

CW1

Problem:

A database storing information on trails is required for a web service.

Approach Formulated

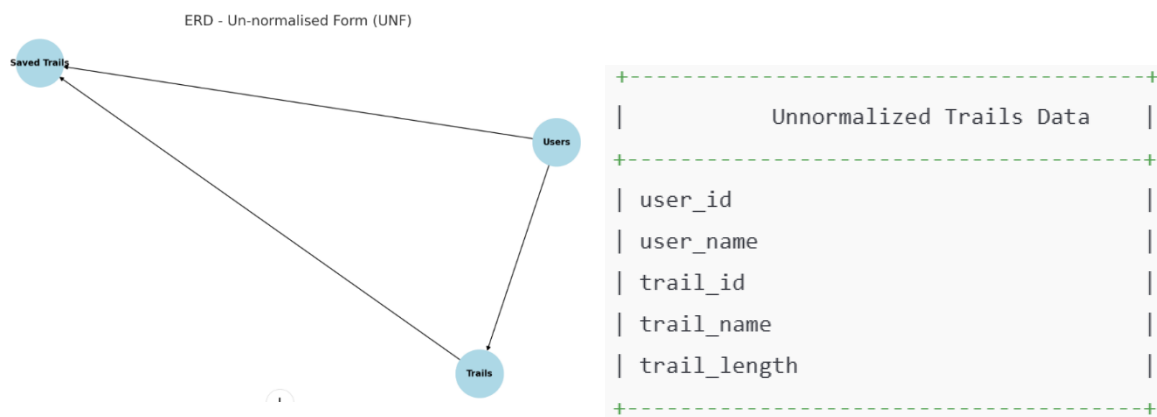
1. Identify Data Items

The first step was to identify the essential attributes required for the trail details micro-service. The primary focus was on:

- **Users:** user_id, user_name
- **Trails:** trail_id, trail_name, trail_length

2. Initial Structure (UNF)

UNF

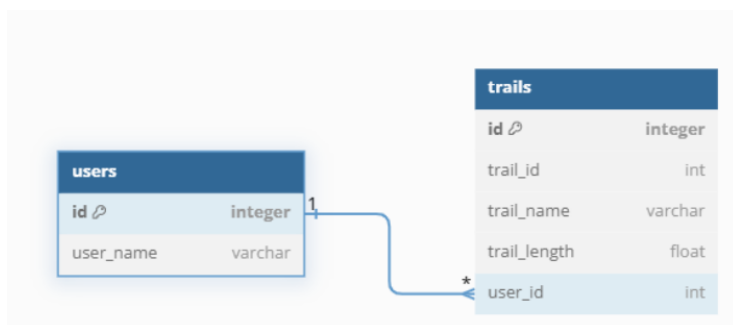
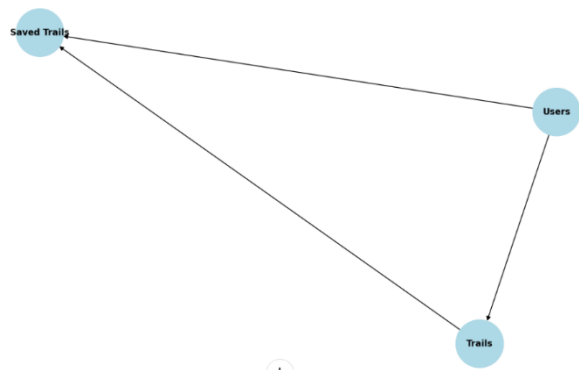


The original data structure was captured in the Un-normalised Form (UNF):

user_id	user_name	trail_id	trail_name	trail_length
1	Alice	1	Scenic Route	5.0
2	Bob	2	Mountain Trail	10.5
2	Bob	1	Scenic Route	5.0
3	Charlie	3	River Walk	3.0
4	David	3	River Walk	3.0
5	Eve	4	Coastal Path	7.2

In this structure, the attributes are not atomic, leading to redundancy and difficulties in maintaining data integrity.

1NF



3. First Normal Form (1NF)

Problem Addressed: The UNF structure had repeating groups and non-atomic values, leading to redundancy.

To achieve 1NF:

Process: Split data into separate tables by creating a users table for user-specific attributes and a trails table for trail-specific attributes.

Structure:

Users Table: Contains unique user_id as primary key, with user_name.

Trails Table: Contains a record_id as primary key, trail_id, trail_name, trail_length, and user_id as a foreign key.

Explanation:

Redundancy Reduction: Redundant user information is removed by splitting into two tables.

Table: Users

user_id (PK) user_name

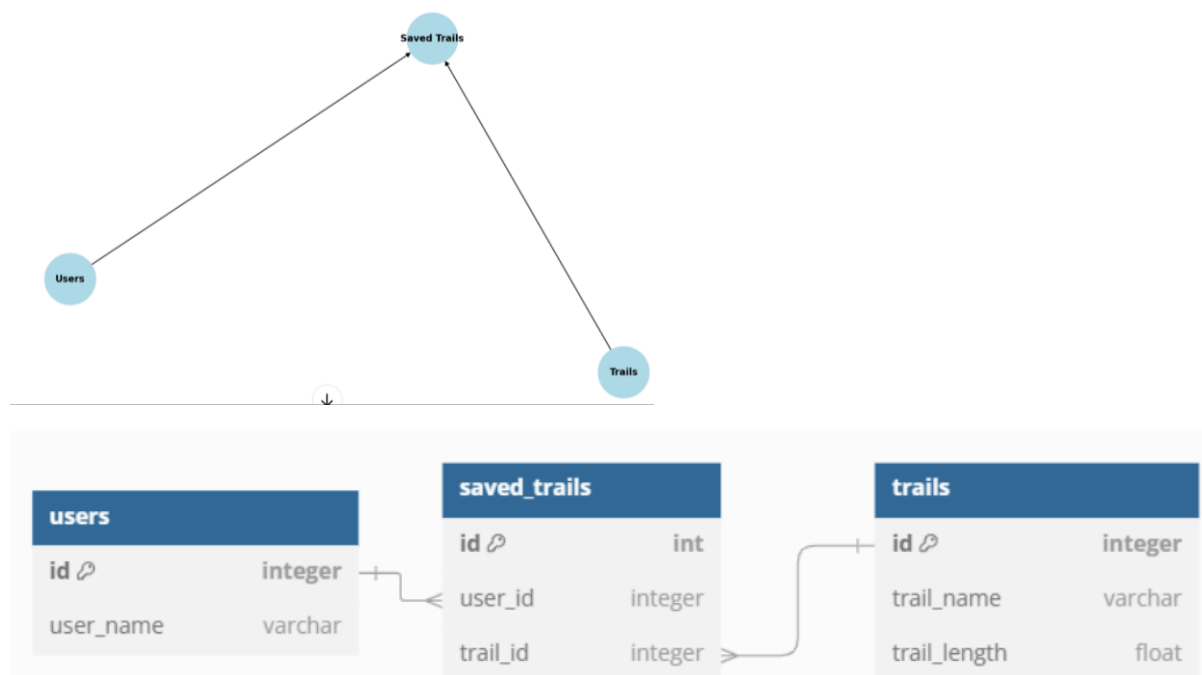
1	Alice
2	Bob
3	Charlie
4	David
5	Eve

Table: Trails

record_id (PK)	trail_id	User_saved	trail_name	trail_length
1	1	1	Scenic Route	5.0
2	2	NULL	Mountain Trail	10.5
3	3	2	River Walk	3.0
4	3	4	River Walk	3.0
5	4	4	Coastal Path	7.2
6	4	4	Coastal Path	7.2

Issues Remaining: Repeating data if multiple users save the same trail. Thus, further normalization is needed.

Second Normal Form (2NF)



Transition to 2NF:

Step: Created an additional linking table, saved_trails, to address redundancy further and introduce unique IDs for each entity.

Structure:

Users Table: Remains as in 1NF.

Trails Table: Now contains only trail-specific information.

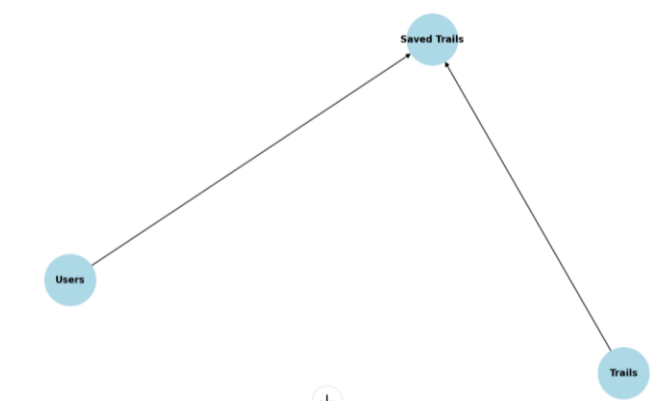
Saved_Trails Table: Introduces id as a unique primary key for saved trails.

Explanation:

Reduction of Redundancy: The saved_trails table eliminates the need to store user-specific information in the trails data.

Issues Remaining: Minor redundancy remains between trails and users, addressed in 3NF.

Third Normal Form 3NF



Transition to 3NF:

Step: Minor refinement, ensuring each field only depends on the primary key of its respective table.

Adjustment: In saved_trails, the use of a unique composite key for user-trail combinations fully encapsulates the relationship.

Final ERD (3NF):

Table: Users Table: Trails Table: Saved_Trails

user_id (PK) trail_id (PK) id (user_id, trail_id) (PK)

user_name trail_name user_id (FK to Users)

trail_length trail_id (FK to Trails)

Explanation:

Optimized Structure: Each table's attributes are fully dependent on its primary key.

Data Integrity: 3NF ensures minimal redundancy and maximum integrity across relationships.

Conclusion and Improvements

This transition from UNF to 3NF structured the data efficiently. Starting with all information in a single entity, we introduced structured relationships progressively through 1NF, 2NF, and finally achieved a fully normalized form in 3NF. This sequence reduces redundancy while providing room for future scalability by keeping data integrity at its core

Field Definition grid for Attributes of Entity ‘Users’

Entity name:

Description: This entity stores independent data of total amount of registered User records

Synonyms: Accounts, people, members

Relationship Link Phrases: Users can save trails

Attribute name	Description	Synonym(s)	Data type	Size (*=max)	Possible data values	Optional?	Validation rules	Key?
id	Unique identifier for each user	UserID	Integer	Default 32 bits	Positive integers	No	>0 & Not Null	Primary Key
user_name	Name of the user	Username	Varchar	Max	Alphanumeric	No	Not empty	-

Field Definition grid for Attributes of Entity ‘Trails’

Entity name: Trails

Description: This entity contains details about different trails, such as name and length, which users can explore and save.

Synonyms: Routes, paths, runs, explorations

Relationship Link Phrases: Trails can be saved by users

Attribute name	Description	Synonym(s)	Data type	Size (*= max)	Possible data values	Optional?	Validation rules	Key?
id	Unique identifier for each trail	PathID	Integer	Default 32 bits	Positive integers	No	>0 & Not Null	Primary Key
trail_name	Name of the trail	TrailName	Varchar	Max	Alphanumeric	No	Not Null	-
trail_length	Length of the trail in km	Length	Float	5, 3	Decimal, 0-50.000	No	>0 & < 50	-

Field Definition grid for Attributes of Entity ‘Saved trails’

Entity name: Saved Trails

Description: This linking entity connects users to the trails they have saved, capturing which trails each user is interested in.

Synonyms: Bookmarked Trails, Favorite Trails

Relationship Link Phrases: Users save trails, Trails are saved by users

Attribute name	Description	Synonym(s)	Data type	Size (*= max)	Possible data values	Optional?	Validation rules	Key?
user_id	Foreign key linking to users table and used as a part of a compound Key	UserID	Integer	Default 32 bits	Positive integers	No	Matches user_id	Primary Foreign Key, part of a compound key
trail_id	Foreign key linking to trails and used as a part of a compound Key	TrailID	Integer	Default 32 bits	Positive integers	No	Matches trail_id	Primary Foreign Key, part of a compound key

Database Development

1. Database, Schema, Tables & Relations

```
Run Cancel Disconnect Change Database: COMP2001_DCebotari
1 CREATE SCHEMA CW1;

Created SCHEMA CW1

Run Cancel Disconnect Change Database: COMP2001_DCebotari
1 -- Create Users table under CW1 schema
2 CREATE TABLE CW1.users (
3     id INT IDENTITY(1,1) PRIMARY KEY,
4     user_name VARCHAR(255) NOT NULL
5 );
6
7 -- Create Trails table under CW1 schema
8 CREATE TABLE CW1.trails (
9     id INT IDENTITY(1,1) PRIMARY KEY,
10    trail_name VARCHAR(255) NOT NULL,
11    trail_length FLOAT
12 );
13
14 -- Create Saved Trails table under CW1 schema
15 CREATE TABLE CW1.saved_trails (
16    trail_id INT NOT NULL,
17    user_id INT NOT NULL,
18    PRIMARY KEY (trail_id, user_id),
19    FOREIGN KEY (trail_id) REFERENCES CW1.trails(id)
20    FOREIGN KEY (user_id) REFERENCES CW1.users(id)
21 );
```

Created Tables Users, Trails and Saved Trails

Assigned relevant attributes to each table, assigned relevant data types and constraints to each attribute. Each ID is a primary key with a unique identifier and one of them is made of 2 foreign primary keys forming a compound key which removes the need for adding an extra attribute such as “saved_trail_id”.

Home > [dist-6-505.uopnet.plymouth.ac.uk](#) > COMP2001_DCebotari

«

New Query

New Notebook

Backup Database (Preview)

Restore Database (Preview)

Refresh

Learn More

Home

Recovery Model : Full

Last Log Backup : Never




Owner : UOPNET\mseymour

Last Database Backup : Never

Compatibility Level : 160

Search

Search by name of type (t, v, f, or sp)

Name	Schema	Type	Actions
 saved_trails	CW1	Table	...
 trails	CW1	Table	...
 users	CW1	Table	...

Created relationships to allow data entry connections and views

2. Testing the database performance

▶ Run ☐ Cancel Disconnect Change | Database: COMP2001_DCebotari ▼

Enable SQLCMD To Notebook

```
1  -- Insert sample data into the users table
2  INSERT INTO CW1.users (user_name) VALUES ('Alice');
3  INSERT INTO CW1.users (user_name) VALUES ('Bob');
4  INSERT INTO CW1.users (user_name) VALUES ('Charlie');
5
6  -- Insert sample data into the trails table
7  INSERT INTO CW1.trails (trail_name, trail_length) VALUES ('Mount
8  INSERT INTO CW1.trails (trail_name, trail_length) VALUES ('Fores
9  INSERT INTO CW1.trails (trail_name, trail_length) VALUES ('Beach
10
11 -- Insert sample data into the saved_trails table
12 -- These entries link users to trails, simulating saved trails
13 INSERT INTO CW1.saved_trails (trail_id, user_id) VALUES (1, 1);
14 INSERT INTO CW1.saved_trails (trail_id, user_id) VALUES (2, 1);
15 INSERT INTO CW1.saved_trails (trail_id, user_id) VALUES (3, 2);
16 INSERT INTO CW1.saved_trails (trail_id, user_id) VALUES (1, 3);
17
```

Messages

12:39:39 AM Started executing query at Line 1
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)

Used SQL Insert queries to populate my database with sample data.

▶ Run ☐ Cancel Disconnect Change | Database: COMP2001_DCebotari ▼

Enable SQLCMD To Notebook

```
1  -- View data in the users table
2  SELECT * FROM CW1.users;
3
4  -- View data in the trails table
5  SELECT * FROM CW1.trails;
6
7  -- View data in the saved_trails table
8  SELECT * FROM CW1.saved_trails;
```

To see if the database works as expected I ran some SELECT queries to display the data from each table.

Old data before commands:

▶ Run

❏ Cancel

🔌 Disconnect

🔄 Change

Database: COMP2001CW1_DCEB... ▾

1

SELECT * FROM CW1.users

2

SELECT * FROM CW1.trails

3

SELECT * FROM CW1.saved_trails

Results

Messages

id	trail_name	trail_length
----	------------	--------------

user_id	trail_id
---------	----------

New data after commands:

	id ▾	user_name ▾
1	1	Alice
2	2	Bob
3	3	Charlie

	id ▾	trail_name ▾	trail_length ▾
1	1	Mountain Trail	12.5
2	2	Forest Walk	8.3
3	3	Beach Path	5.7

	trail_id ▾	user_id ▾
1	1	1
2	1	3
3	2	1
4	3	2

Good Results, as expected

SQLQuery_2 - (158) ...botari) 9+

CW1.trails

Run

Cancel

Disconnect

Change

Database: COMP2001_DCebotari

Enable SQLCMD

To Notebook

```

1  CREATE VIEW CW1.UserSavedTrails AS
2  SELECT
3      u.id AS user_id,
4      u.user_name,
5      t.id AS trail_id,
6      t.trail_name,
7      t.trail_length
8  FROM
9      CW1.saved_trails st
10 JOIN
11     CW1.users u ON st.user_id = u.id
12 JOIN
13     CW1.trails t ON st.trail_id = t.id;
14

```

Wrote an SQL query to create a “VIEW” which is basically a program that visualise your data in a more detailed manner, showing you 1 single Table with all of the linked child/parent data rather than separate individual tables.

Run

Cancel

Disconnect

Change

Database: COMP2001_DCebotari

Estimated

Enable SQLCMD

To Notebook

```

1  SELECT * FROM CW1.UserSavedTrails;
2

```

Results

Messages

	user_id	user_name	trail_id	trail_name	trail_length
1	1	Alice	1	Mountain Trail	12.5
2	3	Charlie	1	Mountain Trail	12.5
3	1	Alice	2	Forest Walk	8.3
4	2	Bob	3	Beach Path	5.7

Have tested the VIEW by writing a select statement again and referencing the view. Worked as expected, all LINKED possible data is visualised. You can only now see which trails have been specially saved by which users.

CRUD

In order to make our microservice more usable I will use Create, Read, Update and Delete SQL queries.

CRUD will make the Creation, reading, updating and deletion of data/records easier.

Here is an example of Creating data/records on the Trails table.

```
▶ Run ☐ Cancel  Disconnect  Change | Database: COMP2001_DCebotari ▼
```

```
1 CREATE PROCEDURE CW1.InsertTrail
2   |   @trail_name VARCHAR(255),
3   |   @trail_length FLOAT
4 AS
5 BEGIN
6   |   INSERT INTO CW1.trails (trail_name, trail_length)
7   |   VALUES (@trail_name, @trail_length);
8 END;
```

This script simply creates a Procedure called “InsertTrail” that says “Insert into table ‘trails’ the following information: trail_name and trail_length upon being executed”.

Now, to run this procedure, we need to write the following SQL query and run:

```
▶ Run ☐ Cancel  Disconnect  Change | Database: COMP2001_DCebotari ▼ |  Estimated Plan  E
```

```
1 EXEC CW1.InsertTrail @trail_name = 'Beautiful Mountain Trail', @trail_length = 10.5;
2
```

“Beautiful Mountain Trail” is our sample trail_name data and “10.5” is our sample trail_length. These can be replaced with anything else and multiple EXEC lines can be spammed and different names and lengths can be entered in order to add multiple records.

It may seem as if we have already done something similar previously using Insert statements, which is true, however, this time we have simplified and made the insertion/creation slightly easier as you can see.

READ

We will now create a procedure that will call a READ program to be able to view certain database data. For this example, I will show all possible data stored in the Users Table.

SQLQuery_1 - (158) ...botari) 1 ● SQLQuery_2 - (72) d...botari) ●

Run Cancel Disconnect Change Database: COMP2001_DCebotari ▼

```
1 CREATE PROCEDURE CW1.GetAllUsers
2 AS
3 BEGIN
4     -- select all records from the users table
5     SELECT *
6     FROM CW1.users;
7 END;
```

SQLQuery_1 - (158) ...botari) 1 ● SQLQuery_2 - (72) d...botari) 1 ●

Run Cancel Disconnect Change Database: COMP2001_DCebotari ▼

```
1 EXEC CW1.GetAllUsers;
```

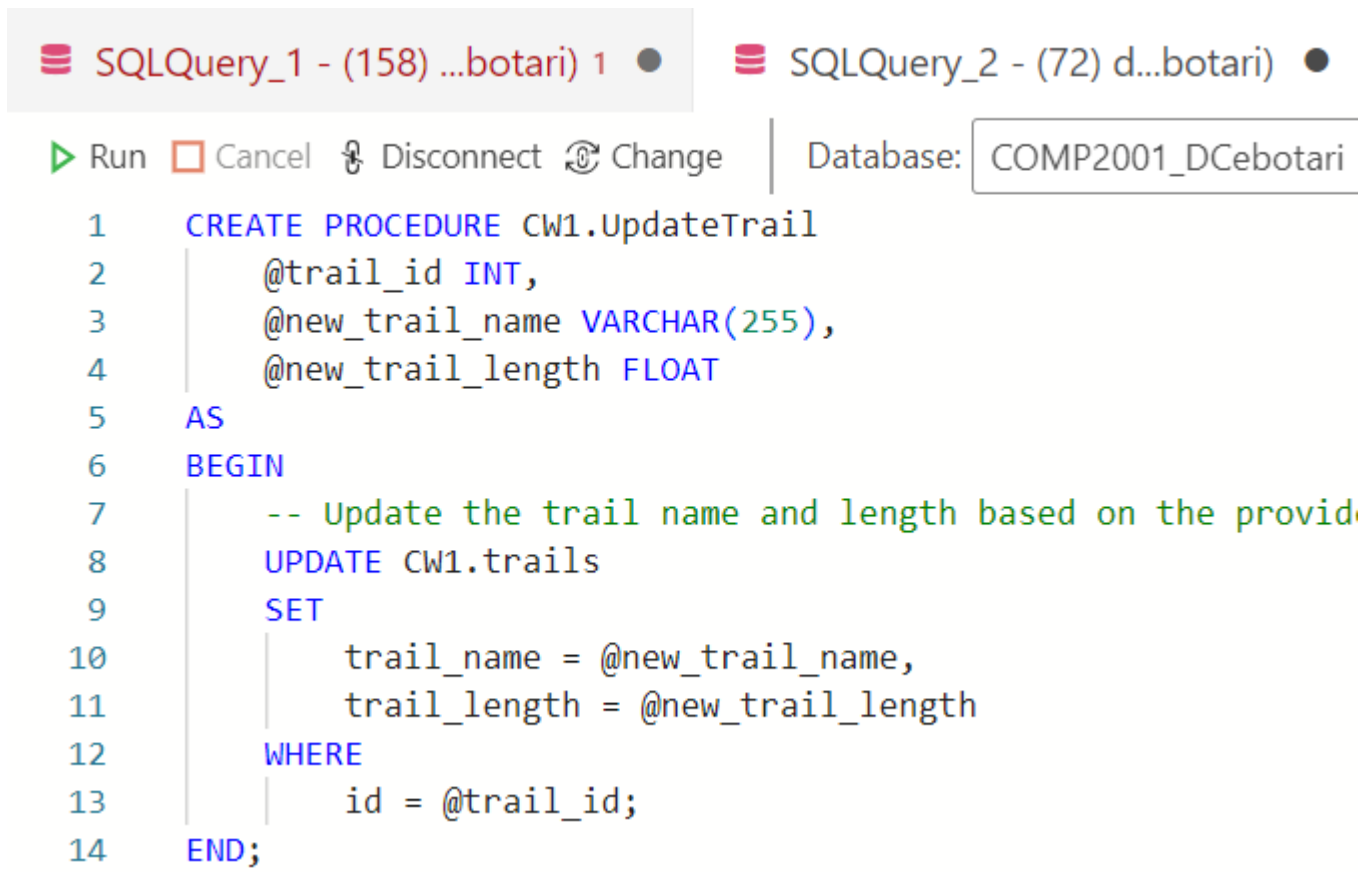
Results

Messages

	id ▼	user_name ▼
1	1	Alice
2	2	Bob
3	3	Charlie

UPDATE

Based on the Trails table, I will demonstrate an UPDATE procedure. I will choose a random Trail ID, and change its relative data such as a trail's name or length.

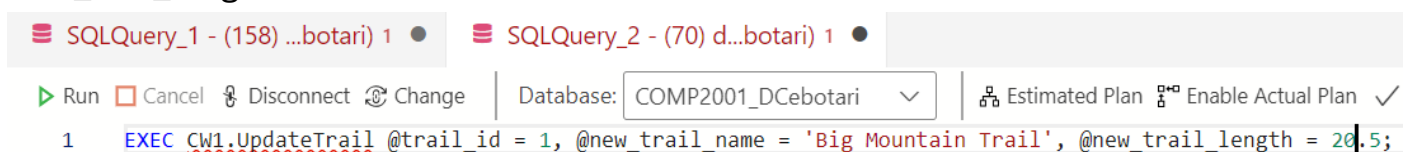


The screenshot shows the SQL Server Enterprise Manager interface. At the top, there are two tabs: 'SQLQuery_1 - (158) ...botari) 1' and 'SQLQuery_2 - (72) d...botari)'. Below the tabs is a toolbar with buttons for 'Run', 'Cancel', 'Disconnect', and 'Change'. To the right of the toolbar is a 'Database:' dropdown menu set to 'COMP2001_DCebotari'. The main area displays the following SQL code:

```
1 CREATE PROCEDURE CW1.UpdateTrail
2     @trail_id INT,
3     @new_trail_name VARCHAR(255),
4     @new_trail_length FLOAT
5 AS
6 BEGIN
7     -- Update the trail name and length based on the provided
8     UPDATE CW1.trails
9     SET
10         trail_name = @new_trail_name,
11         trail_length = @new_trail_length
12 WHERE
13     id = @trail_id;
14 END;
```

How it works, the “PROCEDURE” creates a virtual database table that stores temporary variables. These are, the trail ID we will be modifying, the new trail name we want to give it, and the new trail length we want to use.

Then, it reads the data from CW1.trails, looks up the trail id we are interested in, and if found, it “SETS” the trail_name being the “new_trail_name” and trail_length to new_trail_length.



The screenshot shows the SQL Server Enterprise Manager interface. At the top, there are two tabs: 'SQLQuery_1 - (158) ...botari) 1' and 'SQLQuery_2 - (70) d...botari) 1'. Below the tabs is a toolbar with buttons for 'Run', 'Cancel', 'Disconnect', and 'Change'. To the right of the toolbar is a 'Database:' dropdown menu set to 'COMP2001_DCebotari'. The main area displays the following SQL code:

```
1 EXEC CW1.UpdateTrail @trail_id = 1, @new_trail_name = 'Big Mountain Trail', @new_trail_length = 20.5;
```

Here's how it needs to be executed. I am attempting to modify the trail id number 1's name to Big Mountain Trail and length of 20.5. The old data used to be “Mountain Trail” and 12.5.

Success, Old data:

Run	Cancel	Disconnect	Change	Database: COMP2001_DCebotari	Estimated																														
Enable SQLCMD To Notebook																																			
<pre>1 SELECT * FROM CW1.UserSavedTrails;</pre>																																			
<div>Results Messages</div> <table> <tr> <th></th> <th>user_id</th> <th>user_name</th> <th>trail_id</th> <th>trail_name</th> <th>trail_length</th> </tr> <tr> <td>1</td> <td>1</td> <td>Alice</td> <td>1</td> <td>Mountain Trail</td> <td>12.5</td> </tr> <tr> <td>2</td> <td>3</td> <td>Charlie</td> <td>1</td> <td>Mountain Trail</td> <td>12.5</td> </tr> <tr> <td>3</td> <td>1</td> <td>Alice</td> <td>2</td> <td>Forest Walk</td> <td>8.3</td> </tr> <tr> <td>4</td> <td>2</td> <td>Bob</td> <td>3</td> <td>Beach Path</td> <td>5.7</td> </tr> </table>							user_id	user_name	trail_id	trail_name	trail_length	1	1	Alice	1	Mountain Trail	12.5	2	3	Charlie	1	Mountain Trail	12.5	3	1	Alice	2	Forest Walk	8.3	4	2	Bob	3	Beach Path	5.7
	user_id	user_name	trail_id	trail_name	trail_length																														
1	1	Alice	1	Mountain Trail	12.5																														
2	3	Charlie	1	Mountain Trail	12.5																														
3	1	Alice	2	Forest Walk	8.3																														
4	2	Bob	3	Beach Path	5.7																														

New data:

SQLQuery_1 - (158) ...botari) 1

SQLQuery_2 - (70) d...botari) 1

Run

Cancel

Disconnect

Change

Database: COMP2001_DCebotari

Estimated Plan

1 SELECT * FROM CW1.UserSavedTrails;

Results

Messages

	user_id	user_name	trail_id	trail_name	trail_length
1	1	Alice	1	Big Mountain Trail	20.5
2	3	Charlie	1	Big Mountain Trail	20.5

DELETE

And the final part of CRUD is DELETE.

I will demonstrate on saved_trails table how to delete a record.

We will need to specify at least one primary key out of a compound key formed from 2 primary keys from the registered records in order to delete at least one record.

However, specifying just 1 of the 2 keys is HIGHLY not recommended as that will most likely delete all the records that repeat the 1 primary key mentioned. Therefore, we will be specifying exactly 2 IDs used to identify 1 unique record and that way we will safely remove just one record.

▶ Run
❌ Cancel
🔌 Disconnect
🔄 Change

Database: COMP2001_DCebotari

📊 Estimated

```

1 CREATE PROCEDURE CW1.DeleteSavedTrail
2     @delete_user_id INT,
3     @delete_saved_trail_id INT
4 AS
5 BEGIN
6     -- Delete the record from the saved_trails table based on the provided id
7     DELETE FROM CW1.saved_trails
8     WHERE user_id = @delete_user_id AND trail_id = @delete_saved_trail_id;
9 END;

```

▶ Run
❌ Cancel
🔌 Disconnect
🔄 Change

Database: COMP2001_DCebotari

📊 Estimated Plan
📄 Enable Actual Plan
✓ Parse
🔧 Enable SQLCMD
📖 To Notebook

```

1 EXEC CW1.DeleteSavedTrail @delete_user_id = 1, @delete_saved_trail_id = 2; -- Replace 1 and 2 with the actual ID you want to delete

```

Proof of success:

Old data had 4 records:

▶ Run
❌ Cancel
🔌 Disconnect
🔄 Change

Database: COMP2001_DCebotari

📊 Estimated

🔧 Enable SQLCMD
📖 To Notebook

```

1 SELECT * FROM CW1.UserSavedTrails;
2

```

ResultsMessages

	user_id	user_name	trail_id	trail_name	trail_length
1	1	Alice	1	Mountain Trail	12.5
2	3	Charlie	1	Mountain Trail	12.5
3	1	Alice	2	Forest Walk	8.3
4	2	Bob	3	Beach Path	5.7

New data has only 3 records, you can clearly see only one of “Alice”’s records have been removed.

▶ Run
❌ Cancel
🔌 Disconnect
🔄 Change

Database: COMP2001_DCebotari

📊 Estimated P

```

1 SELECT * FROM CW1.UserSavedtrails;
2

```

ResultsMessages

	user_id	user_name	trail_id	trail_name	trail_length
1	1	Alice	1	Big Mountain Trail	20.5
2	3	Charlie	1	Big Mountain Trail	20.5
3	2	Bob	3	Beach Path	5.7

TRIGGER

I will finalise my report by creating a current final functionality for my microservice which is the addition of a Trigger.

A trigger is like a sensor that detects changes. In our case, I want there to be a trigger present that will record when a user adds a new trail.

Therefore, a trigger will be monitoring changes to the trails table. When a new trail is added, the trigger will run an operation to record that trail to a new table which will be called trails_log.

That table will have information such as what trail was created, by whom, and when(date and time).

The ideal delivery: A program that only registers a new trail if a user has logged their details and entered every required data into the database. Only then will there be new data registered in the trails table and trails_log table.

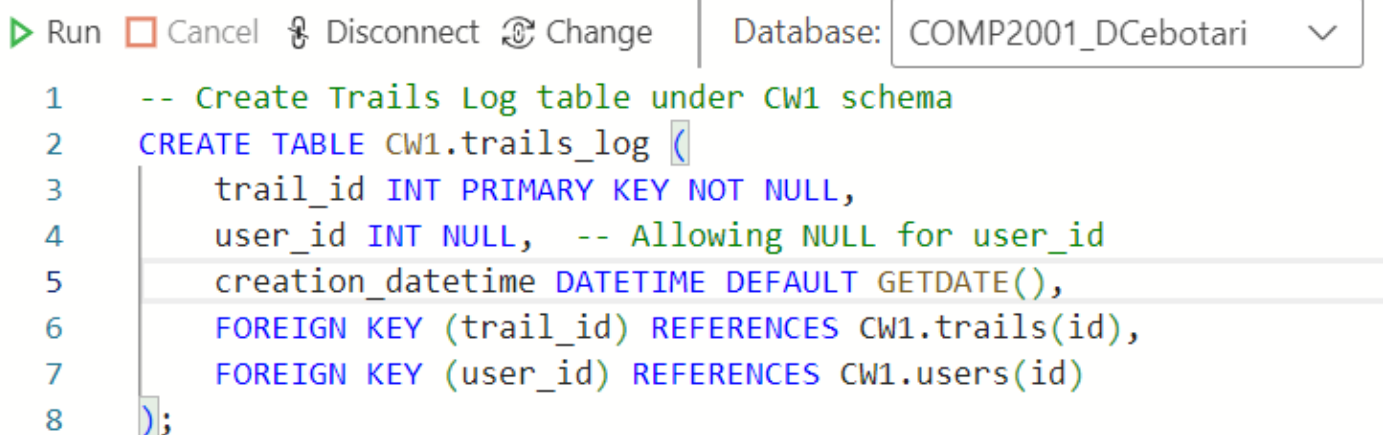
Current possible solution proposed:

Due to SQL Database complications and limitations and earlier proposed ERD and scripts, it is quite difficult to formulate an ideal product as mentioned above.

The possible solution however, is one that allows us to register a trail, it logs a new record inside the trails_log table but may leave some datafields empty for later change.

Specifically, the new trail record, its id will be logged in the trails_log table, but a user doesn't have to be logged yet, this means that trail_id and creation_datetime attributes will be automatically populated/generated by the script but the user_id has to be manually done, and it cannot be done at the same time. The user cannot intervene at this moment. Once a trail_log is created, then the user at any given moment, is allowed to manually add the missing data/ user id in the "user_created" field.

First we create the trails_log table and add its relative attributes,

A screenshot of a SQL IDE interface. At the top, there is a toolbar with buttons: 'Run' (green play icon), 'Cancel' (red square icon), 'Disconnect' (grey plug icon), and 'Change' (circular arrow icon). To the right of the toolbar is a 'Database:' dropdown menu showing 'COMP2001_DCebotari'. Below the toolbar, a SQL script is displayed in a text area with line numbers 1 through 8 on the left. The script is as follows:

```
1  -- Create Trails Log table under CW1 schema
2  CREATE TABLE CW1.trails_log (
3      trail_id INT PRIMARY KEY NOT NULL,
4      user_id INT NULL, -- Allowing NULL for user_id
5      creation_datetime DATETIME DEFAULT GETDATE(),
6      FOREIGN KEY (trail_id) REFERENCES CW1.trails(id),
7      FOREIGN KEY (user_id) REFERENCES CW1.users(id)
8  );
```

As you can see, trail_id is a primary key, but it's also a foreign key, that is because in the trails_log logically should not be repeated foreign trail_ids, therefore, I thought of avoiding the necessity of an extra excessive attribute and just used an existing one and make it a primary key of that table.

As you can see, user_id has a "NULL" meaning it will allow empty data to ensure our Trigger functions and the trail log records most necessary data automatically removing potential of human error.

Run Cancel Disconnect Change Database: COMP2001_DCebotari Estimated Plan Enable Ac

```

1 CREATE TRIGGER CW1.trail_insert_trigger
2 ON CW1.trails
3 AFTER INSERT
4 AS
5 BEGIN
6     -- Insert a new record into the trails_log table
7     INSERT INTO CW1.trails_log (trail_id, creation_datetime)
8     SELECT
9         id, -- Assuming the 'id' column is the primary key for the trails table
10        GETDATE() -- Current date and time
11    FROM inserted; -- 'inserted' is a special table that contains the newly inserted rows
12 END;
13

```

SQL Script Creation of the trigger

Testing the trigger:

I will insert new data in the trails table. The trigger should react and log new data inside the trails_log table.

Run Cancel Disconnect Change Database: COMP2001_DCebotari

```

1 INSERT INTO CW1.trails (trail_name, trail_length)
2 VALUES ('New Beautiful Trail', 4.5);
3

```

As you can see, a new trail was added, let's check the trails log.

Run Cancel Disconnect Change Database: COMP2001_DCebotari

```

1 -- Select rows from a Table or View '[TableOrViewName]' in schema
2 SELECT * FROM [CW1].[trails]
3

```

Results

Messages

	id	trail_name	trail_length
1	1	Big Mountain Trail	20.5
2	2	Forest Walk	8.3
3	3	Beach Path	5.7
4	4	Beautiful Mountain Trail	10.5
5	5	New Beautiful Trail	4.5

As you can see, the trails_log has also been updated, there is an empty user_id which can be modified manually.

Run Cancel Disconnect Change Database: COMP2001_DCebotari

1 -- Select rows from a Table or View '[TableOrViewName]' in schema
2 SELECT * FROM [CW1].[trails_log]
3

Results Messages

	trail_id	user_id	creation_datetime
1	5	NULL	2024-11-04 03:39:23.200

Let's now change the user_id using PROCEDURES and EXEC

```
CREATE PROCEDURE [CW1].[UpdateTrailsLog]
    @trail_id INT,
    @user_id INT
AS
BEGIN
    -- Update the trail name and length based on the provided trail_id
    UPDATE CW1.trails_log
    SET
        user_id = @user_id
    WHERE
        trail_id = @trail_id;
END;
```

Let's now EXECUTE that procedure

Run Cancel Disconnect Change Database: COMP2001_DCebotari

✓ Parse Enable SQLCMD To Notebook

1 EXEC CW1.UpdateTrailsLog @trail_id = 5, @user_id = 2;

Let's now view the changes, as you can see user_id is now 2 instead of NULL.

Run Cancel Disconnect Change Database: COMP2001_DCebotari

✓ Parse Enable SQLCMD To Notebook

1 SELECT * FROM CW1.trails_log

Results Messages

	trail_id	user_id	creation_datetime
1	5	2	2024-11-04 03:39:23.200

This is my final product, there is potential for further improvement such as making the insertion of the userID + trail id in one go which would make this an ideal solution.