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.

General Design Principles

Here are some general principles to adhere to:

1. **Separation of Concerns**. A *concern* is a set of information that affects the code in a computer program. Our program should be separated in sections, each section addressing one concern, and so that a concern is not spread across multiple code sections. This is also called **modular** design.
2. **Single Responsibility**. This is closely related, and it points out that modules should not try to do too much. They should have a single responsibility, and they should do it well.

3. **Information Hiding/Abstraction**. We want to keep design decisions about a specific part of the code entirely local to that piece of code, and provide a consistent way for the rest of the code to get to that information. This way if in the future we decide to change the design, there is only one piece of code that needs to change. The mechanism by which this is achieved is usually called **Encapsulation**.
4. **Model-Controller**. In most web applications there is an important distinction between **models** and **controllers**. Models are modules that control the application's data/state, and they contain the "business logic". Controllers are responsible for interacting with the user and using the models to prepare the user interface. It is imperative that these are kept separate.

Basic Architecture

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.

One important consideration is where the information about whether a task is visible is stored. Should it be part of the task objects, or does it belong in a separate place, that is responsible for the UI?

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 Deciding what to do with labels is a hard decision. Here are some desirable features:

- Labels are fully identified by their text string, so we should never need to have more than one label for a given text string.
- We need to know which labels are used by issues at any given time.
- We need to know a count of how many issues a label is used in, to show on the label list.
- The above two items need to adjust to the "current" task list, which may be filtered (i.e. not all tasks used).
- We need to decide if a label should exist past its use. I.e. if no issues use a label any more, does it need to be removed?
- We need to know if a label is "active" or not, i.e. whether it is currently used for filtering.

We have for the most part two design options:

- Create label objects, that contain a list of the tasks in them along with other status information. Keep the information stored in tasks and that in labels in sync, as they depend on each other. We will opt for this approach here.
- Do not create label objects, keep the information in the tasks, tasklist, and filters. Generate counts when needed by going through the task list. This would certainly have been a viable option.

The constructor for label objects 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.

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, adjusting 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.

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

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:

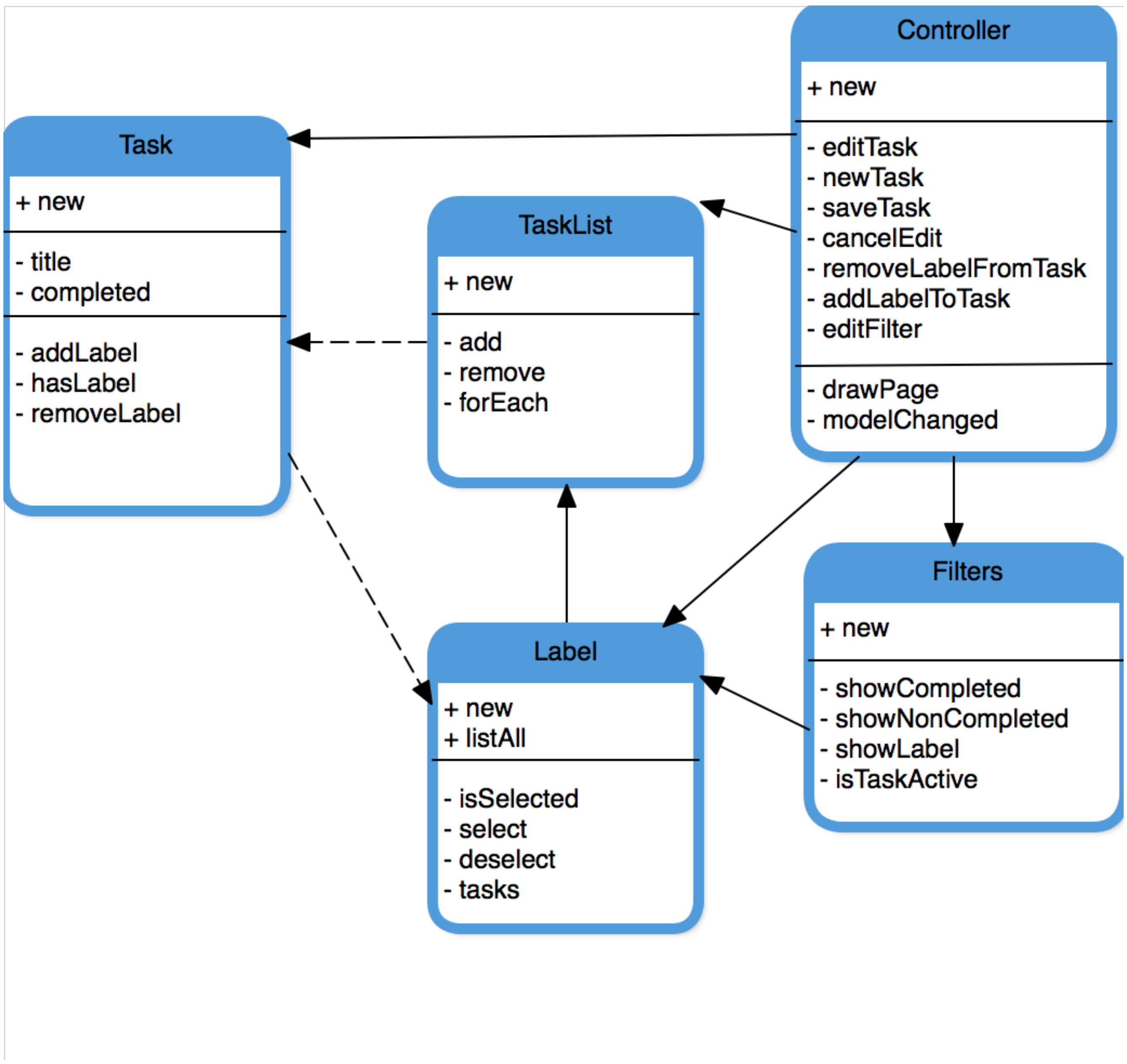

Figure 1: TaskApp planning

```
// 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. Figuring out the correct order for the different modules is actually an interesting program in graph theory, called **topological sort**.

- 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. Essentially we add something like the following on the web page:

  ```
  <script id="taskTemplate" type="text/template">
  <div class="task">
  <p>{{title}}</p><input class="completed" type="checkbox" value="{{completed}}"></input>
  </div>
  </script>
  ```

  The above script can be accessed via its id, and because its type is not set to text/javascript it will not actually be executed by the browser. We can then use its text to create a new item.

  Notice that there are a number of "placeholders", surrounded in double curly braces. We will replace those with suitable values.

- 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.

Initial Code

We now take a look at the starting layout code for our app.

Main Index file We start with index.html:

```
<!DOCTYPE html>
<html>
<head>
   <title>TaskApp: The Task App of the future</title>
   <link rel="stylesheet" type="text/css" href="css/reset.css">
   <link rel="stylesheet" type="text/css" href="css/taskapp.css">
</head>
<body>
   <header>
   </header>
   <main>
      <section id="filters">
         <div class="filter" id="postsChoice">
            <h3>Posts to show</h3>
            <!-- TODO: Add radio buttons -->
         </div>
         <div class="filter" id="labelsList">
            <h3>Labels</h3>
            <ul>
              <!-- Labels will be added dynamically using the template -->
            </ul>
         </div>
      </section>
      <section id="tasks">
      </section>
   </main>
   <footer>
   </footer>
<!-- Normal page info ends here. Some scripts load further down. -->
<!-- This is the template for a task. It is not actually treated as Javascript code
      the curly  braces are placeholders for dynamically generated values -->
<script id="taskTemplate" type="text/template">
<div class="task">
<p>{{title}}</p><input class="completed" type="checkbox" value="{{completed}}"></input>
</div>
</script>
<script id="labelTemplate" type="text/template">
<div class="label">
```

```
<p>{{tag}}<span><!-- count goes here --></span></p><input class="completed" type="checkbox" v
</div>
</script>
<!-- SCRIPT LOADING HERE -->
<script type="text/javascript" src="js/templates.js"></script>
<script type="text/javascript" src="js/tasklist.js"></script>
<script type="text/javascript" src="js/label.js"></script>
<script type="text/javascript" src="js/task.js"></script>
<script type="text/javascript" src="js/filters.js"></script>
<script type="text/javascript" src="js/controller.js"></script>
</body>
</html>
```

Our goal for now is to:

- Delineate the main areas of the page
- Prepare the templates
- Link to a CSS file
- Load the scripts in proper order

Templates Next we look at the template file. It provides us with methods for handling our primitive version of templates. Later on we'll make it use a template library.

```
// templates.js
//
// This file handles the various templates for us.
//
(function(global) {
   var Template, templateStorage, proto;

   if (!global.hasOwnProperty('TaskApp')) {
      global.TaskApp = {};
   }

   // Object keeping the stored templates
   templateStorage = {};

   /*
    * Exported object. `load` is used to store a template, `parse` is used to
    * parse a stored template.
    */
   Template = {
      /*
       * Returns a new template object based on the text in `html`.
       * The `name` can be used to access the template in the future
       * If a template with the same name exists, it prints a warning message
       * and replaces it.
       */
      new: function newTemplate(name, html) {

      },
      /*
       * Returns the template with a given name, or `null` if it does not exist.
       */
      get: function getTemplate(name) {

      }
   };

   /*
    * Prototype object for created templates
    */
   proto = {
      /*
       * Parses the template named ‘name‘, using the ‘values‘ object
       * to resolve parameter entries. For instance {{foo}} will be
       * replaced by values.foo
       */
      parse: function(values) {

      }
   };

   global.TaskApp.Template = Template;
}(typeof window === 'undefined' ? root : window));
```