Overview

In this project, you'll create a cloud-based web application to store data for a movie database (like IMDB). This project will let you experiment with one of the latest industry practices for storing and retrieving data in a public cloud.

ER Diagram for Database

The database you will be creating in this project is modeled using the ER diagram seen in Figure 1 below.

Figure 1. ER diagram for movie database
Requirements

As the database of your application, you will be using Google App Engine’s Datastore, which provides a schema-less data storage mechanism. Your application will have a very simplistic interface, consisting of a text area for the user to enter commands and a “send command” button. You do not need to worry about the interface design, as the provided template already takes care of this. When users visit the application site in their browsers, they will see the following prompt:

Please enter a command in the text area below.

The application should support the following commands (parameters are separated by a single colon):

- add_actor: NAME:GENDER:DATE_OF_BIRTH
- add_director: NAME:GENDER:DATE_OF_BIRTH
- add_company: NAME:ADDRESS
- add_awards_event: EVENT_NAME:YEAR:VENUE
- add_user: USER_ID
- add_movie_rating: USER_ID:MOVIE_TITLE:RELEASE_YEAR:RATING
- add_cast: MOVIE_TITLE:RELEASE_YEAR:ACTOR_NAME:ROLE
- add_nomination_category: CATEGORY_NAME
- add_nomination: MOVIE_TITLE:MOVIE_YEAR:EVENT:EVENT_YEAR:CATEGORY:WON
- get_movies_by_company: COMPANY_NAME
- get_movies_by_director: DIRECTOR_NAME
- get_nominations_for_actor: ACTOR_NAME
- get_movies_of_genre_for_actor: ACTOR_NAME:GENRE
- get_number_of_nominations_for_movie: MOVIE_TITLE:RELEASE_YEAR
- get_average_rating_for_movie: MOVIE_TITLE:RELEASE_YEAR
- get_average_rating_of_user: USER_ID
- delete_company: COMPANY_NAME
- delete_user: USER_ID

You should implement each of the above commands. A detailed description of each command is provided below. For each of the above commands, you can assume that the input will always be formatted correctly, i.e. the number of parameters will be the same as that provided in the command list above and all parameters will be in the required domain. The only error cases you need to check are provided in the description for each command. In the case that something goes wrong with a command (which normally should not be the case, but you never know, life is full of surprises!), other than the errors listed below, please output your own descriptive error message. Note that each field of every table will be stored as a string, to make your job easier.
Description of commands

Please note that the commands provided below will be case-sensitive when testing the application. The following is a list of commands followed by a description of their parameters and the desired effects of invoking them.

• **add_actor:**
  NAME – full name of the actor/actress
  GENDER – gender of the actor (“Female” or “Male”)
  DATE_OF_BIRTH – date of birth of the actor in the format MM-DD-YYYY

  This command should add an actor/actress with the given parameters to the datastore. Addition of duplicate NAME is not allowed. In the case of attempting duplicate NAME addition, your program should print “Error: Actor already exists!”. If the insertion of the actor succeeds, you should print “Command executed successfully!”.

• **add_director:**
  NAME – full name of the director
  GENDER – gender of the director (“Female” or “Male”)
  DATE_OF_BIRTH – date of birth of the director in the format MM-DD-YYYY

  This command should add a director with the given parameters to the datastore. Addition of duplicate NAME is not allowed. In the case of attempting duplicate NAME addition, your program should print “Error: Director already exists!”. If the insertion of the director succeeds, you should print “Command executed successfully!”.

• **add_company:**
  NAME – name of the production company
  ADDRESS – address of the production company

  This command should add a production company with the given parameters to the datastore. Addition of duplicate NAME is not allowed. In the case of attempting duplicate NAME addition, your program should print “Error: Company already exists!”. If the insertion of the company succeeds, you should print “Command executed successfully!”.

• **add_movie:**
  TITLE – title of the movie
  RELEASE_YEAR – the year in which the movie was released
  LENGTH – length of the movie in minutes
GENRE – genre of the movie
PLOT – summary of the movie plot
DIRECTOR – name of the movie's director
COMPANY – name of the movie's production company

This command should add a movie with the given parameters to the datastore. Addition of duplicate {TITLE, RELEASE_YEAR} pairs is not allowed. In the case of attempting duplicate {TITLE, RELEASE_YEAR} addition, your program should print “Error: Movie already exists!”. If the insertion of the movie succeeds, you should print “Command executed successfully!”. For this command, you also need to check for foreign key constraints for the company and director entities, i.e. if the director name and/or company name does not exist in the database, your program should print “Foreign key constraints violated!”.

• add_awards_event:
  EVENT_NAME – name of the awards event
  YEAR – year in which awards event is held
  VENUE – venue at which the awards event is held

  This command should add an awards event with the given parameters to the datastore. Addition of duplicate {EVENT_NAME, YEAR} pairs is not allowed. In the case of attempting duplicate {EVENT_NAME, YEAR} addition, your program should print “Error: Event already exists!”. If the insertion of the event succeeds, you should print “Command executed successfully!”.

• add_user:
  USER_ID – unique identifier for a user of the movie database

  This command should add a user with the given parameters to the datastore. Addition of duplicate USER_ID is not allowed. In the case of attempting duplicate USER_ID addition, your program should print “Error: User already exists!”. If the insertion of the user succeeds, you should print “Command executed successfully!”.

• add_movie_rating:
  USER_ID – identifier for the user adding a movie rating to the database
  MOVIE_TITLE – title of the movie for which the rating is added
  RELEASE_YEAR – release year of the movie for which the rating is added
  RATING – user-provided rating for the movie

  This command should add a movie rating with the given parameters to the datastore. Addition of duplicate {USER_ID, MOVIE_TITLE, RELEASE_YEAR} triplets is not allowed. In the case of attempting duplicate {USER_ID, MOVIE_TITLE,
RELEASE_YEAR} addition, your program should print “Error: Rating already exists!”. If the insertion of the rating succeeds, you should print “Command executed successfully!”. For this command, you also need to check for foreign key constraints for the user and movie entities, i.e. if the user ID and/or {movie title, release year} does not exist in the database, your program should print “Foreign key constraints violated!”.

- **add_cast:**
  MOVIE_TITLE – title of the movie for which the cast is to be added
  RELEASE_YEAR – release year of the movie for which the cast is to be added
  ACTOR_NAME – name of the actor/actress to be added as cast member
  ROLE – role of the actor in the movie

This command should add a movie cast member with the given parameters to the datastore. Addition of duplicate {MOVIE_TITLE, RELEASE_YEAR, ACTOR_NAME} triplets is not allowed. In the case of attempting duplicate {MOVIE_TITLE, RELEASE_YEAR, ACTOR_NAME} addition, your program should print “Error: Cast already exists!”. If the insertion succeeds, you should print “Command executed successfully!”. For this command, you also need to check for foreign key constraints for the movie and actor entities, i.e. if the {movie title, release year} and/or actor name does not exist in the database, your program should print “Foreign key constraints violated!”. Note that for the purposes of this assignment, we assume that each movie has at most one lead actor/supporting actor/lead actress/supporting actress. The role could be one of these strings or the name of the character in the movie.

- **add_nomination_category:**
  CATEGORY_NAME – name of the award nomination category

This command should add an award nomination category with the given parameters to the datastore. Addition of duplicate CATEGORY_NAME values is not allowed. In the case of attempting duplicate CATEGORY_NAME addition, your program should print “Error: Category already exists!”. If the insertion of the nomination category succeeds, you should print “Command executed successfully!”. 

- **add_nomination:**
  MOVIE_TITLE – Title of the movie nominated
  MOVIE_YEAR – Release year of the movie nominated
  EVENT – Name of the awards event
  EVENT_YEAR – Year of the awards event
  CATEGORY – Nomination category
WON – Whether the specified movie won the award in the nomination category (“Yes” or “No”)

This command should add an award nomination with the given parameters to the datastore. Addition of duplicate {MOVIE_TITLE, RELEASE_YEAR, EVENT, EVENT_YEAR, CATEGORY} 5-tuples is not allowed. In the case of attempting duplicate {MOVIE_TITLE, RELEASE_YEAR, EVENT, EVENT_YEAR, CATEGORY} addition, your program should print “Error: Nomination already exists!”. If the insertion succeeds, you should print “Command executed successfully!”. For this command, you also need to check for foreign key constraints for the movie, event and nomination category entities, i.e. if the {movie title, release year} and/or {event, event year} and/or category does not exist in the database, your program should print “Foreign key constraints violated!”.

• get_movies_by_company:
COMPANY_NAME – name of production company

This command should print the title and release year of all movies produced by the company identified by COMPANY_NAME. The output should be a single line with different records separated by a semicolon and space and different fields of a record separated by a comma and space (the order of fields should be: movie title, release year).

e.g.
> get_movies_by_company:Paramount Pictures

• get_movies_by_director:
DIRECTOR_NAME – name of director

This command should print the title and release year of all movies directed by the director identified by DIRECTOR_NAME. The output should be a single line with different records separated by a semicolon and space and different fields of a record separated by a comma and space (the order of fields should be: movie title, release year).

e.g.
> get_movies_by_director:Clint Eastwood
Million Dollar Baby, 2004; Mystic River, 2003; True Crime, 1999; Absolute Power, 1997

• get_nominations_for_actor:
ACTOR_NAME – name of actor/actress
This command should print the awards event name, event year, nomination category and whether the award was won for all movies for which the actor identified by ACTOR_NAME was nominated. To determine those movies for which the specified actor was nominated, you should check that if the gender of the actor is “Male”, the nomination category is “best lead actor” and the role of the actor in the movie is “lead actor” OR the nomination category is “best supporting actor” and the role of the actor in the movie is “supporting actor”. You should use the same reasoning for female actors (i.e. actresses) as well (i.e. category=”best lead actress” and role=“lead actress” OR category=“best supporting actress” and role=“supporting actress”). The output should be a single line with different records separated by a semicolon and space and different fields of a record separated by a comma and space (the order of fields should be: event name, event year, nomination category, whether the award was won).

**e.g.**

```
> get_nominations_for_actor:Daniel Radcliffe
```

- **get_movies_of_genre_for_actor:**
  ACTOR_NAME – name of actor/actress
  GENRE – genre of the movie

This command should print the title and release year of all movies of a specific GENRE in which the actor identified by ACTOR_NAME was a cast member. The output should be a single line with different records separated by a semicolon and space and different fields of a record separated by a comma and space (the order of fields should be: movie title, release year).

**e.g.**

```
> get_movies_of_genre_for_actor:Natalie Portman:Drama
Black Swan, 2010; Hesher, 2010; Brothers, 2009; The Other Woman, 2009
```

- **get_number_of_nominations_for_movie:**
  MOVIE_TITLE – title of the movie
  RELEASE_YEAR – release year of the movie

This command should print the number of award nominations for the movie identified by MOVIE_TITLE and RELEASE_YEAR (note that this includes the total number of nominations, combining all award events and categories). The output should be a single number.

**e.g.**

```
> get_number_of_nominations_for_movie:The Sixth Sense:1999
47
```
• **get_average_rating_for_movie:**
  MOVIE_TITLE – title of the movie
  RELEASE_YEAR – release year of the movie

  This command should print the average user rating for the movie identified by
  MOVIE_TITLE and RELEASE_YEAR. The output should be a single real number
  accurate to one decimal place.

  **e.g.**
  > get_average_rating_for_movie:The Shawshank Redemption:1994
  9.3

• **get_average_rating_of_user:**
  USER_ID – unique identifier for movie database user

  This command should print the average rating the user identified by USER_ID has
  provided for all movies he/she rated. The output should be a single real number
  accurate to one decimal place.

  **e.g.**
  > get_average_rating_of_user:pangin
  7.8

• **delete_company:**
  COMPANY_NAME – name of the production company

  This command should delete the production company identified by
  COMPANY_NAME from the datastore. The deletion should fail and the error message
  “Referential integrity violation!” should be printed if there are any movies produced
  by this company in the database. If the deletion succeeds, the message “Command
  executed successfully!” should be printed.

• **delete_user:**
  USER_ID - unique identifier for movie database user

  This command should delete the database user identified by USER_ID from the
  datastore. The deletion should fail and the error message “Referential integrity
  violation!” should be printed if there are any movie ratings by this user in the
  database. If the deletion succeeds, the message “Command executed successfully!”
  should be printed.
Getting Started

The preferred implementation language for this project is Java. If you haven't used the Google App Engine before, the Google App Engine (GAE) Java tutorial at https://developers.google.com/appengine/docs/java/gettingstarted/introduction will provide all the necessary information for you to get started with the project. Note that you do not need to worry about the user authentication part of the tutorial.

An alternative to using Maven to build your project is to use Eclipse (see https://developers.google.com/appengine/docs/java/tools/eclipse) as the development environment. To install the Google App Engine plugin for Eclipse, go to “Install New Software” under “Help” in Eclipse, put the installation location (e.g. https://dl.google.com/eclipse/plugin/3.7) for your version of Eclipse in the “Work With” textbox and hit Enter. This will bring up a list of plugins in the window below and you only need to select “Google Plugin for Eclipse” and “SDKs” in that list and click “Next”. You should then proceed with the next steps to complete the plugin installation.

Your web application should be named “Movies” and the package for your source code should be named “movies”. Once you are done with the setup of the project (according to the tutorial), you should add the provided “ProcessCommandServlet.java” file to your package (movies), add the provided “movies.jsp” file to your “war” folder and replace the “web.xml” file in the folder “war/WEB-INF” with the web.xml we provide and replace MoviesServlet.java with the MoviesServlet.java file we provide (You will notice ProcessCommandServlet.java corresponds to SignGuestbookServlet, movies.jsp corresponds to guestbook.jsp and MoviesServlet corresponds to GuestbookServlet in the tutorial).

The only file you will need to modify is ProcessCommandServlet.java. The parts you will need to implement in that file are clearly marked, with comments to guide you. For additional information on querying the datastore of the Google App Engine, you can check https://developers.google.com/appengine/docs/java/datastore/queries.

Once you are done with the project and have tested it offline, you should upload your application to the Google App Engine, following the instructions at https://developers.google.com/appengine/docs/java/gettingstarted/uploading. Make sure you do not exceed the free storage and access limits when testing your application online, so we can test it online too. You might also want to delete the data in the datastore generated by your application, but the results of the tests we perform should not be affected even if you do not (unless you coincidentally insert the same data that we will use in our tests).
**Evaluation**

Your project will be evaluated based on how well it meets the specification, i.e. the correctness of the output.

**Submitting Your Work**
Please create a README file that contains identifying information. For example:

CS348 - Project 1

Author:  John Doe  
Login:  jdoe  
Email:  jdoe@cs.purdue.edu  
Application location: http://johns_app_id.appspot.com/

Include here anything you might want us to know when grading your project.

All commands should be implemented in the template file provided (ProcessCommandServlet.java). This file takes care of posting the results of a command to the application website, so you don’t need to worry about that.

To turn in your project, ssh to sslab00.cs.purdue.edu, create a folder named **project1** in your home directory and copy ProcessCommandServlet.java and README.txt to that folder. We should be able to run your application by opening your application location in a web browser. If there happens to be an error with the online version, we should be able to run it in Eclipse, using the movies.jsp, MoviesServlet.java and web.xml provided with the project.

After copying your files in the folder project1, execute the following command in your home directory:

```
turnin -c cs348 -p proj1 project1
```

To verify the contents of your submission, execute the following command:

```
turnin -c cs348 -p proj1 -v
```
Notes:

- Note that order doesn’t matter when printing out the results of a query.
- Be careful about not letting duplicate records in your datastore. Remember duplicate records are identified using the primary key.
- Make sure the application does not crash for any commands.
- You do not need to do any input error checking (on the number of fields in a command, validity of a command etc.) for this project. The project will be tested with correctly formatted input. Also, you do not need to check for domain constraints.
- Note that if an addition or deletion operation is not successful due to foreign key/referential integrity/key constraints, the state of the database should not change as a result of the command.
- To test your application, you can use the sample test to be provided with Project 1. We will be using a completely different dataset than that of Project 1 to test your application, so you should not have to worry about possible overlap of data even if you do not delete the data after your own tests.
- Including your application address in the README.txt is VERY important, so please do not forget to do that.