Hi, guys! That's Miriam from the Front-End team. After carefully looking at the wireframes provided by the Product team, we have been thinking about the nature of the communication between the server (which you have decided to implement via `Python` files) and the client (which is essentially the `HTML` templates we build.) Since we need to either read some information from a database, write into the database, modify existing information, or making some computations, and since these operations are done through the server side and the Back-End team, we came up with the following few tasks that your `Python` files should potentially accomplish:

1. (Displays `Posts` of some `Port`) Given a `Port` id: (1) query the database with this id to receive an `JSON` array of `Posts` ids written via the `port`, and (2) query the database again with those `Posts` ids to receive a `JSON` array of the following info per each `Post`: (1) name of `Post` (2) content (3) image (4) day (5) month (6) year (7) hour (8) minute (9) name of the `User` who wrote it (10) image of the `User` who wrote it, (11) number of votes, and (12) number of `Comments`. The table in the database for the `Post` object has the id of the `User` who wrote the `post`, and his or her name and image in (9) and (10) above can be retrieved by querying the database in the table of `User` objects. While the query for the `Posts` array is done, make sure to receive a **sorted** array based on the date and time of writing the `Posts` (the newest `Posts` are displayed first, so the most recent `Post` will have the index 0 in the `JSON` that you obtain from them.)

2. Do the same as above, except that you obtain a `JSON` array that is sorted based on the highest number of votes.

3. (Display `Posts` Relevant to `User`) Given a `User` id, (1) query the database to obtain the ids of all the `Ports` to which the `User` is subscribed, and (2) per each `Port`, repeat the steps in 1 such that the array of all the `Posts` from all these `Ports` will be sorted based on date and time.

4. Do the same as above, but make sure the `Posts` are sorted based on the highest number of votes.

5. (Display *all* Info about a `Post`) Given a `Post` id, query the database for: (1) name of `Post` (2) content (3) image (4) day (5) month (6) year (7) hour (8) minute (9) name of the `User` who wrote it (10) image of the `User` who wrote it, (11) number of votes, and (12) number of `Comments`. This very `JSON` bulb that you receive should also contain inner (nested) `JSON` bulbs that will represent information about `Comments` to this `Post`. The `JSON` bulb for a `Comment` should include: (1) content (text of `Comment`), (2) day (3) month (4) year (5) hour (6) minute, (7) number of votes to the `Comment`, (8) username of `User` who wrote the `Comment`, (9) the image of the `User`, and (10) inner `JSON` bulbs with `Comments` to the `Comment`. As you probably notice while reading this paragraph, there is a theoretical option for infinitely many nested `JSON` bulbs, which is practically a disaster. Hence, I will ask the Product team if they will allow us to limit the nesting of `Comments` to some level. For example, Facebook allows only two levels of `Comments` (so we have the original `Post`, `Comments` to the `Post`, and `Comments` to those `Comments`, and that's it.) If they will allow this, there will be at maximum two nesting levels in the outer `JSON` code.

6. (`User` Registration) Request information from a form on our template that will contain: (1) `User's` first name (2) `User's` last name (3) username (= login name) (4) email address (5) password, and (6) image. Send this info into the database to create a new `User` instance.

7. (`User` Login) Request information from a form on our template that will contain: (1) username, and (2) password. Authenticate this info with the database, and if it exists and accurate, redirect website to another template that will display (1) `User` image (2) username (3) `Posts` from all `Ports` to which the `User` is subscribed (do the steps in 3), all of which the `Python` file will query from the database's `User` table.

8. (Displaying Randomly Chosen `Ads`) (1) Query the database to find the total number of the `Ad` instances that exist in the `Ad` table. (2) Assuming you called that number n, randomly choose a number from zero to $n - 1$. (3) Assuming your random number is r, retrieve the `JSON` array of information about the rth `Ad` in the database. The information you retrieve shall include: (1) image (or image URL, however you get it) (2) URL of the website of the `Ad`, and (3) caption of the `Ad`.

   If you would like, you could implement a different way of randomly choosing an `Ad` from the database – all as you'd like.

Besides those tasks mentioned above, there will probably be several more that will be concerned with modifying `User` info, unsubscribing from `Ports`, etc., that we will discuss together in class. This general list just gives you an overview of what the `Python` code that your team composes will do. In case anything here is unclear, or you notice that something is missing from here, please feel free to tell us so that we clarify whatever is needed and correct the file.

The Front-End team will upload templates and static files (`JavaScript`, `CSS` and images) one by one as soon as they are ready. Please inform us about the choices of the variable names that you are going to pass into our templates, so that we write them into the templates (the names of the variables that you send and those appearing in our templates should match exactly.) Also, since the key names of the various features of an object are chosen by the Database Team, please inform us about them as well. In our templates, for instance, if we would like to access the content of the most recent `Post`, and the name of the `JSON` variable that you pass to us is called `posts`, we will write the following into our template: {{ `posts[0]['content']` }} to be able to display this. Actually, because the plug-in language that we will use in the template (`Jinja2`) allows us to create loops inside HTML files, it will look more like: {{ `posts[`i`]['content']` }} for every `Post` in the array (we indicate the name of the loop indexing variable i ourselves, so you do not need to bother with this.)

We are so thankful to you for this quality cooperation! Good luck with the project, and have fun!