Final Report

Owen Maynard - 10770644

Introduction

In this report you will find explanations on the micro service, explanations on the design and how the project was planned out, I will outline the legal, social, ethical, and professional issues of the project. Finally, I will discuss the implementation and evaluate the effectiveness of the final product.

Github link

Swagger API Link

Background

The micro service that I have implemented is the profile service, it is used use CRUD procedures on the profiles of the user, while also managing things such as followers and favourite activities.

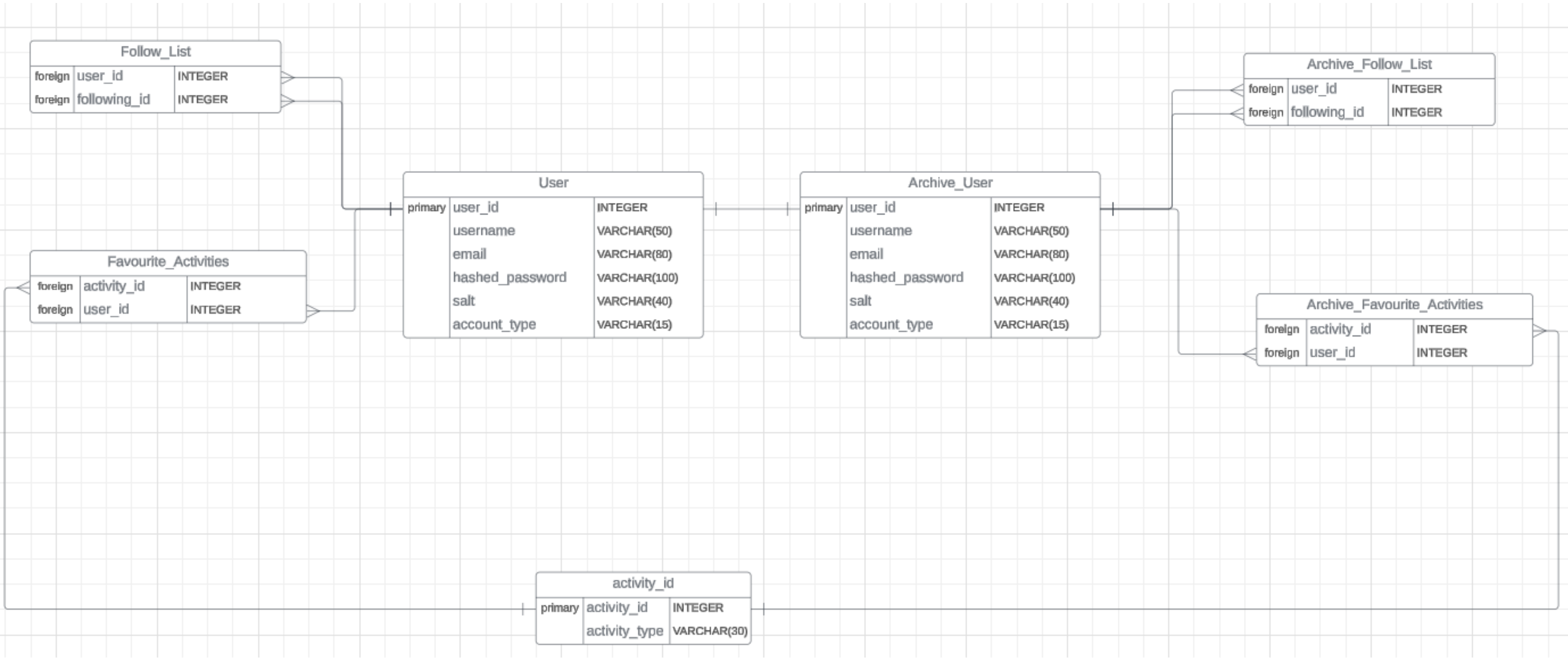
This will all be done via the use of a Swagger interface to show the structure of the API.

I have controllers for: Activities, ArchiveUsers, FavouriteActivities, Follow, FollowersCount, FollowingCount, Login, Logout and finally Users.

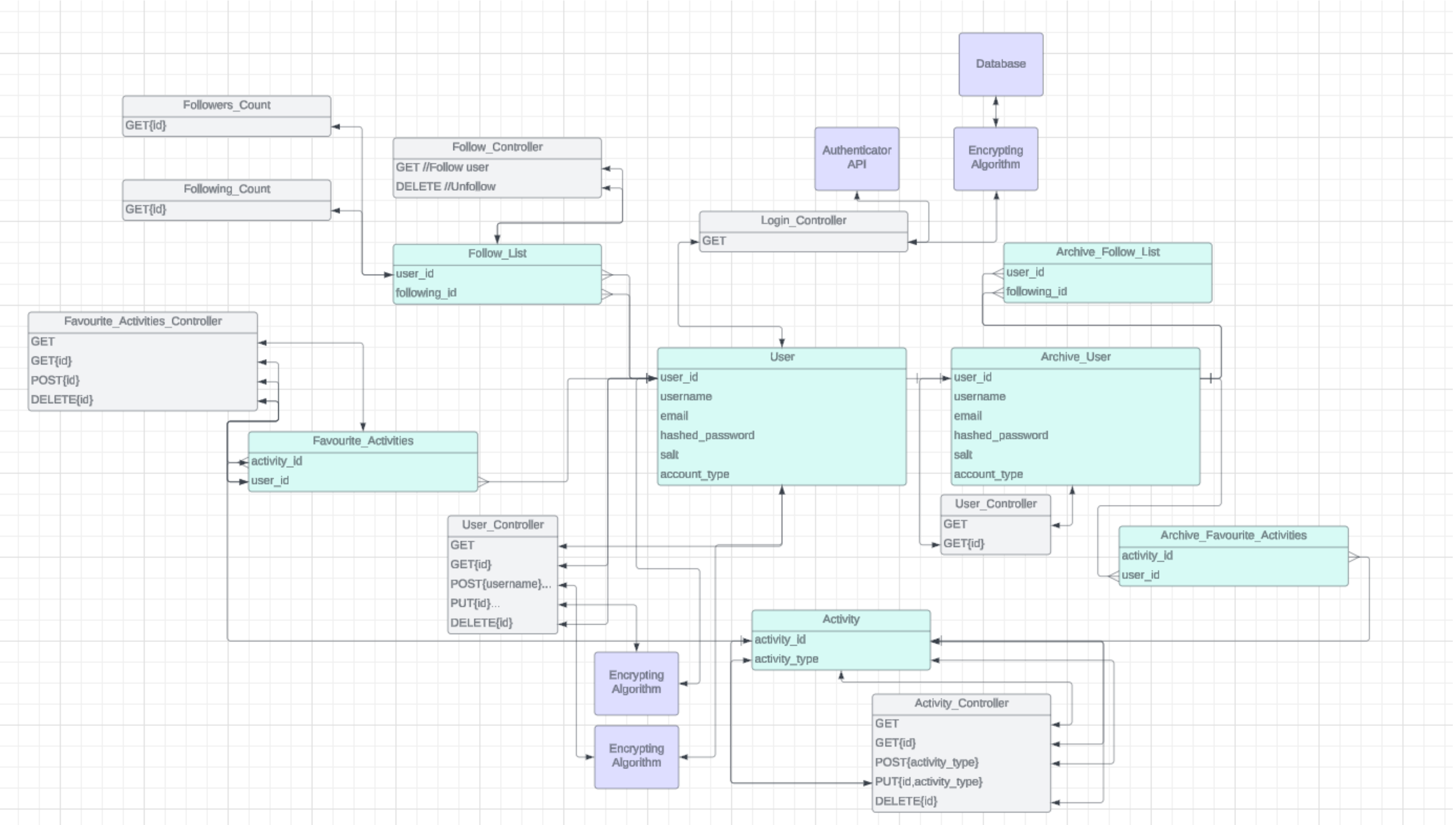
The template that I used to create the project is a C# ASP.NET Web Core API , the purpose of this is to provide CRUD interactions on the database data such as the Users via a web interface. The project has vertical scaling and could be scaled easily. Currently the micro service doesn't interact with any other services however it is designed with the capability to if I were to create other services. It features good security measures such as encryption and authentication. Errors are handled to provide feedback messages to the users. It is well tested and has been deployed onto a hosting server. Documentation could be improved in the future.

Design

Logical ERD:



UML Diagram of the controllers and how they interact with the tables.



Controller Name and Type	Responsibility	Input Data	Return Data
Users - Get	GET All users data		A Json object of all users
Users – Get{id]	Get specific users data	Integer id	A Json object of one user
User – Post{data...}	Create A user	All user data such as String username, String password and String email	A confirmation result on the success status
User – Put{data...}	Edit user data	The new data that you want instead of the old data such as String newEmail	A confirmation result on the success status
User – Delete{id}	Delete user	The integer id of the user to delete	A confirmation result on the success status
Login - Get	Responsible for logging the user in using the authenticator API.	String email String password	A confirmation result if the login was successful
Activities - GET, Get{id}, POST, PUT, DELETE	Responsible for CRUD interactions for the activities data.	Input data such as String activity_id and String activity_type	Get requests return a json object of activities and POST,PUT and DELETE return a confirmation result.
Followers – POST, DELETE, GET{id}//Follower Count, GET{id}//Following Count	Interactions for the followers such as follow unfollow and get follower/following counts.	Integer id for following and unfollowing, uses logged in id Aswell. Integer id for follower/following count too.	Integer for following/follower count. Confirmation result on follow/unfollow.
Favourite Activities – POST, DELETE, GET,GET{id}	Interactions for the favourite activities such as favourite an activity and un favouriting it. Also able to get a list of favourite activities from all or single user.	Integer activity_id Integer id – Get list from specific user.	POST and DELETE are confirmation results. GET and GET id is a json object with a list of activities.

Legal, Social, Ethical and Professional (LSEP)

My implementation makes sure to implement information security, privacy, integrity and preservation. The way that it does this is via encryption, and the permissions that are given to users to help protect unauthorised access to sensitive information.

Encryption helps with information security by ensuring that sensitive data remains confidential and secure from unauthorized access.

It helps with privacy in that it provides users with control over their data as it is well safeguarded, and the user doesn't have to trust other people with their data as much.

It helps with integrity as it helps to prevent unauthorised modifications of data.

It helps with preservation as it is less sought after and is less likely to be modified via unauthorised modifications.

The method that I used to implement the encryption is by double-hashing the passwords that the user enters, along with a randomly generated salt, the hashed password and salt are then stored in the database instead of a plain text password. When the user logs in, the hashed password and salt are retrieved, using the new password with the salt, and checking if the result is the same. This means that the user has entered the correct password.

Another way that my implementation ensures security is via my implementation of session timers that expire if the user has not performed an action within 10 minutes, this helps to make sure that a user's account is secure and cannot be used by other potential bad actors.

My data items are designed in a way that enforces integrity, privacy and security as the controllers have limited access to sensitive information and is designed in a way that maintains data integrity via good use of normalisation and best practices in database management.

I have also mitigated some of OWASP top 10 using methods described here.

Legal – My programs legal issue would be the data protection act as I am storing sensitive information such as email and passwords. I mitigate this by not using real data and by encrypting the passwords. Another one could be the disability act however my program is using swagger which is already compatible for users.

Social - The social issue in my program is that there is currently no implementation for other languages, however this could be easily done as swagger has features that allow you to switch between languages easily.

Ethical - While the disability act is supposed to be for legal reasons, it also becomes an ethical issue when the application does not conform to the standards needed for some users to properly use the application. This again would not be that much of an issue due to swagger already having some tools for accessibility.

Professional - The professional issue that is present in my application is that there are no design choices based on where the API is being used, for example a different country, this however is not that much of an issue as the API would just be used for endpoints and therefore does not really need design.

Implementation

General Information

All of the actions in the API check if the user is logged in first, otherwise they are not allowed to perform the action.

All of these also have links to the code.

Users

GET – Get all users, returns the main view on the database with followers and favourite activities shown

GET{id} – Gets a specific user from main view

POST – Creates a user with the inputs provided, can be used without logging in

PUT – Edits the current logged in user, only admin can set usertype to admin

DELETE – Deletes (Archives) a user, admin only

Login/Logout

GET – Logs in the user using the auth api and the database with the passwords

GET – Logout, clears all login data

Archive Users

GET – Get all archive users, admin only

GET{id} – Get a specific archived user, admin only

Follower/Counts

POST – Follow user

DELETE – Unfollow user

GET{id} – Get Follower count of a user

GET{id} – Get Following count of a user

Activities

GET – Get a list of all activities

GET{id} – Get a specific activity

POST – Create an activity, admin only

PUT – Edit activity, admin only

DELETE – Delete activity, admin only

FavouriteActivities

GET – Get a list of all users and their favourite activities

GET{id} – Gets a list of all favourite activities from a specific user.

POST{id} – Favourites an activity

DELETE{id} – Unfavourite an activity

Logout timer after 10 mins – On login, the time is noted down +10 mins here.

Then, every time a user performs an action this function is called to check if the session is expired. This returns true if the session is expired, if the session is not expired then the session time is reset, and it returns false.

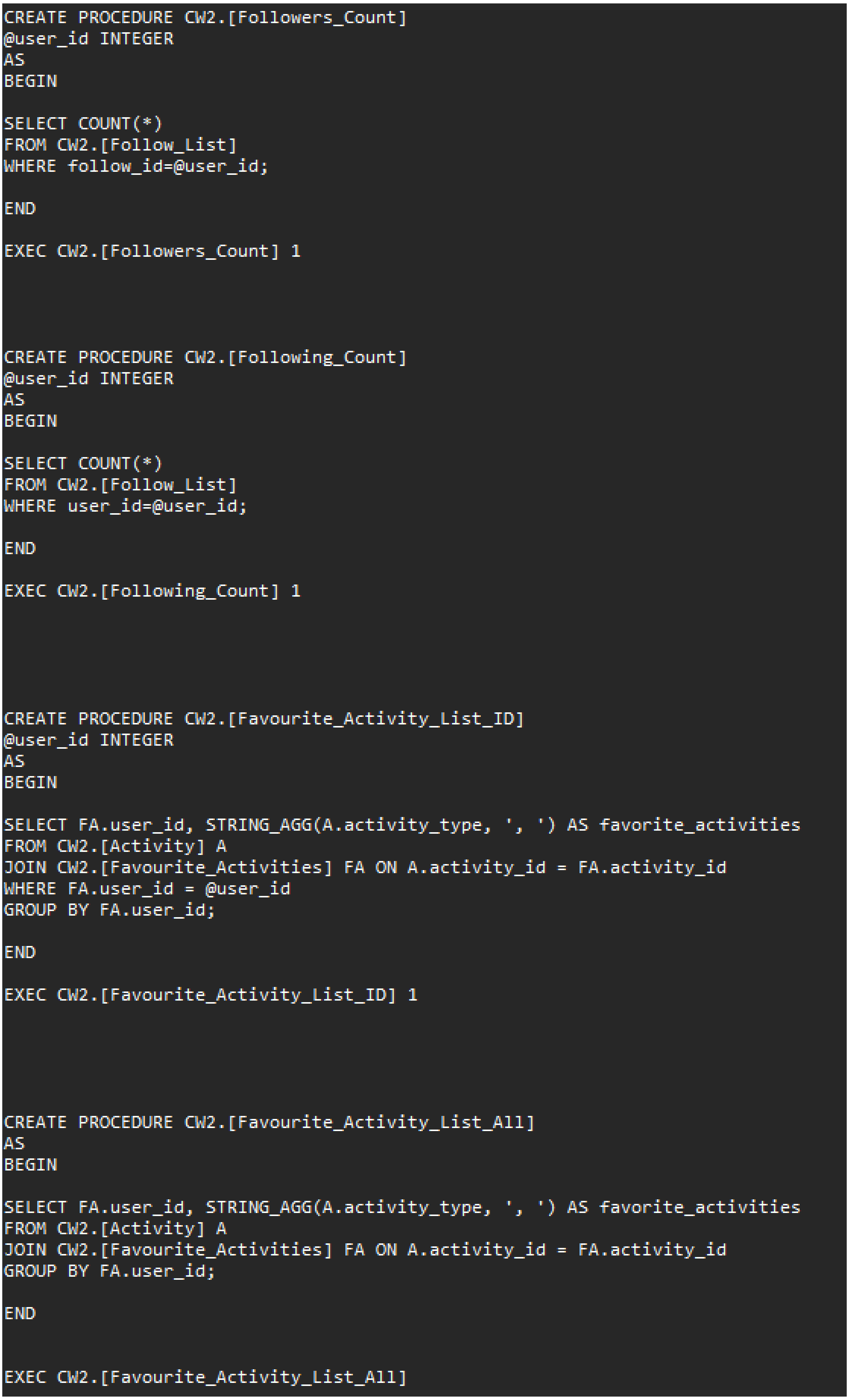
This result is then used to check whether the function needs to be cancelled early, here is an example of the code used for that: code

Implementation of the encrypting of a password and the login into the authentication API.

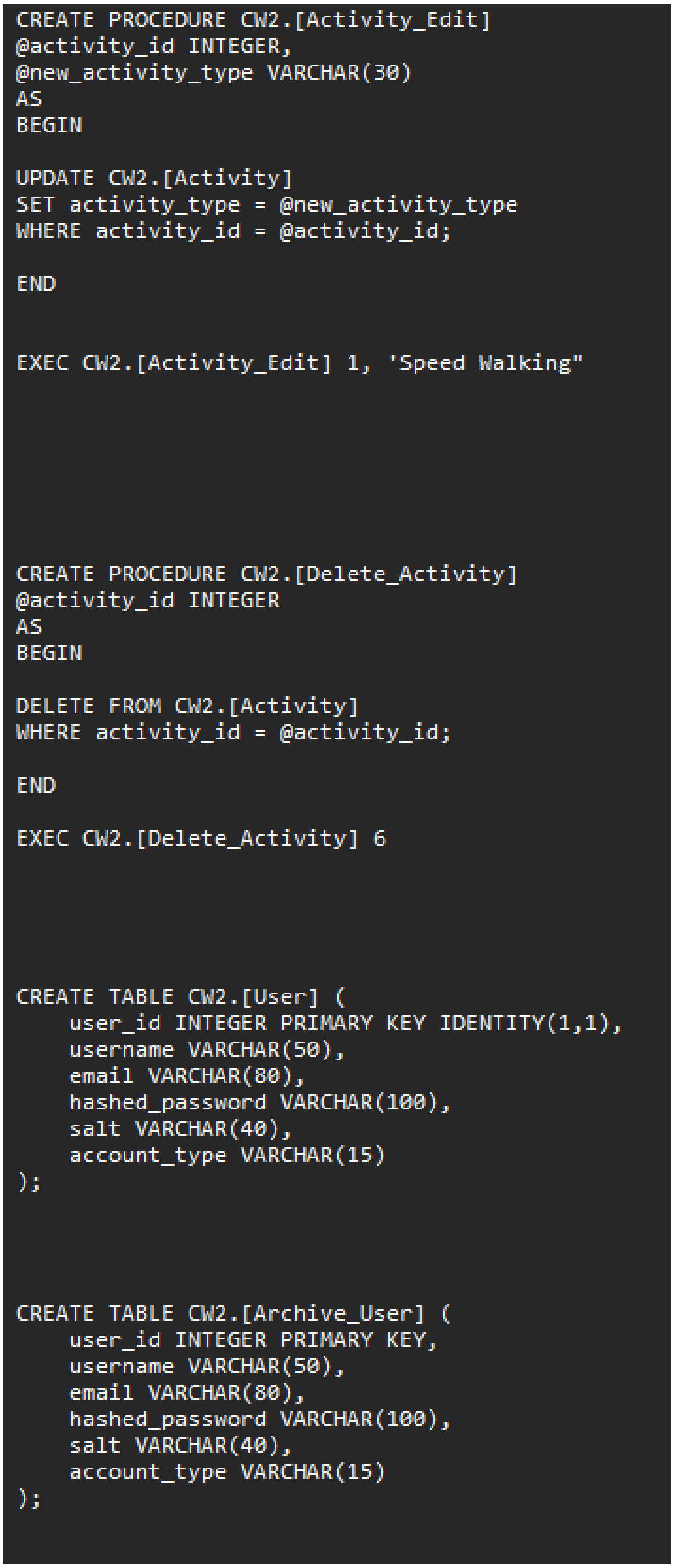
First the authentication api is contacted via this function from this class, which Is a simple http request using the email and password, if the result is true then we log in and set the variable Login.isLoggedIn to true, this is then used in the login controller to see if the login was successful.

After this we log into the database, this is done by getting the hashed password and salt from the api, code here. The code then hashes the login password that has been entered, using the salt from the database too, it then compares the result with the hashed password from the database. If they are the same then the user is allowed to login and the rest of the data from the database is stored.

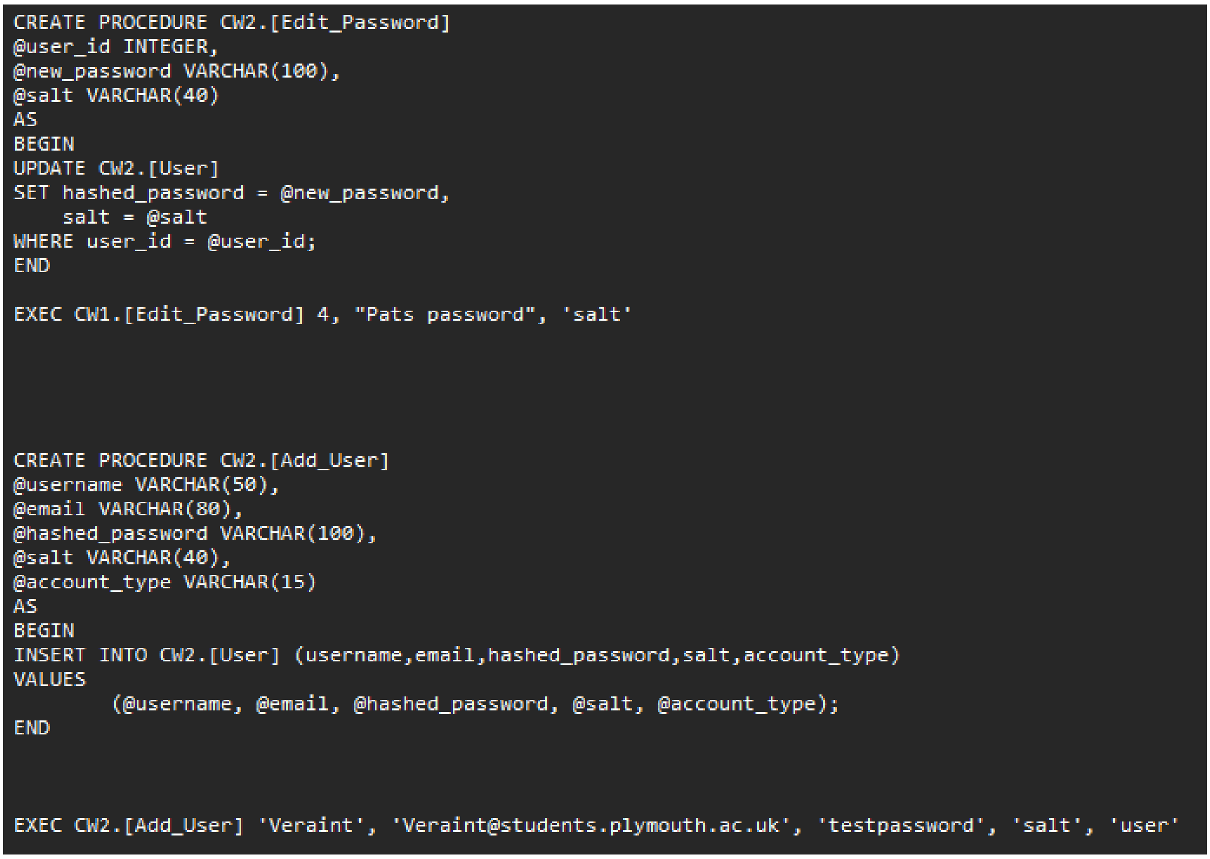
New SQL Stored Procedures



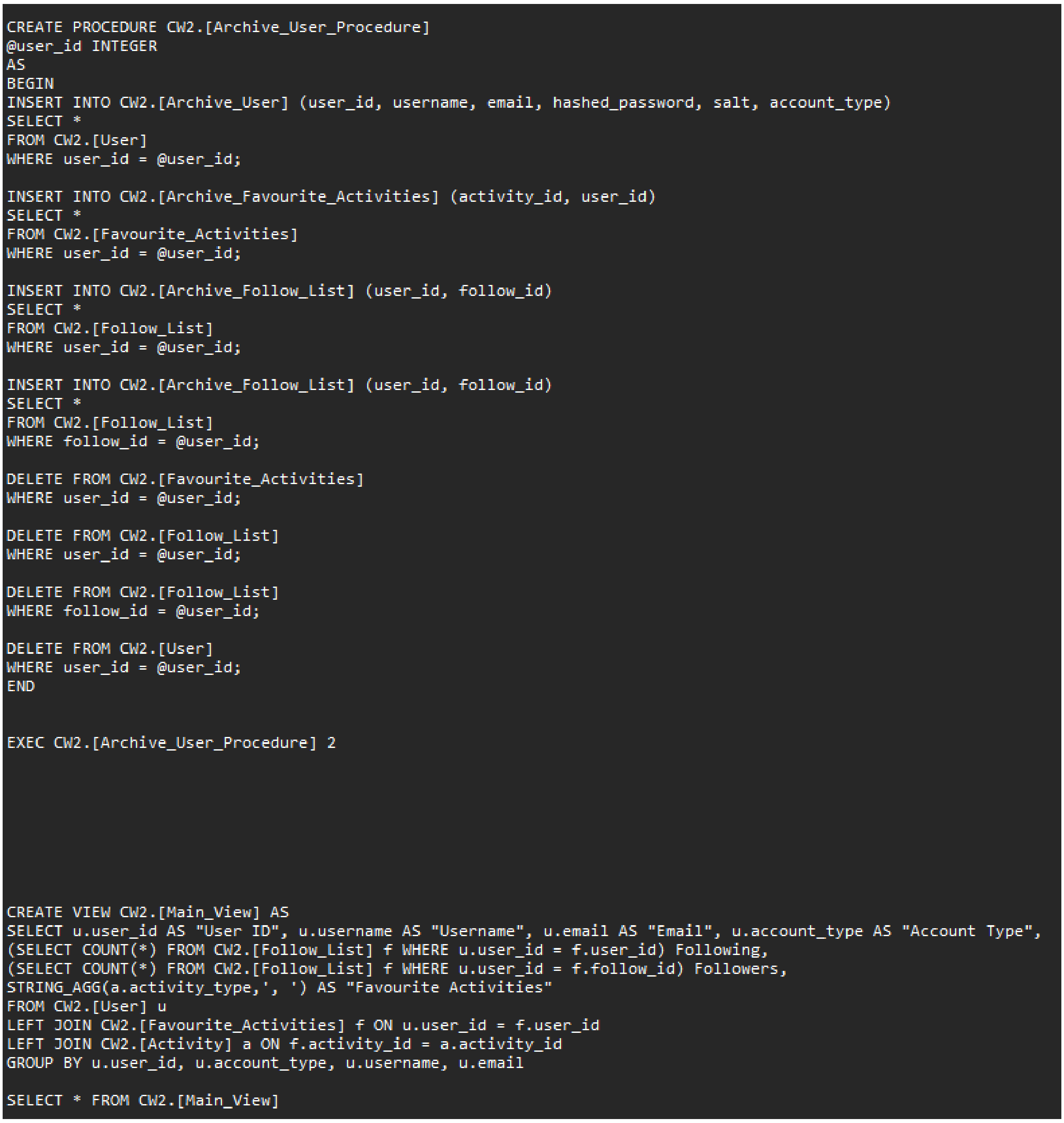
```
CREATE PROCEDURE CW2.[Followers_Count]
@user_id INTEGER
AS
BEGIN

SELECT COUNT(*)
FROM CW2.[Follow_List]
WHERE follow_id=@user_id;

END

EXEC CW2.[Followers_Count] 1




CREATE PROCEDURE CW2.[Following_Count]
@user_id INTEGER
AS
BEGIN

SELECT COUNT(*)
FROM CW2.[Follow_List]
WHERE user_id=@user_id;

END

EXEC CW2.[Following_Count] 1





CREATE PROCEDURE CW2.[Favourite_Activity_List_ID]
@user_id INTEGER
AS
BEGIN

SELECT FA.user_id, STRING_AGG(A.activity_type, ', ') AS favorite_activities
FROM CW2.[Activity] A
JOIN CW2.[Favourite_Activities] FA ON A.activity_id = FA.activity_id
WHERE FA.user_id = @user_id
GROUP BY FA.user_id;

END

EXEC CW2.[Favourite_Activity_List_ID] 1





CREATE PROCEDURE CW2.[Favourite_Activity_List_All]
AS
BEGIN

SELECT FA.user_id, STRING_AGG(A.activity_type, ', ') AS favorite_activities
FROM CW2.[Activity] A
JOIN CW2.[Favourite_Activities] FA ON A.activity_id = FA.activity_id
GROUP BY FA.user_id;

END


EXEC CW2.[Favourite_Activity_List_All]
```



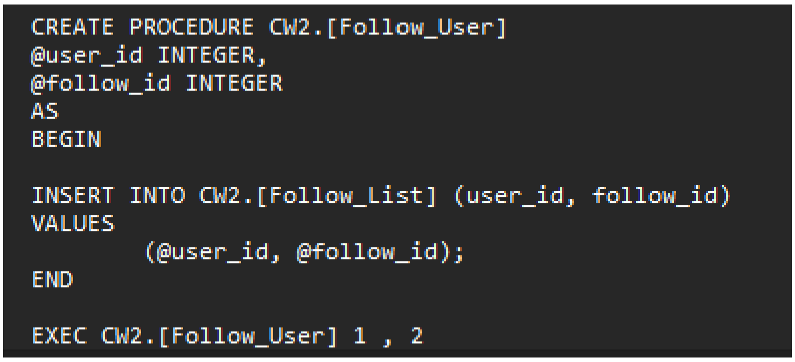
```
CREATE PROCEDURE CW2.[Activity_Edit]
@activity_id INTEGER,
@new_activity_type VARCHAR(30)
AS
BEGIN

UPDATE CW2.[Activity]
SET activity_type = @new_activity_type
WHERE activity_id = @activity_id;

END


EXEC CW2.[Activity_Edit] 1, 'Speed Walking"







CREATE PROCEDURE CW2.[Delete_Activity]
@activity_id INTEGER
AS
BEGIN

DELETE FROM CW2.[Activity]
WHERE activity_id = @activity_id;

END

EXEC CW2.[Delete_Activity] 6





CREATE TABLE CW2.[User] (
    user_id INTEGER PRIMARY KEY IDENTITY(1,1),
    username VARCHAR(50),
    email VARCHAR(80),
    hashed_password VARCHAR(100),
    salt VARCHAR(40),
    account_type VARCHAR(15)
);




CREATE TABLE CW2.[Archive_User] (
    user_id INTEGER PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(80),
    hashed_password VARCHAR(100),
    salt VARCHAR(40),
    account_type VARCHAR(15)
);
```



```
CREATE PROCEDURE CW2.[Edit_Password]
@user_id INTEGER,
@new_password VARCHAR(100),
@salt VARCHAR(40)
AS
BEGIN
UPDATE CW2.[User]
SET hashed_password = @new_password,
    salt = @salt
WHERE user_id = @user_id;
END

EXEC CW1.[Edit_Password] 4, "Pats password", 'salt'






CREATE PROCEDURE CW2.[Add_User]
@username VARCHAR(50),
@email VARCHAR(80),
@hashed_password VARCHAR(100),
@salt VARCHAR(40),
@account_type VARCHAR(15)
AS
BEGIN
INSERT INTO CW2.[User] (username,email,hashed_password,salt,account_type)
VALUES
        (@username, @email, @hashed_password, @salt, @account_type);
END



EXEC CW2.[Add_User] 'Veraint', 'Veraint@students.plymouth.ac.uk', 'testpassword', 'salt', 'user'
```



```
CREATE PROCEDURE CW2.[Archive_User_Procedure]
@user_id INTEGER
AS
BEGIN
INSERT INTO CW2.[Archive_User] (user_id, username, email, hashed_password, salt, account_type)
SELECT *
FROM CW2.[User]
WHERE user_id = @user_id;

INSERT INTO CW2.[Archive_Favourite_Activities] (activity_id, user_id)
SELECT *
FROM CW2.[Favourite_Activities]
WHERE user_id = @user_id;

INSERT INTO CW2.[Archive_Follow_List] (user_id, follow_id)
SELECT *
FROM CW2.[Follow_List]
WHERE user_id = @user_id;

INSERT INTO CW2.[Archive_Follow_List] (user_id, follow_id)
SELECT *
FROM CW2.[Follow_List]
WHERE follow_id = @user_id;

DELETE FROM CW2.[Favourite_Activities]
WHERE user_id = @user_id;

DELETE FROM CW2.[Follow_List]
WHERE user_id = @user_id;

DELETE FROM CW2.[Follow_List]
WHERE follow_id = @user_id;

DELETE FROM CW2.[User]
WHERE user_id = @user_id;
END


EXEC CW2.[Archive_User_Procedure] 2








CREATE VIEW CW2.[Main_View] AS
SELECT u.user_id AS "User ID", u.username AS "Username", u.email AS "Email", u.account_type AS "Account Type",
(SELECT COUNT(*) FROM CW2.[Follow_List] f WHERE u.user_id = f.user_id) Following,
(SELECT COUNT(*) FROM CW2.[Follow_List] f WHERE u.user_id = f.follow_id) Followers,
STRING_AGG(a.activity_type,', ') AS "Favourite Activities"
FROM CW2.[User] u
LEFT JOIN CW2.[Favourite_Activities] f ON u.user_id = f.user_id
LEFT JOIN CW2.[Activity] a ON f.activity_id = a.activity_id
GROUP BY u.user_id, u.account_type, u.username, u.email

SELECT * FROM CW2.[Main_View]
```



```
CREATE PROCEDURE CW2.[Follow_User]
@user_id INTEGER,
@follow_id INTEGER
AS
BEGIN

INSERT INTO CW2.[Follow_List] (user_id, follow_id)
VALUES
        (@user_id, @follow_id);
END

EXEC CW2.[Follow_User] 1 , 2
```

Evaluation

An improvement that could be made is that instead of using double hashing to help remove clusters and reduce collisions I could have instead used uniform probing which is asymptotically equivalent to double hashing. [1] This would have an improvement that means the computational requirements would be a lot less.

Another thing that would be a massive improvement to the project would be to implement good documentation in the API, this would help people that want to use the endpoints be able to understand the data types and how to interact with the API a lot better.

I also could have created a global function to contact the database instead of using repeating code. However, this is not that important as different functions require different handling of the data and therefore the code is not as repetitive as it seems.

Finally, I could heavily improve on fault tolerance, the reason for this is that my program doesn't handle errors in a perfect manor and could be improved to be clearer in what went wrong.

I think that despite the clear improvements that could be made the project is overall well implemented and is incredibly functional for the requirements needed.

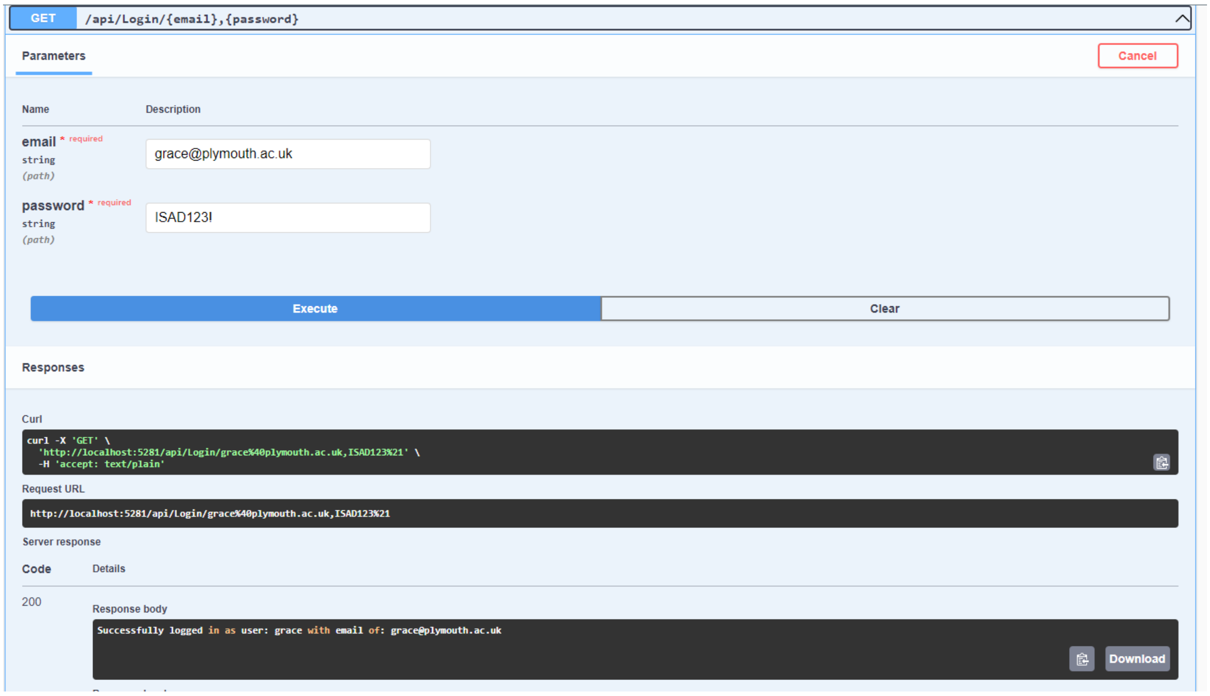
Testing

I have created a testing table to ensure thorough testing of the API.

What is being tested	Inputs	Result Expected	Actual Result	Changes/ Fixes
Login Controller - GET	Incorrect email and password	Does not login to user and does not allow user to perform other actions	Expected Result	N/A
Login Controller - GET	Correct email and password	Logs into system and allows user to perform other actions.	Expected Result	N/A
Login Controller – POST{data…}	New user data such as email password and username	User is successfully created into the database.	Expected Result	N/A
Login Controller – PUT{id…}	Changed user data such as newUsername, using the user_id to edit	User data is successfully edited in the database.	Expected Result	N/A
Login Controller – DELETE{id}	Id of user to delete	Archive the user into the archive tables	Expected Result	
User Controller- GET and GET{id}	Valid id input	Returns a JSON object of the users in the database	Returns a JSON object of the users in the database	N/A
Activity Controller – GET and GET{id}	Valid id input	Returns a JSON object of the	Expected Result	N/A

		activities in the database		
Activity Controller – POST	String activityType	Creates new activity in the table	Expected Result	N/A
Activity Controller – PUT	Int activity_id String activityType	Edits the activity type	Expected Result	N/A
Activity Controller – DELETE	Int activity_id	Deletes the activity	Expected Result	N/A

Login



email * required
string
(path)
grace@plymouth.ac.uk

password * required
string
(path)
ISAD123!

Curl

```
curl -X 'GET' \
  'http://localhost:5281/api/Login/grace%40plymouth.ac.uk,ISAD123%21' \
  -H 'accept: text/plain'
```

Request URL

```
http://localhost:5281/api/Login/grace%40plymouth.ac.uk,ISAD123%21
```

Server response

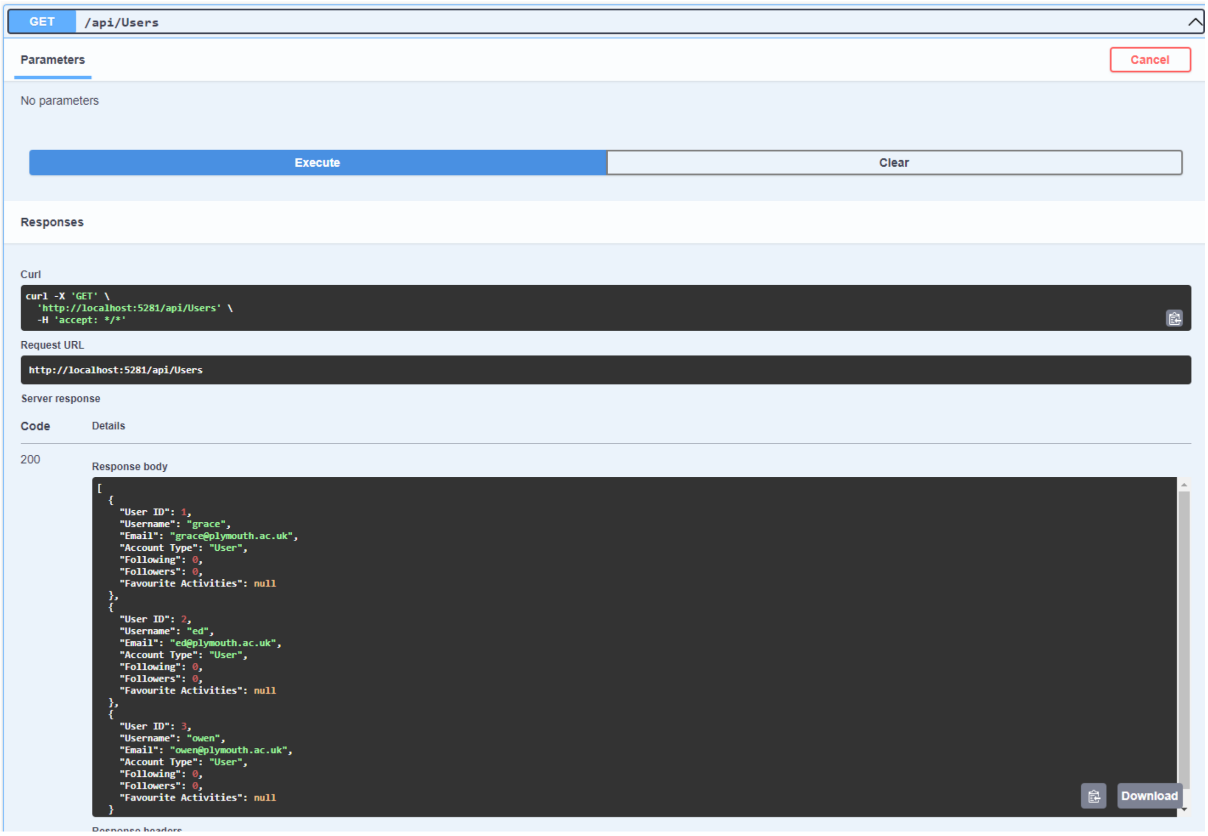
Code Details

200 Response body

```
Successfully logged in as user: grace with email of: grace@plymouth.ac.uk
```

Logout

Get All Users



No parameters

Curl

```
curl -X 'GET' \
  'http://localhost:5281/api/Users' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:5281/api/Users
```

Server response

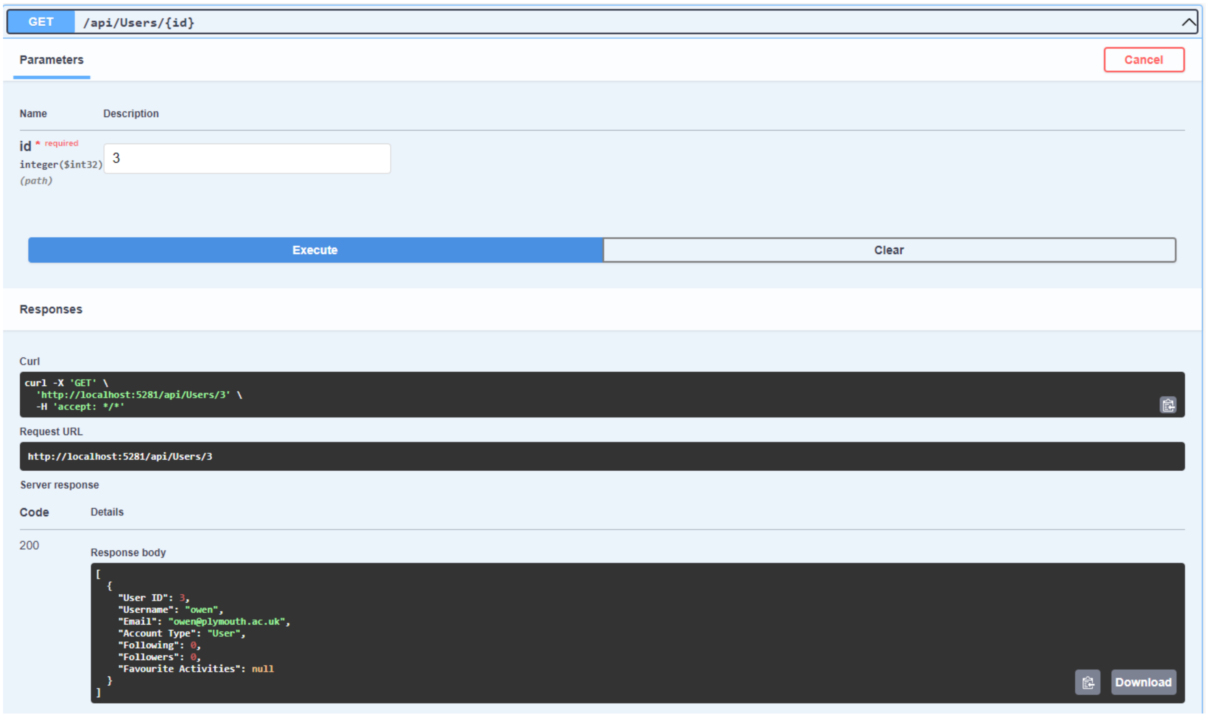
Code Details

200 Response body

```
[
  {
    "User ID": 1,
    "Username": "grace",
    "Email": "grace@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  },
  {
    "User ID": 2,
    "Username": "ed",
    "Email": "ed@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  },
  {
    "User ID": 3,
    "Username": "owen",
    "Email": "owen@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  }
```

Response headers

Get Specific User



Name Description

id * required integer($int32) (path): 3

Curl

```
curl -X 'GET' \
  'http://localhost:5281/api/Users/3' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:5281/api/Users/3
```

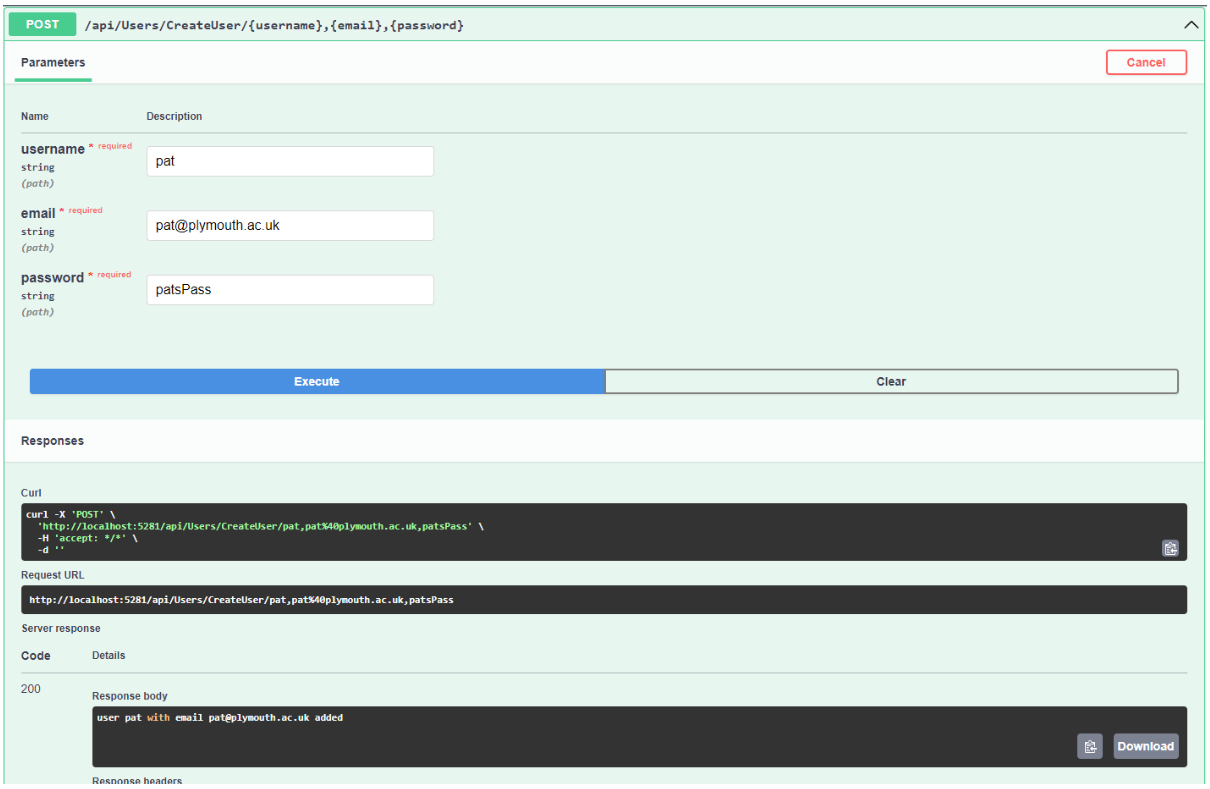
Server response

Code Details

200 Response body

```
[
  {
    "User ID": 3,
    "Username": "owen",
    "Email": "owen@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  }
]
```

Post User (Create)



/api/Users/CreateUser/{username},{email},{password}

Parameters

Name	Description
username * required string *(path)*	pat
email * required string *(path)*	pat@plymouth.ac.uk
password * required string *(path)*	patsPass

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5281/api/Users/CreateUser/pat,pat%40plymouth.ac.uk,patsPass' \
  -H 'accept: */*' \
  -d ''
```

Request URL

```
http://localhost:5281/api/Users/CreateUser/pat,pat%40plymouth.ac.uk,patsPass
```

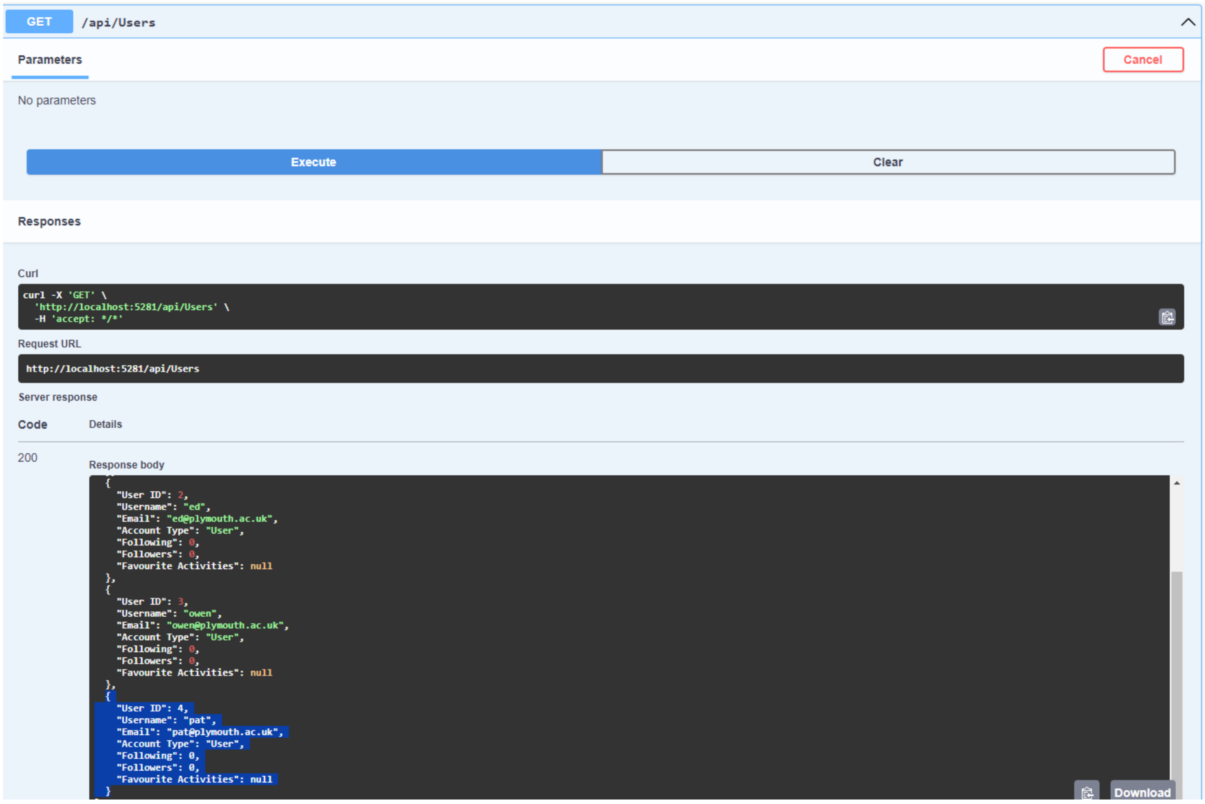
Server response

Code	Details
200	Response body

```
user pat with email pat@plymouth.ac.uk added
```

Response headers

New user is now present in database:



/api/Users

Parameters

No parameters

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5281/api/Users' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:5281/api/Users
```

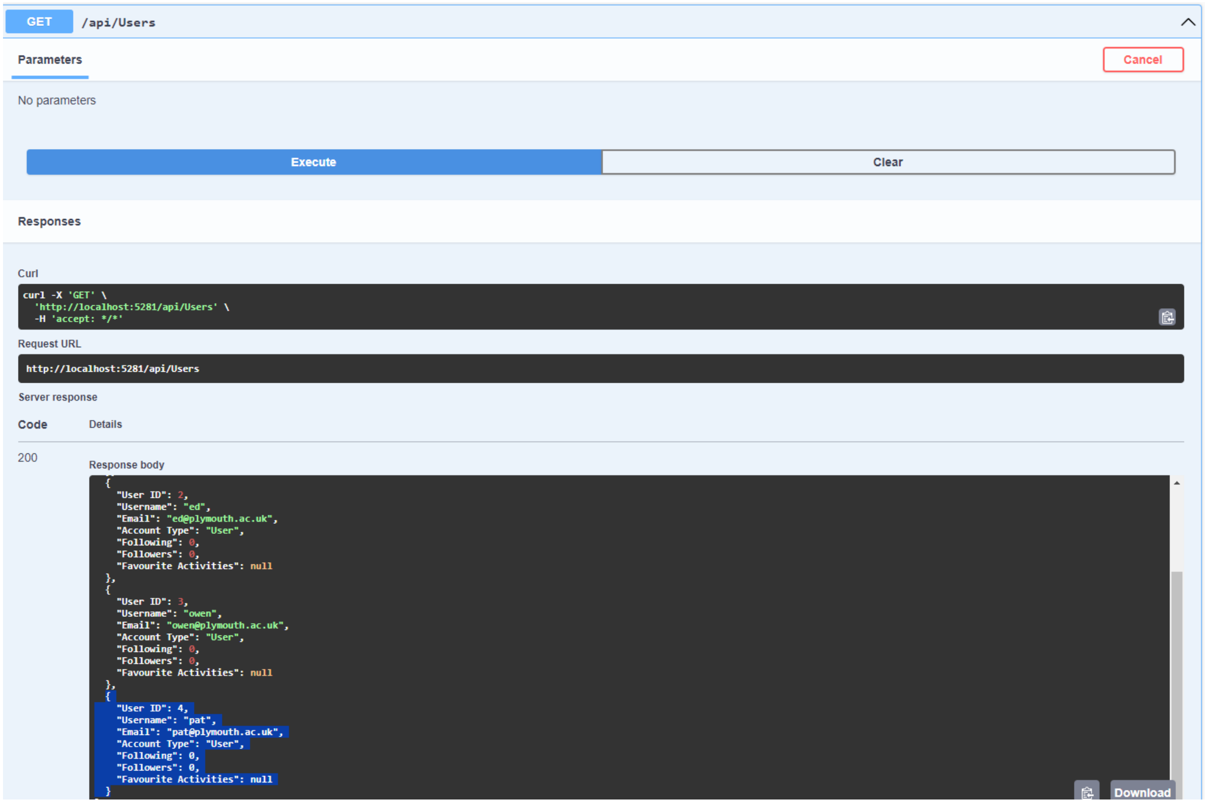
Server response

Code	Details
200	Response body

```
  {
    "User ID": 2,
    "Username": "ed",
    "Email": "ed@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  },
  {
    "User ID": 3,
    "Username": "owen",
    "Email": "owen@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  },
  {
    "User ID": 4,
    "Username": "pat",
    "Email": "pat@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  }
```

Edit User

Before editing:



Curl

```
curl -X 'GET' \
  'http://localhost:5281/api/Users' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:5281/api/Users
```

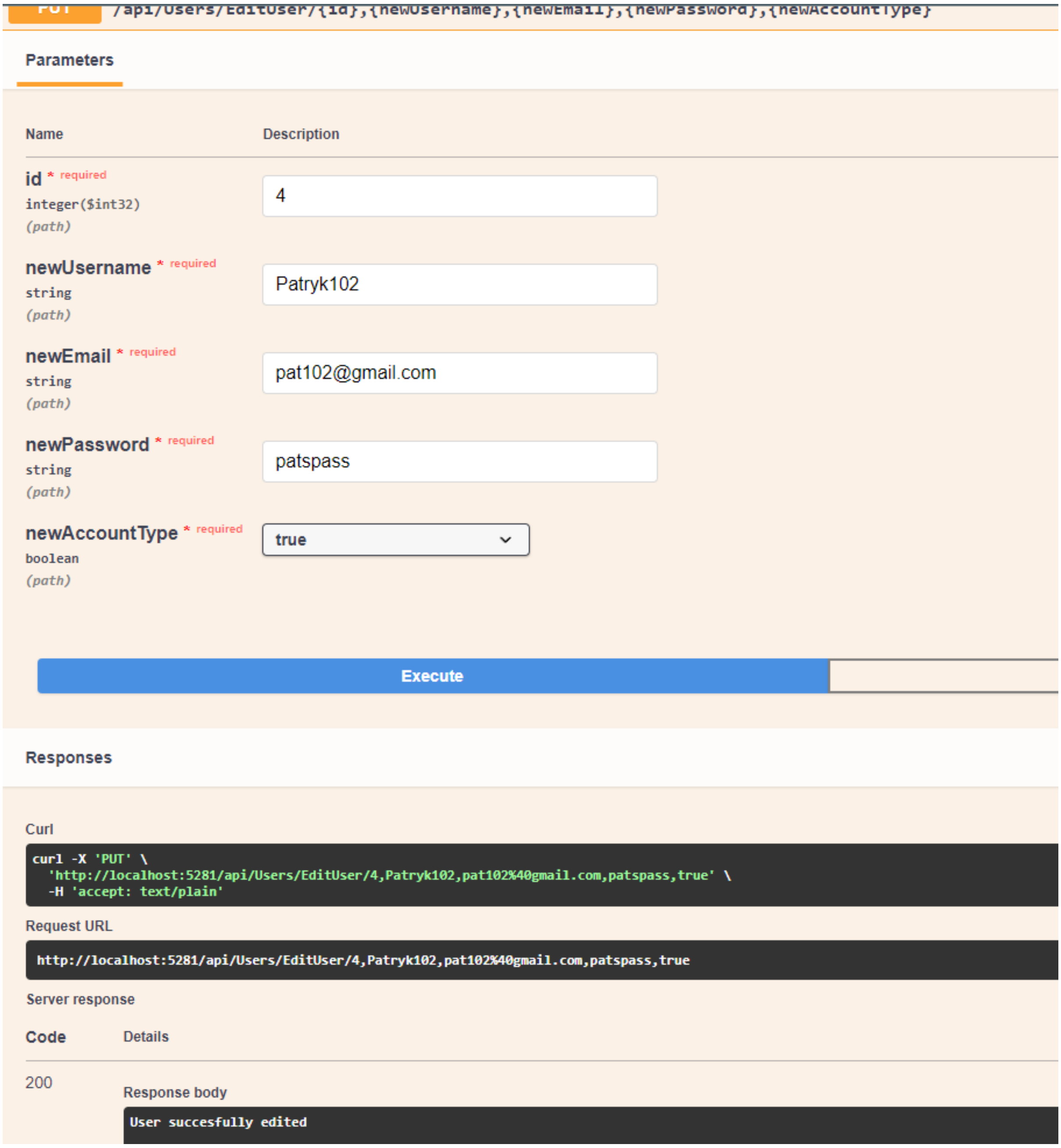
Server response

200

Response body

```
  {
    "User ID": 2,
    "Username": "ed",
    "Email": "ed@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  },
  {
    "User ID": 3,
    "Username": "owen",
    "Email": "owen@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  },
  {
    "User ID": 4,
    "Username": "pat",
    "Email": "pat@plymouth.ac.uk",
    "Account Type": "User",
    "Following": 0,
    "Followers": 0,
    "Favourite Activities": null
  }
```

Editing Data:



PUT /api/Users/EditUser/{id},{newUsername},{newEmail},{newPassword},{newAccountType}

Name	Description
id * required integer($int32) *(path)*	4
newUsername * required string *(path)*	Patryk102
newEmail * required string *(path)*	pat102@gmail.com
newPassword * required string *(path)*	patspass
newAccountType * required boolean *(path)*	true

Curl

```
curl -X 'PUT' \
  'http://localhost:5281/api/Users/EditUser/4,Patryk102,pat102%40gmail.com,patspass,true' \
  -H 'accept: text/plain'
```

Request URL

```
http://localhost:5281/api/Users/EditUser/4,Patryk102,pat102%40gmail.com,patspass,true
```

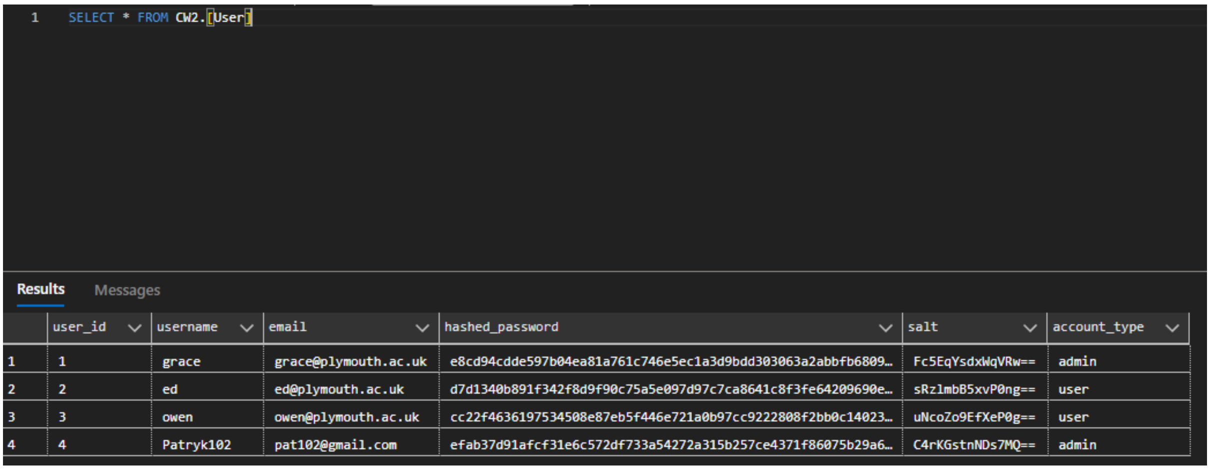
Server response

Code Details

200 Response body

```
User succesfully edited
```

After editing:



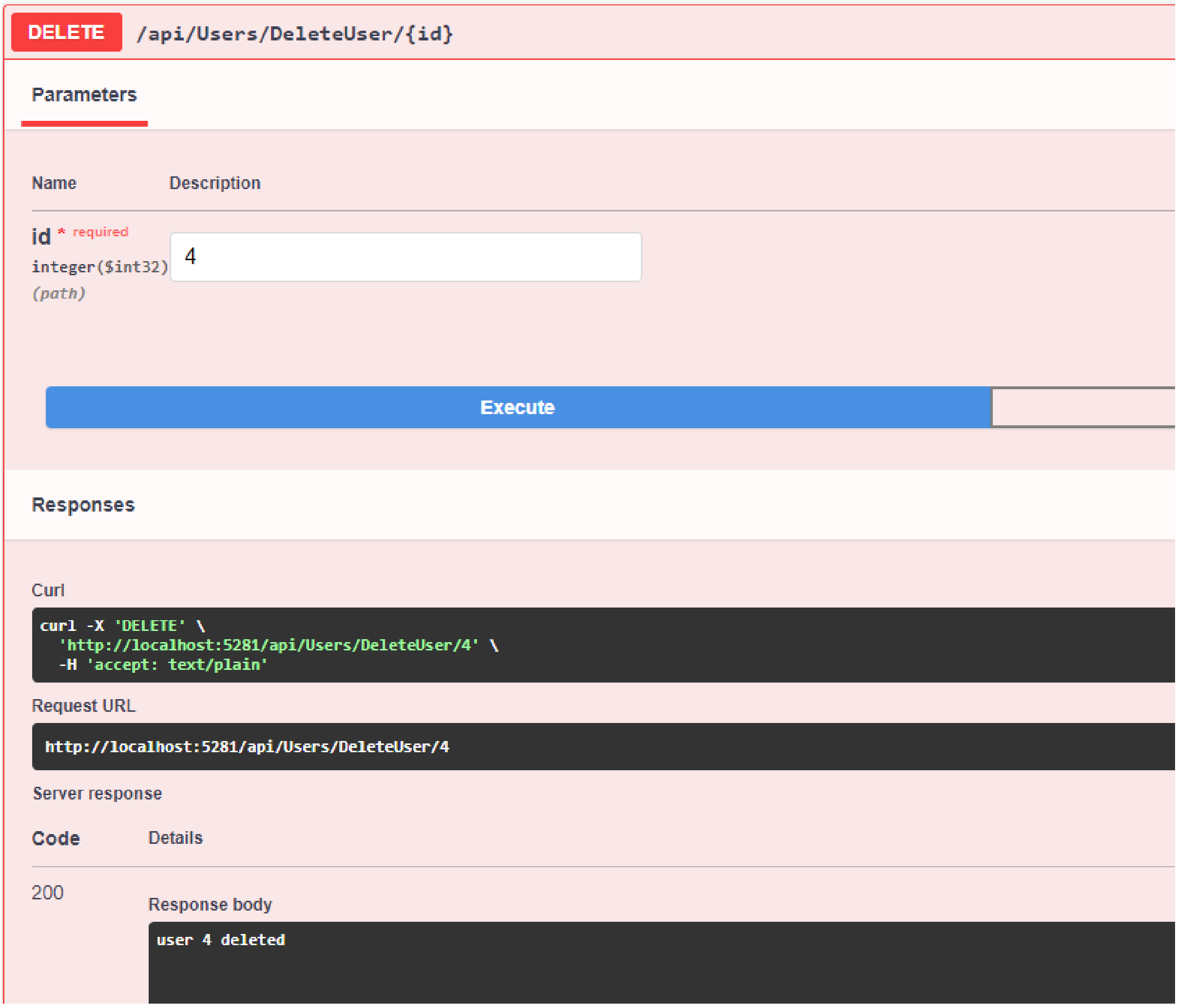
```
SELECT * FROM CW2.[User]
```

	user_id	username	email	hashed_password	salt	account_type
1	1	grace	grace@plymouth.ac.uk	e8cd94cdde597b04ea81a761c746e5ec1a3d9bdd303063a2abbfb6809…	Fc5EqYsdxWqVRw==	admin
2	2	ed	ed@plymouth.ac.uk	d7d1340b891f342f8d9f90c75a5e097d97c7ca8641c8f3fe64209690e…	sRzlmbB5xvP0ng==	user
3	3	owen	owen@plymouth.ac.uk	cc22f4636197534508e87eb5f446e721a0b97cc9222808f2bb0c14023…	uNcoZo9EfXeP0g==	user
4	4	Patryk102	pat102@gmail.com	efab37d91afcf31e6c572df733a54272a315b257ce4371f86075b29a6…	C4rKGstnNDs7MQ==	admin

This is the data stored in the database and as you can see the data has been changed to be the new data as pat is now an admin with the new email and encrypted password.

Delete User (Archive User)

Logged in as grace we are going to delete pat now.



id: 4

Curl

```
curl -X 'DELETE' \
  'http://localhost:5281/api/Users/DeleteUser/4' \
  -H 'accept: text/plain'
```

Request URL

```
http://localhost:5281/api/Users/DeleteUser/4
```

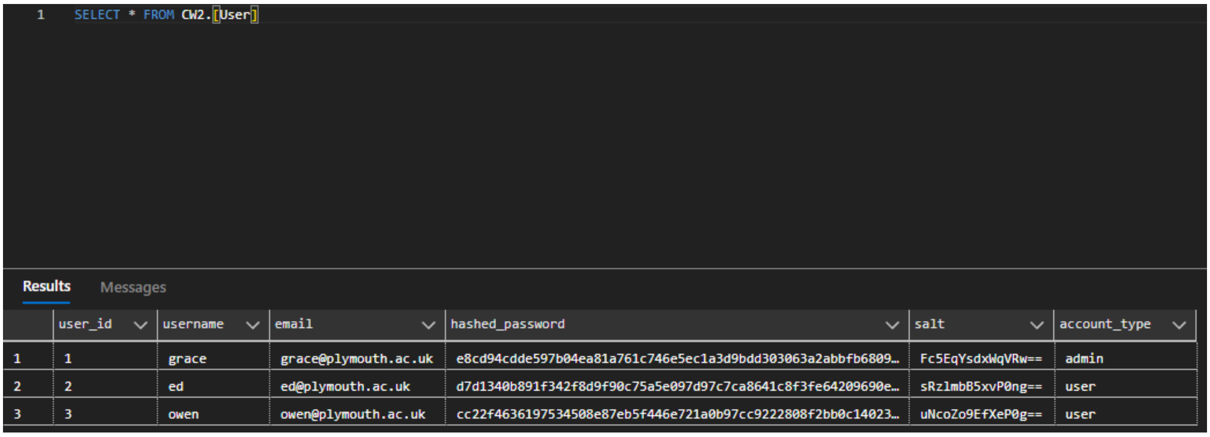
Server response

Code: 200

Response body

```
user 4 deleted
```

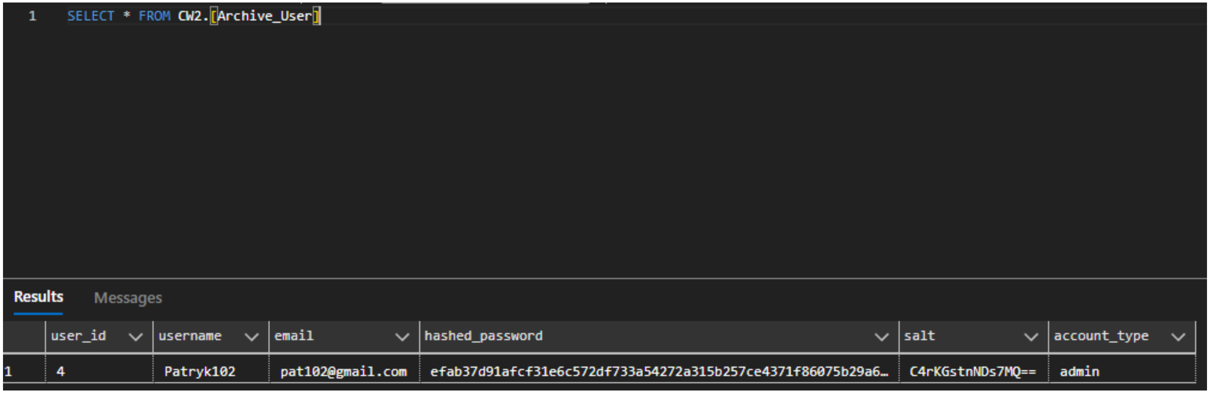
As you can see pat is now deleted from the main users table



```
SELECT * FROM CW2.[User]
```

	user_id	username	email	hashed_password	salt	account_type
1	1	grace	grace@plymouth.ac.uk	e8cd94cdde597b04ea81a761c746e5ec1a3d9bdd303063a2abbfb6809…	Fc5EqYsdxWqVRw==	admin
2	2	ed	ed@plymouth.ac.uk	d7d1340b891f342f8d9f90c75a5e097d97c7ca8641c8f3fe64209690e…	sRzlmbB5xvP0ng==	user
3	3	owen	owen@plymouth.ac.uk	cc22f4636197534508e87eb5f446e721a0b97cc9222808f2bb0c14023…	uNcoZo9EfXeP0g==	user

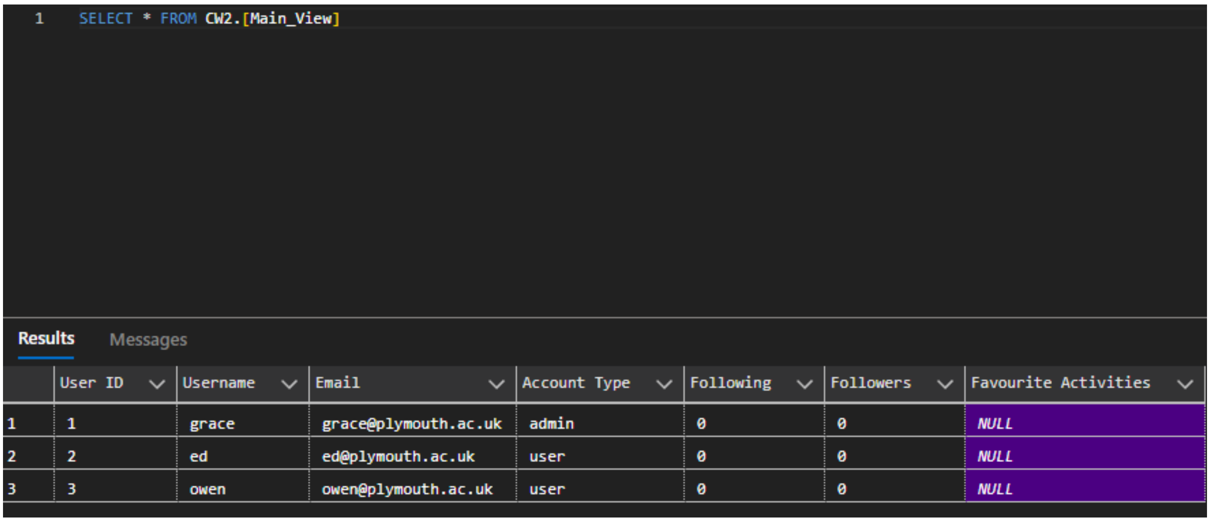
And is present in the archive table



```
SELECT * FROM CW2.[Archive_User]
```

	user_id	username	email	hashed_password	salt	account_type
1	4	Patryk102	pat102@gmail.com	efab37d91afcf31e6c572df733a54272a315b257ce4371f86075b29a6…	C4rKGstnNDs7MQ==	admin

Follow

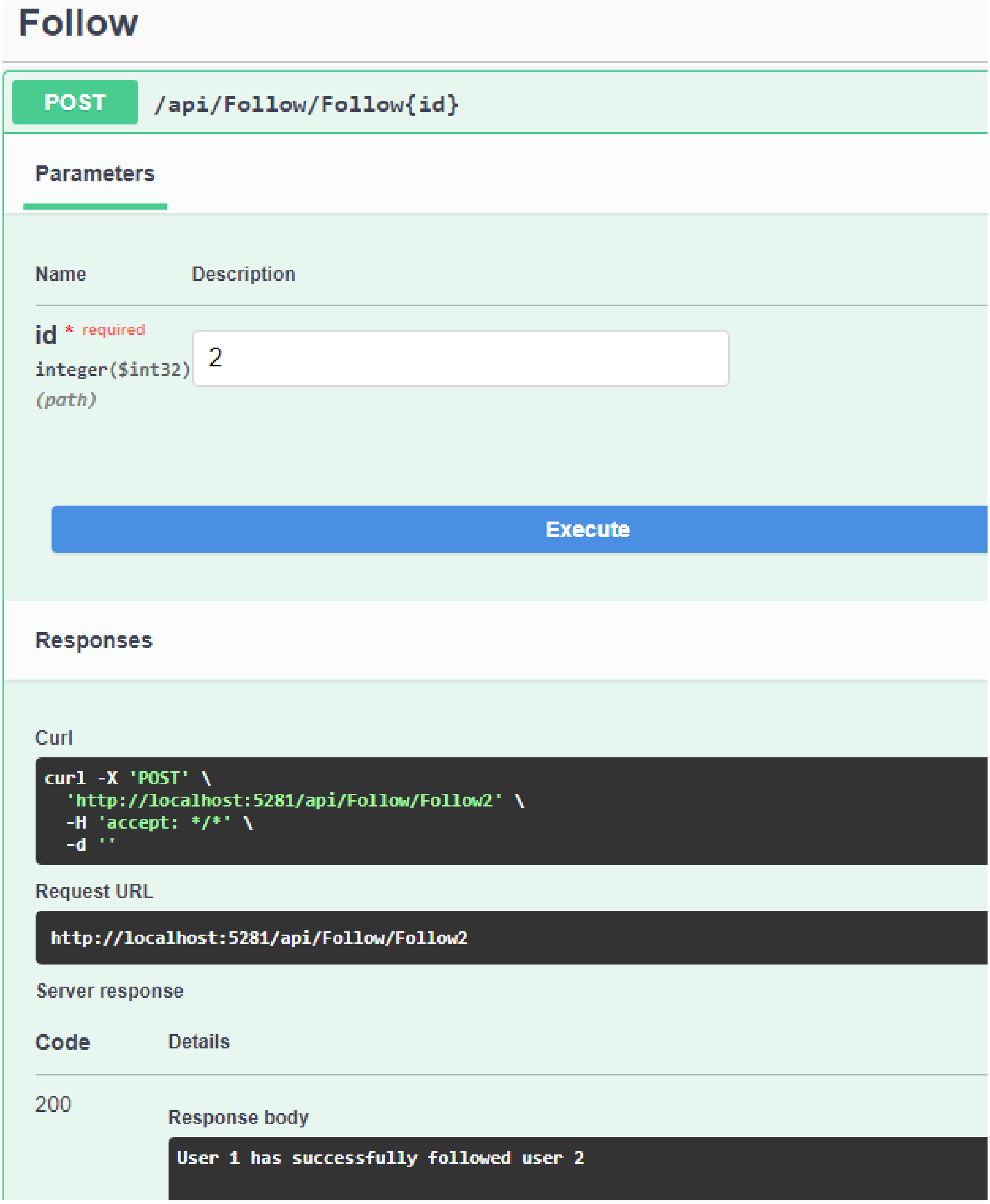


```
SELECT * FROM CW2.[Main_View]
```

	User ID	Username	Email	Account Type	Following	Followers	Favourite Activities
1	1	grace	grace@plymouth.ac.uk	admin	0	0	NULL
2	2	ed	ed@plymouth.ac.uk	user	0	0	NULL
3	3	owen	owen@plymouth.ac.uk	user	0	0	NULL

Before following a user with login of grace

D



```
curl -X 'POST' \
  'http://localhost:5281/api/Follow/Follow2' \
  -H 'accept: */*' \
  -d ''
```

Request URL

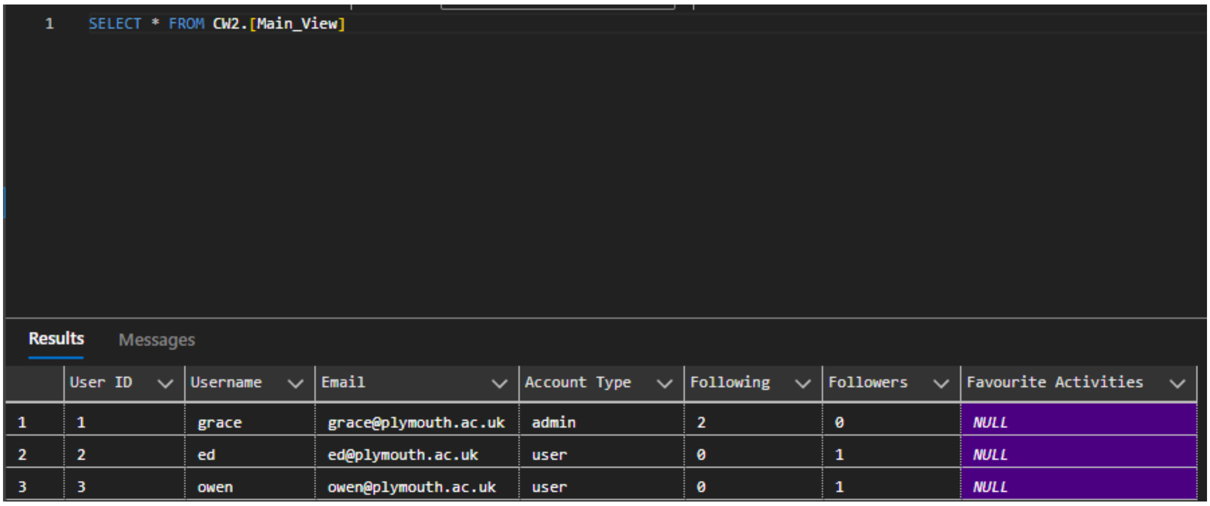
```
http://localhost:5281/api/Follow/Follow2
```

Server response

Code	Details
200	Response body

```
User 1 has successfully followed user 2
```

After executing with id 2 and 3:

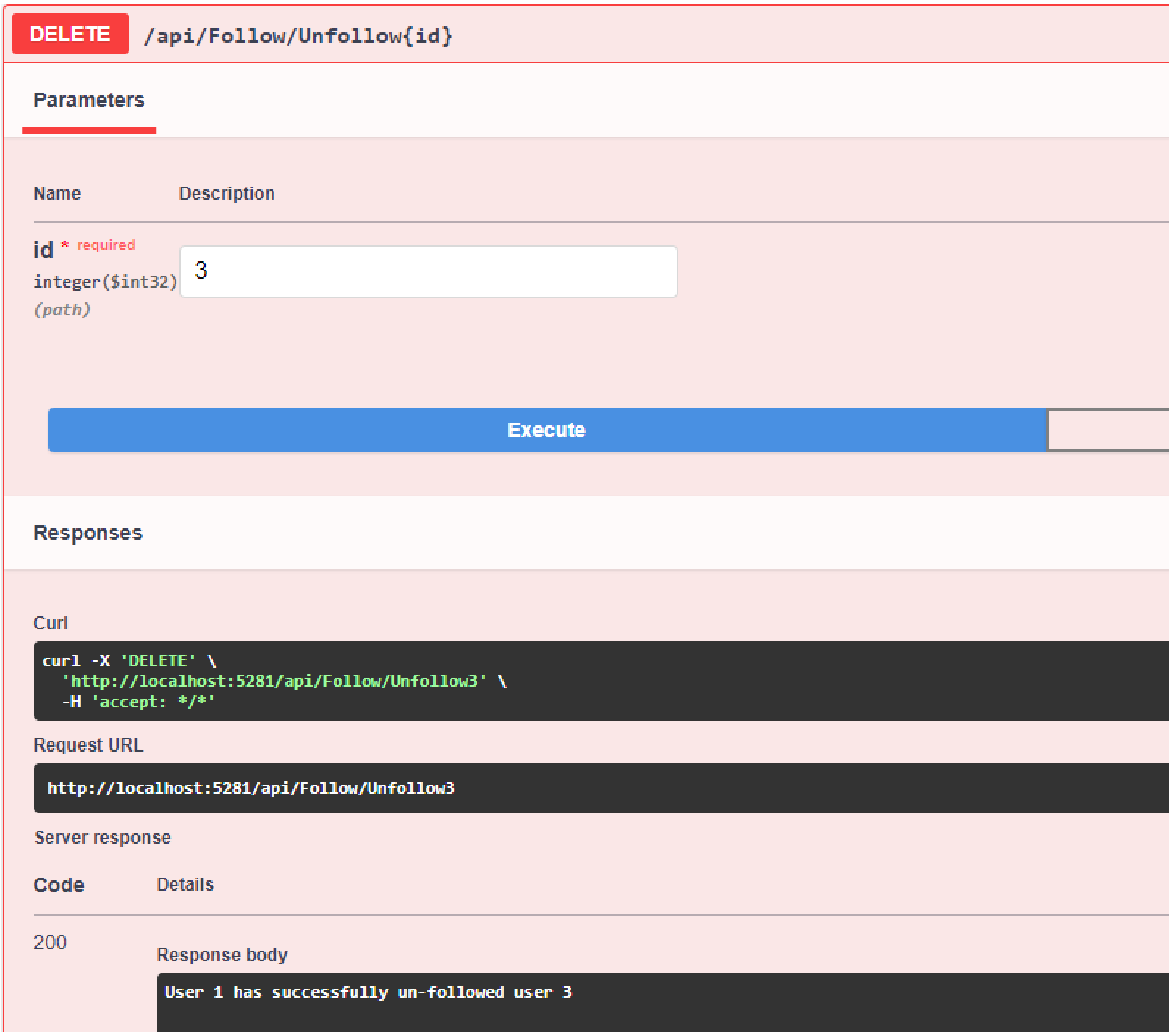


```
SELECT * FROM CW2.[Main_View]
```

Results Messages

	User ID	Username	Email	Account Type	Following	Followers	Favourite Activities
1	1	grace	grace@plymouth.ac.uk	admin	2	0	NULL
2	2	ed	ed@plymouth.ac.uk	user	0	1	NULL
3	3	owen	owen@plymouth.ac.uk	user	0	1	NULL

Unfollow



/api/Follow/Unfollow{id}

id * required
integer($int32)
(path)

3

Curl

```
curl -X 'DELETE' \
  'http://localhost:5281/api/Follow/Unfollow3' \
  -H 'accept: */*'
```

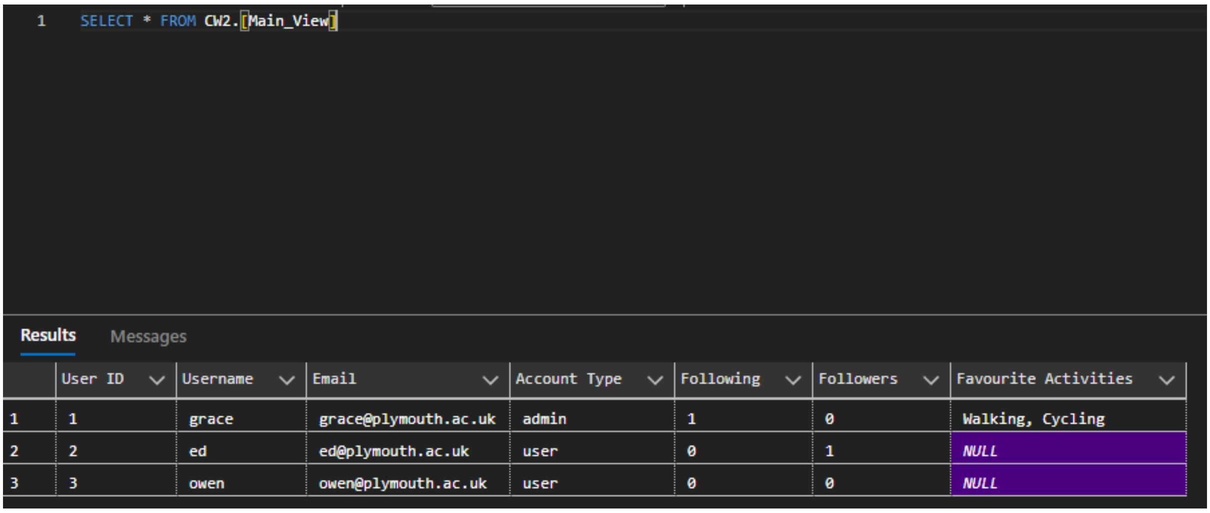
Request URL

```
http://localhost:5281/api/Follow/Unfollow3
```

Server response

Code	Details
200	Response body

```
User 1 has successfully un-followed user 3
```



```sql
SELECT * FROM CW2.[Main_View]
```

	User ID	Username	Email	Account Type	Following	Followers	Favourite Activities
1	1	grace	grace@plymouth.ac.uk	admin	1	0	Walking, Cycling
2	2	ed	ed@plymouth.ac.uk	user	0	1	*NULL*
3	3	owen	owen@plymouth.ac.uk	user	0	0	*NULL*

References

1.

Leo J. Guibas and Endre Szemeredi. 1976. The analysis of double hashing(Extended Abstract). In Proceedings of the eighth annual ACM symposium on Theory of computing (STOC '76). Association for Computing Machinery, New York, NY, USA, 187–191. https://doi.org/10.1145/800113.803647