This document first describes the aims of this project followed by a brief overview. It then lists the requirements as explicitly as possible. It describes the files with which you have been provided. Finally, it provides some hints as to how the project requirements can be met.

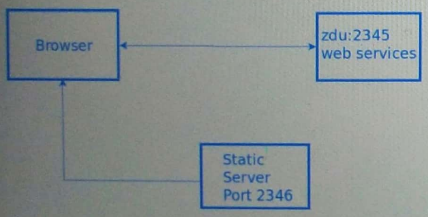
Aims

The aims of this project are as follows:

- To expose you to the DOM API.
- To implement *custom HTML elements*.
- To use the fetch API.

Overall Architectures

In this project you will be interacting with three programs shown in the following block diagram:



The project involves two servers:

- A server which implements web services for accounts. This is the server you essentially implemented in your previous project. You should use the server running on *<https://zdu.binghamton.edu:2345>* which is a solution to your project.
- A development server run by the *parcel bundler* on port 2346 which makes it possible to load all your project files into a browser. Note that this server is not used once the project has been loaded into your browser, making the project a *Single-Page App*.

All the code you implement in this project will run entirely within a browser (typically running on your VM). The JavaScript code will be loaded into the browser from the parcel server. Once loaded into the browser, this code will make direct requests to the web services server.

Requirements

You must push a `submit/prj4-sol` directory to your github repository such that typing `npm ci` within that directory followed by `npm start` will start a web server on `localhost` at port 2346. usually `1239`. This server should provide access to a **single-page app** which allows searching, creating and viewing accounts.

The operation of the app is illustrated in this video.

You will need to modify the provided accounts-app.mjs file to implement a custom HTML `<accounts-app>` component. You are being provided with almost all the HTML for your project either as a static string or generated dynamically by calling content functions.

Provided Files

The prj4-sol directory contains a start for your project. It contains the following files:

accounts-app.mjs

A skeleton file for your project. You will be doing much of your development within this file.

ws.mjs

A skeleton file which serves as a client for some of the web services developed in *Project 3*. You will be doing some development within this file.

html-content.mjs

A file which provides static and dynamic HTML for this project. You should not modify this file.

accounts-services.mjs

This provides all the services provided by the web services. The file is largely the same as from your previous project. You should not need to modify this file, but you will need to use the exported functions which are made available via the `services` parameter in `accounts-app`.

index.html

The top-level entry page for the application.

setup.html

A page which allows specifying the web services URL before entering the top-level page. Note that the default web services URL is `https://zdu.binghamton.edu:2345`.

setup.html

A page which allows specifying the web services URL before entering the top-level page. Note that the default web services URL is `https://zdu.binghamton.edu:2345`.

index.mjs

The top-level JavaScript file which is included by `index.html`. It sets up the web-services URL and then creates the `accounts-app` custom HTML element.

defs.mjs and *util.mjs*

Unchanged from your previous project.

accounts.css

A top-level stylesheet. You should not need to change 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.

Changes from Project 3

A minor change has been made to the accounts-services.mjs file to filter out parameters with empty values.

The solution to Project 3 has been deployed on `zdu:2345` with the following changes from the source code provided for Project 3:

- The `cors` plugin in `accounts-ws.mjs` has been modified to ensure that a `Location` header is returned from the server to the client.
- A bug in the handling of `INTERNAL_SERVER_ERROR`'s has been fixed.

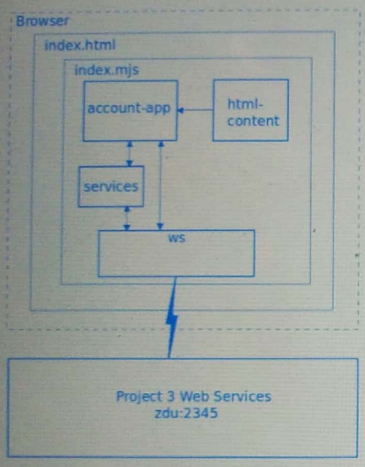
See the README in the solution to Project 3.

Application Architecture

The overall application architecture is as shown in the following figure:

Application Architecture

The overall application architecture is as shown in the following figure:



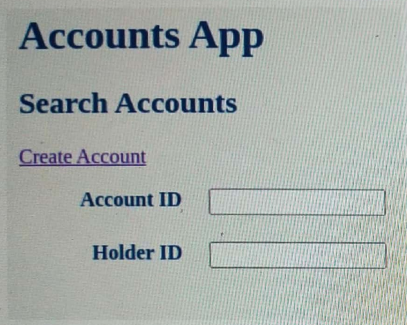
1. The main page `index.html` is loaded into the browser.
2. The `index.html` main page includes `index.mjs` as a module.
3. The code in `index.mjs` gets the URL (default `https://zdu.binghamton.edu:2345`) to use for Project 3 web services. It uses that URL to create an instance of `ws` which serves as a browser-side interface to the Project 3 web services.
4. The code in `index.mjs` uses the created `ws` instance to create an instance of the account-services (where `ws` plays the role of DAO from the previous project).
5. Finally, both the `services` and `ws` instances are used to create an `<account-app>` custom element.
6. The `account-app` paints the three application `sections` on the screen. Only one section is displayed at any point in time:

Accounts Search Section

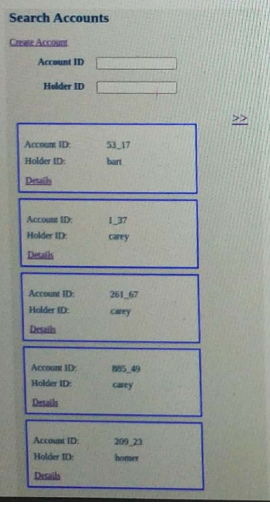
This section allows searching for an account by account `id` and `holderId`:

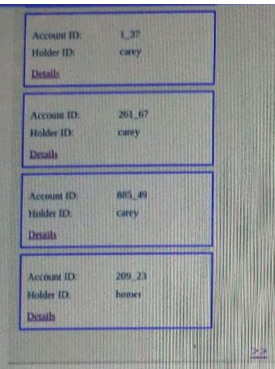
Accounts Search Section

This section allows searching for an account by account `id` and `holderId`:

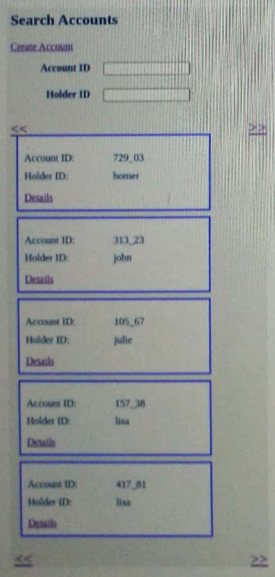


An accounts search is performed when a `blur` event is detected in any of the input widgets. The following figure shows the first set of search results for searching for all accounts (with both input widgets empty):





Note the use of the >> navigation widgets at the top and bottom of each set of search results. Clicking on one of those widgets results in the following screen:

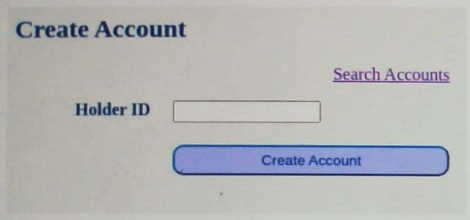


Now the navigation controls include the << to allow returning to the previous set of search results.

Now the navigation controls include the << to allow returning to the previous set of search results.

Account Create Section

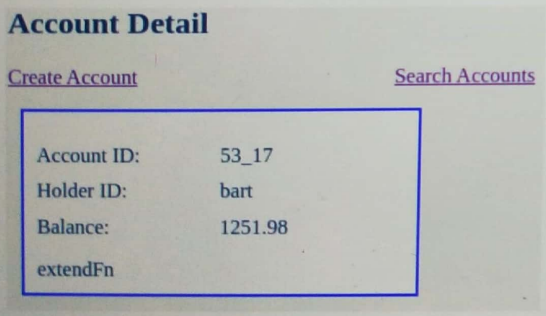
Clicking on the `Create Account` link in the **Accounts Search Section** results in the display of the **Account Create Section**:



Providing a Holder ID and clicking the button will result in creating a new account which is displayed on the **Account Detail Section** described next.

Account Detail Section

This section displays the details of an account including its `id`, `holderId` and `balance`:



The above figure shows the details of the first account from the search shown above.

The detail section contains an `extendFn` placeholder for the next project. It is currently set up to call a function when clicked. The provided function merely logs a message containing its parameters on the console.

Most of the HTML has been provided for you in the html-content.mjs module. This makes it easy for you to implement the content and styling which is not the emphasis of the project. The bulk of the HTML is static (exported as constant `HTML`) while other HTML elements are built dynamically by calling functions.

Error Architecture

Project 3 web services return an error response of the form:

```
{ status: Number,
  errors: [ { message, options?: { code?, widget? }, } ],
}
```

It is also possible that the web service itself fails, resulting in an exception. That exception will need to be converted into a response of the above form so that the consumer of the web services need not handle exceptions.

Given an erroneous response, the application needs to display the errors. There are two variations of errors:

1. The error object mentions a particular widget in its `options` property. In that case, the error message should be displayed adjacent to the offending widget on the page.

   This project has such a scenario when account creation is attempted without providing a holder ID.

2. The error object does not mention any particular widget in its `options` property. In that case, the error message is displayed in a separate area of the application display which is dedicated to such generic errors.

This project is set up for this error architecture using the provided `reportErrors()` function.

Browser API

It is a good idea to review some of the browser API's you will need to use in this project:

URL

The URL API provides an easy way to build properly escaped URLs. Of particular interest is the `searchParams` property which allows you to build up query parameters.

The usage will be something like

```
const url = new URL(path, wsBaseUrl);
for (const [k, v] of Object.entries(qParams)) {
```

```
const url = new URL(path, wsBaseUrl);
for (const [k, v] of Object.entries(qParams)) {
  url.searchParams.set(k, v);
}
//url.href contains properly escaped URL
```

Fetch API

The fetch() API will be used in this project to access the Project 3 web services from the `ws` module.

GET

```
fetch(url)
```

is used to perform an HTTP `GET`. The `url` can be built using the aforementioned `URL` to include query parameters. This is an asynchronous call returning a Response. A further asynchronous `json()` call on the returned response will return the web service result.

This can be used to access the account-info and accounts-search web services.

POST

```
fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data)
})
```

is used to perform an HTTP `POST` sending `data` as JSON in the request body. This will be used for the create-account web service. For that service, it is sufficient to merely look at the `Location` header:

```
const response =
  await fetch(url, { method: 'POST', ...} );
if (response.ok) { //status is 2xx
  return response.headers.get('Location');
}
else { //return response errors
  return await response.json();
}
```

DOM API

DOM API

To build HTML dynamically in the browser you may find some of the following APIs useful:

- *Web Components*. This project creates a custom `<accounts-app>` element which extends `HTMLElement`. It uses the shadow DOM to ensure isolation from the rest of the document.

  The project does not need to use any of the lifecycle methods of a custom element. All the set up of the element is done within the constructor.

- el.querySelector(sel): This returns the first element selected by CSS selector `sel` within element `el`.

- el.querySelectorAll(sel): This returns a `NodeList` containing all elements selected by CSS selector `sel` within element `el`.

  Note that `NodeList` is not a full-fledged JavaScript array. Though it does allow iteration using `for-of` and `forEach()`, it does not support array methods like `map()` and `join()`. A `nodeList` can be converted into a full-fledged array by spreading it into an array using `[...nodeList]`.

- addEventListener() can be used to add an event listener to a target element.

  The event listener (aka handler) will be called with an event `ev` as its first argument. It is important to distinguish between `ev.currentTarget` and `ev.target` as they may not be the same. The former refers to the element which registered the event, whereas the latter refers to the element which initiated the event.

  For example, assume that a `<button>` element contains a `<span>` element and the `<button>` element registers a `click` event. If the user clicks on the contained `<span>` element, then `ev.target` will reference the `<span>` element, whereas `ev.currentTarget` will reference the `<button>` element. This is important to keep in mind if the event handler tries to extract some attributes from the event target. See this *MDN article*.

  It is also important to understand `this` within an event handler. If the handler is defined using an arrow function, then as usual, `this` will be inherited from the context within which it was defined. OTOH, if the handler is defined using a `function` keyword, then `this` will be always be set to `ev.currentTarget`.

- *new FormData(formElement)*: This is convenient to collect all the data from the widgets within the form-element `formElement`. It can be converted into a plain JavaScript object using `Object.fromEntries()`.

- createElement(tagName): Can be used to create an element.

- *el.setAttribute(name, value)*: This can be used to set attribute *name* to *value* for element *el*.

- el.append(...elementOrText): The can be used to append one-or-more elements or text strings to the content of element `el`.

- el.classList: This read-only property provides access to an object representing the list of whitespace-separated identifiers in the `class` attribute of element `el`. Particularly useful methods on the object include `add(className)` and `remove(className)`.

Parcel Bundler

This project will use the parcel bundler to bundle multiple JavaScript, HTML and CSS files together and serve them to the browser in a single bundle. Parcel is given the root content of an app like an `index.html` page and bundles together all direct and indirect dependencies. It also provides a development server with *Hot Module Replacement* HMR which makes the development experience reasonably pleasant.

Parcel's HMR is not 100% reliable. If you are getting behavior which does not make sense, it is probably a good idea to stop the parcel server, remove the `.cache` and `dist` directories and restart with a clean slate.

If you have syntax errors in your code, the browser displays the errors on a black background. Again, if those errors don't make sense, it is a good idea to try restarting the parcel server to see if the errors disappear.

Hints

The following points are worth noting:

- There are two parts to this project:

    1. Accessing the Project 3 web services using the browser's `fetch()` API. For this project, you will need the `searchAccounts()` and `newAccount()` services as well as a generic `get()` service.

    2. Using the DOM API to add behavior to HTML. Mostly, you will not be writing HTML; you will be adding behavior to either static HTML or to dynamic HTML generated by calling functions with which you have been provided. You will need to set up the following handlers:

        a. Navigation handlers navigating between the different sections of the app by turning the visibility of the different sections on or off. This handler will be triggered by the `click` event on the navigation links present at the start of each section.

        This handler is simplest in that it is handled entirely within the app and does not require any calls to the web services.

        b. A handler which creates a new account, triggered by the `submit` event from the create account form.

        c. A handler which calls the `extendFn()` provided to the `account-app`. This handler will be triggered by a `click` event on the `extendFn` text within the account-detail display.

        d. A handler which searches accounts. This handler will be triggered by the `blur` event occurring on any of the input widgets in the search form.

        e. A handler which displays account details in the details section, triggered by a `click` event on the `Details` link displayed in the account-search results.

e. A handler which displays account details in the details section, triggered by a `click` event on the `Details` link displayed in the account-search results.

f. A handler which scrolls back-and-forth through account search results. This handler will be triggered by `click` events on the scroll widgets output before and after each set of search results.

- You could run this project using your own server for Project 3 web services by providing the URL using the provided setup.html file. However, the default uses the server installed on `zdu:2345`. You may use this server, but to avoid having multiple students stepping over each other please provide a distinctive part of your name as the holder ID when creating an account.

  Please note that the `zdu:2345` server may be periodically restarted, resulting in all newly created accounts being deleted.

- Note that your `accounts-app` is being painted within a `shadowRoot`. Hence you should treat `this.shadowRoot` rather than `document` as the root of your element hierarchy.

- Note that for security reasons, the browser will refuse to make calls to the web services because they use a locally generated certificate. To work around this limitation, point your browser directlu to the web services URL (`https://zdu.binghamton.edu:2345` if using the default) and click through to `Advanced` and tell the browser that you are okay with it accepting the local certificates.

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

1. Read the project requirements thoroughly. Look at the video to make sure you understand the necessary behavior. Review the material covered in class including the *authentication app*.

   Glance through the docs for APIs *linked earlier* in this document.

2. Set up your `prj4-sol` branch and `submit/prj4-sol` as per your previous projects.

3. Initialize your `package.json` using the `npm init -y` command, and set up your `scripts` section as follows:

```
"scripts": {
  "start": "parcel src/index.html  --port 2346",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

   Additionally, add in the following section after the `"scripts":` section:

```
"browserslist": [
  "since 2020-12"
],
```

   This ensures that parcel does not attempt to polyfill recent JavaScript features like `??`.

3. Initialize your `package.json` using the `npm init -y` command, and set up your `scripts` section as follows:

```
"scripts": {
  "start": "parcel src/index.html  --port 2346",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Additionally, add in the following section after the `"scripts":` section:

```
"browserslist": [
  "since 2020-12"
],
```

This ensures that parcel does not attempt to polyfill recent JavaScript features like `??`.

Then install `parcel-bundler` as a development dependency and `~/cs544/lib/cs544-js-utils` as a runtime dependency.

4. You should now be able to run the project:

```
$ npm start
```

This will run the parcel developmental server on port 2346. If you point your browser to `http://localhost:2346`, you should see the search section of your app being displayed without any of the necessary dynamic functionality.

5. Record the timestamp at which you copied the files by running the `changes.sh` script:

```
$ ~/cs544/bin/changes.sh ~/cs544/projects/prj4/prj4-sol
```

This ensures that subsequent runs of the script will report changes made after this time.

Note that the script will report changes if changes were made between the time the project was published and the time you started work on your project. Your copy should already contain those changes unless you were very unlucky and those changes were made between the time you performed the previous step and the time you performed this step!

Be sure the rerun this script each time you restart work on your project. If there are no changes, there should be no output. If there are changes, you will need to apply the diffs manually.

6. Replace the xxx entries in the `README` template and commit your project to github.

7. Fill out the skeleton `setNavHandlers()` method. Use the DOM querySelectorAll() method to select the **create** and **search** navigation links anywhere within the app's `shadowRoot`. Add a `click` handler which `selects()` the appropriate app section to each navigation link. Test.

8. Handle POST requests to the web services by implementing the generic `post()` method in `ws.mjs`.

7. Fill out the skeleton `setNavHandlers()` method. Use the DOM querySelectorAll() method to select the **create** and **search** navigation links anywhere within the app's `shadowRoot`. Add a `click` handler which `selects()` the appropriate app section to each navigation link. Test.

8. Handle POST requests to the web services by implementing the generic `_post()` method in `ws.mjs`.

   Note that `fetch()` returns non-2xx statuses as regular responses, but you will get an exception if you encounter a network error. Make sure you handle such exceptions by wrapping the above code within a `try`-block and convert the exception to our standard `{ errors: [ { message } ] }` object. You should probably also log the error to standard error.

9. Implement the `newAccount()` method in `ws.mjs`. This implementation will be a simple wrapper around the `_post()` method implemented in the previous step.

   Note that by implementing this method within ws, you have also implemented it in accounts-services since the latter is a wrapper around the former.

10. Implement the `setCreateHandler()` within `accounts-app.mjs`. You will need to select the create form from the app and set up a `submit` handler using addEventListener(). Don't forget to have the handler instruct the browser to prevent the submission of the form to the server. The handler should extract the form data from the form and send it on to the `services.newAccount()` method you implemented in the previous step.

    Ensure that you check for errors by calling `reportErrors()` on the result of the `newAccount()` call. Verify that this working by submitting an empty create form.

    We will need to defer handling a successful result until after the next step.

11. Implement the `get()` method in your copy of the `ws.mjs` file.

    a. Use the URL API to build up the url. First use the URL constructor to combine the relative `path` argument with the absolute `_urlBase` instance property. Then iterate through the key-value entries in `q` and each pair to the `url.searchParams` using its `set()` method.

    b. Call the web-service at the constructed URL using the fetch() API.

    Don't forget to catch any network errors using a `try-catch` as implemented in your `_post()` method.

12. Return to completing the handling for a successful account creation. by using the returned `Location` URL to display the account details in the detail section of the app. Since you will need to perform the identical action if the user clicks on the `Details` link in the search results, factor this out into an auxiliary method which takes the account URL as a parameter.

    The auxiliary method should perform a `get()` to the account URL. If there are no errors, it should extract the `id`, `holderId` and `balance` from the returned result. It should build an HTML element for the details using the provided `makeAccountDetail()` function and insert it into the detail section.

13. Add a click handler to the `extendFn` you just added in the previous step. Set up your click handler to call the provided `extendFn` parameter to your `accounts-app` component passing in the account-id and the DOM id of the `extendFn` element. Test by ensuring that an appropriate message appears in

13. Add a click handler to the `extendFn` you just added in the previous step. Set up your click handler to call the provided `extendFn` parameter to your `accounts-app` component passing in the account-id and the DOM id of the `extendFn` element. Test by ensuring that an appropriate message appears in the console log.

14. In `ws.mjs` implement the `searchAccounts()` methods as a trivial wrapper around the previously implemented `get()` method.

    Note that by implementing this method within ws, you have also implemented it in accounts-services since the latter is a wrapper around the former.

    Test your implementation by adding a call to `service.searchAccounts()` within an asynchronous IIFE in the accounts-app constructor, simply logging the result of the call to the console.

15. Attach blur handlers to the input widgets in the search-form.

    a. Select the input widgets in the search-form by calling the querySelectorAll() method on the `shadowRoot` of the app-component with a suitable CSS

    b. Attach `blur` handlers to each of the selected widgets using addEventListener(). For now simply have your handler log a message to the console. Test by blurring one of the search widgets.

    c. Change your `blur` handler to call the skeleton `search()` method instead.

    d. Implement the `search()` method to distinguish whether or not the `url` parameter is provided.

    - If it is not provided then use the form-data from the search form to call the previously implemented `service.searchAccounts()` method.
    - If provided, then call `get(url)`.

    In both cases, you will have a result of the form:

```
{ links: [ SelfLink, NextLink?, PrevLink? ],
  result: [ { links: [ SelfLink ],
              result: { id, holderId },
            },
          ],
}
```

    e. If there are no results, display a message like `No results` in the results area of the search section.

    f. If there are one-or-more results, extract the `id`, `holderId` and `SelfLink.href` of each result, and use the provided `makeSearchResult()` function to create an element containing the result. Then append() that result to the results area of the search section.

    Test. It is probably a good idea to factor out the results display into a separate method since the search handler will need to add event handlers to

f. If there are one-or-more results, extract the `id`, `holderId` and `SelfLink.href` of each result, and use the provided `makeSearchResult()` function to create an element containing the result. Then append() that result to the results area of the search section.

   Test. It is probably a good idea to factor out the results display into a separate method since the search handler will need to add event handlers to the search results for account details and scrolling results.

16. For each search result attach a click handler to its `Details` element which simply calls your previously implemented auxiliary function to display account details given the URL for the account.

17. Use the provided `makeScrollElement()` to add a scroll element before and after the search results.

18. Add a `click` handler to each scroll control which simply calls your previously implemented `search()` method passing in the url for the scroll link.

19. Iterate until you meet all requirements.

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 `prj4-sol` branch into your `main` branch and submit your project to the TA via github.

Submit as per your previous project. Before submitting, please update your README to document the status of your project:

- Include the hash of your last commit in the `README`. Since you will now need to commit this `README` change, it won't be the absolutely last commit hash, but will let the TA know that your submission is the one you want graded.
- Document known problems. If there are no known problems, explicitly state so.
- Anything else which you feel is noteworthy about your submission.

If you want to make sure that your github submission is complete, clone your github repo into a new directory, say `~/tmp`. Since the project depends on the file system to provide the cs544 library packages, you should ensure that this clone repository can access them. This can be done by simply setting up a symlink to the `cs544` course directory at the same level as your cloned github project:

```
$ cd ~/tmp
$ ln -s ~/cs544 .
```

This should ensure that the relative links in the project's `package.json` file can find the cs544 library package. You should now be able to do a `npm ci` to build and run your project.

To be turned in as a PDF using the submission link which will be set up on mycourses. If you write code, you can include the code as separate files.

The paper may be on any topic relevant to the course not explicitly covered in class. Some examples:

- Details of non-trivial JavaScript features not covered in class.
- Comparison of JavaScript's prototype inheritance with traditional OO `class`'es.
- A summary giving technical details of any web programming framework or technology not covered in class.

Cutting-and-pasting of material from external sources is not permitted except for clearly marked short quotations. The paper **must** contain a reference section specifying **all** references used.

The paper will be graded between a 1 and a 3. An attempt at some original work gives you a better chance of getting a 3; an example would be writing some code to illustrate or measure some concepts.

The body of the paper may not exceed 4 pages. This limit does not apply to any supplementary material like references, code listings or logs.