University of British Columbia, Vancouver
Department of Computer Science

CPSC 304 Project Cover Page

Milestone #: 4

Date: Nov 28, 2024

Group Number: 54

Name	Student Number	CS Alias (Userid)	Preferred E-mail Address
Julia Sangster	29688934	y1o9i	juliasangster@hotmail.com
Annie Chung	38565115	r9l7z	aachung@student.ubc.ca
Momin Kashif	13718895	q1d5z	mominkas@student.ubc.ca

By typing our names and student numbers in the above table, we certify that the work in the attached assignment was performed solely by those whose names and student IDs are included above. (In the case of Project Milestone 0, the main purpose of this page is for you to let us know your e-mail address, and then let us assign you to a TA for your project supervisor.)

In addition, we indicate that we are fully aware of the rules and consequences of plagiarism, as set forth by the Department of Computer Science and the University of British Columbia.

Repository link

https://github.students.cs.ubc.ca/CPSC304-2024W-T1/project_q1d5z_r9l7z_y1o9i

SQL script

You can run this script to initialize all tables and values in the database: initialize.sql

You can run this from the command line with:
`psql -U postgres -d dnd -f [path to initialize.sql]`

Project description

Our application provides a database solution for managing key components of Dungeons & Dragons campaigns by modeling campaigns, players, characters, classes and species. Players can look up character stats and detailed information regarding character classes and species. Game masters can pull up summaries of their managed campaigns. Participants can also see interesting stats about their fellow players through summaries we've created.

The largest change to our schema was in the participant entity. We wanted a way to manage what a game master and game player could access, in terms of operations. To do so, we now require that users log into the application before accessing data. This required a new password field to be added to the participant table.

```
CREATE TABLE Participant (
    participant_id SERIAL PRIMARY KEY,
    location VARCHAR(1000) NULL,
    name VARCHAR(100) NOT NULL,
    password VARCHAR(255) NOT NULL,
    experience_level INTEGER NULL
);
```

Another minor change was adding the requirement for difficulty level of campaigns to be NOT NULL since we wanted to perform several queries based on difficulty level and it made sense for a campaign to require one.

```
CREATE TABLE Campaign (
      campaign_id SERIAL PRIMARY KEY,
      campaign_name VARCHAR(100) NOT NULL,
      meeting_location VARCHAR(100) NULL,
      meeting_time TIME NULL,
      setting VARCHAR(1000) NULL,
      difficulty_level VARCHAR(100) NOT NULL,
      max_num_players INTEGER NOT NULL,
      current_num_players INTEGER NOT NULL,
      description VARCHAR(1000) NULL,
      date_created DATE NOT NULL,
      game_master_id INTEGER NOT NULL,
      FOREIGN KEY (game_master_id) REFERENCES Game_Master(game_master_id)
);
```

Lastly, we added a unique constraint to the name field in the participant table. The name serves as the username (and also what game masters see when trying to add a user to a campaign) which is why we did not want to allow duplicates.

```
CREATE TABLE Participant (
      participant_id SERIAL PRIMARY KEY,
      location VARCHAR(1000) NULL,
      name VARCHAR(100) NOT NULL UNIQUE,
      password VARCHAR(255) NOT NULL,
      experience_level INTEGER NULL
);
```

Queries

Note that the heading of each query contains a link to the code containing/generating that query.

2.1.1 INSERT

This SQL statement allows a user to add a class (name and level) to the system, which they input via a form. Name is a FK in class description, and level is a FK in class level features. This operation handles the case where either FK does not exist and rejects the tuple with an error message that appears on the user-interface: "This class name or level does not exist."

```
Inputs:
   -  $1 - user specifies class name to add
   -  $2 - user specifies class level to add

INSERT INTO class (name, level) VALUES ($1, $2)
```

2.1.2 UPDATE

This SQL statement allows a user to edit/update a previously added character. This is done via a pre-populated input form with the currently-held values. Fields which are FKs (class_name, species_name, level) have restrictions on the UI that only allow valid entries, via sub-queries also called with UPDATE call.

```
Inputs: array of inputs with indices...
   -  $1 - name               as specified by users
   -  $2 - hair_color         as specified by users
   -  $3 - eye_color          as specified by users
   -  $4 - level              as specified by users, from valid FK options
   -  $5 - position           as specified by users
   -  $6 - class name         as specified by users, from valid FK options
   -  $7 - species name       as specified by users, from valid FK options
   -  $8 - hp                 as specified by users, or by RNG rolling of
                              specified class's hit-die
   -  $9 - character_id       as specified by users, upon selecting edit

UPDATE character
SET   name              = $1, hair_color  = $2,
      eye_color         = $3, level       = $4,
      position          = $5, class_name  = $6,
      species_name      = $7, hp          = $8
WHERE character_id = $9
RETURNING name
```

2.1.3 DELETE

On the Class page, under Class Level Features, the user can delete any row in the table. This delete will cascade and also delete any Class and Character associated with the level, via the CREATE TABLE code. A modal will pop up when a user deletes a class level feature so the user can confirm deletion.

```
Input:
   - $1 - user specifies which level to delete

DELETE FROM class_level_features WHERE level = $1
```

2.1.4 SELECTION

On the Species page, the user can search for rows by selecting a combination of conditions by using attribute names, operators and providing values. Species that meet the search criteria are returned.

```
Inputs:
   - conditions comprising an attribute, an operator, a value and a clause
   - attributes ⊆ [name, description, weight, height, type]
   - operators ⊆ [=, <>, <, >, <=, >=, like]
   - values ⊆ [string, number]
   - clauses ⊆ [AND, OR]

SELECT * FROM species WHERE conditions
```

2.1.5 PROJECTION

On the Classes page under Class Descriptions, this query is used to show the user all class descriptions, but only the attributes from the “selected columns to view” that the user has selected.

```
Inputs:
   - attributes ⊆ [name, description, primary_ability, weapon_proficincy,
     armor_proficiency, hit_die, saving_throw_proficiency]

SELECT attributes FROM class_description
```

2.1.6 JOIN

On the Campaigns page, this query is used to display all the campaigns the user is a part of (both game player and game master roles). The user is able to choose between the difficulty of the campaigns to display via a dropdown on the campaigns page (all, easy, medium, hard). Only campaigns of the chosen difficulty are selected to display to the user.

```
Inputs:
   -  attributes ⊆ ['c.campaign_id', 'c.campaign_name', "c.meeting_location",
      'c.meeting_time', 'c.setting', 'c.difficulty_level',
      'c.max_num_players', 'c.current_num_players', 'c.description']

SELECT * FROM (
      SELECT attributes, 'Game Master' as role
      FROM campaign c
      JOIN game_master gm ON c.game_master_id = gm.game_master_id
      WHERE gm.game_master_id = $1

      UNION

      SELECT attributes, 'Player' as role
      FROM campaign c
      JOIN enroll e ON c.campaign_id = e.campaign_id
      JOIN game_player gp ON e.game_player_id = gp.game_player_id
      WHERE gp.game_player_id = $1

      ) combined_campaigns
${difficulty && difficulty !== 'all' ? 'WHERE difficulty_level = $2' : ''}
```

2.1.7 Aggregation with GROUP BY

On the home page, this query returns the name of each species and the number of characters belonging to that species. It groups the results by species name, counting the number of characters for that species and sorts the results in descending order based on the number of characters.

```
SELECT s.name AS name, COUNT(*) AS count
FROM character c
JOIN species s ON c.species_name = s.name
GROUP BY s.name
ORDER BY count DESC
```

2.1.8 Aggregation with HAVING

On the home page, this query returns the participant ID, name and average character level for each participant with an average character level greater than 2. It groups the results by each participant's ID and name, calculates the average character level for each group and filters out participants who don't meet the condition. The results are sorted in descending order of average level.

```
SELECT p.participant_id, p.name AS name, AVG(c.level) AS avg
FROM participant p
JOIN character c ON p.participant_id = c.participant_id
GROUP BY p.participant_id, p.name
HAVING AVG(c.level) > 2
ORDER BY avg DESC
```

2.1.9 Nested aggregation with GROUP BY

On the home page, this query returns the most popular difficulty level. The query computes the ratio of current to enrolled players for each difficulty level across campaigns and then selects the difficulty level with the highest ratio. This provides an overall measure of the experience of all the users.

```
SELECT
      c1.difficulty_level,
      AVG(c1.current_num_players * 1.0 / c1.max_num_players) AS avg_ratio
FROM campaign c1
GROUP BY c1.difficulty_level
      HAVING AVG(c1.current_num_players * 1.0 / c1.max_num_players) >= ALL (
            SELECT AVG(c2.current_num_players * 1.0 / c2.max_num_players)
            FROM campaign c2
            GROUP BY c2.difficulty_level);
```

2.1.10 DIVISION

On the home-page this query identifies and returns the name, location, and experience level of all participants who have at least one character in every class. This is meant to identify "masterful" players

---

```
SELECT p.name, p.location, p.expereience_level
FROM Participant p
WHERE NOT EXISTS (
      (SELECT c.name
       FROM class c)
       EXCEPT (SELECT c2.name
               FROM character char
               JOIN class c2 ON char.class_name = c2.name
               WHERE char.participant_id = p.participant_id))
```