**Sisyphus**

Jesse Bernoudy, Bob McHenry, Mario Rodriguez, Chris Wilson, Grant Zukowski

Sisyphus is a mobile application to help organize projects and to collaborate with others in the workplace, or to simply help you get more organized at home with your daily tasks. Its basic structure will be a landing page that shows your root-level projects or tasks, with the ability to add new projects or tasks, edit/delete them, or nest other items inside projects. There are several different fragments that correlate with different layouts, and a main activity page that the user will see when they open the application. Each page/component will be explained in detail below.

The application is built with an MVC architecture, which separates the view from the controller. The view communicates back and forth with the model, as does the controller. The following classes are in our view:

**StartPage.java, TaskViewer.java, TaskItemAdapter.java, TaskEditFragment.java, TaskDetailFragment.java, TaskListFragment.java, ProjectEditFragment.java, and ProjectDetailFragment.java**. All of the layouts in the res/layouts directory are also considered part of the view.

The following classes make up the model, which is responsible for object creation and communication with the controller:

**ToDoItem.java, Task.java, Project.java, User.java**

The following classes make up the controller:

**JSONController.java**

The application stores its data in a tree-like model, with a “user” object being the main root. This information is stored in the Environment.DIRECTORY\_DOCUMENTS storage medium, in a .json file. The Controller, when called (which has a bunch of static methods), will pull that .json file from storage, and then call on the model to build a Root user object, consisting of userName, userID, and rootProject fields. Then, recursively, the controller starts calling the model to build any ToDoItems that have been stored in the .json file, and adding them to each parent object – after which these root projects will be shown on the landing screen. Projects or tasks can be added to the root User object’s “rootProject” field in an array, Projects and Tasks can be added to parent level projects, but nothing can be added to Task objects – they are considered leaf objects. Once all of these objects have been built, the user will see their root item screen and can start interacting with the UI. Each component is detailed below.

**View**

**StartPage.java**

Main, default launcher activity. Creates an ArrayList to store all ToDoItems for use in the initial ListView and for UI interaction. This activity content is activity\_start\_page.xml in the res/layouts directory. During its onCreate(), it sets the content view, and adds an action bar. The action bar can take the user home, or go “back” one level by popping the last fragment transition from the back stack. The activity will query the model to get all tasks, the model will query the controller to get the users .json file and start building objects, and the model will finally return the user’s root Task ArrayList. Once the initial list view is loaded and the user can see their root items via the showTaskViewer() fragment, the onTaskItemSelected() method responds to clicks by checking if that task is a Task or a Project object, and will call either showTaskDetails() or showTaskViewer() to load the correlated fragments. The back button pops the last fragment transition to display its fragment. All fragments are shown in a fragment container.

**TaskListFragment.java**

This class is a static class that will return an instance of itself. Its onCreateView() method inflates the fragment\_task\_list.xml layout, creates a ListView and then binds an adapter to it – the adapter contains the tasks ArrayList from the start page. It also creates a button to view details.

**TaskItemAdapter.java**

Its job is to check the ArrayList for the exact position that was clicked, gets the data for that position in the array, checks to see if the view is being reused, and if not, inflates the view for that data with either list\_item\_project.xml or list\_item\_task.xml depending on what type of object the data was. It then sets the objects name and description data on the TextViews that are in the xml file that is being used. After all of this it returns this entire component as a View to the class that called it.

**TaskEditFragment.java, TaskDetailFragment.java, ProjectEditFragment.java, ProjectDetailFragment.java**

These are simply fragment classes that inflate their corresponding layout views (fragment\_task\_edit\_view.xml, fragment\_task\_detail\_view.xml, fragment\_project\_edit\_view.xml, and fragment\_project\_detail\_view.xml). These are loaded into a fragment container for the viewer to see, instead of creating new activities left and right.

**Model**

**User.java**

This is a root level object. It contains simply three fields: userName, userId, and rootProject. It has several contructors including one for manual instantiation and assignment of values to fields, as well as one that takes JSONObject information to build the user and start building all of its root projects.

**ToDoItem.java**

Abstract class that is a superclass to both Project.java and Task.java. Contains all essential fields and getters/setters for all fields: itemID, itemName, itemDescription, createdBy, category, dateAdded, dueDate, priority. These are fields that are used when the user is adding/editing a Project or a Task.

**Project.java**

Extends ToDoItems.java abstract class. Contains an additional field called childItems (ArrayList), which contains all child items in this project instance. The class has all the normal getters/setters, and has several constructors, one which takes a JSONObject to build this project and all its children, and one that takes manual inputs of all field values. This class also contains a two-dimensional array for keeping track of completion of items and progress of total items.

**Task.java**

Extends ToDoItems.java abstract class. Contains additional fields assignedTo, and isComplete. This class has all the normal getters/setters, and several constructors, one which takes a JSONObject to build this task, and one that takes manual intputs of all field values. This class also has a few extra methods to check if it is complete, and set complete.

**Controller**

**JSONController.java**

This is the controller, responsible for communication with the model. Its job is to pull .json information from the phone’s storage environment, and to pass JSONObjects/data to the model for building User/Project/Task objects. This class is private and contains only static methods to be called on-the-fly. There is no instantiation of this class. It can get and return JSON data to the model when requested. It also uses an InputStreamReader to grab the stored .json file and parse the data stored in it, or can write the the file in the same format.

**Screen Layouts**

The following screenshots will detail each screen shown to the user:

|  |  |
| --- | --- |
| Launch Page: | ListView: |
| DetailView: | EditView: |