# 1 Project 1

**Due**: Jun 30 by 11:59p

**Important Reminder**: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. This is followed by a log which should help you understand the requirements. Finally, it provides some hints as to how those requirements can be met.

[Please note that intra-document links do not work within the PDF version of this document.]

## 1.1 Aims

The aims of this project are as follows:

- To ensure that you have set up your VM as specified in the *VM Setup* and *Git Setup* documents.
- To get you to write a non-trivial JavaScript program.
- To give you some exposure to using `webpack`.
- To familiarize you with manipulating the DOM using jquery.

## 1.2 How To Read This Document

This document is rather long as it attempts to specify the project in detail and also provide step-by-step instructions. Hence it is not a good idea to attempt to understand the document in detail initially.

1. Read the following **Background** section thoroughly.
2. Scan through the rest of the document so that you understand what kind of information is provided and know where to find it.
3. Run the working application and relate the displayed content with the provided meta.js.
4. Look at the *provided code* in detail.
5. Start working on the project, and refer back to details provided in different parts of the document as needed.

## 1.3 Background

Instead of writing web forms directly in HTML, it is possible to represent them using a JavaScript data structure and then directly write the corresponding DOM objects into the DOM. For example, a simple google search form could be represented as:

```
{ type: 'form',
  attr: { action: 'https://www.google.com/search', },
  items: [
    { type: 'input',
      text: 'Search Terms',
      attr: {
        name: 'q',
        title: "enter search terms",
      },
      required: true,
    },
    { type: 'submit', text: 'Search Google', },
  ],
}
```

which would generate DOM objects **equivalent** to the following HTML:

```
<form action="https://www.google.com/search">
  <label for="/googleSearch/items/1/items/0">
    Search Terms
  </label>
  <div>
    <input name="q" title="enter search terms" type="text"
           id="/googleSearch/items/1/items/0"/>
    <div class="error"
         id="/googleSearch/items/1/items/0-err"></div>
    <div></div>
    <button type="submit">Search Google</button>
  </div>
</form>
```

The data structure is much less verbose than the HTML. More importantly, it is much easier to build up dynamically.

The conversion can be done using a templating system like mustache or reactjs. This project takes a different approach and generates the DOM objects directly in the document's DOM using jquery. Note that HTML is never explicitly

generated; only the corresponding DOM objects are created.

## 1.4 Requirements

You must push a `submit/prj1-sol` directory to your `master` branch in your github repository such that typing `npm ci` followed by `npm run start` within that directory starts a server. Accessing that server using a browser should display a default page with content as specified by meta.js. The behavior of your program is illustrated here.

The detailed requirements for the code are quite long. Hence we first list some auxiliary requirements to ensure that they do not get buried under the lengthy main requirements:

1. Your `submit/prj1-sol` directory **must** also contain a `vm.png` image file to verify that you have set up your VM correctly. Specifically, the image must show a x2go client window running your VM. The captured x2go window must show a terminal window containing the output of the following commands:

```
$ hostname; hostname -I
$ ls ~/projects
$ ls ~/cs544
$ ls ~/i?44
$ crontab -l | tail -3
```

2. You may not use any external runtime libraries other than jquery.
3. There is no requirement that you check whether the `meta.js` input is error-free.

Now for the detailed main requirements.

The project should be set up to display a page corresponding to `Meta[ref]` where `Meta` is the object exported by `meta.js` and `ref` is the value of the `ref` query parameter for the page URL. If `ref` is not specified, then it should default to `_`.

We specify the different kinds of content using a semi-formal notation. The descriptions below are organized in order of increasing complexity. Note that property names are used consistently across multiple `type`'s; hence the explanation for a property is not repeated when it is used subsequently.

We use `Content` to represent arbitrary content generated by the system. The different kinds of content are identified using a `type` property which defaults to `block`.

- One of the simplest kind of content is a **block**:

```
{ type: 'block', attr: Attr, items: List<Content>, }
```

```
=> <div attr>=items=</div>
```

The above notation means that in `meta.js`, a block would be represented as a hash with a required `type` property with value `block`, an optional `attr` property which is a map of attribute names to values and an `items` property which is a list of `Content`.

The `=>` separates the input specification from the HTML which is equivalent to the generated objects. In this case, the generated objects should consist of a `div` element with attributes specified by the input `attr` map and content which is a recursive expansion of the list of items.

If a content description does not have a `type` property, then the `type` defaults to `block`.

- We have a similar specification for a paragraph with type **para**:

```
{ type: 'para', attr: Attr, items: List<Content>, }
=> <p attr>=items=</p>
```

- A text **segment** contains an arbitrary `String` or a recursive expansion of `items`.

```
{ type: 'segment', attr: Attr, text: String, items: List<Content>,
}
    => if text is present
       then
         <span attr>text</span>
       else
         <span attr>=items=</span>
       end
```

The output specification following the `=>` uses a pseudo-code `if-then-¬else-end` construct to distinguish between the case when `text` is present and when it is not. Note that if `text` is present, then `items` is ignored.

- A **header** can be used to display a heading:

```
{ type: 'header', attr: Attr, level: [1-6], text: String,
}
=> <`h${level}` attr>text</`/h${level}`>
```

The `level` property should be an integer between 1 to 6 inclusive. The output specification uses JavaScript's template literal notation to indicate generation of HTML tags `h1` to `h6`.

- A **link** to another page:

```
{ type: 'link', attr: Attr, ref: String, text: String, }
```

```
=> <a href="url(ref)" attr>text</a>
```

where `url(ref)` should return a URL which is identical to the URL of the current page except that it has a `ref` query parameter set to the value of the `ref` property.

- A **form**:

```
{ type: 'form', attr: Attr, items: List<Content>, }
=> <form attr>=items=</form>
```

When an **error-free** form is submitted, the values of the widgets should be output on the console using `console.log()` as JSON with an indent of 2. The value output should be an object containing a property for each active `name` in the form. The values of widgets which are potentially multi-valued **must** be output as a list.

- Now we get into form controls or widgets. Usually, a form will have some kind of **submit** button for submitting the form:

```
{ type: 'submit', attr: Attr, text: String, }
=>
<div></div>
<button type="submit" attr>text</button>
```

If `text` is not specified, it should default to `Submit`.

[The empty `div` element is necessary to place the submit correctly with the provided stylesheet. The stylesheet assumes that each widget is preceeded by a `label` object. In this case, the empty `div` substitutes for the lack of a `label` object.]

- We allow specifying simple **input** controls for allowing the input of text:

```
{ type: 'input', attr: Attr, subType: SubType,
  text: String, required: Boolean,
  chkFn: Function, errMsgFn: Function,
}
=> if (subType === 'textarea') then
   then <label for=ID>text*</label>
        <div>
          <textarea attr id=ID></textarea>
          <div class="error" id=ID></div>
        </div>
   else <label for=ID>text*</label>
        <div>
          <input attr id=ID/>
          <div class="error" id=ID></div>
         </div>
```

```
    end
```

The `SubType` can be `textarea` or any value supported for the `type` attribute of `<input>` elements in HTML5. It defaults to iiteral value `"text"`.

The `text` property specifies the label for the widget.

The remaining input properties specify validation constraints on the input:

- The `required` property is specified as **true** if the input must contain at least one non-whitespace character. If `required` is **true**, then the input must be checked whenever the widget is `blur`'d or the form is submitted. If the widget value is empty, then an error message

  ```
  `The field ${text} must be specified.`
  ```

  must be added to the DOM as explained below. `${text}` refers to text used to label the widget.

- The `chkFn` property should specify a JavaScript function used for validating the input. It takes up to 3 parameters:

  1. **val**: The value provided for the widget by the user.
  2. **info**: The complete specification for the widget; i.e. the portion before the `=>` in the specification above.
  3. **meta**: The complete data structure defined in `meta.js`.

  This validation function should be called only when a non-empty value is provided for the widget whenever the widget is `blur`'d or the form is submitted. It should return truthy iff **val** is valid.

- The `errMsgFn` property must specify a function having the same parameters as the `chkFn` property. It must return a suitable error message when the widget value fails validation by `chkFn()`.

  If `chkFn` is specified, but `errMsgFn` is not specified, then when the widget value fails validation by `chkFn()`, the error message:

  ```
  `invalid value ${val}`
  ```

  should be displayed (where `val` is the value provided for the widget).

The output specification following the `=>` uses a pseudo-code `if-then-¬ else-end` construct to distinguish between the case when the `subType` is `textarea` and everything else. The expected HTML should be quite clear but a few points are worth noting:

  - The `ID` must be an ID which is guaranteed to be unique across the entire HTML page. If the `attr` specifies an `id`, then the value of that property should be used.
  - The `text*` notation means that if the `required` property is truthy, then a `*` should follow the `text` labelling the widget.
  - Each widget is followed by an empty `div` with `class="error"`. This `div` will be used for displaying validation errors for the widget.

- We have a control specified with `type` **uniSelect** which selects a single value from a list of values:

```
{ type: 'uniSelect', attr: Attr, items: List<KeyText>,
  text: String, required: Boolean,
  chkFn: Function, errMsgFn: Function,
}
=> if (items.length > (_options.N_UNI_SELECT || 4)
   then <label for=ID>text*</label>
        <div>
        <select attr id=ID>OptionItems(items)</select>
          <div class="error" id=ID></span>
        </div>
   else <label for=ID>text*</label>
        <div>
          <div class="fieldset">
            InputItems(items, 'radio', attr)
          </div>
          <div class="error" id=ID></span>
        </div>
   end
```

`List<KeyText>` means that `items` must be a list of objects having `key` and `text` properties, both of which are simply strings. Either `key` or `text` may be omitted in which case it defaults to the other.

HTML provides two form controls for selecting one value from a list of values: using a dropdown select box or radio buttons. The choice of whether to display a dropdown or radio buttons is based on the lenght of `items`. If it greater than some threshold, then a `select` dropdown is used, otherwise radio buttons are used. The threshold is specified by `_options.N_UNI_SELECT` where `_options` is a top-level property in the object exported by `meta.js`. If not specified, then it should default to 4.

  - The notation `OptionItems(items)` should render each `KeyText` `item` in `items` as follows:

```
<option value='"${item.key"'>'${item.text}'</option>
```

- The notation `InputItems(items, 'radio', attr)` should render each `KeyText item` in `items` as follows:

```
<label for="ID">`${item.text}`</label>
<input value=`"%{item.key}"` type="radio"
       attr id="ID"/>
```

The validations specified by `required` and `chkFn()` should be done on form submission or whenever the widget value **changes**.

- The `type` **multiSelect** is used for selecting multiple values from a list of values:

```
    { type: 'multiSelect', attr: Attr, items: List<KeyText>,
text: String,
      required: Boolean, chkFn: Function, errMsgFn: Function,
    }
    => if (items.length > (_options.N_MULTI_SELECT || 4)
       then <label for=ID>text*</label>
            <div>
              <select attr multiple="true" id=ID>
                OptionItems(items)
              </select>
              <div class="error" id=ID></span>
            </div>
       else <label for=ID>text*</label>
            <div>
              <div class="fieldset">
                InputItems(items, 'checkbox', attr)
              </div>
              <div class="error" id=ID></span>
            </div>
       end
```

This is very similar to the `uniSelect type` except that depending on the number of items, it generates either a `<select>` widget with its `multiple` attribute set to `"true"` or a set of `checkbox` widgets.

## 1.5 WebPack

The `<script>` element can be used within a HTML page for loading external scripts into a browser. This entails a network request which can be relatively slow. Given that the fact that modern websites may involve using tens or even hundreds of external libraries, loading all these libraries can result in a non-peformant website.

A solution to this problem in the early days of the web was to simply concatenate

multiple JavaScript files and libraries together. This resulted in fewer network requests, but given the lack of modules in the original version of the language, this was often difficult to do correctly. Attempts to reduce the size of each network request results in **minifiers** which removed comments and non-significant whitespace as well as shortening identifiers.

This evolution resulted in the development of **bundler** programs which bundled multiple resources like scripts, stylesheets, images into a single bundle. One of the most popular of the bundlers currently being used is webpack.

The concepts underlying webpack are quite straightforward. An application may have one or more top-level **entry** points; all dependent resources reachable from these entry points are **output** into a bundle. Resources are loaded using **loaders** and recursively traversed while looking for additional resources which need to be added to the bundle. The core capabilities of webpack can be extended using its **plugin** architecture to install plugins which provide services like transformations. For example, it is possible to write code in modern JavaScript or even a totally different language and then use a webpack plugin to transpile it into a version of JavaScript which runs even in relatively old browsers.

## 1.6 Provided Files

The prj1-sol directory contains a start for your project. It contains the following files:

**index.js** A partial implementation of your program. Areas you need to implement are indicated by `@TODO` comments. Add code as necessary, feeling free to restructure the file.

**webpack.config.js** A simple configuration file for webpack. It is commented extensively and should be quite understandable. You should not need to modify this file.

**meta.js** A file which specifies the content to be produced by your program. Note that even though this file will be bundled along with your code, it should still be regarded as an **input** to your program. It is entirely possible that your program will be tested using a totally different `meta.js`.

You may modify this file when testing your program, add more pages and forms.

**style.css** A style sheet which ensures that your generated content is reasonably readable. You should not need to modify this file.

**README** A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

If your project is not complete, document what is not working in the `README`.

**.gitignore** A file which lists paths which should be ignored by git. You should not need to modify this file.

## 1.7 Hints

You should feel free to use any of the functions from the standard *JavaScript library*, *basic browser API*'s like URL, as well as those from jquery. Note that you are not allowed to use any external library other than jquery at runtime.

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Set up your course VM as per the instructions specified in the *VM Setup* and *Git Setup* documents.

   The subsequent steps assume that you are working on a properly setup VM.

2. Read the project requirements thoroughly. Compare the provided meta.js. with the program behavior. Reconcile both with the requirements for the types.

   Look at the provided code. Look at least briefly at webpack.config.js. Look at the provided code in index.js much more thoroughly as you will need to understand it in order to write your own code.

3. Familiarize yourself with your browser's debugger. This can usually be invoked using F12. In chrome, the `Elements` tab will be useful to examine the objects you have injected into the DOM, the `Console` tab for looking at runtime errors and the `Sources` tab for examining source code and setting breakpoints. Note that you will find your original source code under `webpack-internal://`. In firefox, you will find equivalent tabs `Inspector`, `Console` and `Debugger`.

4. Copy over the provided files into your `i?44` github project:

```
$ cd ~/i?44                          #cd to dir for github repo
$ git checkout -b prj1-sol           #create new git branch
$ mkdir -p submit                    #create submit dir if not exist
$ cd submit                          #change over to submit dir
$ cp -pr ~/cs544/projects/prj1/prj1-sol . #copy provided files
$ cd prj1-sol                        #go into project dir
```

   Note the `.` indicating the current directory at the end of the second-last command.

5. Capture a `vm.png` image as instructed in the requirements. Transfer this image over to your prj1-sol.

6. Use your favorite editor to replace the `XXX` entries in the `README` template.

7. Commit your project to github:

```
$ git add .                    #add all files to git staging
$ git commit -a -m 'started prj1' #commit them to local git
 $ git push -u origin prj1-sol     #push to github,
                                          #tie local prj1-
sol branch to remote
```

8. Set up your project as an npm project:

```
$ npm init -y
$ npm install --save-dev \
  webpack webpack-cli webpack-dev-server \
  css-loader style-loader html-webpack-plugin
$ npm install jquery
```

   (a) The first command initializes your project. It will create a `package¬ .json` file containing default information.

   (b) The second command installs dependencies you need for developing your project.

   - The `--save-dev` option indicates that the dependencies are needed only for **development**. These libraries will not be loaded into the browser at runtime.
   - The `webpack` module provides the webpack core; `webpack-¬ cli` provides a command-line interface to webpack (run it from the command-line using `npx webpack --help` from within your project directory), `webpack-dev-server` provides an HTTP server which supports **Hot Module Replacement** HMR, `css-loader` loads css files into your bundle and `style-loader` injects the loaded css into your HTML, `html-webpack-plugin` generates an empty HTML page which sucks in the generated bundle.

   Ignore warnings re. fsevents which is not needed on linux. See *this stackoverflow article* and *this issue* for more information.

   (c) The last command installs `jquery`. Note that since the `--save-dev` option is not provided, it installs it as a **runtime** dependency which will be included in the bundle sent to the browser.

   [Older npm documentation specifies the `--save` option, but that is no longer necessary].

   Running these commands will create a `node_modules` directory, register

the dependencies in your `package.json` file and create a `package-lock.json` file (the last filename is somewhat misleading).

9. Edit the `package.json` file to add the following line:

   ```
   "start": "webpack-dev-server -d --mode development"
   ```

   after the `scripts.test` entry. Note that you will need to add a comma `,` after the `test` entry to ensure that the file continues to be valid json.

10. You should now be able to run the project. In a separate terminal window (or tab), start the `webpack-dev-server` via the script you just installed in `package.json`:

    ```
    $ npm run start
    ```

    You will get a message telling you that a server has been started at port 8080. Point a browser running on the same machine to `http://localhost:8080` and you should see a page listing all the different forms listed by `meta.js`.

    Note that the page `Form Links` should be displayed completely because the provided code implements all the necessary functionality. That will not be the case for the individual forms: the first 3 only display the headers and the last displays the header and some miscellaneous text.

11. Verify that HMR is working. Go into a source code file like `src/index.js` and introduce a syntax error by adding say a stray right parentheses `)` after the first `import` keyword. You should see an error in both your terminal and the browser. Remove the error and you should again see the clean page in the browser and a clean compile in the terminal.

    Note that only compile-time errors are displayed in the browser. Runtime errors will be displayed in the console.

12. Commit your updates into github:

    ```
    $ git add .
    $ git commit -a -m 'project started working'
    $ git push
    ```

    Note that the `node_modules` directory should not get committed as it is set up to be ignored by the provided .gitignore.

13. Fire up the browser debugger using F12. If you go into the **Sources** tab (chrome) or **Debugger** tab (firefox) and dive into `webpack-internal://` you should see your source code:

    Navigate the debugger into `src/index.js` and add a breakpoint on the third line in the `header()` function which starts with `$e.text` (simply click on the line number). Refresh the page being displayed in the browser. The breakpoint should be hit, the browser page will be displayed as empty

while the program is paused. Mouse over variables in the source code displayed in the debugger window; you should see their current values.

Remove the breakpoint by clicking the line number once again and continue execution by clicking on **pause/resume** control (the arrow-like symbol towards the top-right of your debugger window). Execution should continue and the page should be redisplayed.

As you work on your project, be sure to use the other controls you will find near the pause/resume control. The **step** control steps to the next line in the current function, the **step into** steps into a function while **step out of** runs until control returns from the current function. While the program is stopped you can resume till the next breakpoint using the **pause/resume** control.

Whew!! You are now set up and ready to start developing your project code. The subsequent steps start out with simple `type`'s and build up towards the more complex ones. To start with, we will ignore the requirements on event handlers.

14. Add code for the submit `type`. Use `makeElement()` to create an empty `div` and `append()` it to `$element`. Extend the provided `meta.attr` (using *Object.assign()*) to add in a `{ type: 'submit' }` attribute. Then create a `button` element with the extended attributes. Assign the provided `meta.text` to it using its text() method. Finally append the button element to `$element`.

    Test by displaying the forms; you should now see suitable submit buttons.

15. Add code for the input `type`. Pull out `meta.text`, extending it with a `*` if `meta.required` is truthy. Create an `id` using `makeId()` if the incoming attributes `meta.attr` do not already have one. Append a suitable `label` element to `$element` followed by a currently empty `div` element. Then add either a `<textarea>` or `<input>` element to the `div` depending on the value of `meta.subType`. Be sure to extend the incoming `meta.attr` suitably as in the previous step. Finally add an empty error `div` to the earlier `div`.

    Test by displaying the forms; you should now see suitable `<input>` fields.

16. Add code for the uniSelect `type`. It is just a more complicated version of the previous steps but should be quite straightforward when simply following the specification.

17. Add code for the multiSelect `type`. You should realize that it is very similar to the code for the `uniSelect type`. Use the *Extract Function* refactoring to extract the common code into a function with suitable parameters which accounts for the difference. Now the `uniSelect()` and `multiSelect()` functions can become simple *wrapper functions* which call this extracted function.

You should now be able to display all the provided forms. We will now implement the required dynamic behavior by adding event handlers.

18. Add code to implement the required *dynamic behavior* when the form is submitted. The provided `form()` function contains the start of the handler. For now, ignore the requirement that the form be error-free. Simply use jquery's serializeArray() method on `$form` to get the widget values from the form. Print the results on the console after converting to JSON. You should see all the form widgets, but the result of `serializeArray()` is an array whereas the requirements want an object.

    It is a simple matter to massage the array into the required object. For each `name` in the array, lookup the corresponding widget within the form using a selector `$(`[name="${name}"]`, $form)`. Then if the widget has an attribute `multiple` which is truthy or an attribute `type` which is `checkbox`, it is a potentially multi-valued widget and the values should be stored in the object as an array; otherwise the value is stored directly in the object.

    Once you have massaged the array into an object `obj`, print it out on the console using `JSON.stringify(obj, null, 2)`.

19. Now we need handlers to validate individual widgets on blur events (for `type input`) or `change` events (for `uniSelect` and `multiSelect type`'s). A little reflection will show that the required validation behavior is exactly the same except for the event which triggers the behavior. So create a common function to set up a handler and attach the handler to the widget using jquery's on().

    The handler should be defined within a context where it has access to the meta-info for the widget. This ensures that the dynamic handler has access to the static validation information associated with the widget.

    To start with, simply have the handler log something on the console and verify that you are catching `blur` and `change` events as specified.

    Now set up the validation behavior of the handler. Note that it will be called with its first argument specifying the event. Note that if neither `required` or `chkFn` are specified for the widget, then no validation is possible and the handler can simply return.

    (a) You can get the jquery-wrapped event target `$target` using `$(event.target)`.

    (b) For all widgets except checkboxes, you can get the widget value using `$target.val()` (we will consider checkboxes later).

    (c) Get hold of the error `div` for the widget using the id which you specified when you created that error `div`. You can then set the error message by using jquery's text() method on the error `div`.

(d) If the trim()'d value is empty, then if `required` is true for the target widget, then display a suitable error, else display an empty error.

(e) If the trimmed value is non-empty and there is a `chkFn()`, then if `chkFn()` succeeds set the error to empty, otherwise set it to a suitable error message (use `errMsgFn()` if it is specified, otherwise the specified default message).

(f) If the trimmed value is non-empty and there is no `chkFn()` there is no error.

For checkboxes, you can get a jquery list of all checked boxes for a particular `name` in a form `$form` using a selector like `$(‘input[name="${name}"]:checked‘, $form)`. You can then massage that jquery list into a JavaScript array of values and then use the array of values for the validation.

20. Now that we have validations working, we need to implement the requirement that the submit handler should only be run for an **error-free** form. We can do so using the handlers set up in the previous step. Specifically, we first trigger those handlers:

```
$('input,select,textarea', $form).trigger('blur');
$('input,select', $form).trigger('change');
```

Then we check whether we have any errors within the form by looking at the results of the selector `$('.error', $form)`. If there are any non-empty error messages, the results of the form submission should not be logged on the console.

21. Iterate until you meet all requirements.

22. If necessary, update your README with your project status or any information which you want to convey to the grader.

It is a good idea to commit and push your project periodically whenever you have made significant changes. When complete, please follow the procedure given in the *git setup* document to merge your `prj1-sol` branch into your `master` branch and submit your project to the grader via github.