**DOCUMENTATION**

We developed an online game store, which provides a platform to players for playing games hosted on third-party servers, as well as developers for hosting these games on the site. The two user types are mutually exclusive; a user cannot be both a player and a developer.

**Database structure:**

For our application, we have 3 tables – User, Games and Scores. The User table has been extended to include an additional field called “**usertype**”, which specifies if the user is a player or a developer. This field has been added as a part of a Usertypes table, which hence shares a one-to-one relationship with the User table. The User fields used in our project are:

1. **username**
2. **first\_name**
3. **last\_name**
4. **email**
5. **password**
6. **is\_active**

The Games table, like User, is a “master” table of all the games available in the store. This table has the following fields:

1. **name** – Unique for each game
2. **category** – For our store, we have five types of games – adventure, football, puzzle, racing and sports.
3. **url** – this is the URL of the game as hosted by a third-party server
4. **developer** – A User object of “developer” type representing the developer who developed this application. This field shares a foreign key relationship with the User table.
5. **price** – the price of each game, as set by the developer

Unlike the User and Games tables, which are essentially static in nature, the Scores table is more dynamic, “transactional” in nature. This table is regularly updated with the latest score of each user for each game he/she plays, as well as the high scores if this latest score is amongst his/her 5 highest scores achieved for that game. The fields in this table are:

1. **game** – A Game object representing the game the user plays. This field shares a foreign key relationship with the Games table.
2. **player** – A User object of “player” type representing the user registered to play the aforementioned game. This field shares a foreign key relationship with the User table.

By this design, the combination of the “game” and “player” fields is unique for each Scores object.

1. **high\_score\_1** – an integer field storing the highest score achieved by the user playing this game**.**
2. **high\_score\_2** – an integer field storing the 2nd highest score achieved by the user playing this game**.**
3. **high\_score\_3** – an integer field storing the 3rd highest score achieved by the user playing this game**.**
4. **high\_score\_4** – an integer field storing the 4th highest score achieved by the user playing this game**.**
5. **high\_score\_5** – an integer field storing the 5th highest score achieved by the user playing this game**.**
6. **last\_score –** the last score achieved by the player playing this game.

**Authentication:**

The homepage of our website provides a common login facility to the user. The user must indicate if he/she is a player or developer, as depending on that the player or developer homepage opens.

New users register on a separate page, entering details such as username, password, and email, and the user type of choice. Upon all these details and clicking the “Register” button on the file “registration.html”, the user’s details are saved to the database, but the “is\_active” attribute of the user is set to False. As long as this is False, the user will not be able to enter the player or developer homepage (depending on the user type selected) from the home page. That can be done only when the is\_active field is set to True. This is done by clicking on a verification URL sent to the user’s entered email address. For our project, we have used the Django console backend as the email server, and so the email appears in the terminal. The user has to copy-paste that URL in the browser’s address bar. If the verification is successful, the user’s is\_active attribute is set to True, and he/she is redirected to the login page, wherefrom the user can now view the appropriate homepage (i.e. player/developer) upon successful login.

The verification URL consists of the server name, followed by a “/” and then the username and password separated by a “:::” character. Upon entering this URL in the browser, a verify() view is called in the server which parses the characters following the aforementioned “/” character. The parsing is done based on the “:::” character. The username and hashed password (Django stores a hash of the password in the database, instead of the password itself) are used to query a the User table, which should return only a single User object, as the username+password combination is unique. We felt this combination would be secure, as even if an attacker got the username, it would require an extraordinary brute force attack to decipher the password from the hash. On the other hand, using only the password hash in the URL would be insufficient in case 2 users registered with the same password.