Project 3 - Shirk: A Forgiving Task Manager - Phase 3
6.170 Software Studio | FA14
Andre Aboulian, Tricia Divita, Lily Seropian

Motivation

Shirk is a to-do list web application built for procrastinators, designed to help individuals stay organized and productive. It aims to replace the paper to-do list by providing a platform for users to record, view, and complete tasks. It supports grouping, scheduling, prioritizing and shirking (neglecting) tasks.

Purposes

1. **Help users remember what they need to do.** Procrastinators tend to let things pile up, and having a list of everything that needs to get done will allow procrastinators to put off tasks without worrying about forgetting to do them. Being able to mark things as done is necessary for any to-do list.
2. **Help users remember when they need to do things.** *Shirk* makes it easy for the user to keep track of what has to be done when by having a deadline associated with each task. This will not make sense for all tasks, so the deadline is optional on a per-task basis.
3. **Let users procrastinate.** Shirking is a concept unique to our to-do list: when a deadline comes up, if the user has not completed the task, it is considered shirked. *Shirk* keeps track of these tasks and persistently reminds the user that they need to be done. In this way, procrastinators can make sure that they'll still get everything done, even if it's late.
4. **Help users get organized.** *Shirk* facilitates creation of multiple lists of related tasks (errands to run, chores to do around the house, etc.). This helps separate tasks that have nothing to do with each other so that the user is not overwhelmed by all of their tasks at once. *Shirk* also allows users to selectively display tasks using predefined filters, such as tasks whose deadline is today.
5. **Help users focus on important tasks.** *Shirk* lets users prioritize their tasks and easily view the most important ones. This will help procrastinators figure out exactly where they should focus their time when they finally get down to work.

The first three purposes constitute a Minimum Viable Product: the first two are essential for a functioning to-do list, and the third is *Shirk*'s unique hook. The last two are nice features to have, but not absolutely necessary.

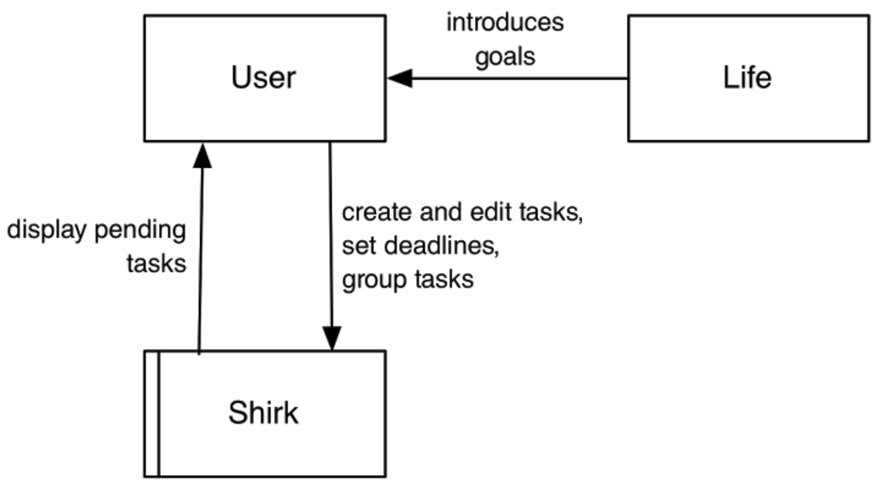
Existing solutions

There are a few to-do lists marketed for procrastinators already in existence. The ones with the most notable procrastinator-oriented features are Finish, Do It (Tomorrow), CARROT, and Three ToDo. Each app approaches the problem of designing a to-do list for procrastinators in a different way:

- *Finish* uses timeframes: tasks are either short-term, mid-term, or long-term, and automatically get moved from one category to another as the due date gets closer. However, *Finish* does not say how to handles tasks that are not completed by their due date.
- *Do It (Tomorrow)* has two lists: stuff to do today and stuff to do tomorrow, with the option to move tasks from today to tomorrow. This does not support long-term planning or a notion of due dates.
- *CARROT* has an "AI" that gets mad at the user when they don't complete tasks on time. This is a cute idea but may make the user feel bad about themselves.
- *Three ToDo* only shows 3 tasks at a time, which you can drop, defer, or mark as done. The app then infers a priority for the tasks based on what you do. This can interfere with long-term planning and let important tasks be forgotten if they are never reached.

Shirk combines the best of traditional to-do lists and the above ideas by providing all the long-term planning benefits of a traditional to-do list with the ability to procrastinate short-term without forgetting about a task or getting yelled at by an angry app.

Context



Concepts

Task: A specific chore or undertaking, constituting a single unit of work. Each task has a title and may have a more detailed description and a deadline. *Shirk* does not support recurring tasks.

List: A group containing individual tasks, which usually share a common purpose. Each task belongs to exactly one list, and is permanently assigned to that list at creation.

Deadline: The date by which the user aims to complete a specific task. Assigning a deadline is optional. There is no time of day associated with a deadline. Deadlines are used to determine whether tasks are overdue.

Shirking: The act of ignoring a task. The number of days an unfinished task has been shirked is defined as the number of days that have passed since its deadline.

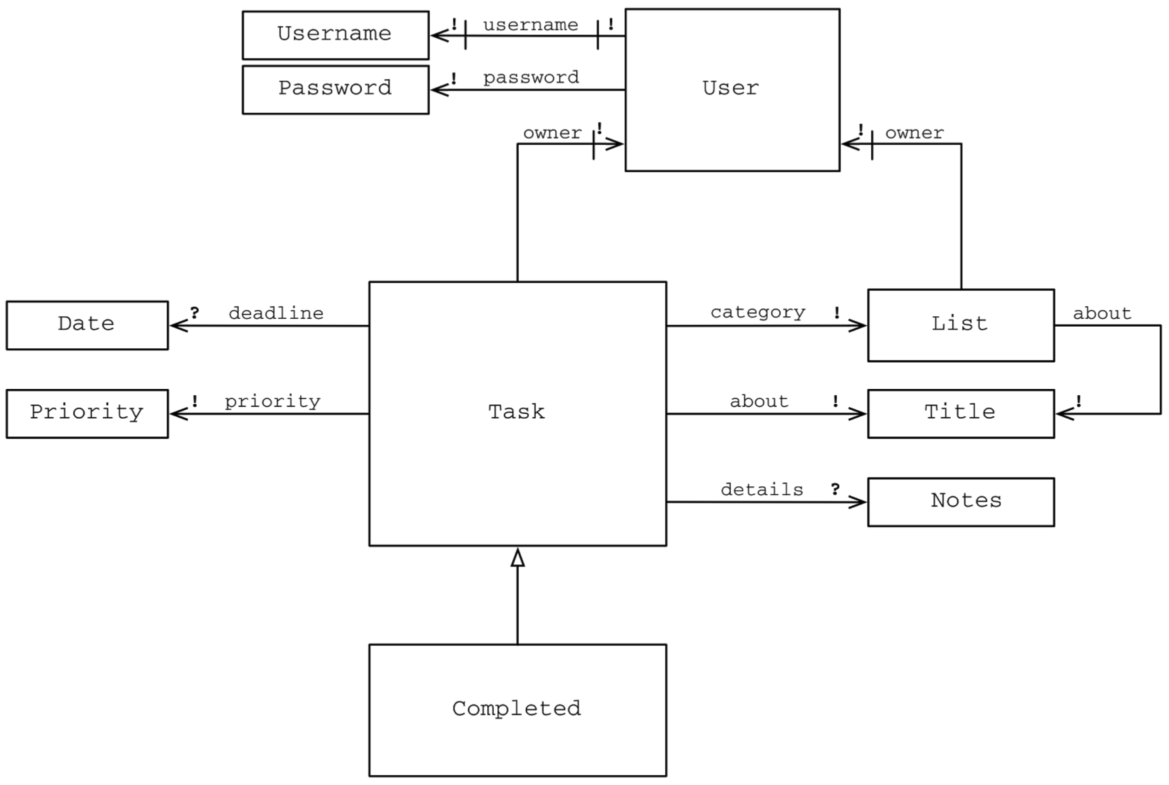
Completing: The act of marking a task as finished. This action does not affect how the task can be mutated; its list and deadline can still be changed, and its status can even be set back to incomplete.

Priority: The importance of a specific task. Each task can optionally be marked by the user as “high” or “low” in priority, and otherwise will default to “neutral”.

Filter: A built-in category defined by certain criteria, such as tasks that are high in priority. A listing of tasks that meet these criteria is generated as needed.

User: An individual (ostensibly a procrastinator) interacting with the system with a specific account. Each user has their own tasks associated with them, which they control completely. Users have no interaction with other users; there is no viewing of other users’ tasks, sharing of tasks, etc.

Data Model



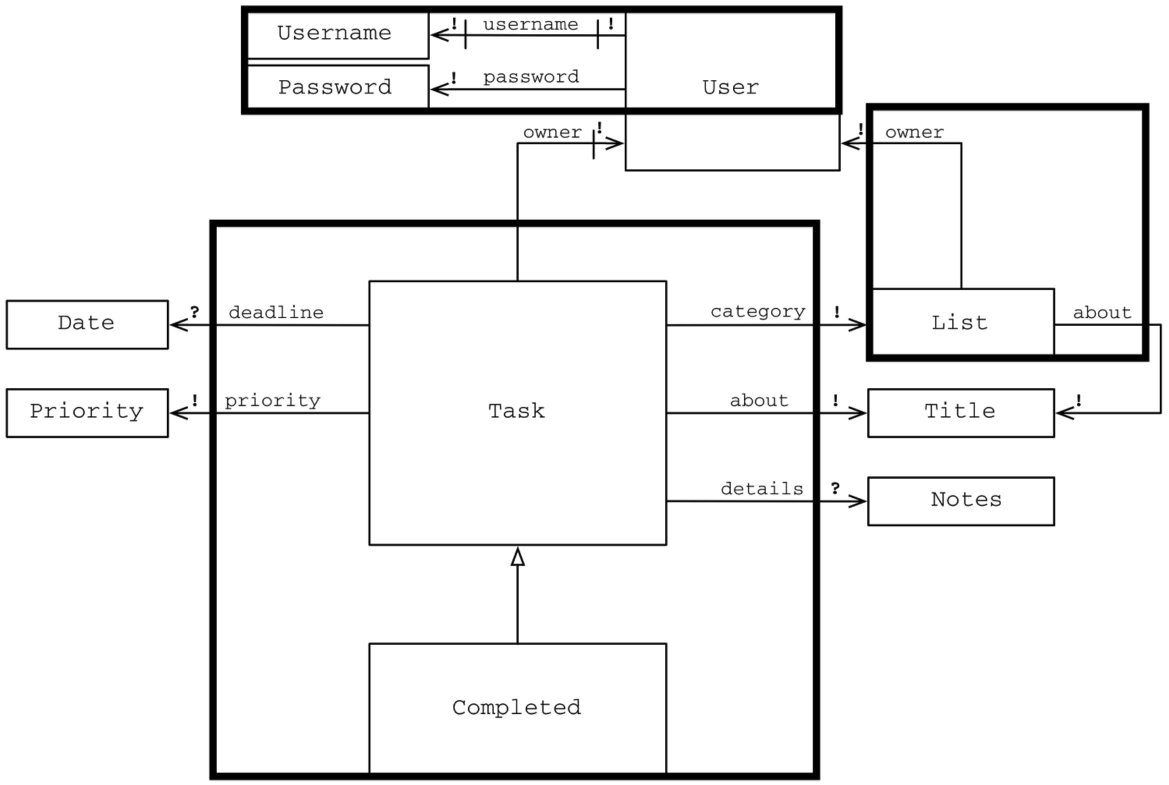
Additional Constraints: All *Tasks* referencing a *List* must have the same *owner* as the *List*.

Data Design

Shirk has three models: Users, Lists, and Tasks. Users are a separate model because both Tasks and Lists have an associated User, so it would not make sense to group Users with either Lists or Tasks but not the other. Lists and Tasks both need an associated User because users may only access their own Lists or Tasks, and there are routes that allow accessing both.

Lists are a separate model because it helps explicitly separate Tasks that are in different Lists (and each Task is in exactly one List). It allows easily allows filtering the Tasks in a given List.

Tasks have a list property indicating which List they belong to, rather than having Lists have a list of their Tasks. This approach scales much better when there are Lists with many Tasks. It would also make changing which list a task is in easier, although we decided not to allow that.



API Specification

Everything will be validated through passport.js. Therefore each user will only be able to access and modify tasks and lists that belong to them.

User-Specific Routes

POST /users

Description	Create a new user.

Request Body	`username: String`
	`password: String`

Response	`200 - User creation successful.`
	`400 - Username/password blank or username is already used.`

POST /login

Description	Attempt to login the user.

Request Body	`username: String`
	`password: String`

Response	`200 - Login successful.`
	`401 - Username/password not found.`

POST /logout

Description	Attempt to logout the user.
Response	`200 - Logout successful.`

Task-Specific Routes

For the following routes, consider Task to be an object as follows:

```
{
      _id: TaskID (String) - only in response
      title: String
      notes: String
      list: ListID (String)
      deadline: Date [optional] (YYYYMMDD)
      priority: Number [optional] (-1,0,+1)
      completed: Boolean
}
```

GET /tasks/:id

Description	Retrieve the specified Task.
Params	Path Params: `id` - TaskID identifier string
Response	`task: Task`

PUT /tasks/:id

Description	Modifies the specified Task.
Params	Path Params: `id` - TaskID identifier string
Request Body	`task: Task`
Response	`task: Task`

Error	`400 - Put unsuccessful.`
	`404 - Task not found.`

DELETE /tasks/:id

Description	Deletes the specified Task.
Params	Path Params: `id` - TaskID identifier string
Response	`taskID: String`

Error	`500 - Server error while deleting.`
	`401 - Current user is not owner of task.`
	`404 - Task not found.`

GET /tasks/

Description	Retrieves all tasks, optionally filtered by query parameters.
Params	Query Arguments: `limit` - max number of tasks to return (default: Infinite) `startDate` - deadlines after Date (default: Any) `endDate` - deadlines before Date (default: Any) `completed` - completion status (options: 0 or 1, default: Any) `priority` - task priority (options: -1 or 0 or 1, default: Any)

Response	`200/OK`
	`tasks: Task[]`

POST /tasks/

Description	Create a new Task.
Request Body	`task: Task`
Response	`task: Task`
Error	`400 - Task not created.`

<u>List-Specific Routes</u>

For the following routes, consider List to be an object as follows:

```
{
      _id: ListID (String) - only in request
      title: String
}
```

GET /lists/:id

Description	Retrieves the specified List and its associated Tasks.
Params	<u>Path Params</u>: `id` - ListID identifier string <u>Query Args</u>: `limit` - max number of tasks to return (default: Infinite) `startDate` - deadlines after Date (default: Any) `endDate` - deadlines before Date (default: Any) `completed` - completion status (options: 0 or 1, default: Any) `priority` - task priority (options: -1 or 0 or 1, default: Any)

Response	`list: List`
	`tasks: Task[]`

PUT /lists/:id

Description	Modifies the specified List.
Params	Path Params: `id` - ListID identifier string
Request Body	`list: List`
Response	`list: List`

Error	`400 - Put unsuccessful.`
	`404 - List not found.`

DELETE /lists/:id

Description	Deletes the specified List and its associated Tasks.
Params	Path Params: `id` - ListID identifier string
Response	`listID: String`

Error	`401 - Current user is not owner of list.`
	`404 - List not found.`

GET /lists/

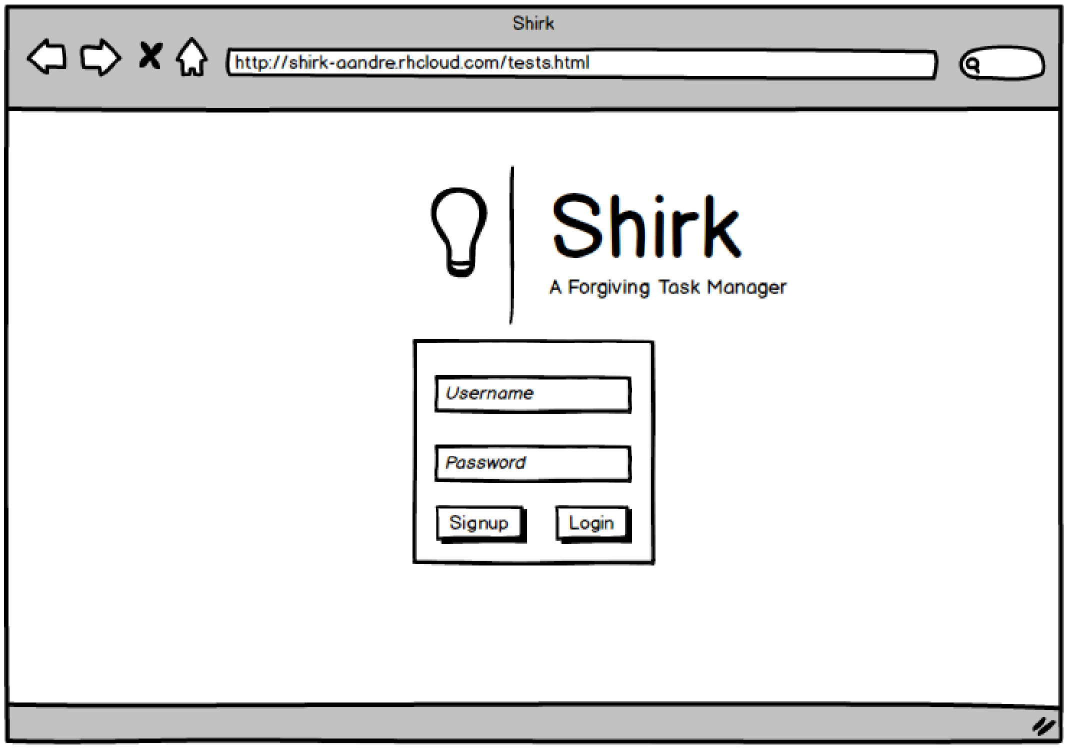
Description	Retrieve all Lists.
Response	`lists: List[]`

POST /lists/

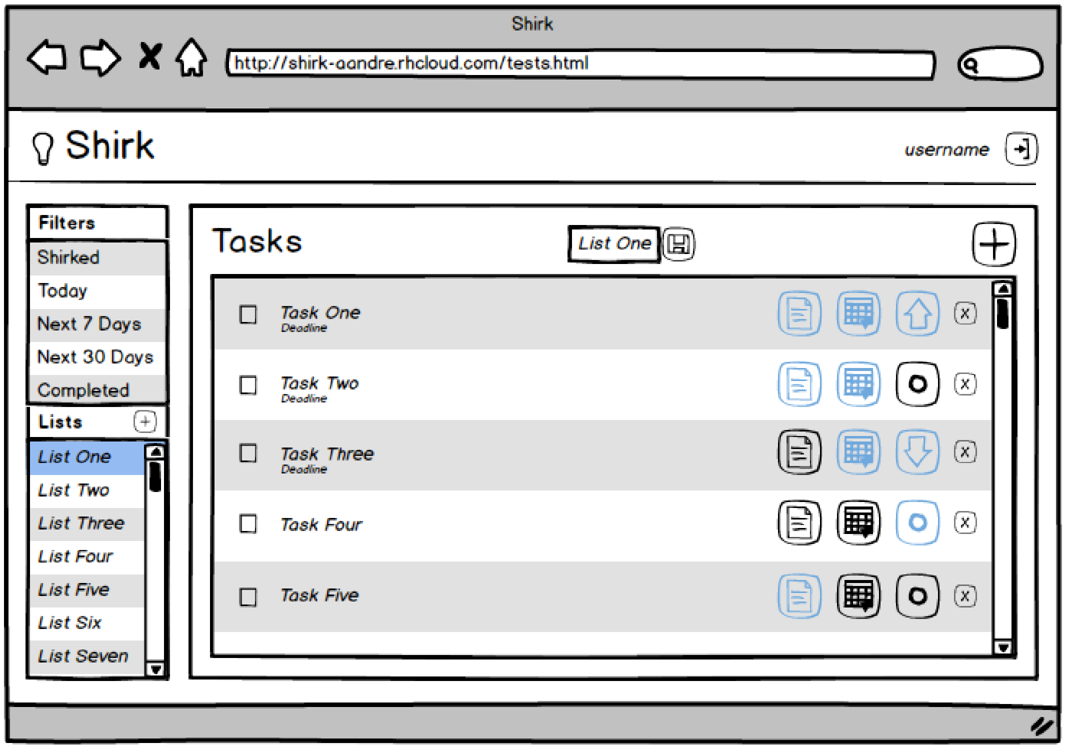
Description	Create a new List.
Request Body	`list: List`
Response	`list: List`
Error	`400 - List not created.`

UI Wireframes

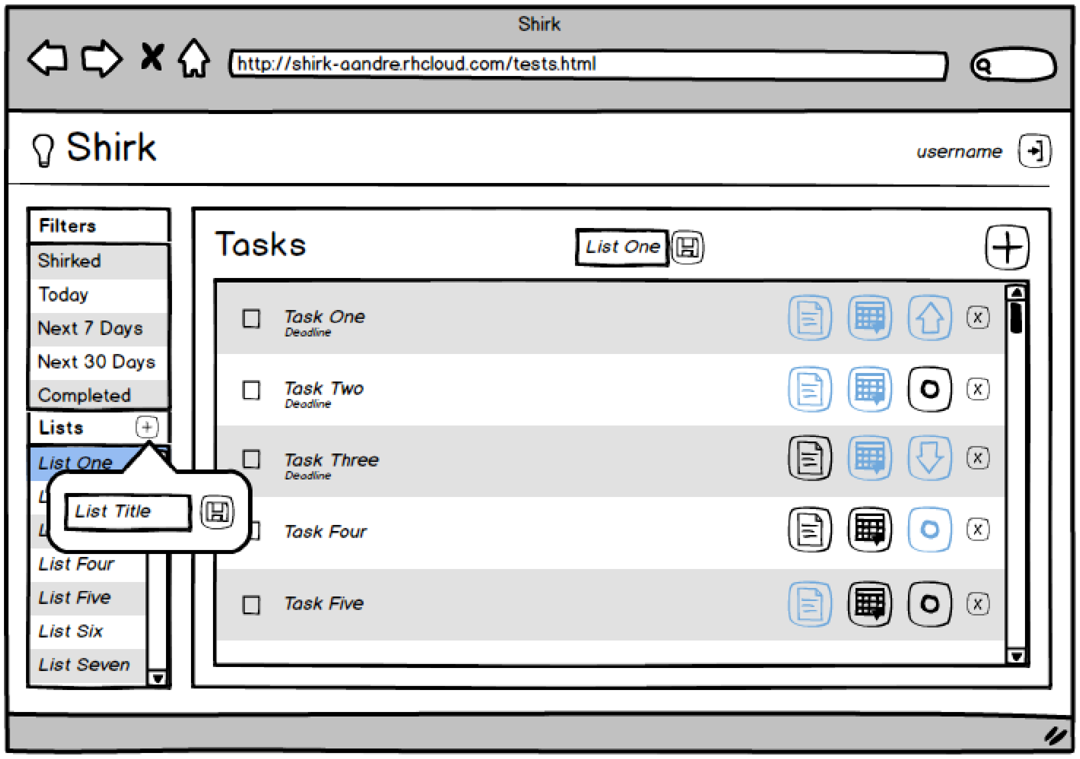
Login Page:



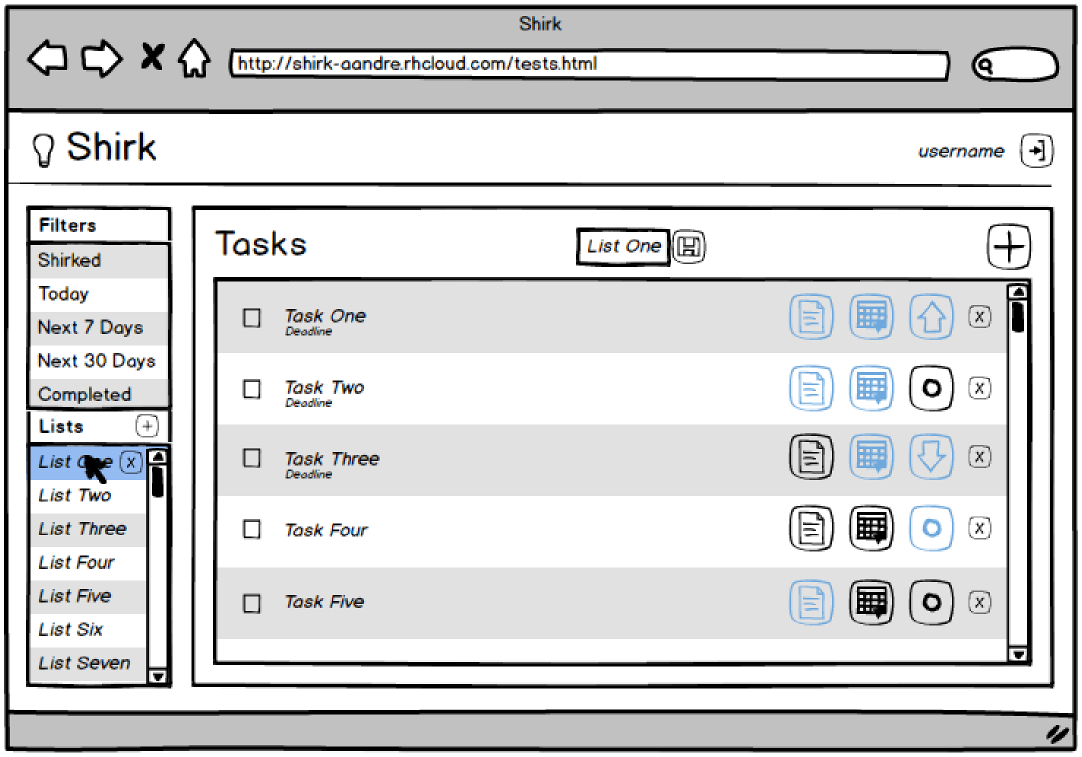
Main Page:



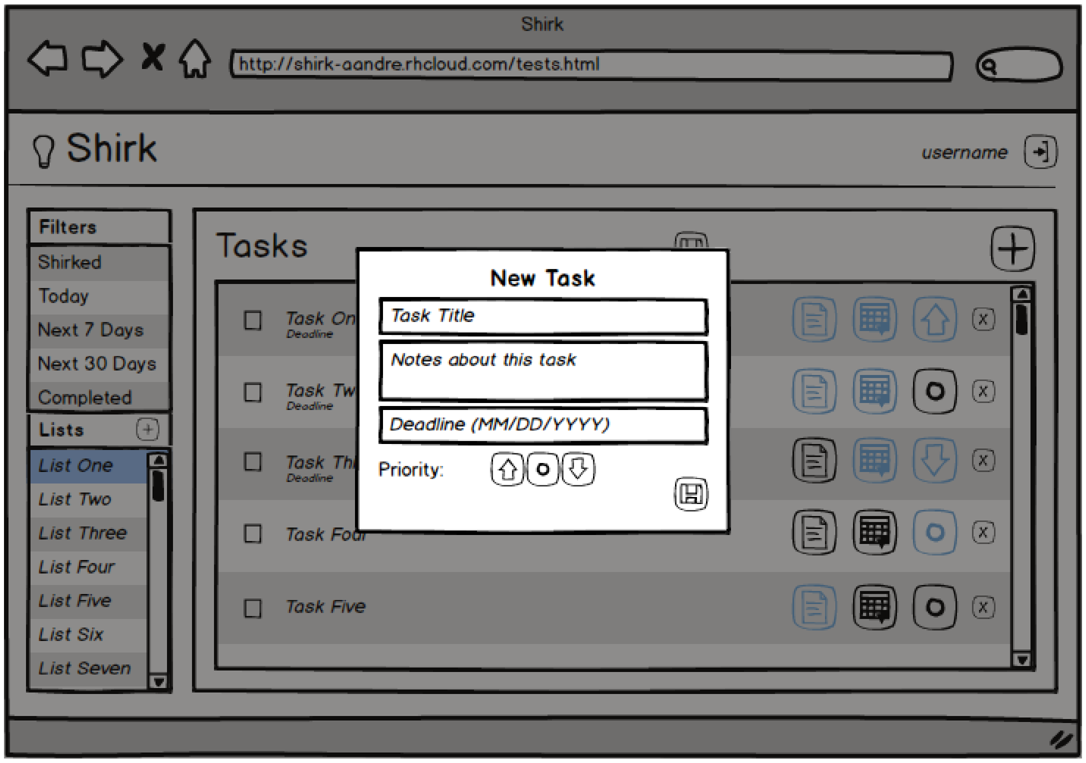
Add a New List:



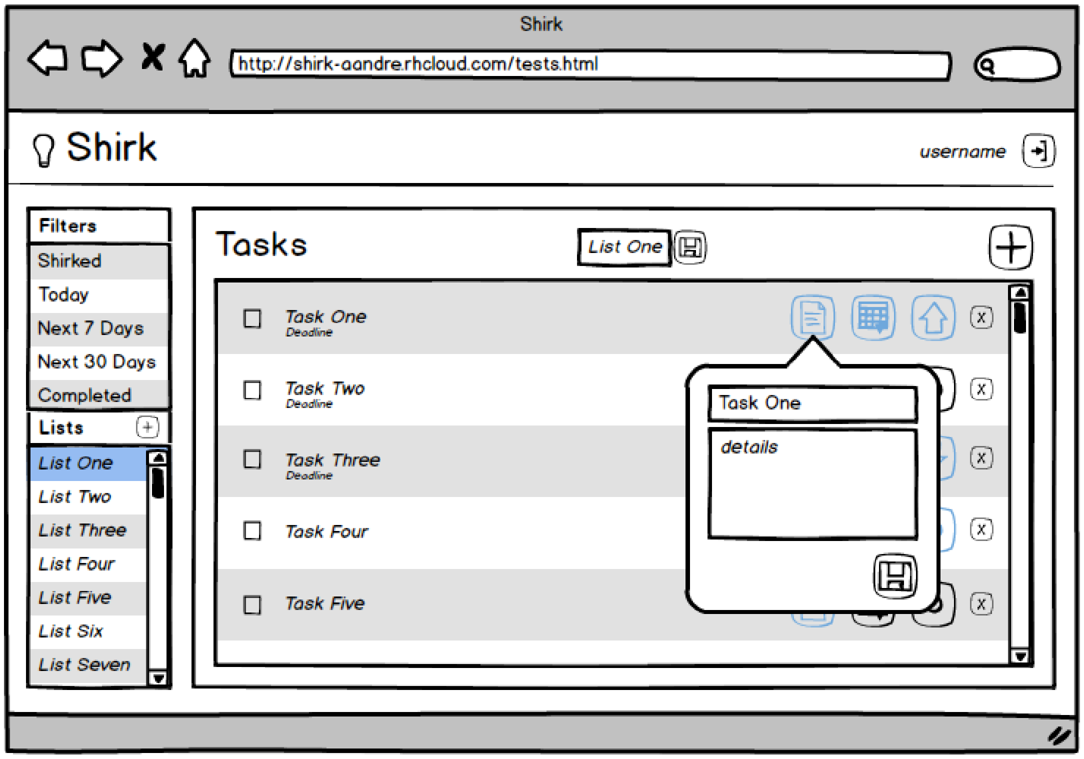
Delete a List:



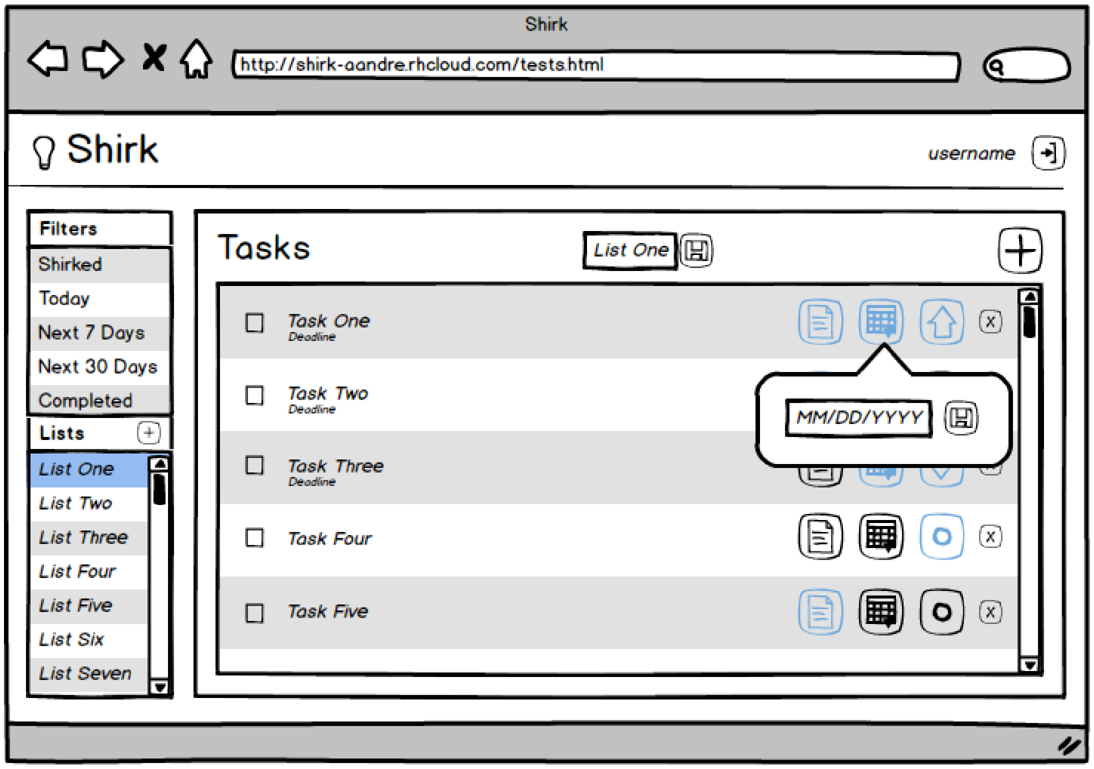
Add a New Task:



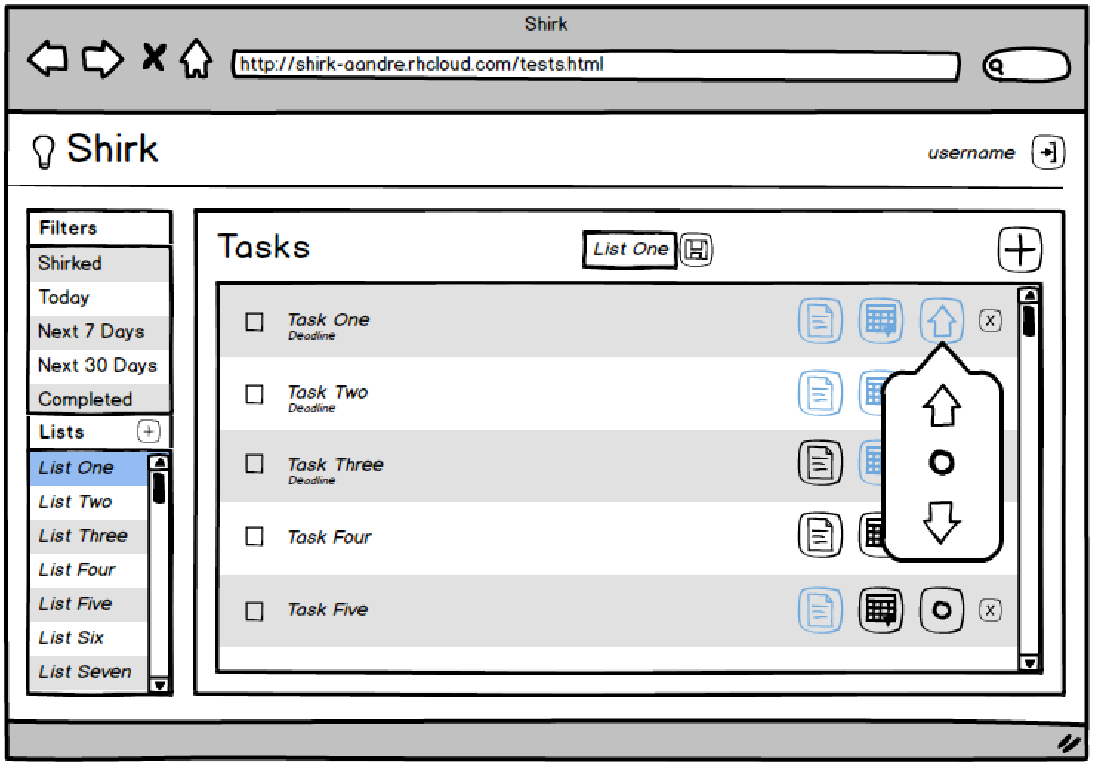
Edit Task Title and Notes:



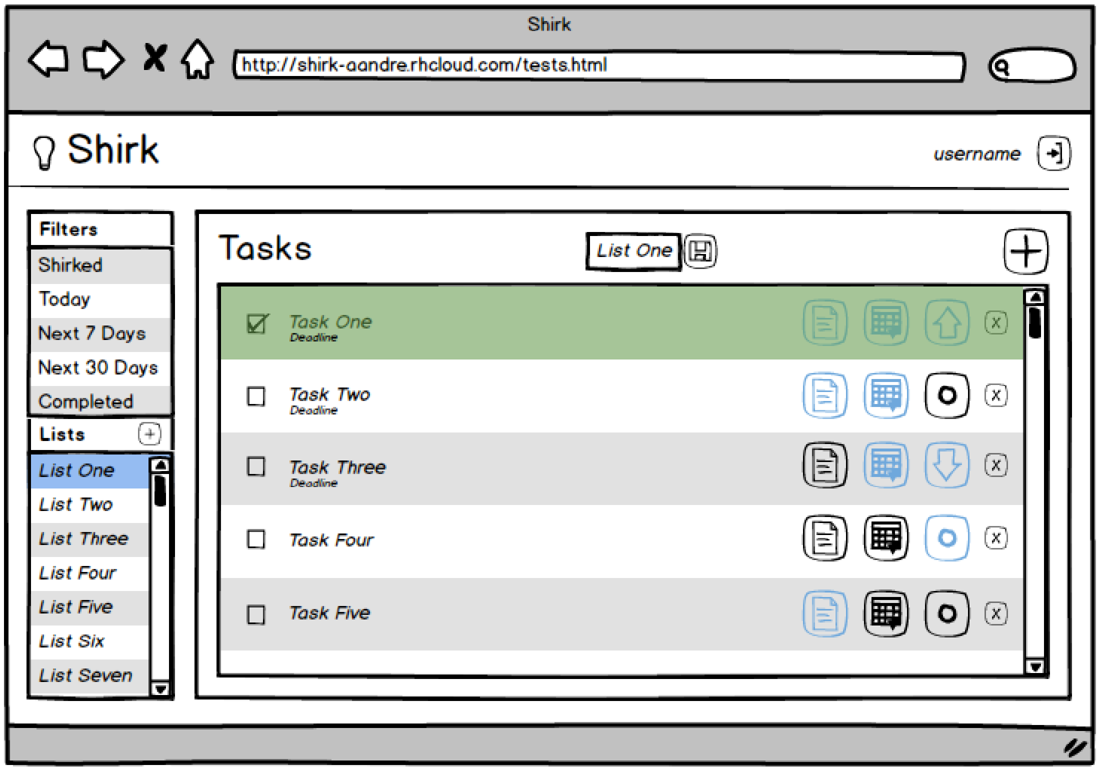
Edit Task Deadline:



Edit Task Priority:

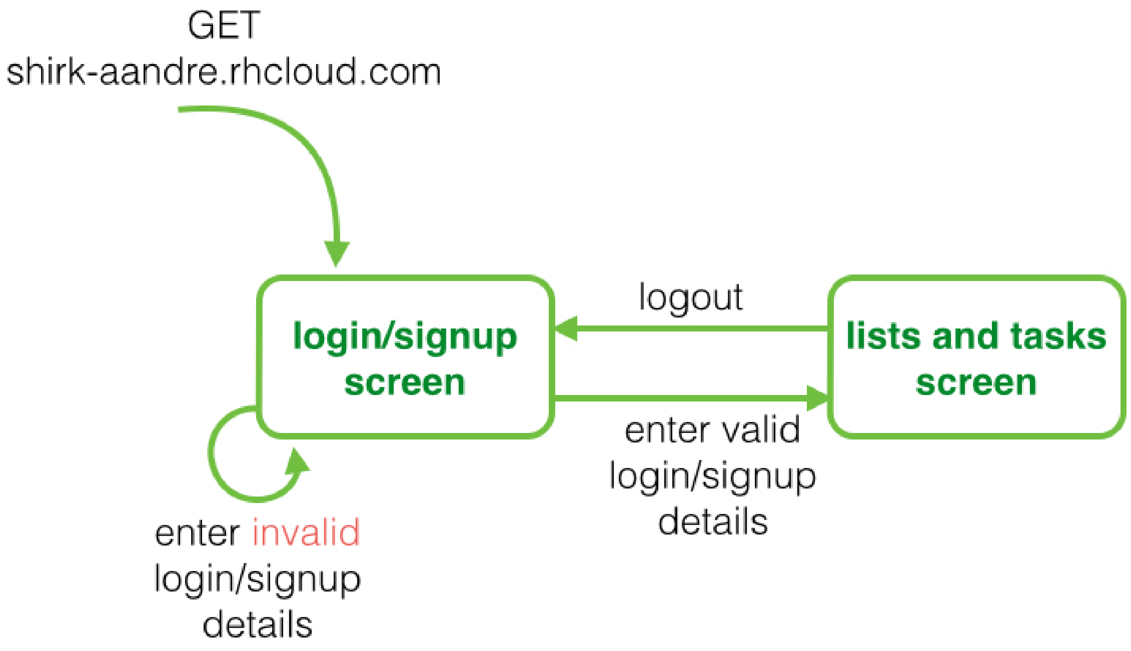


Mark a Task Complete:

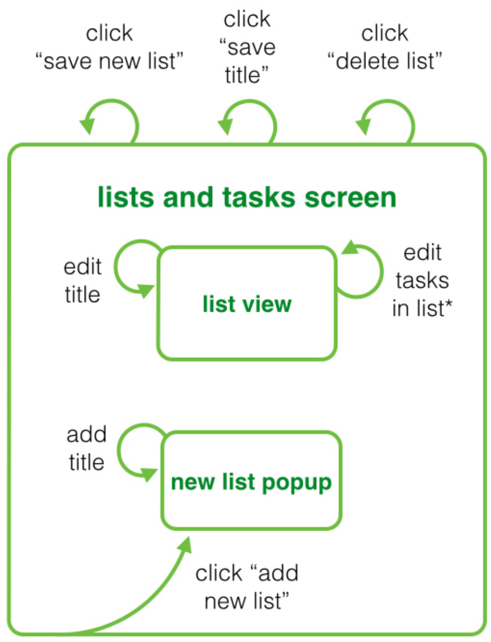


UI Flow/State Modification

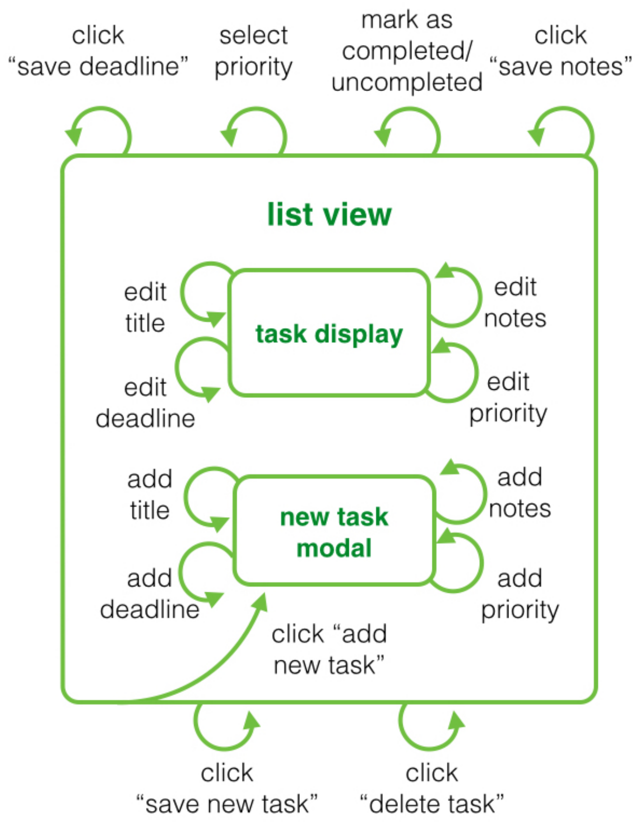
Overall flow:



Details of lists and tasks screen - what you see after you log in



Details of list view - what you see on the lists and tasks screen once you click on a list



Challenges

Shirking Tasks

The concept of shirking, or delaying, a task can be represented in varying ways. It can either be a conscious refusal to complete a task or the act of neglecting to do it.

Users consciously shirking tasks on their deadline is not a very feasible option. For one, if the user forgets to shirk a task, there would emerge a distinction between *delayed* (shirked) and *overdue*. For instance, the task may be several days past its deadline, whereas the user has only requested to shirk it a single day. The number of days a task has been shirked would also need to be recorded in order to apply the appropriate time pressure to the user. This would be complicated to determine if the user modified a deadline. The benefit of this approach is that the user is definitely aware of the work they have left undone, since each task must be individually shirked.

A better approach is to make *overdue* tasks synonymous with *shirked* ones, meaning that all tasks not completed by their deadline would be automatically shirked. This would require no intervention from the user, and would allow *Shirk* to calculate the number of days that have passed since the deadline, a usable indicator of urgency. Changed deadlines would not be an issue, since the number of shirked days would simply be recomputed. However, since the process is automatic, the user may not be aware of all of their overdue tasks. *Shirk* will address this by displaying a noticeable, persistent reminder of all shirked tasks.

Determining Task Importance

Pending tasks may have an associated priority or deadline, or both. The idea of relative "importance" becomes relevant when comparing tasks across these two indicators. The problem lies with ordering and representing these tasks corresponding with this importance metric.

The tasks can all be combined into a single listing and ordered by the widely-accepted 'urgent and important matrix'. Urgent/high tasks would be most important, followed by urgent/low, not urgent/high, and not urgent/low. Since we represent priority as high/neutral/low and deadlines can be early/timely/late, we would need to extrapolate on the accepted standard to accommodate for these added values. This is further complicated by the fact that tasks can omit having a deadline. In any case, the user is unlikely to agree with the formulated ordering for a subjective view of ordering.

Since priority and deadline are not directly comparable, it is possible to instead leave them separated. Within each date period (overdue, due today, next seven days, etc.) the tasks can be ordered by the user-set priority. With this simpler representation, the analysis of importance is delegated to the user. The only shortcoming of this approach is that viewing the wrong date range (ex. next month) may allow the user to overlook what is urgent/high. However, the UI can be designed such that it will be very clear to the user what they are looking at and what they should be looking at.

Filtering Tasks

It is often necessary to display tasks that pertain to a certain logical set -- “high priority” or “overdue”, for instance. The difficulty lies in selecting these tasks in a way that is reflective of the data’s current state.

One method is to create entities in the data model that correspond to each of these sets. Suppose “overdue”, “on time”, “early” are subsets of Deadline and a task can correspond to at most one of these subsets. Displaying overdue tasks then becomes a simple matter of querying the appropriate entity. However, a problem quickly arises once a single day has elapsed; yesterday’s “on time” tasks may now be “overdue”. This method would require the references in the database to be updated regularly, which is infeasible and not an atomic operation.

A much better approach would entail filtering existing data to obtain these sets. The data model always contains factual data about each task’s deadline and priority. Searching for tasks with these properties would be a matter of searching through all tasks, without needing to modify any data. This method would find the relevant tasks when needed, eliminating the problem of stale references. New filters can also be created without requiring modification to the data model.

Another issue that arises is determining where filtering takes place: on the List level (filter all the Tasks in a List), on the Task level (filter all Tasks), or both. *Shirk* provides routes for both functionalities. The same results could be obtained by only filtering on Tasks, and providing an option to specify the List in the Task filter, but this does not provide as clean a separation between Lists and Tasks.

Getting Lists

When a user gets one of their lists, we had to decide how much information about the tasks for that list should be gotten at the same time. There seemed to be two logical possible approaches - either go ahead and return both the list and all the tasks associated with it, or just find all the IDs for the tasks on that list, return them along with the list, and then have to call again later to look each task up if they were needed. For the purposes of our application, we decided that the first approach made much more sense. The main reason for this was that there is no use case in our application for getting a list where you would not also immediately want the tasks, as getting a list will be used for displaying the list title along with all the tasks in that list. Therefore, it would not make sense to return partial information, only to have to turn around and get the rest of it immediately afterward. This would have been both less efficient and more complicated than just getting all the tasks in the first place.

Lessons Learned

The Importance of Good API Design

One thing that became very apparent during this project was that having a complete, well-thought-out API design early on is incredibly helpful. We spent a good bit of time on ours, and it really seemed to pay off when we got to the front end, and everything we needed was just already there. We were essentially able to just make the UI work, and then having it actually communicate with the backend was relatively easy, because that was all already in place. It definitely saved us time in the long run to have been thorough about that initially.

Things that Seem Simple Can Take a Long Time Anyway

We thought that the front-end part of this project would be the easiest part, but it actually ended up taking us substantially longer than we thought it would. Everything ended up being a lot more work than we expected, and we ended up needing to put in more hours on this part than on the API part. Fortunately, we had started early enough to realize this with time left to still get things done, but it was a warning for the future not to underestimate front-end.

Working Together Makes Things Happen Faster

Especially during Phase 3 of this project, we were reminded of the usefulness of working together, as in being physically in the same place while working. We did a lot of the implementation for Phase 2 on our own after meeting up to plan what to do, but for Phase 3, where things were taking a longer time to do, we decided to meet up and code together in the same room. This allowed us to solve problems and get things done substantially faster than we would have otherwise, because we were able to just ask someone next to us any time we got stuck or were uncertain of something, and usually someone else knew the answer. Also, because we were working on the UI, it was really nice to be able to get immediate feedback from teammates on UI elements and make sure they looked and functioned the way we wanted them to. Overall, working together seemed to make things a good bit easier than they could have been.