Introduction:

This guide is designed to help new developers understand the code of the gamified eco-friendly app, Eco-Mystery, developed with Django. It provides an overview of the project structure, key resources used, and coding conventions. By following this guide, new developers should be able to quickly get up to speed with the project and contribute effectively.

Apps:

The project structure of our Django project consists of three main apps: **groupdjangoproject**, **game**, and **users**. The **groupdjangoproject** app is the main app of the project, which contains the core functionalities and connects the other two apps.

The **game** app is responsible for handling all the game-related functionalities, such as game logic, scoring, game levels and the unity application. The **user's** app handles user authentication and registration, as well as user profiles and preferences. Each app follows the standard Django directory structure, with their own **models.py**, **views.py**, **templates/**, and **static/** directories. The **groupdjangoproject** app contains the main **urls.py** file, which connects all the URLs from the different apps and routes them to the appropriate views. The project also includes a **settings.py** file, which contains all the project-wide configurations, such as database settings, secret keys, and installed apps. This project structure allows for an effective organization of the project's constituent parts, making it easier to maintain and extend the application in the future.

Game Models:

The **Challenge** model is used to store information about challenges in the game. It has three attributes: **challengeId**, **challengeDesc**, and **challengeType**. *challengeId* is an integer, *challengeDesc* is a text field, and *challengeType* is a text field with options ('Bin', 'Green Areas', 'Walking').

The **Fact** model is used to store facts of the day. It has two attributes: date and fact. date is a date field, and fact is a text field.
The Bin model is used to store information about bin locations in the game. It has five attributes: bin_number, latitude, longitude, poi_info, and challenge. bin_number is an integer, latitude and longitude are float fields, poi_info is a text field, and challenge is a foreign key to the **Challenge** model.

The **Suspect** model is used to store information about suspects in the game's mystery minigame. It has four attributes: story, number, name, and brief. story is a foreign key to the Story model, number is an integer, name is a text field, and brief is a text field.
The Story model is used to store information about the story in the game's mystery minigame. It has nine attributes: *MAX_SUSPECTS*, story_number, *sprite_1*, *sprite_2*, *sprite_3*, *sprite_4*, sprite_5, *clues*, and *culprit*. *MAX_SUSPECTS* is a constant used to limit the number of suspects in the game, story_number is an integer, sprite_1-5 are character sprite codes used to customize the appearance of the game, clues is a text field used to store clues, and culprit is an integer used to represent the suspect who is guilty in the game.
Each model has a string representation function (**str**()), which returns a string representation of the model for display purposes, specifically compatibility with the Unity engine. The **Suspect** model also has a Meta class, which is used to define metadata for the Django admin site. The Suspect model also has a save function that overloads the existing models.Model.save() function.

Users Models:

The **users** models.py file in Django is where the models for user accounts are defined. The file contains three model classes: **Account**, **Profile**, and **ChallengeTracker**.
The **Account** model is used to store user information such as their email, username, password, level, points, and other attributes related to their progress in the app. This model has several methods including *current_level()*, *level_progress()*, *is_my_bool_field_true()*, *account_dailypoints()*, and *account_points()*.
The **Profile** model is used to store additional information related to a user such as a profile picture, and it is linked to the built-in Django User model with a OneToOneField.
Finally, the ChallengeTracker model is used to store the progress of users on various challenges that are available in the app. This model has an account field, a challenge field, and a completed field.
The models are defined using Django's built-in model structure, which includes various types of fields such as CharField, IntegerField, BooleanField, and ForeignKey, to represent the different types of data that are stored in the models. The models also include various methods to manipulate and retrieve data stored in the models.

Game Views:

The game ‘views.py’ begins with import statements that import necessary modules such as math, Django's built-in render and redirect functions, and various models and forms from the application. The first function defined in the module is **green_checker**, which checks whether any incomplete challenges related to green areas have been completed

by the user, based on the user's green counter and the target number of green areas for the challenge.
The **home** function serves the homepage for the game application. It collects information about the logged-in user and their competition information with other users in the same accommodation. It also finds and displays the current fact of the day, calculates the blur strength of the fact, and computes the user's progress towards completing the fact. Finally, it renders the 'overview.html' template with the collected context variables.
The **leaderboard** function serves the leaderboard page for the game application. It collects and displays the sorted points of all users within the same accommodation as the current user, as well as the sorted point totals for all accommodations. Finally, it renders the *leaderboard.html* template with passed context variables.
The ***map()*** function requires a user to be logged in (decorated with the **@login_required** decorator). It serves a web page that displays a map of the game area, with markers indicating the locations of trash bins. It also checks if a user has clicked on a green area in the map, which is defined by a polygon. If the user is inside the green area, their *greenCounter* (number of times they entered a green area) and points are increased, and a message is returned as a JSON object.
The ***news()*** function also requires a user to be logged in. It serves a web page that displays the latest environmental news obtained through a third-party API.
The ***challengeManager()*** function also requires a user to be logged in. It serves a web page that displays the user's completed challenges and the challenges they have yet to complete. It also displays the number of green areas, bins, and walks that the user has completed.
The ***QR()*** function requires a user to be logged in. It serves a web page that displays a QR code that the user can scan to start a walk.
The ***update_points()*** function, requires a user to be logged in. It updates the user's record in the database with the current number of points they have earned.

Users Views:

The **register** function is defined in this file. It handles requests sent to the URL **/register.html**. If the request is a POST request, it means the user has submitted a registration form. The form data is first cleaned using the **UserRegistrationForm** from **forms.py**, then validated to ensure that the user has provided a valid email address.
If the validation is successful, a new user account is created by instantiating the **Account** model from **models.py** and saving it to the database. A success message is then displayed to the user on the front end, and they are redirected to the home page of the site. If the validation fails, a warning message is displayed to the user on the front end.

If the request is not a POST request, a blank registration form is rendered using the **UserRegistrationForm**, and the user is presented with the registration page.
The function returns an **HttpResponse** object, which serves the **register.html** template with the context of the instantiated **UserRegistrationForm** if the request is a GET request, or redirects the user to the home page if the request is a POST request and registration is successful.

Groupdjangoproject Urls:

The **urls.py** file in the **groupdjangoproject** directory is responsible for routing URLs to the appropriate views in the project. It defines the urlpatterns list, which maps specific URLs to views that handle those URLs. The first few lines in the file contain imports of necessary modules such as the **admin** and **auth_views** modules. Additionally, the **user_views** module is imported from the **users** app. The **urlpatterns** list defines the routing patterns for the application. The first pattern maps the **admin/** URL to the Django admin site. The second pattern maps the root URL to the **sitePage** app's **urls.py** file using the **include()** function. The third pattern maps the **register/** URL to the **user_views.register** function defined in the **views.py** file of the **users** app.
The fourth and fifth patterns map the **login/** and **logout/** URLs to the Django built-in **LoginView** and **LogoutView** classes respectively. The sixth pattern maps the **game/** URL to the **game** app's **urls.py** file using the **include()** function.

Game Urls:

- **path('', player_views.home, name='overview')**: This path maps to the **home** view in the **player_views** module and serves as the default homepage of your game app.
- **path('leaderboard/', player_views.leaderboard, name="leaderboard")**: This path maps to the **leaderboard** view in the **player_views** module and serves the leaderboard page for your game.
- **path('profile/', player_views.profile, name='profile')**: This path maps to the **profile** view in the **player_views** module and serves the profile page for your game.
- **path('map/', player_views.map, name='map')**: This path maps to the **map** view in the **player_views** module and serves the map page for your game.
- **path('news/', player_views.news, name='news')**: This path maps to the **news** view in the **player_views** module and serves the news page for your game.

- **path('challenges/', player_views.challengeManager, name='challengeManager')**: This path maps to the **challengeManager** view in the **player_views** module and serves the challenges page for your game.
- **path('QR/', player_views.QR, name='QR')**: This path maps to the **QR** view in the **player_views** module and serves the QR code page for your game.
- **path('update_points/', player_views.update_points, name='update_points')**: This path maps to the **update_points** view in the **player_views** module and serves as an endpoint for updating player points in your game.
- **path('unity/', player_views.unity, name='unity')**: This path maps to the **unity** view in the **player_views** module and serves the Unity game page for your game.
- **path('Receiver/', player_views.Receiver, name='Receiver')**: This path maps to the **Receiver** view in the **player_views** module and serves as an endpoint for receiving data from the game client.
- **path('get_Directions/', player_views.get_Directions, name='get_Directions')**: This path maps to the **get_Directions** view in the **player_views** module and serves as an endpoint for getting directions in your game.

Map JavaScript:

- Setting up the Mapbox API access token:
  - **mapboxgl.accessToken** sets the access token for Mapbox API, which is required to access the maps and other features of the Mapbox platform.
- Creating a GeoJSON object:
  - **const geojson** is a JavaScript object that defines a GeoJSON feature collection. It includes four point features with coordinates and properties for each point.
- Creating a map object:
  - **const map** creates a new instance of the **Map** class from the Mapbox API.
  - **container** specifies the HTML element where the map will be rendered.
  - **style** specifies the Mapbox style to use for the map. The 'streets-v12' style is used in this example.
  - **center** specifies the initial center of the map.
  - **zoom** specifies the initial zoom level of the map.
- Adding markers to the map:

    - A for-loop is used to iterate through each feature in the GeoJSON object.
    - For each feature, a new HTML element is created and assigned a CSS class.
    - A new marker is created using the **Marker** class from the Mapbox API.
    - The marker's position is set to the feature's coordinates, and a popup is added with the feature's title and description.
    - The marker is added to the map using the **addTo** method.
- Adding additional markers to the map:
    - Another for-loop is used to iterate through an array of **bin_info**.
    - For each element in the array, a new marker is created using the **Marker** class from the Mapbox API.
    - The marker's position is set to the latitude and longitude coordinates in the array, and a popup is added with the text from the third element in the array.
    - The marker is added to the map using the **addTo** method.
- Adding green-areas (polygons) to the map:
    - Four new data sources are added to the map using the **addSource** method.
    - Each data source includes a GeoJSON object that defines a polygon feature with coordinates for each point.
    - The **addLayer** method could be used to style the polygons and add them to the map.

- **getLocation()** function: This function checks if the browser supports geolocation or not. If it does, it calls the **getCurrentPosition()** method to retrieve the user's current position. This function passes the position object returned by **getCurrentPosition()** to the **showPosition()** function, which extracts the latitude and longitude values and sends them to the **locationParserToReceiver()** function using an AJAX POST request. This function is called when the page loads, and it retrieves the user's current location data.
- **getFinalLocation()** function: This function is similar to **getLocation()**, but it retrieves the user's final location data. This function is called when the user clicks a button, and it retrieves the user's final location data.
- **showPosition()** function: This function extracts the latitude and longitude values from the position object returned by **getCurrentPosition()**. It then calls the **locationParserToReceiver()** function, which sends an AJAX POST request to the server to save the user's location data.
- **showFinalPosition()** function: This function is called when the **getFinalLocation()** function retrieves the user's final location. It extracts the

latitude and longitude values from the position object and sends them to the **locationParserToget_Directions()** function using an AJAX POST request.

- **locationParserToReceiver()** function: This function sends an AJAX POST request to the server to save the user's location data. It sends the latitude and longitude values as data and includes the CSRF token to prevent CSRF attacks.
- **locationParserToget_Directions()** function: This function sends an AJAX POST request to the server to retrieve directions based on the user's location data. It sends the latitude and longitude values as data and includes the CSRF token to prevent CSRF attacks.

QR JavaScript:

- **const scanner = new Html5QrcodeScanner('reader', { ... });**: This creates a new instance of the **Html5QrcodeScanner** class and assigns it to the variable **scanner**. The constructor takes two parameters: the first is the ID of the DOM element where the scanner will be rendered (**'reader'** in this case), and the second is an object that specifies the dimensions of the scanning box and the frames per second to attempt a scan.
- **scanner.render(success, error);**: This method call starts the scanner and takes two callback functions as parameters: **success** and **error**. The **success** function is called when a valid QR code is detected, while the **error** function is called if there is an error in the scanning process.
- **const Bins = ['qjdkiivbbunmue625ljyjy04w941jy', ...];**: This creates an array of valid QR code strings and assigns it to the variable **Bins**.
- **function success(result) { ... }**: This is the callback function that is called when a valid QR code is detected. It takes the result of the scan as a parameter (**result**).
- **for (i = 0; i < Bins.length; i++) { ... }**: This is a **for** loop that iterates over the **Bins** array to check if the **result** matches any of the valid QR code strings.
- **if (result == Bins[i]) { ... }**: This **if** statement checks if the **result** matches the current QR code string in the **Bins** array. If it does, it sets the inner HTML of an element with the ID **result** to a block of CSS styles that create a button.
- **else if (i == 8) { ... }**: If the **result** does not match any of the valid QR code strings after the loop completes, this **else if** statement sets the inner HTML of the element with the ID **result** to a message indicating that the QR code is invalid and provides a button to try again.
- **scanner.clear();**: This method call clears the scanning instance.
- **document.getElementById('reader').remove();**: This removes the **reader** element from the DOM since it is no longer needed.

Forms JavaScript:

showTab(currentTab); Display the current tab This function displays the current tab on the screen. It takes the currentTab variable as an argument.
function showTab(n): This function displays a specific tab on the screen. It takes an integer n as an argument, which represents the index of the tab to be displayed.
function nextPrev(n): This function handles switching between tabs when the next or previous button is clicked. It takes an integer n as an argument, which is either 1 for next or -1 for previous.
function validateForm(): This function checks the value and validity of the form fields. It returns true if the current tab fields are valid, and false if they are not.
function fixStepIndicator(n): This function removes the active class from all steps and adds it to the current step. It takes an integer n as an argument, which represents the index of the current tab.
function emailCheck(): This function checks if the email is a valid email for the University of Exeter by checking if the text after the '@' is 'exeter.ac.uk'. It returns true if the email is valid, and false if it is not.
toggleWrongEmail(num): This function displays a message when there is an invalid email. It takes an integer num as an argument, which is either 1 to show the wrong email message, or anything else to hide it.