Programming Assignment 1: Model Hub Platform

CENG351 Data Management and File Structures, Fall 2025-2026

Due date: 17 December 2025, Wednesday, 23:59

Contents

1 Introduction

In this assignment, you will complete a SQL case study based on a real-world scenario. Imagine that you applied for a lead database administrator position at a research-driven AI company that manages an online Model Hub platform similar to Hugging Face. After passing the initial interview stages, the company has given you a final technical task. To move forward in the hiring process, you must design and query the database that powers the platform.

You are asked to implement the Model Hub platform similar to Hugging Face Platform database using Java and SQL.Your goal is to create the tables, insert data, and answer various SQL queries through specific Java methods. A project structure and several helper classes are provided for you, and your job is to complete the required methods.

This assignment will help you practice SQL queries, database interactions in Java, Maven project management, and working with an embedded H2 database. Even if you are not fully familiar with Java development, detailed explanations and instructions are included to guide you through the tasks.

2 Background

The platform is designed as a Model Hub similar to Hugging Face, where machine learning models are designed for tasks, organizations release these models, models have model versions, users have profiles, users follow each other, users upload datasets, users run specific model versions on datasets, runs may create results, and results may be included in publications. The system should track **users, profiles, organizations, models, model versions, tasks, datasets, results, and publications.**

Every model is designed for one or more tasks, and each task is supported by one or many models (for example, both CodeBERT and GraphCodeBERT models are designed for the code understanding task; moreover, CodeBERT is also designed for other tasks such as documentation generation and code search). Since a model may be designed for multiple tasks, "is_primary" information should be kept indicating whether a task is the model's primary task.

An organization may release zero or more machine learning models, and each model must be released by exactly one organization.

A model has one or more model versions. Each model version is identified by a "version_no" that is unique within its model (the same "version_no" may appear in different models) and cannot exist without its model.

Each profile must belong to exactly one user, and each user may have at most one profile, since some user may choose not to create a profile page on the platform.

Users may follow other users on the platform. Organizations cannot follow anyone. A user can follow zero or more other users and can be followed by zero or more users. Note that a user cannot follow itself.

A user may upload zero or more datasets, and each dataset must be uploaded by at least one user. Multiple users can upload the same dataset. For each upload of a dataset by a user, the user's "role" for this upload (e.g., owner, contributor, validator, annotator) is stored in the system.

A user runs a specific model version on a dataset. In each run, one model version is executed on one dataset by a single user. Any user, model version, or dataset may participate in zero or more runs.

Each **run** may create zero or more results because it can evaluate multiple "hyperparameter_config" values; different hyperparameter configurations (such as learning_rate, batch_size, optimizer) create different results. Each result contains metrics (such as accuracy and F1 score) and the hyperparameter configuration used. Some runs may create no results (e.g., aborted or failed). Every result must be created by exactly one successful run.

Each publication recorded on the platform must include one or more results. A result may appear in zero or more publications, as some may remain unpublished. In published articles, results from runs executed by users using different model versions on datasets under varying hyperparameter configurations may be included and compared using metrics like accuracy and f1_score.

3 Database Schema

The Model Hub Platform uses the following relational database schema:

- **Users** (PIN, user_name, reputation_score)
- **Organizations** (OrgID, org_name, rating)
- **Profiles** (ProfileID, bio, avatar_url, PIN) *PIN REFERENCES Users(PIN)*
  Note: PIN NOT NULL and UNIQUE
- **follows** (followerPIN, followeePIN, following_date) *followerPIN REFERENCES Users(PIN), followeePIN REFERENCES Users(PIN)*
- **Models** (ModelID, model_name, license, size, OrgID) *OrgID REFERENCES Organizations(OrgID)*
  Note: OrgID NOT NULL
- **ModelVersions** (ModelID, version_no, version_date) *ModelID REFERENCES Models(ModelID)*
- **Tasks** (TaskID, task_name)
- **designed_for** (ModelID, TaskID, is_primary) *ModelID REFERENCES Models(ModelID), TaskID REFERENCES Tasks(TaskID)*
- **Datasets** (DatasetID, dataset_name, modality, number_of_rows)
- **uploads** (PIN, DatasetID, role) *PIN REFERENCES Users(PIN), DatasetID REFERENCES Datasets(DatasetID)*
- **runs** (PIN, ModelID, version_no, DatasetID, run_type) *PIN REFERENCES Users(PIN), (ModelID, version_no) REFERENCES ModelVersions(ModelID, version_no), DatasetID REFERENCES Datasets(DatasetID)*
- **Results** (ResultID, accuracy, f1_score, hyperparameter_config, PIN, ModelID, version_no, DatasetID) *(PIN, ModelID, version_no, DatasetID) REFERENCES runs(PIN, ModelID, version_no, DatasetID)*
  Note: (PIN, ModelID, version_no, DatasetID) NOT NULL
- **Publications** (PubID, title, venue)
- **includes** (PubID, ResultID, placement_type, placement_section) *PubID REFERENCES Publications(PubID), ResultID REFERENCES Results(ResultID)*

Each table represents an entity in the system, with primary keys underlined and foreign key relationships specified.

4 Project Structure

The project is organized as follows:

```
ModelHubPlatform/
  pom.xml
  src/
    main/
      java/
        ceng/
          ceng351/
            ModelHubPlatform/
              User.java
              Organization.java
              Profile.java
              follow.java
              Model.java
              ModelVersion.java
              Task.java
              designed_for.java
              Dataset.java
              upload.java
              run.java
              Result.java
              Publication.java
              include.java
              ModelHubPlatform.java
              IModelHubPlatform.java
              Evaluation.java
              FileOperations.java
              QueryResult.java
      resources/
        data/
          Users.txt
          Organizations.txt
          Profiles.txt
          follows.txt
          Models.txt
          ModelVersions.txt
          Tasks.txt
          designed_for.txt
          Datasets.txt
          uploads.txt
          runs.txt
          Results.txt
          Publications.txt
          includes.txt
```

Each database table is represented by a corresponding Java class. These classes are already implemented for your convenience. Further, the classes `Evaluation`, `FileOperations`, and `QueryResult` are provided for testing purposes. You need to complete the `ModelHubPlatform.java` class. Then, running `Evaluation.java` will create and write the return values to the file `Output.txt` under the `output` directory.

5 Maven and Embedded H2 Database

The project uses Maven to manage dependencies and build configurations, simplifying development and execution. Maven handles all required libraries automatically, and the project is configured to run with **Java 14**. Maven is a build automation tool used primarily for Java projects. The `pom.xml` file contains all the necessary information to build the project.

We utilize the embedded H2 in-memory database to manage all database operations without requiring any external installation. The H2 instance runs entirely in RAM, meaning that the database exists only while the JVM and JDBC connection remain active; once the program terminates, all data is automatically removed. For debugging and inspection, the H2 Web Console can also be accessed through the browser at `http://localhost:8082` using the JDBC URL `jdbc:h2:tcp://localhost/mem:ModelHubPlatformdb`, with the default username `sa` and an empty password. Through this interface, it is possible to browse tables, inspect data, and execute SQL queries interactively, providing a convenient environment for testing and verification during development.

The H2 database is included as a dependency in the `pom.xml` file. This ensures that the database is ready to use without any extra setup. We strongly recommend using an advanced IDE like IntelliJ IDEA, which is installed on the department computers (*ineks*). Opening the `pom.xml` file in IntelliJ IDEA will automatically set up the project for you.

6 Tasks and Implementation Details

You are required to implement the methods defined in the `IModelHubPlatform` interface within the `ModelHubPlatform.java` class. Below are the tasks to be implemented:

6.1 Task 1: Create Database Tables (5 pts)

Method: `createTables()`
You will create all the database tables according to the schema described in Section 3. Ensure that all **primary keys and foreign keys are properly defined.** To avoid integrity-constraint violations during initialization, tables must be created in a strict dependency order: first independent tables (`Users`, `Organizations`, `Tasks`, `Datasets`, `Publications`), then tables referencing them (`Profiles`, `follows`, `Models`, `uploads`), followed by model-dependent tables (`ModelVersions`, `designed_for`), then run-dependent tables (`runs`), and finally tables referencing results (`Results`, `includes`).

Notes:

- A user cannot follow itself.

- Profiles table: PIN must be NOT NULL and UNIQUE.
- Models table: OrgID must be NOT NULL.
- Results table: (PIN, ModelID, version_no, DatasetID) must be NOT NULL.

Output: The number of tables created successfully.

6.2 Task 2: Insert Data into Tables (5 pts)

Implement the following methods to insert rows (data) from the provided `.txt` files into the corresponding tables. All input data is located under `src/main/resources/data`, and a total of 14 `.txt` files are provided.

Methods:

- `insertUsers(User[] users)`
- `insertOrganizations(Organization[] organizations)`
- `insertTasks(Task[] tasks)`
- `insertDatasets(Dataset[] datasets)`
- `insertPublications(Publication[] publications)`
- `insertProfiles(Profile[] profiles)`
- `insertfollows(follow[] follows)`
- `insertModels(Model[] models)`
- `insertuploads(upload[] uploads)`
- `insertModelVersions(ModelVersion[] modelVersions)`
- `insertdesigned_fors(designed_for[] designedFors)`
- `insertruns(run[] runs)`
- `insertResults(Result[] results)`
- `insertincludes(include[] includes)`

Important Note: All insert methods are executed before the evaluation of any other task. If any rows fail to be inserted due to SQL errors or foreign-key violations, the remaining tasks will operate on incomplete data and produce incorrect results. Therefore, correct and complete data insertion is essential for passing the evaluation tests.

Note: You are not required to implement any file-reading logic. All input files are read by the methods in `FileOperations.java`. The `Evaluation` class automatically calls these methods and passes the resulting objects to your insertion functions. Your responsibility in this task is only to insert the provided objects into the corresponding tables.

Output: For each method, return the number of rows inserted successfully.

6.3 Task 3: Find Users Without Profiles (5 pts)

Method: `getUsersWithoutProfiles()`
Retrieve all users who do not have a corresponding profile entry in the `Profiles` table.
Note: Sort the result by `reputation_score` in **descending** order, and for users with the same reputation score, sort by `PIN` in **ascending** order.
Output: An array of `User` objects containing `PIN`, `user_name`, and `reputation_score`.

6.4 Task 4: Decrease Reputation for Users Without Profiles (5 pts)

Method: `decreaseReputationForMissingProfiles()`
Reduce the `reputation_score` of all users who do not have an associated profile. A deduction of **10 points** is applied only if the user's current reputation_score is ≥ 10.
Output: An integer value representing the number of updated rows in the `Users` table.

6.5 Task 5: Find Users With Specific Bio Keywords (5 pts)

Method: `getUsersByBioKeywords()`
Retrieve all users who have a profile and whose `bio` contains at least one of the following words (case-insensitive): *"engineer"*, *"scientist"*, or *"student"*.
Note: You may find it useful to convert the `bio` field to lowercase (e.g., using `LOWER(P.bio)`) to perform case-insensitive keyword matching.
Note: Sort the results by `PIN` in **ascending** order.
Note: Keyword matching is performed using substring search. For example, a bio containing *"engineering"* will still match *"engineer"* because the expression `LIKE '%engineer%'` returns true.
Output: An array of `QueryResult.UserPINNameReputationBio` objects, where each object includes `PIN`, `user_name`, `reputation_score`, and the associated profile `bio`.

6.6 Task 6: Find Organizations With No Released Models and Low Rating (5 pts)

Method: `getOrganizationsWithNoReleasedModelsAndLowRating()`
Find all organizations that have *not released any models* and whose rating is strictly smaller than 2.5. Organizations with a rating of exactly 2.5 should **not** be included.
Note: Sort the results by `OrgID` in **descending** order.
Output: An array of `Organization` objects containing `OrgID`, `org_name`, and `rating`.

6.7 Task 7: Delete Organizations With No Released Models and Low Rating (5 pts)

Method: `deleteOrganizationsWithNoReleasedModelsAndLowRating()`
Delete all organizations that have *not released any models* and whose rating is strictly smaller than 2.5. Organizations with a rating of exactly 2.5 should **not** be deleted.
Note: No other tables should be modified.
Output: The number of deleted rows.

6.8 Task 8: Retrieve Models and Their Primary Task Information (5 pts)

Method: `getModelPrimaryTaskInfo()`
In the platform, each model must have exactly one primary task, meaning that in the `designed_for` table there should be only one entry per model where `is_primary = true`. For each model, the primary task count should be calculated to verify this constraint. This task requires retrieving the `ModelID`, the `model_name`, the model's `primary_task_name`,

and the `primary_task_count`, which indicates how many tasks the model has with `is_primary = true`.
Note: Since every model must have exactly one primary task, the value of `primary_task_count` is expected to be **1 for all models**.
Note: Sort the results by `ModelID` in **ascending** order.
Output: An array of `QueryResult.ModelPrimaryTaskInfo` objects.

6.9 Task 9: Compute User Popularity Score (5 pts)

Method: `getUserPopularityScore()`
In this task, you will compute a *popularity score* for each user in the platform. The popularity score is defined as:

$$popularity\_score = follower\_count - followee\_count$$

The `follower_count` is the number of users who follow this user (i.e., number of entries where `followeePIN = PIN`), and the `followee_count` is the number of users the user follows (i.e., number of entries where `followerPIN = PIN`). If a user follows no one, the followee count should be treated as 0. If a user is followed by no one, the follower count should also be treated as 0.
Note: The result must include the `PIN`, `user_name`, and the computed `popularity_score`.
Note: Sort the results by `popularity_score` in **descending** order.
Note: If two users have the same score, sort them by `PIN` in **ascending** order.
Note: Return only the top 20 popularity score users.
Output: An array of `QueryResult.UserPopularityInfo` objects.

6.10 Task 10: Comprehensive Model Information (5 pts)

Method: `getComprehensiveModelInfo()`
Write an SQL query that retrieves detailed information for every model in the platform. For each model, your query must return:

- `ModelID`
- `model_name`
- the organization's `org_name` that released the model
- `license`
- `size`
- the model's `primary_task_name`
- the model's `total_number_of_versions`
- the `latest_version_no`
- the `latest_version_date`

Note: Every model in the platform is guaranteed to have at least one model version and exactly one primary task.

Note: You can safely assume that the version dates in the `ModelVersions` table follow a strictly increasing chronological order. In other words, a later version of a model is guaranteed to have a version date that is not earlier than any of its previous versions. Furthermore, no two versions of the same model share the same `version_date`. This ensures that the latest version of each model can be correctly identified using the maximum version date.

Note: Sort the results by `ModelID` in **ascending** order.

Output: An array of `QueryResult.ComprehensiveModelInfo` objects.

6.11 Task 11: Dataset Statistics by Modality (5 pts)

Method: `getDatasetStatisticsByModality()`
For each modality in the `Datasets` table, compute the total number of datasets and the average number of rows across those datasets. The output must include the `modality`, `dataset_count`, and `average_rows`.
Note: Sort the results by `average_rows` in **descending** order, and by `modality` in **ascending** order when equal.
Output: An array of `QueryResult.DatasetStatisticsByModality` objects.

6.12 Task 12: Retrieve Large-Parameter Model Versions Within a Date Range (5 pts)

Method: `getLargeModelVersionsByDateRange(String start_date, String end_date)`
In this task, you are asked to retrieve information about *large-parameter models*.
A model is considered "large" if its `size` value **ends with the character 'B'**. For example, sizes such as `0.4B`, `1.5B`, `7B`, or `70B` are all included.
For each large-parameter model, retrieve all model versions whose `version_date` falls within the given date range (inclusive).

The output must include the `ModelID`, `model_name`, `size`, `version_no`, and `version_date`.

Note: A large model that has no versions in the specified date range should not appear in the output. Since a model may have multiple qualifying versions, the same model may appear multiple times in the result list, once for each model version whose date is within the given date range.

Note: The `start_date` and `end_date` parameters are given in the format `YYYY-MM-DD`, and the range is inclusive (i.e., versions on the boundary dates are also included).

Note: Sort the results by `ModelID` in ascending order and then by `version_date` in ascending order.

Output: An array of `QueryResult.LargeModelVersionInfo` objects.

6.13 Task 13: Find Dataset(s) with Maximum Upload Count (5 pts)

Method: `getDatasetsWithMaxUploadCount()`
In this task, you are asked to find the dataset or datasets that have been uploaded the **MAXIMUM** number of times by users. For each such dataset, retrieve the `DatasetID`, the `dataset_name`, and the total number of uploads (`upload_count`).
Note: The result may contain more than one dataset if multiple datasets share the same maximum upload count.
Note: The output must be sorted by `DatasetID` in ascending order.
Output: An array of `QueryResult.DatasetMaxUploadInfo` objects.

6.14 Task 14: Find Complete Dataset(s) with All Roles (5 pts)

Method: `getCompleteDatasets()`
In this task, you are asked to find all **complete dataset(s)**. A dataset is considered **complete** if it has been uploaded by users with **all distinct roles** that exist in the system.
Note: For a dataset to be complete, there must be at least one upload recorded for *each* distinct role present in the `uploads` table.
Note: The output must be sorted by `DatasetID` in ascending order.
Output: An array of `Dataset` objects.

6.15 Task 15: Find Users Who Uploaded Datasets with Role 'creator' or 'contributor' but Never 'validator' and Have Reputation $\geq$ 60 (5 pts)

Method: `getUsersCreatorOrContributorButNotValidator()`
In this task, you are asked to retrieve all **User** objects representing users who have:

- performed at least one upload where the role was **'creator'** *or* **'contributor'**,
- never performed any upload where the role was **'validator'**, and
- have a **reputation score greater than or equal to 60**.

Note: The resulting list must be sorted by `PIN` in ascending order.
Output: An array of `User` objects.

6.16 Task 16: Find Users Who Ran All Versions of at Least One Model (5 pts)

Method: `getUsersWhoRanAllVersionsOfModels()`
Retrieve all users who executed runs on **every version of at least one model**. A user is included if they have run **all versions belonging to some model**, and for each such user-model-version_no triple, return their `PIN`, `user_name`, `ModelID`, `model_name`, `version_no`, and the model's `license`.

Note: A user is included only if they have run *every* version of at least one model. Since a model may have multiple versions, a qualifying user will appear *multiple times* in the output, with one entry for each version of that model. If a user has run all versions of multiple models, they will also appear multiple times in the results, with one row for each model version.

Note: The final output must be sorted by `PIN`, `ModelID`, and `version_no` in ascending order.

Output: An array of `UserModelVersionInfo` objects.

6.17 Task 17: Run-Type Statistics (5 pts)

In this task, you are asked to compute summary statistics for each distinct `run_type` recorded in the `runs` table. For each `run_type`, retrieve:

- the total number of **results created** from runs performed under that run type (`total_number_of_results`),
- the average `f1_score` of all such results (`average_f1_score`).

Note: Each result is created by exactly one run, but a run may create zero or more results. Therefore, only run types that actually have at least one associated result will appear in the output, since the statistics are computed over existing results.

Note: The final output must be sorted by `run_type` in **descending** order, and each result must contain `run_type`, `total_number_of_results`, and `average_f1_score`.

Output: An array of `QueryResult.RunTypeStats` objects.

6.18 Task 18: Find Publications That Include Results From Runs of a Dataset (5 pts)

Method: `getPublicationsUsingDataset(String dataset_name)`
Return all publications that include results created by **runs** that used the specified `dataset_name`. A publication should be returned only if:

- It includes at least one result that was **created by a run** using the given `dataset_name`, and
- That result has an accuracy value greater than or equal to **0.70**.

Note: The final result must be sorted by `PubID` in **ascending** order.

Output: An array of `Publication` objects.

6.19 Task 19: Find Top 10 Highly-Reputed Users (5 pts)

Method: `getTopTenHighlyReputedUsers()`
In this task, you are asked to compute a custom `user_score`, for every user in the system. The score is defined as the **sum** of the following components:

- **number_of_owner_uploads**: the number of datasets uploaded by the user with the role "owner",
- **number_of_publications_including_user_results**: the number of publications that include at least one result created from any run performed by that user,

- **reputation_score**: the user's reputation score from the `Users` table.

You must compute the score for all users, then return the **top 10 users** with the highest `user_score`, sorted in **descending** order of the score. If two users have the same score, break ties by sorting in ascending order of `PIN`.

Note: Users who have no owner uploads or no associated publications must still be included in the computation. In such cases, the missing values should be treated as zero. This is handled by using `CASE WHEN` (or an equivalent NULL-handling expression) to convert NULL values into 0 before computing the user score.

Output: An array of `QueryResult.HighlyReputedUser` objects, each containing the fields `PIN`, `user_name`, and `user_score`.

6.20 Task 20: Find Vulnerability Detection Publications (5 pts)

Method: `getVulnerabilityDetectionPublications()`
A research assistant is conducting a literature review study in the field of **Vulnerability Detection**. To support this study, the assistant needs to retrieve all publications that include results created by runs of model versions of models that are designed for the task `''Vulnerability Detection''` (task_name is 'Vulnerability Detection').

This query enables the research assistant to systematically retrieve all publications that report high-quality experimental results for the `''Vulnerability Detection''` task.

A publication should be returned only if:

- it includes at least one result created by a run whose model version belongs to a model that is designed for the task `''Vulnerability Detection''`, and
- the corresponding result has both `accuracy` and `f1_score` values greater than or equal to **0.70**.

For each qualifying publication, retrieve the following information:

- `PubID`
- `ResultID`
- `title` of the publication
- `venue` of the publication
- `run_type`
- `f1_score` of the result
- `accuracy` of the result
- `hyperparameter_config` of the result
- `placement_type`
- `placement_section`
- `user_name` of user who executes the run

- model_name
- size of the model
- version_no of the model
- dataset_name

Note: A publication may appear multiple times in the output because a single publication can include multiple qualifying results, and each qualifying result must be returned as a separate row.

Note: Return the results in **ascending order of** PubID, and for the same publication, in **ascending order of** ResultID.

Output: An array of QueryResult.TaskSpecificPublication objects containing the fields listed above.

7 Statements and Prepared Statements

In Java, SQL queries can be executed using either Statement or PreparedStatement objects.

7.1 Statement

A Statement object is used to execute static SQL statements that do not require parameters. It is suitable for simple, fixed SQL queries.

7.2 PreparedStatement

A PreparedStatement object is used for executing precompiled SQL statements that include parameters. It offers several advantages:

- Allows parameters to be set dynamically.
- Provides better performance because the SQL statement is precompiled.
- Improves security by preventing SQL injection attacks.

7.3 Example Methods

Below are two example methods demonstrating the use of Statement and PreparedStatement.

7.3.1 Example 1: Get Organizations with Rating Above 4.5

Using Statement:

```
public Organization[] getOrganizationsWithRating45() {
    String query = "SELECT * FROM Organizations WHERE rating > 
        4.5";
    List<Organization> orgList = new ArrayList<>();

    try {
        Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            int id = rs.getInt("OrgID");
            String name = rs.getString("org_name");
            double rating = rs.getDouble("rating");

            Organization org = new Organization(id, name, rating
                );
            orgList.add(org);
        }

        rs.close();
        stmt.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return orgList.toArray(new Organization[0]);
}
```

7.3.2 Example 2: Get Organizations with Rating Above a Given Threshold

Using PreparedStatement:

```
public Organization[] getOrganizationsWithRatingAbove(double
   threshold) {
    String query = "SELECT␣*␣FROM␣Organizations␣WHERE␣rating␣>␣?
        ";
    List<Organization> orgList = new ArrayList<>();

    try {
        PreparedStatement pstmt = connection.prepareStatement(
            query);
        pstmt.setDouble(1, threshold);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            int id = rs.getInt("OrgID");
            String name = rs.getString("org_name");
            double rating = rs.getDouble("rating");

            Organization org = new Organization(id, name, rating
                );
            orgList.add(org);
        }

        rs.close();
```

```
            pstmt.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return orgList.toArray(new Organization[0]);
}
```

Explanation:
In the first example, the condition (rating > 4.5) is fixed, so a `Statement` is sufficient. In the second example, the threshold is provided at runtime, so a `PreparedStatement` is used to safely insert the parameter and protect against SQL injection.

8 Implementation and Submission Instructions

For PA1, you are required to implement only the methods inside the `ModelHubPlatform.java` file. You must not modify any of the other source files (`Evaluation.java`, data classes, or the project structure).

During grading, an automated testing script processes each student's submission one by one. For every student, the submitted `ModelHubPlatform.java` file is taken from the student work folder and automatically placed into the official Maven project used for evaluation. The project is then compiled and executed to generate the student's `Output.txt`. This output is compared task by task against the expected `Output.txt`. If a task's output matches exactly, you receive **5 points** for that task; otherwise, you receive **0 points**.

8.1 Working as a Maven Project

You may import the provided project into your preferred IDE (such as IntelliJ or Eclipse) by opening the directory where the `pom.xml` file is located.

Alternatively, you can run the system directly from the command line as a Maven project. **This step is critical because the automated tester also builds and executes your code using these exact Maven commands.** Therefore, you must ensure that your final implementation works correctly with the following commands:

```
cd directory\where\your\pom.xml\file\is
mvn exec:java "-Dexec.mainClass=ceng.ceng351.ModelHubPlatform.Evaluation"
```

If your implementation does not run successfully with this Maven execution, the tester will fail to generate your `Output.txt`, resulting in zero points for all tasks.

8.2 Submission Requirements

- You must submit only the file: `ModelHubPlatform.java`.

- Do **not** submit the entire project or any additional files.
- Do **not** modify any other class; doing so may result in compilation errors or automatic loss of points.
- Ensure that your code successfully generates a valid `Output.txt` when executed with the provided Maven command.

9 Regulations

- **Programming Language:** Java (version 14).
- **Database:** Embedded H2 Database.
- **Cheating:** We have a zero-tolerance policy for cheating. Individuals involved in cheating will be punished according to university regulations.
- **Evaluation:** Input files are guaranteed to be correctly formatted. Similar (and larger) data will be used during evaluation. Your program will be evaluated automatically using a "black-box" technique, so ensure that you adhere strictly to the specifications. You must accomplish tasks using SQL queries within the Java methods, not with other Java programming facilities.

10 Grading

Your implementations will undergo black-box testing. For your submissions to receive credit, the results must **exactly match** the expected outputs. That is, adhering precisely to the output specifications, including the order of outputs, is critical.

All tasks from **Task 1 to Task 20** are worth **5 points each**.
Be aware that a different dataset will be used for evaluation purposes.