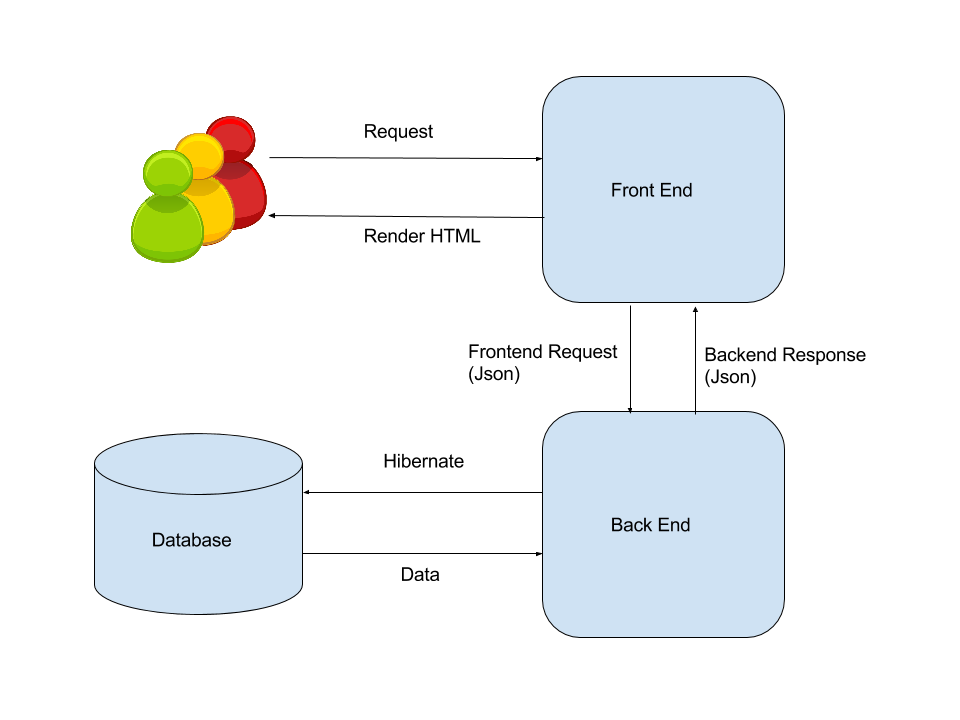
**18-655 Project 3 Team 7 & 8**

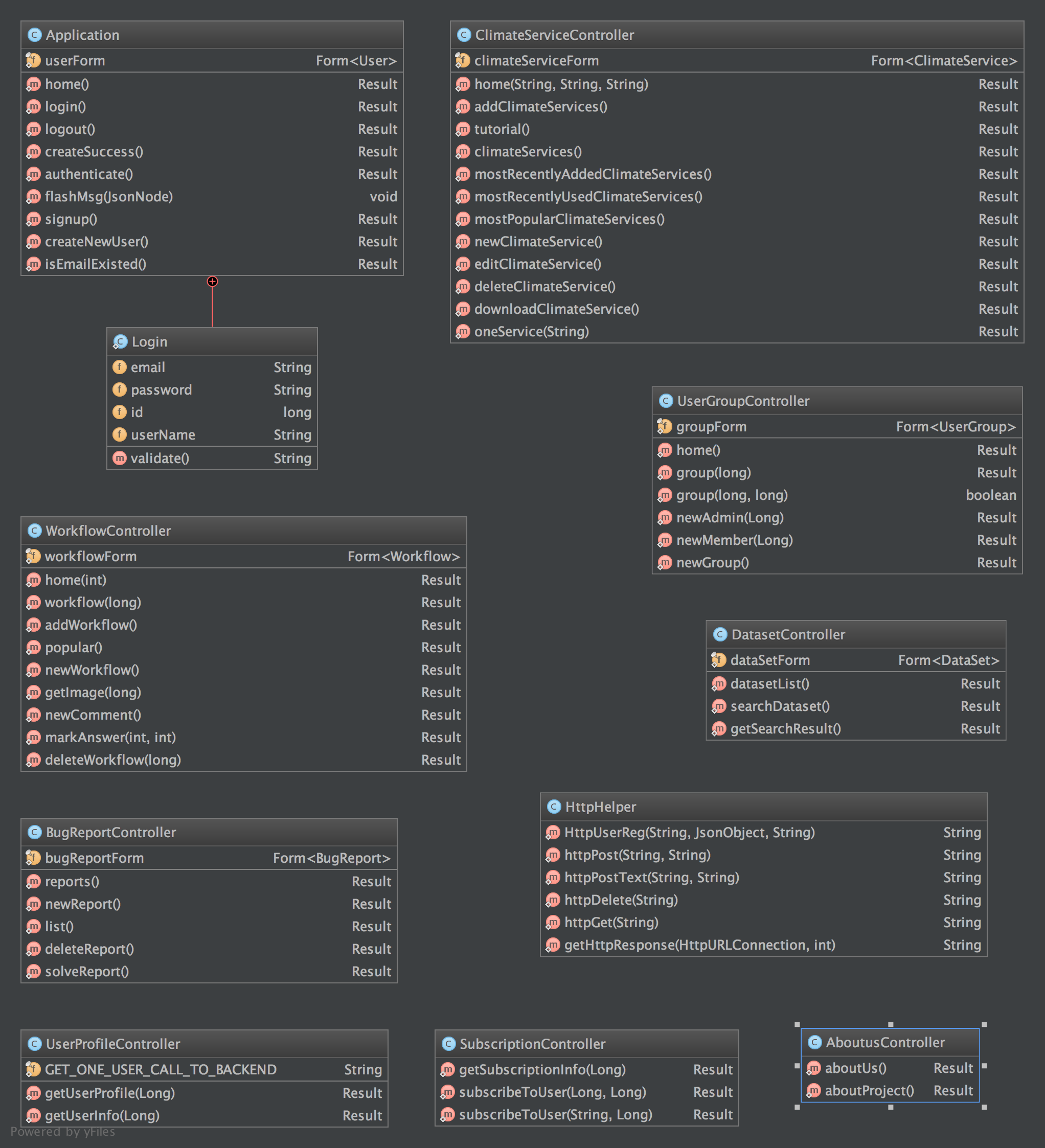
**Design:**



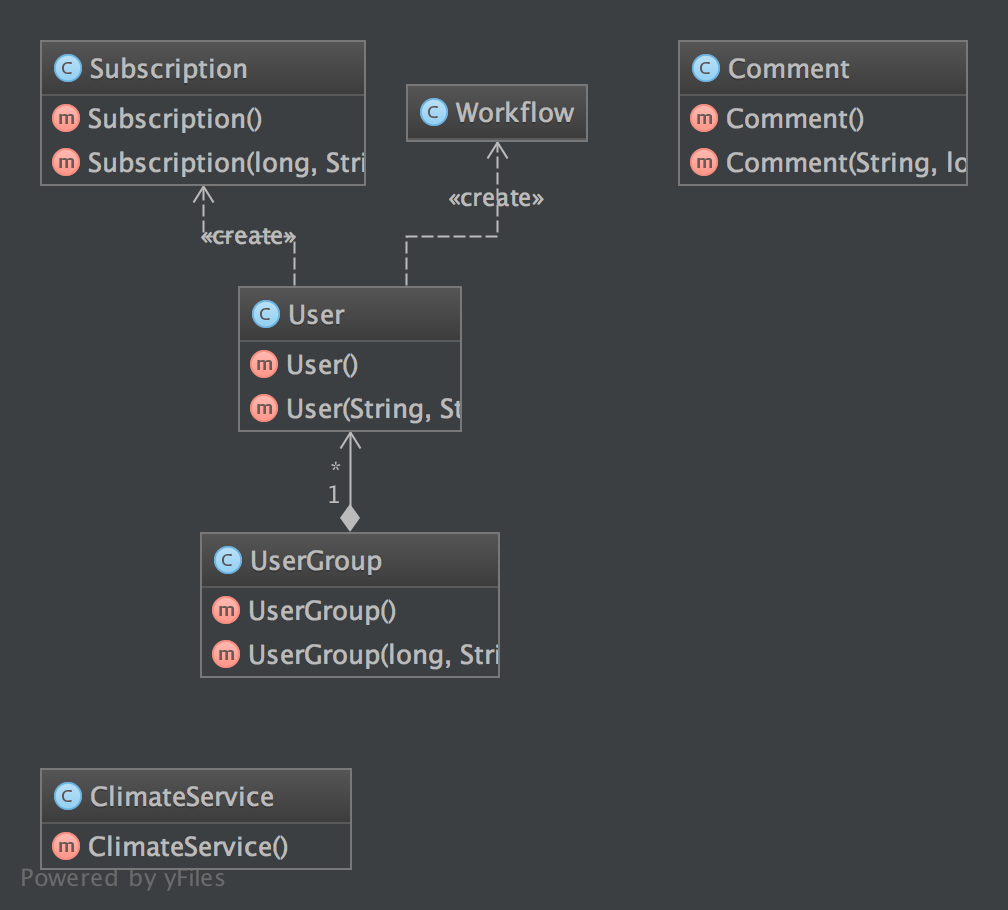
When client send HTTP request to the front end, front end controller use the method in the corresponding model class and call backend API to send request to backend. Backend controller look into the database and populate or fetch the data into or from database. Controller then read data into model, serialized the model into json format and send to front end as response. The front end controller receive the reply and then render the HTML to the client.

**4.1 Class Diagram**

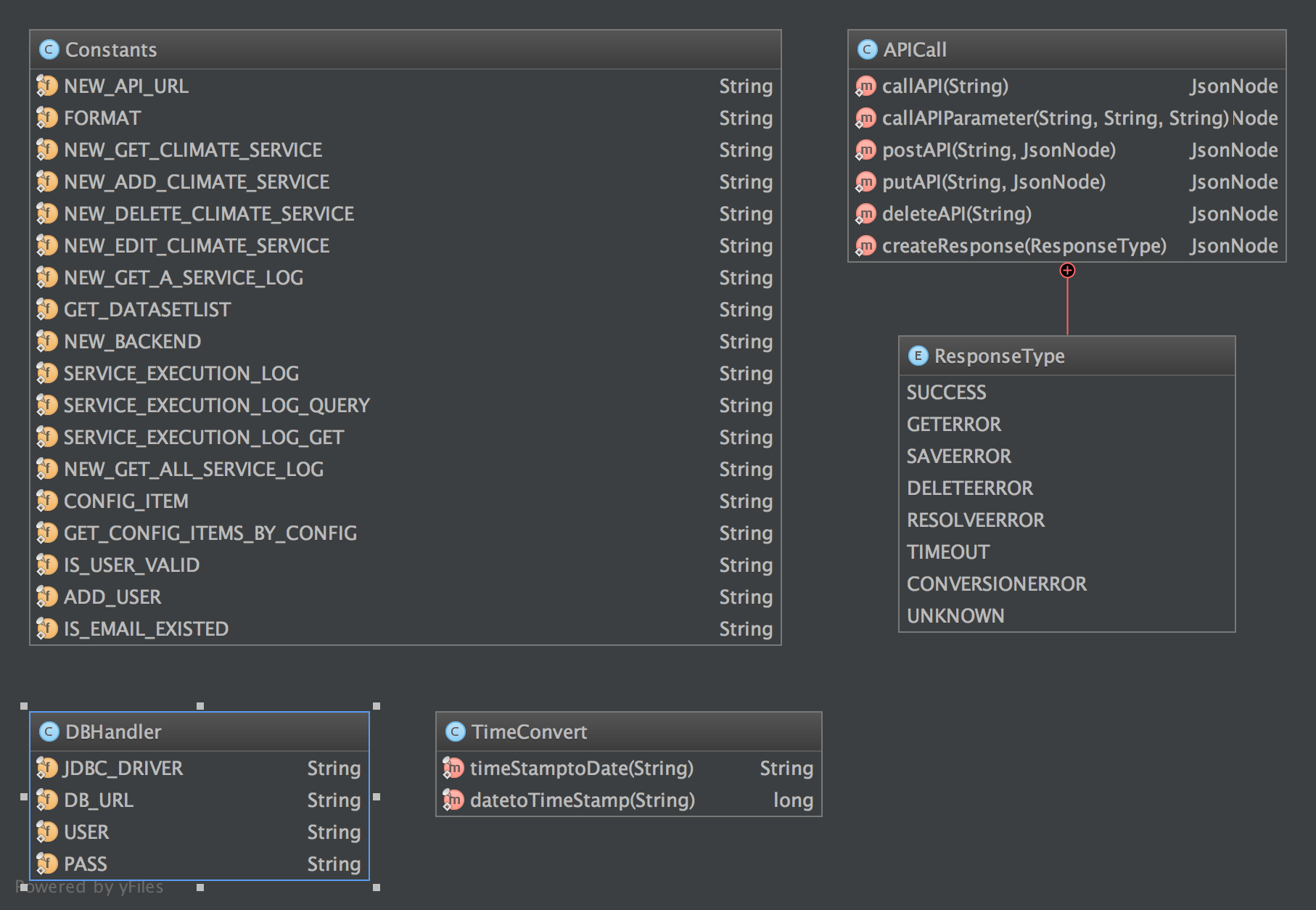
Following is the class diagram of our controllers package, on top of the code base provided, we have implemented several classes including: WorkflowController, UserProfileController, UserGroupController, SubscriptionController. In addition, we added methods of login and authenticate in ApplicationController. These controllers are the Controller part in the MVC framework which in charge of manipulate models and update views.



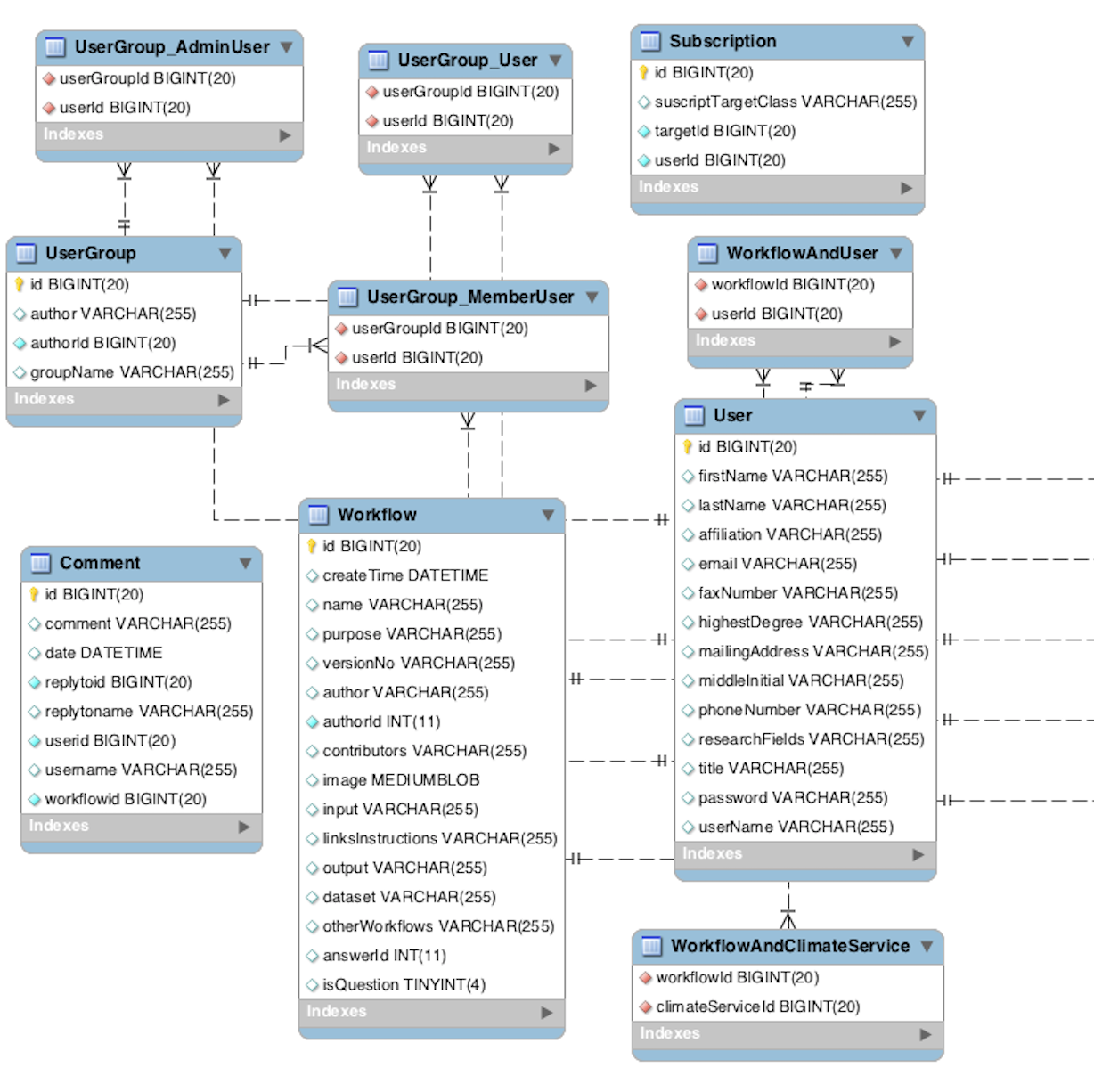
The figure below demonstrated the structure in our Metadata package under Models. They store contents and data that will be used by this project and were also used when accessing the back end.



Following is the class diagram of the util Package, the APICall class was used when making API calls to the back end, and the Constant class were used to store frequent used constants.



**4.2 Database Design**



Above is our database design UML. This UML only shows the new tables that we created for the project.

To implement the functions for workflows, we created the Workflow table to store the properties of the workflow, such as name, version number, contributors, image, etc. The other information of the workflow (Which climate services is included, which user is related to the workflow) are stored in separate tables (WorkflowAndClimateService, WorkflowAndUser), connecting to the workflow table using foreign key.

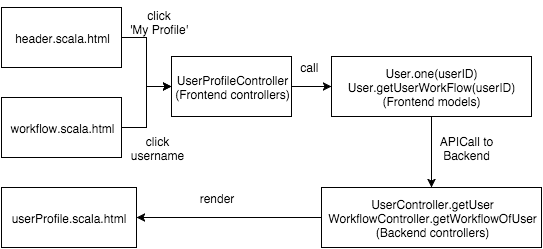
To implement the functions for UserGroup, we created the UserGroup table to store the properties of the group, such as author, description, etc. The group-user relationship is stored in 3 tables, connecting to the usergroup table using foreign key. The UserGroup\_AdminUser table stores the administrator member list information of a group; The UserGroup\_MemberUser stores the ordinary member list information of a group; The UserGroup\_User stores the creator information of a group.

As for the Subscription functions, we set a column named subscriptionTargetClass to store which kind of information is being subscripted(Workflow, UserGroup, User, etc.). It also stores the subscriber’s id and user id.

For the comment functions, we create the table comment. It stores the comment information, the id of the workflow that the comment belongs to, and also the replyToID information, so that it can be displayed in the proper workflow page, with the proper reply relationship.

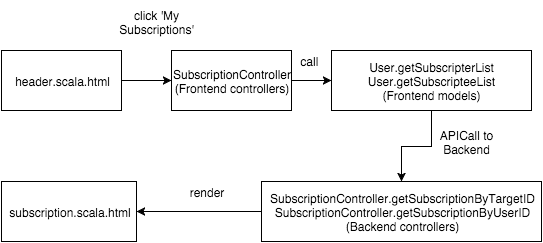
1. **System implementation**

**User Profile Implementation**

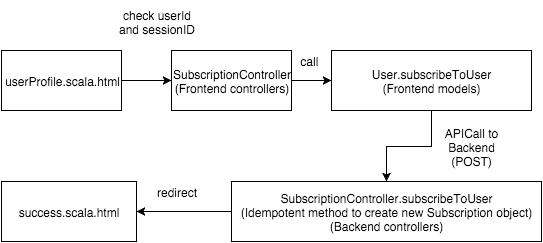


The user profile can be accessed from multiple places: the navigation bar, workflow page, and subscription page. The frontend server will fetch the User object from the backend server and render the content with userProfile.scala.html. Currently it does not support profile edit and deletion, so the users are only allowed to register and view profiles.

**Subscription Implementation**

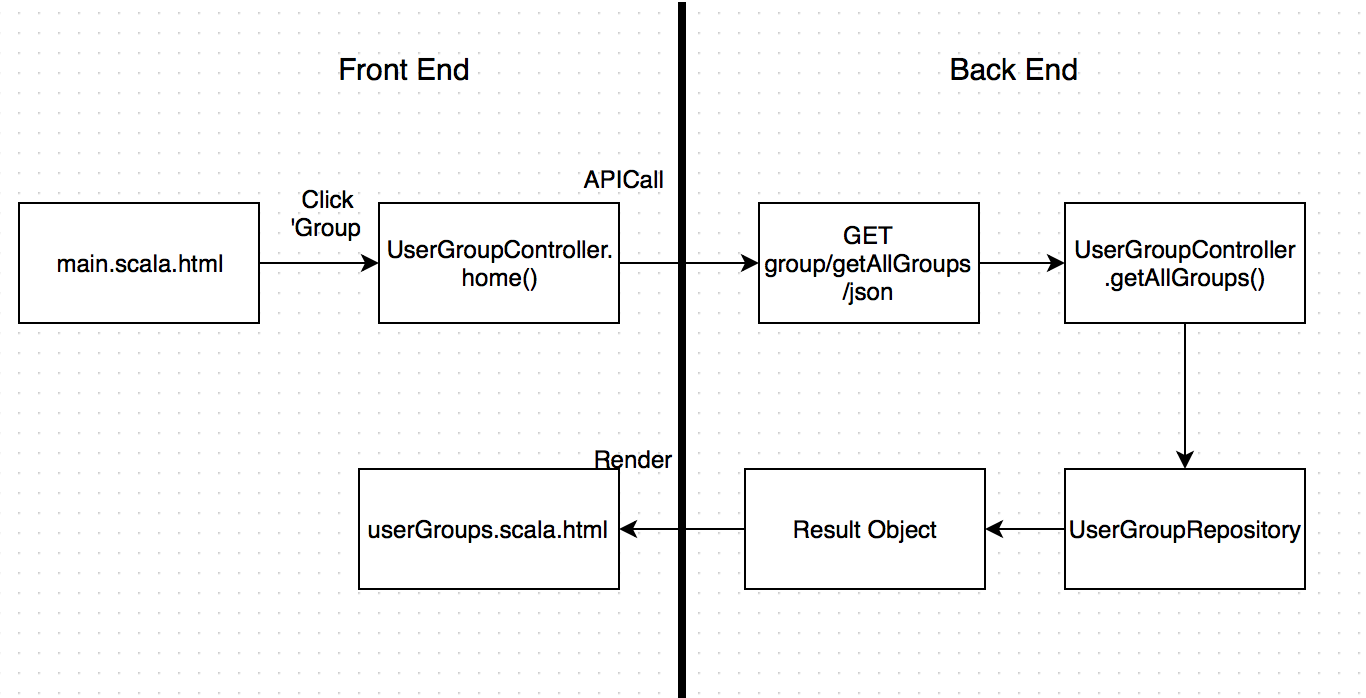


The above diagram shows the flow to get the list of a user’s subscription. Listing a specific collection’s members is a common operation in RESTful design, so it will not be explained in detail. Then the list of subscription will be rendered via subscription.scala.html.

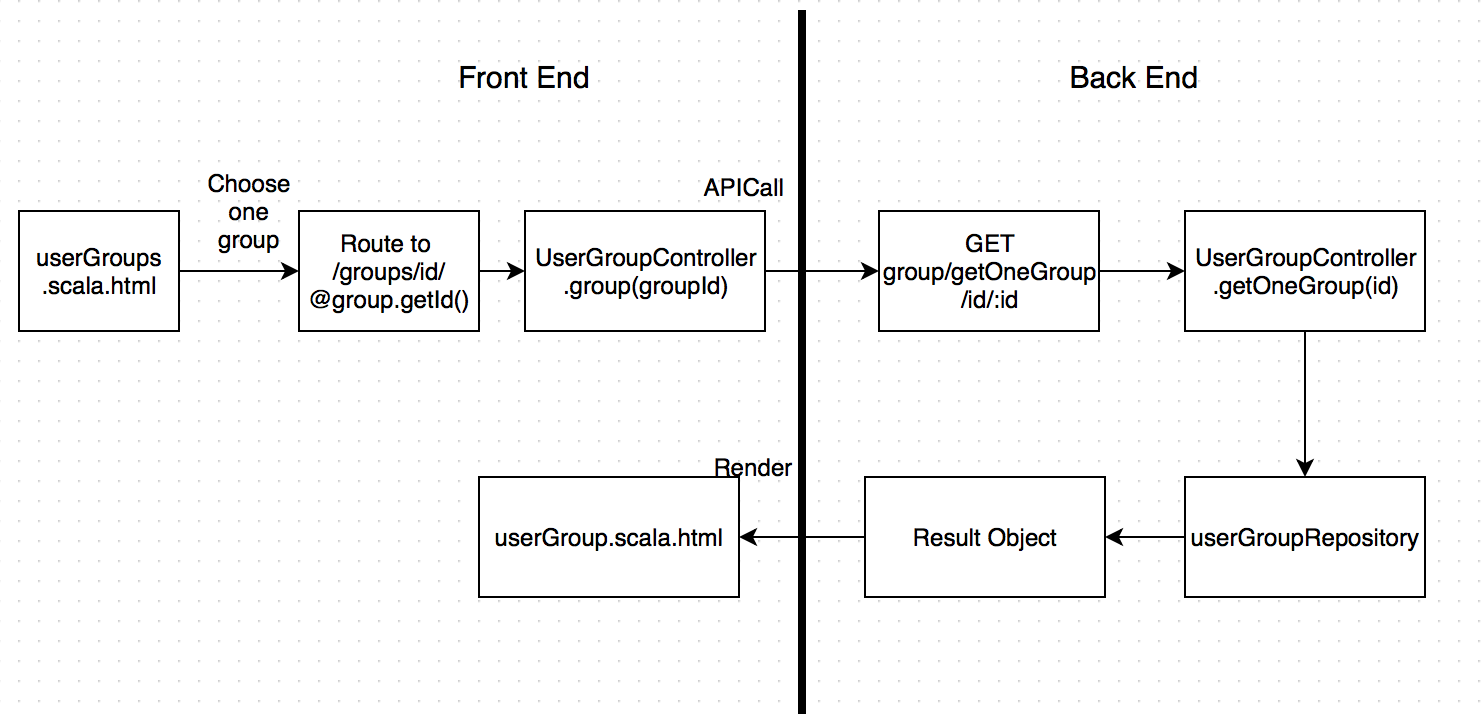


To subscribe a user, the follower has to click the subscription button on the followee’s profile page, and the request will be processed by the SubscriptionController. A recorded will be added into the subscription table.

**UserGroup Implementation**



User group can be accessed in main page by clicking on the “Group” button in the header. The page will be redirected to the the front end controller, which calls home() method and makes an API call to back end router on the URL “group/getAllGroups/json”. The router will calls the back end controller on getAllGroups() method, which makes a query to the database and get the group list as a Result object. Then the object can be rendered to the userGroup.scala.html and serves as a view on the browser.



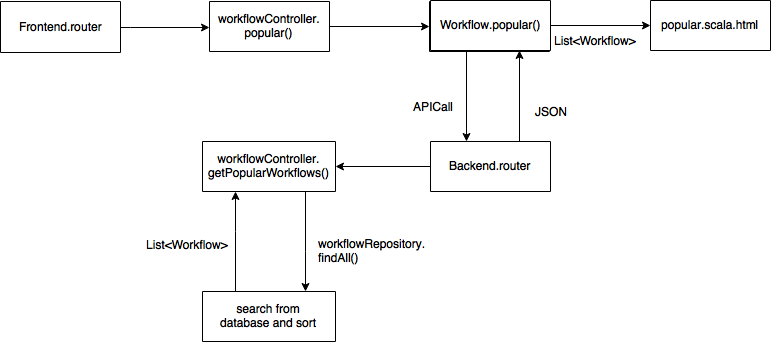
Within userGroup page, a list of available groups is rendered on the browser. User can select a group on the page to view the group details, or create his/her own group on the page.

When clicking on one group, the page will call the front end route on “/groups/id/:id”, which calls the User Group front end controller’s group() method. It makes an API call to the backend router to perform a GET operation on the URL /group/getOneGroup/id/:id, which calls the backend controller’s getOneGroup() method. This makes a query to the backend database, and get the group information as a Result object, which is then rendered to the userGroup.scala.html.

At the end of userGroup page, there is an area for creating a user group. A user can type in the new group name and click “create group” button. Then the page would refresh itself and a new group will be created. The creator is by default the administrator of the group.

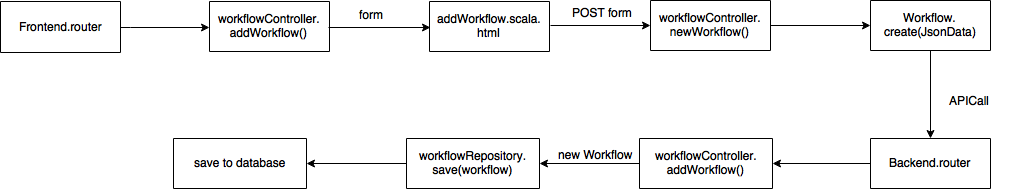
Within a group page, the basic group information is provided. When the creator is visiting this page, he/she will be shown the additional content to add new administrator or add new ordinary member. When a member logs in and visits this page, he/she could only visit the basic information. When a user is not logged in or he/she is not part of the group, he/she will be redirect to the error page, stating that he/she does not have the proper permission to view this information.

**Popularity Implementation**



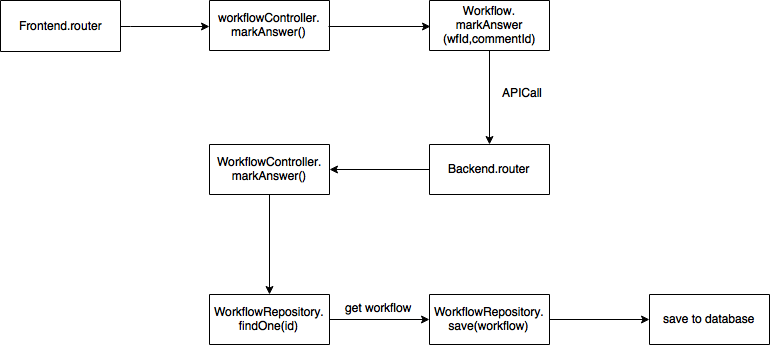
We provide a page with which the user can browse the top popular workflows. The popularity is calculated simply based on the number of views. The more views the workflow has, the more popular the workflow is. When accessing popularity browsing page, everything is identical as browsing the workflows except that the workflows are sorted based on their popularity.

**Post Workflow**



When client want to post a new workflow, he access the /workflow/new/workflow, filled out the form and post to the same url. When front end receive the post request, the newWorkflow() method is called and form data are synthesized into a model, then serialized into a json format string and sent to backend/workflow/newWorkflow API. Backend controller addWorkflow() is called, a new workflow model is created and use the Workflow model class’s save method to populate the data into database.

**MarkAnswer Implementation**



Our implementation is different from the requirement because we misunderstand its meaning. In our implementation, a user can mark a workflow as a question during the creation. The author can pick up one comment associated with the workflow as the answer. The operation behind this functionality is simply to set the comment id in a workflow.