Task 1

Associative Tables
I noticed that there were going to be multiple one-to-many relationships. For example, one topic could have many posts and one forum could have many topics.

To normalise these tables, I created the following associative tables (Topics_In_Forum, Posts_In_Topic) containing only the primary keys of their respective tables.

This removes a lot of the redundant data, so less memory is required for storage and it is also faster to search through. There is also less duplication thereby reducing the risk of mistakes.

API
In order to aid the API, I ensured both the Topic and Post tables contained a ‘postedAt’ field containing the exact time to the second that the topic was posted. The time and username combination is then used to get the unique identifier of the topic after the entry has been created in the table.

If somehow, multiple posts are created by the same user during the same second a database error will be returned and the data will not be inserted into the table.

Other Changes
The table structures for Topic, Forum and Post more-or-less followed their respective Java class structure but I made some changes to improve normalisation.

I ensured each table had a unique primary key and that there were no functional dependencies within each table to achieve third normative form.

I included data validation to prevent users entering data that would be too long for the table to hold or an invalid I hardcoded the size limits (in regards to the length of certain fields) to prevent magic numbers and make the code easier to understand.

Task 3: Adding a ‘Like’ Functionality

Part A:
The proposed functionality enables a user to like either a post or a topic. This functionality can essentially be thought of as a one to many relationship between a user and a post or topic.

I created two tables **(User_Likes_Posts** and **User_Likes_Topics)** containing the user id and the respective topic or post id of the liked item.

I ensured the primary key was composite to prevent data duplication. For example, a user liking the same topic or post more than one time should not be possible. Each user-post (or user-topic) combination within each table should be unique.

Part B:

Recording a ‘like’ for a post:
Below query demonstrating a user with id of 1 liking a post with id of 3.

```
INSERT INTO User_Likes_Posts (postid, userid)
```

```
VALUES (3, 1);
```

Getting the total number of topics and posts liked by a specific user.

Below query finds all total number of topics and posts for user with id of 1.

```
WITH LikedPosts AS (
        SELECT COUNT(User_Likes_Posts.postid) AS TotalPosts FROM Person
        JOIN  User_Likes_Posts ON User_Likes_Posts.userid =  Person.id
        WHERE Person.id = 1
        ),

LikedTopics AS (
        SELECT COUNT(User_Likes_Topics.topicid) AS TotalTopics FROM Person
        JOIN  User_Likes_Topics ON User_Likes_Topics.userid =  Person.id
        WHERE Person.id = 1
        ),

Totals AS (
SELECT TotalPosts FROM LikedPosts
UNION ALL
SELECT TotalTopics FROM LikedTopics
)

SELECT SUM(TotalPosts) AS TotalLikes FROM Totals;
```

Getting the names of all people who have liked a topic, ordered alphabetically.

```
SELECT Person.name FROM Person
JOIN  User_Likes_Topics ON User_Likes_Topics.userid =  Person.id
ORDER BY Person.name;
```

Part C:

Deploying new schema functionality to an existing application can be dangerous as changes to tables might cause the requests to break or behave differently. For example, removing or adding fields or changing their criteria may cause existing queries to be invalid by referencing fields that are no longer there or inserting values which can no longer meet the field criteria.

The new schema did not modify at all the previously existing tables and therefore should not impact the functioning of the current application.