update to **connect the backend to the database**:

FILE	RESPONSABILITIES
/.env	securely store your MySQL database credentials
/utils/ database.js	Holds the MySQL connection setup logic <i>Responsible for creating and exporting a connection pool that other parts of the application can use.</i>
/repository/sqlArticleRepository.js	Provides a clean, reusable interface to interact with the articles table in your MySQL database . <i>Encapsulates all the SQL queries related to articles and exposes them as functions that the rest of your application can call.</i>

Here is what you need to do:

- **.env file**

Create a .env file to securely store your MySQL database credentials.

See <https://www.npmjs.com/package/dotenv>

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=complete this line
DB_NAME=complete this line
PORT=4000
```

- **utils/database.js**
 - Create a **MySQL connection pool** using the credentials from the .env file.
<https://sidorares.github.io/node-mysql2/docs#using-connection-pools>
 - Export this connection pool so it can be used by other modules in the project.
- **repositories/sqlArticleRepository.js**

Implement the following functions to interact with the articles table in the database:

```
getAll() – fetch all articles  
getById(id) – fetch one article by ID  
create(article) – insert a new article  
update(id, article) – update an existing article  
remove(id) – delete an article by ID
```

Use the connection pool from database.js to **execute the SQL queries** inside these functions.

As an example, to implement getAll() :

```
export async function getArticles() {  
  const [rows] = await pool.query("SELECT * FROM articles");  
  return rows;  
}
```

Q6 - Test the endpoints

To test the implementation of MySQL repository (*create, update, remove, get articles...*)

- First, perform tests using a **REST API client** (thunder or postman)
- Then, run the **front-end project** and asset the views work properly

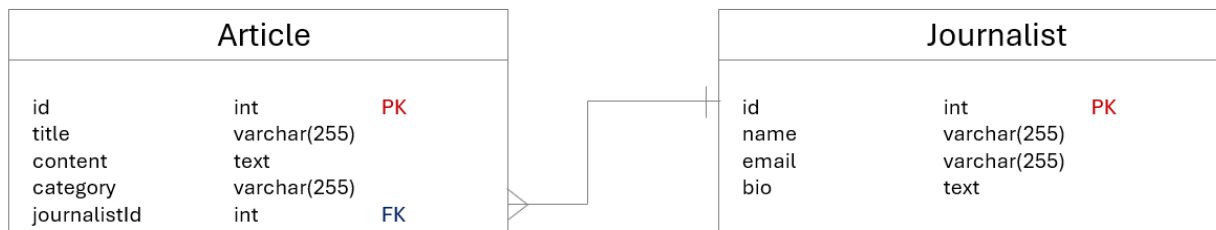
EXERCISE 3 – Handle Journalists

For this exercise, you continue on the previous exercise code.

Now, users want to see **who wrote each article** to better understand the source.

- You will need to update the app, so the article page shows the **journalist's name and info**.
- You will need to provide a journalist view, showing all articles written by a specific journalist.

Database



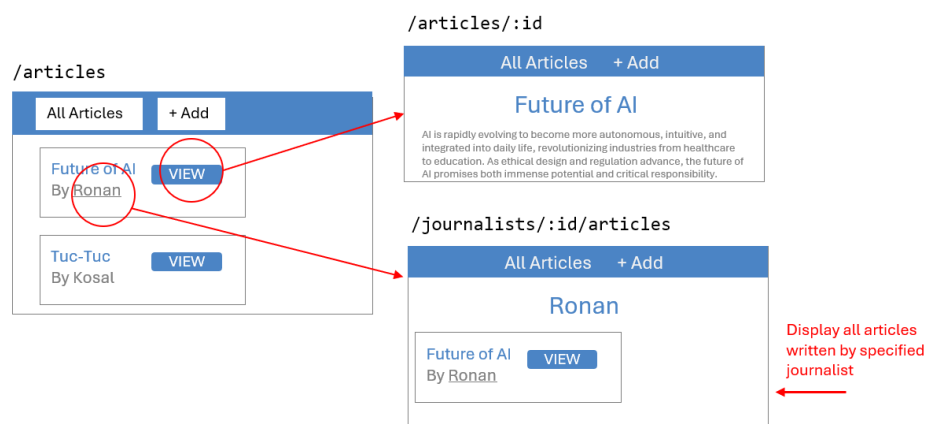
Update your database structure to handle the journalist table

- Create **journalists** table with fields: id, name, email, bio
- Update **articles** table to include **journalist_id** foreign key
- **Populate the database** with some fake data

Back End

- Implement **repository** methods:
 - Fetch articles with joined journalist name (using **SQL JOIN**)
 - Fetch all articles written by a specific journalist name (using **SQL JOIN**)
- Add **controller** functions:
 - Get all articles by journalist ID
- Define **new API routes**:
 - GET `/api/articles/:id` article + journalist name.
 - GET `/api/journalists/:id/articles` articles list by journalist.(display journalist info and articles)

Front End



An additional view shall display all articles written by the specific journalist

- Update **Article Details page**:
 - Display journalist name alongside the article.
- Create **Journalist Articles List page**:
 - Display all articles by selected journalist.
- Add navigation:
 - From Article Details page, allow users to click journalist name to view **that journalist's articles**.
- Update API calls:
 - Fetch combined article + journalist data.
 - Fetch articles filtered by journalist ID.

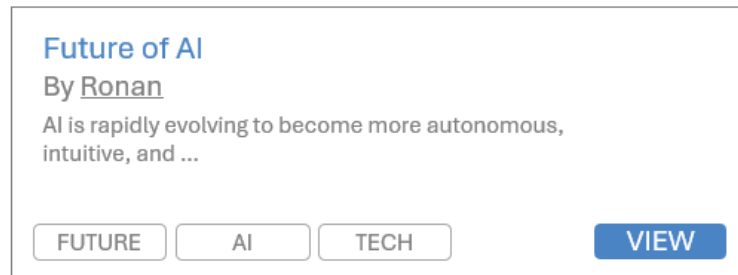
EXERCISE 4 – Handle Tags

BONUS

For this exercise, you continue on the previous exercise code.

Now, users want to easily **assign tags to articles**.

The users can then filter articles by selecting different tags.



You will need to add categories to articles and let users filter the article list by selecting a category.

Database

- Create a new **table** Category (id, name).
 - *What kind of relationships do we have between articles and categories?*
 - *The relationship between articles and categories is a many-to-one relationship, where each article is linked to one category, and each category can be linked to multiple articles.*

Back End:

- Implement repository methods to:
 - Retrieve all categories.
 - Retrieve all articles filtered by category, using JOIN to include category name.
- Add a new API endpoint to get articles by category ID.

Front End:

- Add a **multiple categories filter UI component** on the **article list page** (multiple choice dropdown, chipset selector).
- When categories are selected, fetch and display only articles in those categories.
- Display categories names alongside articles in the list.