**DOCUMENTATION**

**Group members:**

Alapan Mukherjee –

Gergely Horváth – 432885

Habtamnesh Goite –

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. A user initially logs in as a player, and has the option to upgrade to a developer, in which case he/she can both play and create games.

**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 “**developer**”, which is a boolean field specifying 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 the login facility to a user. When the account is first created, the user is a player, with the “developer” field of the Usertypes table set to False. Later on, the player can update his/her profile to become a developer as well. This sets the “developer” field to True. However, regardless of the value of this field, when a user logs in, the player homepage always appears. An upgraded player (i.e. developer) has the option to change over to the developer homepage, which in turn provides the option to change back to the player homepage. The options are provided through hyperlinks. One point to note here is that there is no option available to

On entering player\_homepage.html, the player can see a link “Update your profile”, which the user clicks on to go to editprofile.html, in whch the user can update his/her details. It is necessary to write the password in the password field (which is the only field to be blank by default in the page) of editprofile.html,

New users register on a separate page, entering details such as username, password, and email. 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 (initial) player homepage 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 Gmail's SMTP server. 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 player homepage 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.

We have used Django's in-built authentication mechanism to authenticate the user.

**Player functionality**

The player homepage is the default homepage for any new user. For a player not having any games, the page loads with a message indicating that this player has not bought any games as yet. The rest of the page is the same as that of a not-logged-in user. A player who has already bought some games will see a page having a table of the games bought. For each game, there is a “Game info” link, on which the player can click to view some information about the game (game\_info.html). As the player has already bought the game, game\_info.html has a “Play this game” link, on which the user clicks to go to game.html, which loads the actual game as a third-party.

The users have the opportunity to search for games as well. In the navigation sidebar there are the pages for each category, through which the corresponding games are displayed in a table. If the user is looking for a specific game a textual search field is available in all of the pages. It is searching through the database for any match in game names, which contains any characters entered in the search input. It does not matter how the name is typed in, the input is normalized.

When a player selects a game and the game\_info.html is rendered and this game is not yet bought by this player, pressing the “Buy <game\_name>” button will direct to the payment page. The start\_buy.html is rendered where there is a hidden form with all the necessary information about the game and about the user for the payment to be successful. If the user presses the “Continue” button the payment service comes up, where the player can accept or decline the pament. If it is canceled then the payment/cancel.html page is rendered with the information that the payment was canceled. If the player accepts the payment the payment service sends back the payment id, the reference number and the checksum to our service. The payment/success.html is rendered and the success\_view handles the checks and database updating. First it checks the checksum to verify that the response was coming from the payment service. If it matches then it checks the payment id. The payment id is fabricated from the user id, a separator character “\_” and the game id. Based on this the service identifies both the user and the game and then creates a new Scores object which contains the user object, the game object and the current time and date of the registration and saves it into the Scores table. The Scores table besides storing the scores for each player and each game, is responsible for handling the authentication of which player has which game.

**Developer functionality**

When a player has upgraded to a developer, he/she has the option to change over to the developer homepage. The link to the developer page appears for the upgraded user. After the link has been clicked and the developer page loaded, initially the player (now developer), finds an empty table meant to display a list of games present in the developer's inventory. At the beginning, there are no games added, and so the table is blank. The developer can then click on “Add new game” to go to addgame.html, where a new game can be added by entering details such name of the game, category, URL and price. As the name is unique for each game, if Django finds that a game with that name already exists, the user is notified and the entry not saved to the database.

For each game in the list of games in the developer homepage, 3 links/buttons exist which allow the developer to edit game details, delete the game, and view game statistics. The latter consists of showing how many copies of the game were sold on a day, and the total number of copies sold.

**Game-service interaction**

There are 5 types of JSON messages being passed between the game in game.html and our service. The types, along with their uses, are described below:

1. SCORE – This message is used to display the last score in the score.html page. It is sent by the game to the service.
2. SAVE – SAVE messages save the gamestate - comprising of the game-specific player items and the last score – to the database. When doing this, the last score is compared to the existing to the 5 highest-score fields. If it is found to be larger than any of them, that high score is replaced by the last score. It is sent by the game to the service.
3. LOAD\_REQUEST – LOAD\_REQUEST is sent by the game to the service to load the gamestate for this game and player, if it exits
4. LOAD – The service sends a LOAD message to the game with the retrieved gamestate when there is one for the player and the game. The contents of this message are parsed and displayed on game.html.
5. MESSAGE – If no gamestate exists for the game and player, a MESSAGE response is sent from the service to the game indicating that no gamestate. This message is displayed on game.html as well.

**Other features**

1. RESTful API – We implemented a RESTful service for displaying the high scores on the page highscores.html. When the view highscores(request, id) receives a GET request, it

**Front end design**

We created a draft design of the web site layout before writing the HTML and CSS files. Based on the design sketch we first created a kind of template that contains the core contents that will be present in all the pages. This is so that all pages have a uniform content. The template included the title bar which includes the title and the logo of the page, left and right side bars and footer.

The design took the color consistency as well. We have chosen bright colors as the sight is entertainment website and should be pleasant and attractive colors.

We used the template as a base to create all pages . The template was helpful as it provided us the core elements . All pages contain features unique to the theme and also the basic core elements.

The CSS is a separate file saved in a static folder in our django project. All Html files included a link to the CSS. Having a separate CSS file make the code clear and helped us use same css for all the pages. We divided the page in four parts and created classes . These classes were use to position contents of every page in the desired location as well as apply customs such as color and size.

We used images for categories , log and profile form external source and have included the source of the images in the footer of all the pages as the images are licensed under a “Creative Commons Attribution 3.0 Generic License”.

Although the plan included external authentication with Facebook , we were unable to make the login work because of error “Unicode-objects must be encoded before hashing”. As a result we remove the login with facebook option we had.

**Technical design**

We have adhered to the Model-View-Template architecture as far as possible by using very little code in the templates, and making the database queries and business logic in the views. For pages like registration.html, Django's ModelForm feature proved to be extremely useful, as it did the rendering of the fields and their validations itself.

While most of the code involved Python, HTML and CSS, a small amount of Ajax was used to display details like game statistics in developers\_homepage.html. The latter was needed to create sections whose contents changed dynamically, such as due to the click of a button.

We have generally tried not to repeat the same content in different HTML files. Any message (e.g. error, logout, verification) has been displayed on the same page responsible for calling the view, which generated the message (e.g. login.html).

**Features implemented with points**

**Authentication:** We have implemented all the requirements, and the e-mail validation is using SMTP server.

**200/200 points**

**Basic player functionalities:** All requirements are fulfilled, payments are handled and checked, players are able to play the test game, game-service interaction works, search feature is implemented with categories and with textual search.

**300/300 points**

**Basic developer functionalities:** Adding, setting, modifying and removing a game is possible, game inventory is displayed with all the functionalities easily accessible, basic game statistics displayed for each game with sold copies and dates.

**200/200 points**

**Game/service interaction:** All messages are implemented, scores are saved for each player. Global high-score for games are not implemented due to lack of attention.

**175/200 points**

**Quality of work:** All features are well commented, structure is coherent and consistent, DRY principle is applied in every possible place, M-V-T separation applied, user experience is thoroughly tested and style is changed many times to be the best possible.

**90/100 points**

**Save/load feature:** With the game/service interaction, the service supports basic save and load functionality.

**100/100 points**

**RESTful API:** We have implemented a RESTful API for displaying the highscores fot each game.

**100/100 points**

**Non-functional requirements:** Documentation contains all the information about the implemented features and the way they work, the service is well tested and prepared with sample data for the demo, project management was mainly based on the Niksula repository, however we have arranged multiple face-to-face meetings, where we worked together, designed and implemented features together.

**175/200 points**

**Instructions to use our application**

Note that in this website all players must be registed and logged in to play a game.

**General**

* On the right sidebar can a user log in or register
* Registering is through the link “Sign up”
* After successful registration the user receives a verification e-mail, in which there is a verification url through the account can be verified
* After verification the user can log in
* After log in first the player homepage is displayed for all types of users
* On the left sidebar the user can navigate on the site
* Logout button is in the right sidebar
* A user can be only a player but not a developer, or a developer but then it is a player as well by default
* All the games added by a developer are instantly available in their player inventory as well
* Changing between the player and developer homepages is done via the “Developer homepage” or “Player homepage” links in the right sidebar, where the value depends on what homepage is currently visible to the user

**Players**

* Players can only play games they paid for
* If the user does not have any games, the most popular games are listed
* Otherwise their own inventory is displayed
* Players can search for games either in the left navigation sidebar or through textual search
* Clicking the “Go to game” link directs the user to the game information page
* If the user have the game in their inventory they can click “Play game”
* Otherwise there is a button “Buy <game\_name>” through which the player can buy the game
* Choosing the “Play game” will direct the user to the game page, where the game can be played
* Game state and scores are listed while the player plays, and if the game is saved then the scores are saved to a highscore table
* Players can change their personal information via the “Update your profile” link
* Here can players upgrade themselves to developer

**Developers**

* On the developer homepage the developers can add new games buy clicking on the “Add game” link
* After that a form comes up with all the required fields for game information
* Saved added games are shown in a table for developers
* There is a button for each game “View game statistics” which displays the actual games basic sales figures under the games table
* Editing a game is through the “Edit” link
* Deleting a game is through the “Delete this game” link

**Heroku URL to the deployed application:** <http://gladiatorgames.herokuapp.com/>

**Division of work**

The sections were divided amongst the various team members as follows -

1. Front-end (HTML, CSS), third-party authentication, implementing views, database design – Habtamnesh Goite
2. Front-end (HTML), player functionality (payment, textual search, category search, related information pages, popular games), e-mail validation, database design - Gergely Horváth
3. Developer functionality (add/edit/delete games), user authentication, e-mail validation, RESTful service for high scores, database design – Alapan Mukherjee