TaskApp

In this section we build our first app, and discuss the various steps in creating an app like that.

Notes

App Overview

We start with a high-level description of the app we have in mind. It will be an application that helps us manage a list of "tasks". Here is some desired functionality:

- The application manages "tasks", which have some associated "title" string to them.
- We can see a listing of all the tasks.
- We can mark tasks as completed, or delete them altogether.
- We can associate "labels" to a task.
- Users can add/remove labels from a task.
- We can choose to only see the tasks that have certain label(s).

We will look at optional behavior later.

Before we describe one approach, you should work on it first. You should answer the following questions:

1. What are the different **modules** we may need? Modules are collections of functions, constructors, variables, that all share a common purpose.
2. What are the relationships between your modules? Which module needs to know about which other module? A key goal is to keep these relationships at a minimum.
3. Describe some key elements of each module (methods, variables).
4. Identify which modules need to directly respond to user input and/or update the page. Ideally these behaviors would NOT be spread across multiple modules.

Based on the above description, let us consider some key components of such a project:

Task We will need to have some "Task" objects. They should contain a title, a completion status, and a list of labels. Tasks objects have methods for adding/removing a label, setting the title and so on.

In an extended version of the app, tasks might be stored in some way, for instance in a database.

TaskList We will need to have a way to manage a list of objects. This could be as simple as an array underneath, but we will provide a specific interface.

A taskList should have ways to add/remove tasks, and a way iterate over the tasks.

Label Even though labels are simple word strings, there is value in having them be their own objects. This constructor will be unusual in the sense that for any given string there can be only one label for that string. Our constructor will therefore be enforcing that, and possibly return an existing object if it is asked to create an existing label.

Each label also contains the information as to whether it is currently "active" or not.

Because of this unusual behavior of labels, we will not use a separate object to manage the list of labels. Instead, the Label class will be providing us with a "list" view of the labels when asked.

Filters We'll need some structure to keep track of the filters that are in place. These would include the list of the labels, and whether we should show only uncompleted or only completed tasks.

A key question is who is responsible for deciding if a task should be shown or not. Is this a question that the tasks should be able to answer for themselves, given a Filters structure, or is this a question that the Filters structure should be able to answer, given a task? The latter makes more sense.

Controller A controller is the component responsible for responding to user input, and adjust the backend structures as needed, as well as updating the interface.

In more complex applications we may have multiple controllers. In this instance we will use a single controller, that manages all the different page parts.

Overall Code Structure

We need to decide how to organize our code. Here are some key considerations:

- Each component should be in its own file, so its code can be considered/tested/altered without needing any of the other parts.
- In general components need to be as independent of each other as possible, but some components will need to know about other components. We therefore need a mechanism for components to "require" other components. We will see in more detail how to do this in the future.
- We should try not to pollute the global space as much as possible. One way to avoid that is to create one global variable as a "namespace for our application". Then we add components to that. For example, a start code for each component could be something like this:

```
// Immediate function invocation. Closes at the end of the file.
// "global" will be the global object, which is either "window" (browser) or "root" (nod
(function(global) {
    if (!global.hasOwnProperty('TaskApp')) {
        global.TaskApp = {};            // Create namespace if it does not already exist.
    }

    // Can declare local variables/functions etc here
    var Task;
    // If Task depended on other modules, e.g. TaskApp.TaskList, we can
    // check if those are set and throw an error or delay running the rest of the code.

    // Define the Task class here.
    // All the meaty stuff goes here.

    // We "export" the Task class at some point
    global.TaskApp.Task = Task;
}(typeof window === 'undefined' ? root : window));
```

We will use this format in each of our files.

- For the above code to work, we need to manually manage dependencies, and load the files in the correct order. So the order in which we include script tags in the file matters.
- We will need a way to create new "task" items. For that we will need a "template" to copy from. We will see one way of doing that for now, via a script tag. We'll learn more about templates later.
- We will need to decide on a file structure. We will use the following:

```
– projectRoot
    – css
        cssfiles in here
    – js
        javascript files in here
    – lib
        folder for 3rd–party libraries like jQuery
    – tests
        test files go here
    index.html
    .eslintrc.json
    other key root level files
```

This is by no means the only way to do it, but it is one way.

- The functionality that actually requires a web-browser should be restricted to the Controller. The other classes (models), should be self-sufficient, and we should be able to write tests for them directly.

Here's a basic graph for an initial setup for the different modules:

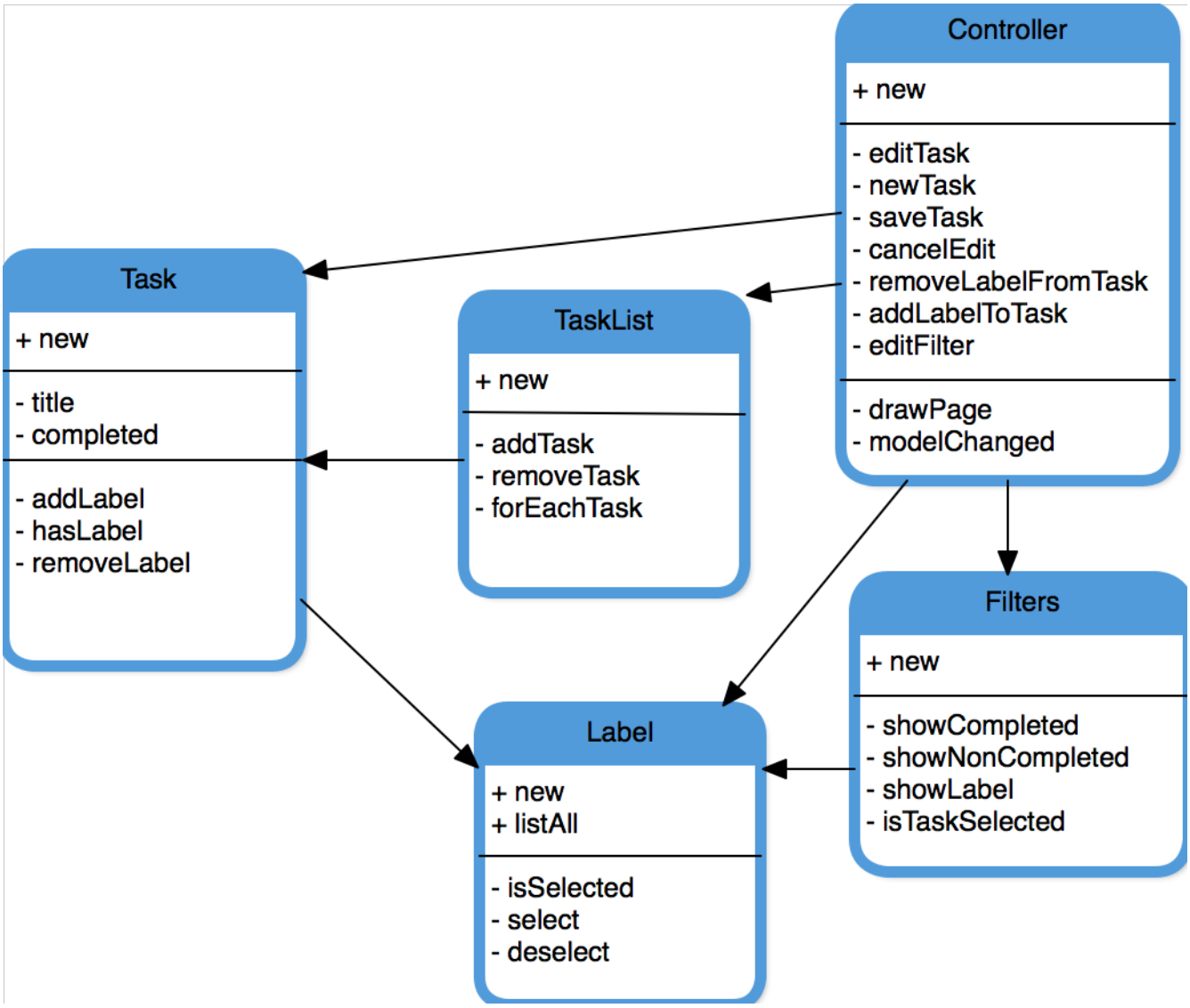

Figure 1: TaskApp planning