OWASP OWTF - Web Interface Enhancements

MOHIT SHARMA

ms892075@gmail.com

TABLE OF CONTENTS

OVERVIEW

The current web interface of OWTF is non-functional and some of its pages are not yet implemented. This project is about implementing a full functional and responsive webui for all the pages of the app written in ReactJs with Redux as its state manager. The project also includes introducing new features in the app with refinement of the current layouts ensuring excellent reliability and performance. Implementing an automated testing environment with a good Unit/Integration test coverage is also an important part of the project. Project also demands adding Typescript to the app (if time permits) to eliminate a large no. of errors from the code.

PRE-GSoC PROJECT INVOLVEMENT

I am constantly in touch with this project and quite familiar with it, below are my contributions to this project:-

Pull requests:

- A few basic unit tests written in Jest and Enzyme are added and babel upgraded to v7.0(required for babel-jest)
    - Initial tests added and babel upgraded to v7.0 https://github.com/owtf/owtf/pull/1017 (PULL)
- Implemented the Transaction Page using Evergreen components with minor bug fixes.
    - Implemented Transaction Page in evergreen-ui https://github.com/owtf/owtf/pull/1011 (PULL)
- Implemented the Settings Page using Evergreen components.
    - Settings Page ported to evergreen-ui https://github.com/owtf/owtf/pull/1008 (PULL)
- Implemented the Help Page using Evergreen components.
    - Help page ported to evergreen-ui

https://github.com/owtf/owtf/pull/1006 (PULL)

- Porting Targets page to react using react-bootstrap, along with APIs calling(for backend) using react-sagas.
    - Implements Reports Page in the React application https://github.com/owtf/owtf/pull/999 (PULL)
- Porting Targets page to react using react-bootstrap, along with APIs calling(for backened) using react-sagas.
    - Implements Targets Page in the React application https://github.com/owtf/owtf/pull/990 (PULL)
- Porting Transaction page to react using react-bootstrap, along with APIs calling(for backened) using react-sagas.
    - Implements Transactions Page in the React application https://github.com/owtf/owtf/pull/989 (PULL)
- Ported settings page to react using react-bootstrap, along with configurations load and patch using react-sagas.
    - Implements SettingsPage in the React application https://github.com/owtf/owtf/pull/987 (PULL)
- Code for creating a new session has been implemented and backend functionality for deleting a session has been added along with documentation.
    - Adds session creation and deletion functionality https://github.com/owtf/owtf/pull/980 (PULL)
- After applying filter for a work from the worklist and pressing delete button, work having the selected type will get deleted instead of all the works.
    - Delete selected filter in worklist added https://github.com/owtf/owtf/pull/930 (PULL)

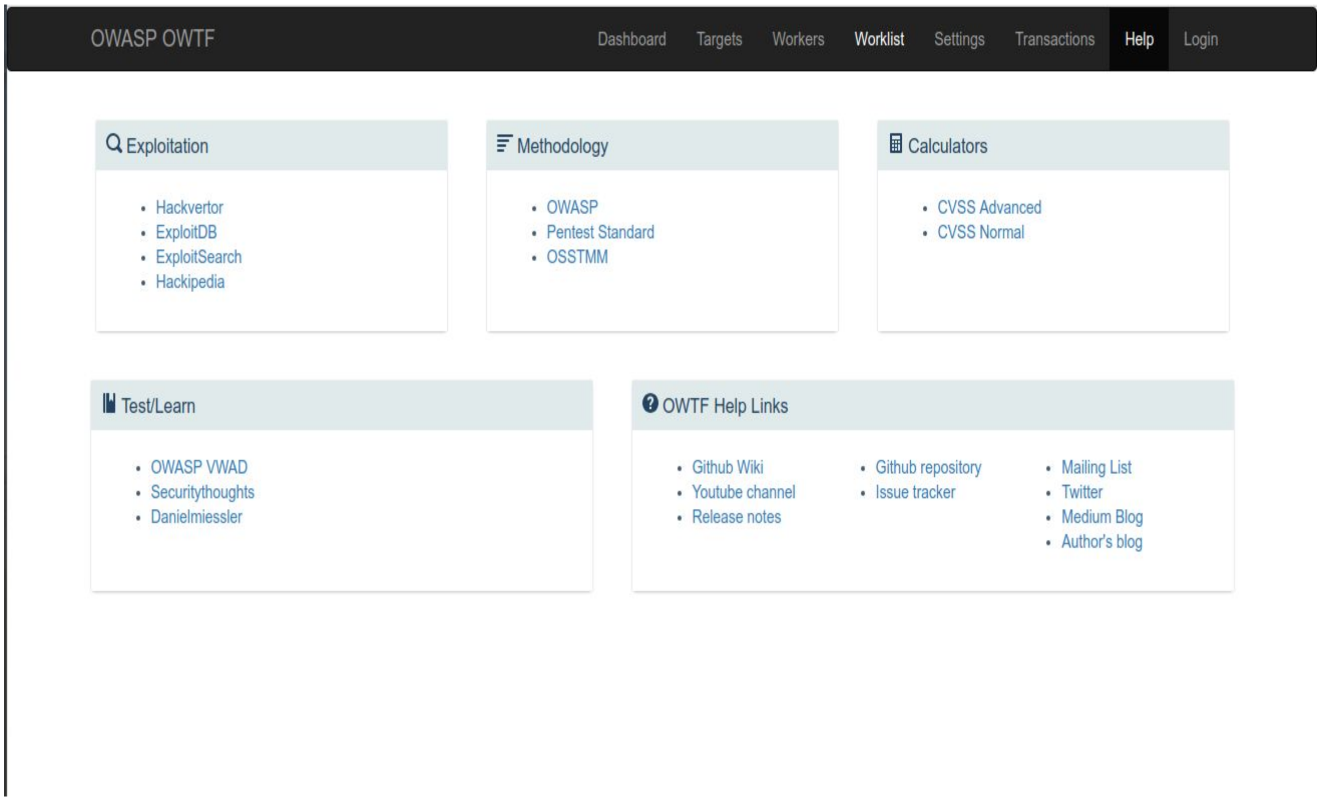
Reported Issues:

- Improper orientation of page describing individual targets https://github.com/owtf/owtf/issues/926

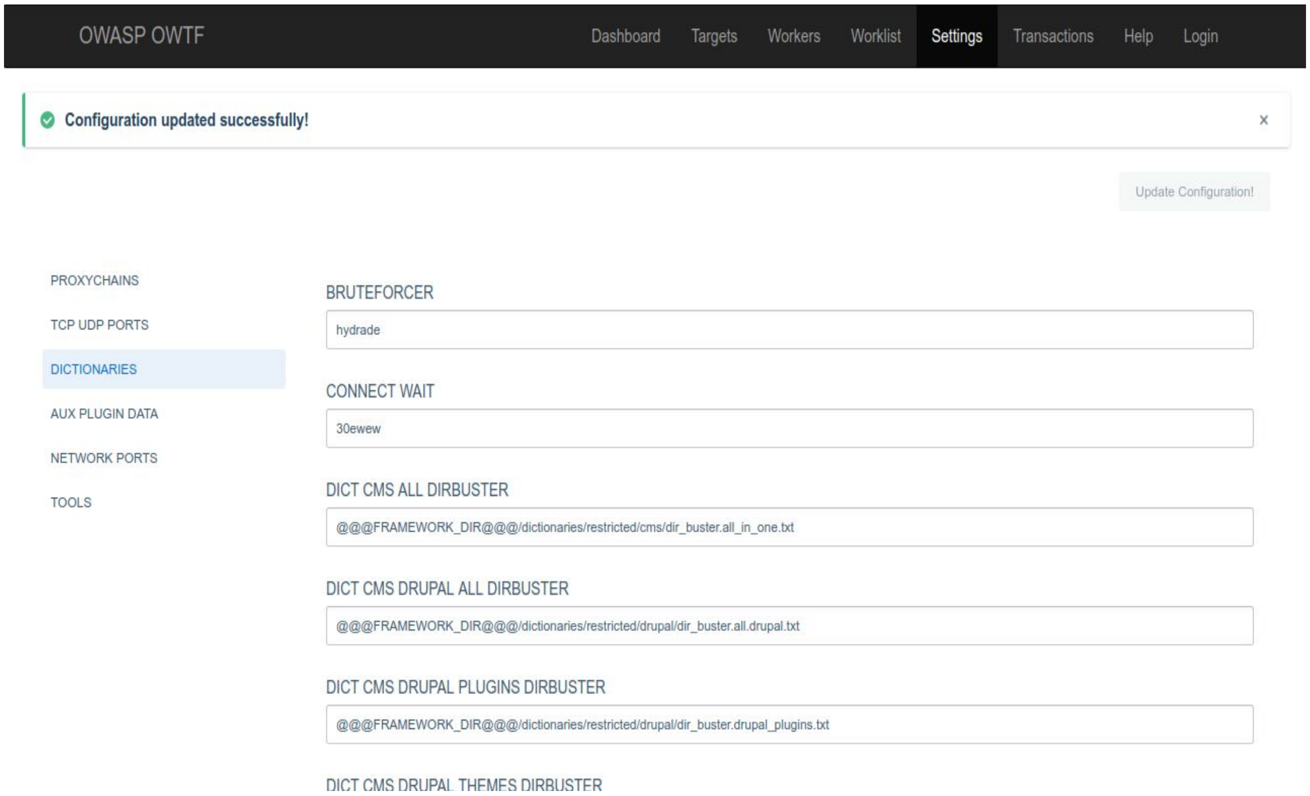
PRE-GSoC IMPLEMENTATION RESEARCH

I've been looking at the older version of the OWTF (master branch) to get an idea of how the UI is working and what's the best way to implement the same functionality in the new UI using React-Redux with excellent user experience. The old webui was implemented in React along with some jinja templates with styling done using **vanilla-bootstrapping**. The best way of getting a good understanding of the project is to implement a small portion of the UI in react-redux. So I began with porting some simple pages to React with UI components built using **Evergreen-ui**. The final layouts of the ported pages are shown below:-

Help page



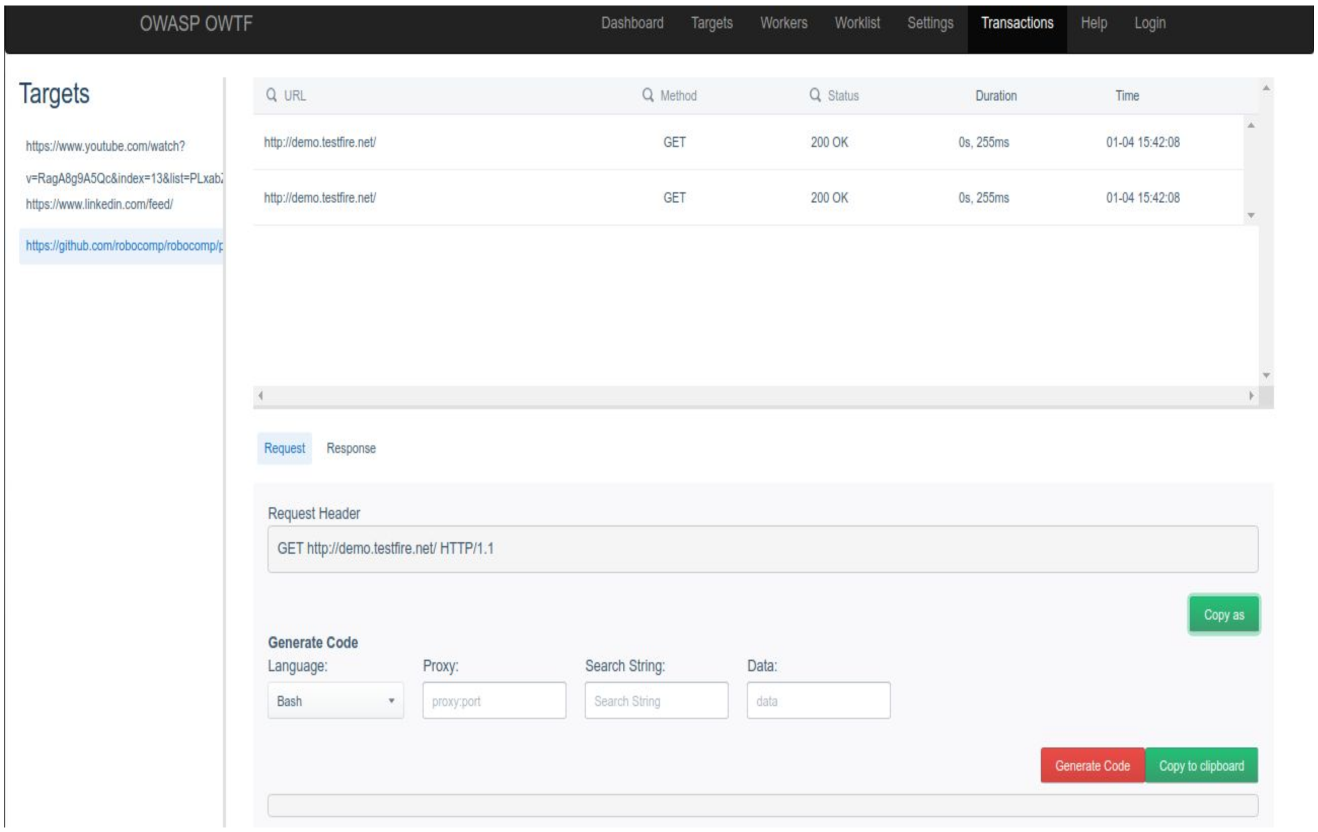
Settings Page



Configuration updated successfully!

BRUTEFORCER	hydrade
CONNECT WAIT	30ewew
DICT CMS ALL DIRBUSTER	@@@FRAMEWORK_DIR@@@/dictionaries/restricted/cms/dir_buster.all_in_one.txt
DICT CMS DRUPAL ALL DIRBUSTER	@@@FRAMEWORK_DIR@@@/dictionaries/restricted/drupal/dir_buster.all.drupal.txt
DICT CMS DRUPAL PLUGINS DIRBUSTER	@@@FRAMEWORK_DIR@@@/dictionaries/restricted/drupal/dir_buster.drupal_plugins.txt
DICT CMS DRUPAL THEMES DIRBUSTER	

Transaction Page



https://www.youtube.com/watch?v=RagA8g9A5Qc&index=13&list=PLxab
https://www.linkedin.com/feed/
https://github.com/robocomp/robocomp/

URL	Method	Status	Duration	Time
http://demo.testfire.net/	GET	200 OK	0s, 255ms	01-04 15:42:08
http://demo.testfire.net/	GET	200 OK	0s, 255ms	01-04 15:42:08

Request Header

GET http://demo.testfire.net/ HTTP/1.1

Generate Code

Language:	Proxy:	Search String:	Data:
Bash	proxy:port	Search String	data

IMPLEMENTATION PLAN

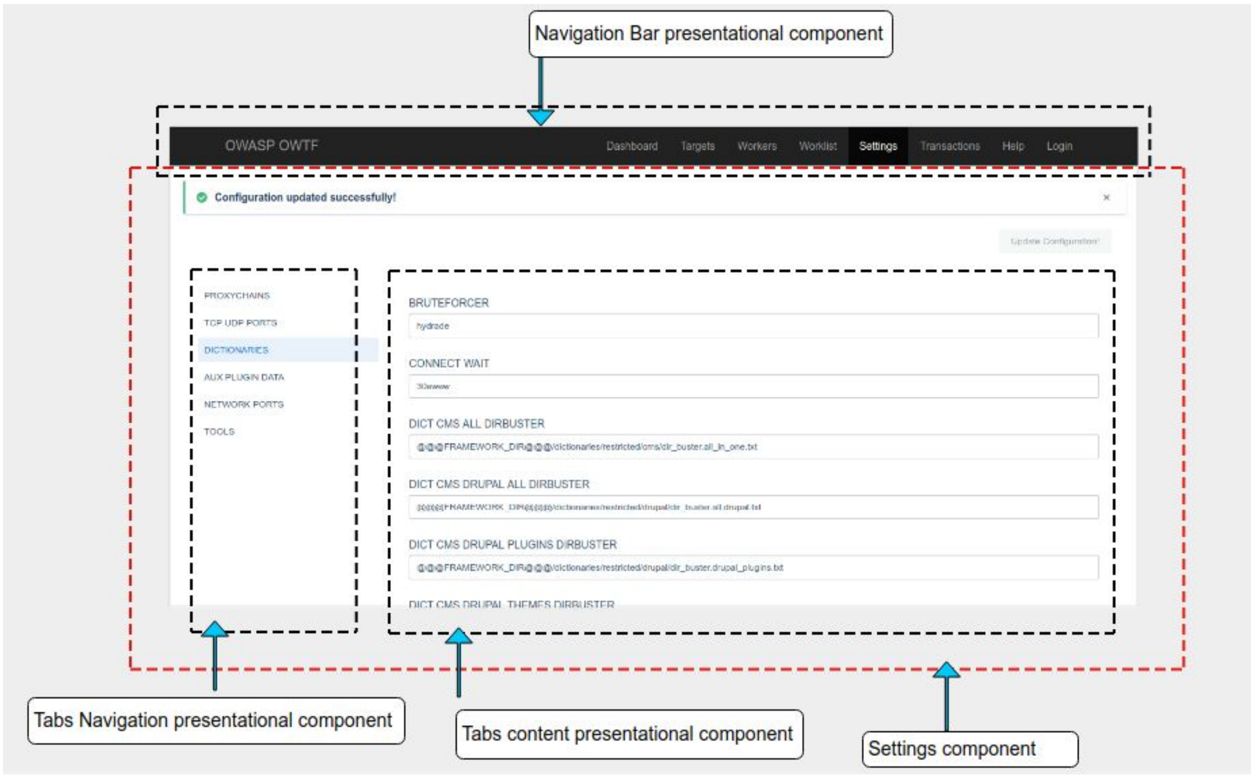
SWITCHING UI TO REACT-REDUX

1. Breakdown of the individual pages into components

The main part of the project is to divide each page of the webapp into chunks of components based on the overall purpose of each component. To write a more clean and maintainable code, it is beneficial to keep React and Redux logic separately which further divide components into -

- **Presentational Components -** Concerned with how things look. Their only function is presentational markup. In a Redux-powered app, a presentational component does not interact with the Redux store.
- **Container Components -** Deals with Redux logic, dispatch "Actions" and more. It passes the data to the presentational component via **props**, handle events, deal with React on behalf of Presentational component.

  Ex :- The breakdown of the **Settings Page** into presentational and container components is shown below:



Configuration updated successfully!

BRUTEFORCER
hydrade

CONNECT WAIT
30wwww

DICT CMS ALL DIRBUSTER
@@@FRAMEWORK_DIR@@@/dictionaries/restricted/cms/dir_buster.all_in_one.txt

DICT CMS DRUPAL ALL DIRBUSTER
@@@FRAMEWORK_DIR@@@/dictionaries/restricted/drupal/dir_buster.all.drupal.txt

DICT CMS DRUPAL PLUGINS DIRBUSTER
@@@FRAMEWORK_DIR@@@/dictionaries/restricted/drupal/dir_buster.drupal_plugins.txt

DICT CMS DRUPAL THEMES DIRBUSTER

Legend: Black dotted lines = “Presentational” components. Red dotted lines = “Container” components.

2. **Writing async Actions creators and Reducers while making Api calls with redux-saga (Async Redux)**

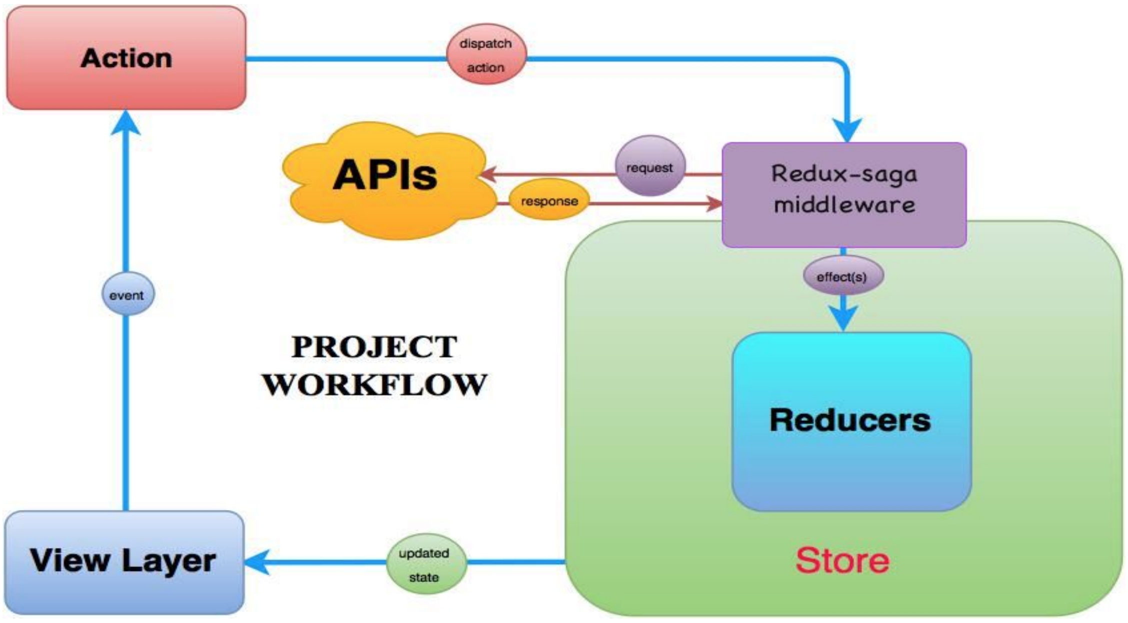
In order for our app to function, it needs to make Api calls and reflect the outcome in the components. Since Redux out of the box supports only synchronous actions, the standard way to do make asynchronous api calls with Redux is to use **Redux Middlewares** (middleware is what comes in between dispatching an action and updating the store) like redux-thunk, redux-promise, redux-saga.

I’ll be using redux-saga to make api calls as it allows us to test our asynchronous flows easily and also maintains purity of our actions as Saga’s listen independently to actions unlike in the case of thunks.

Redux-saga is designed to make handling side effects in our redux app nice and simple. It achieves this by leveraging an ES6 feature called **Generators**, allowing us to write asynchronous code that looks synchronous, and is very easy to test.

The workflow of async redux redux is as follows:

- A component calls an action creator
- The action creator then emits an action using a specific type
- The watcher sagas all listen for any actions emitted and intercept an action that it is listening for. It then calls the appropriate worker saga.
- The worker saga makes an API call and dispatches an action to the reducers with the type of action and the payload.
- The reducer listens for any dispatched actions and if it matches, it then uses the supplied data to update the state in the store.



Courtesy -Link

The App has generally three states:

STATE(before call) →STATE(during call) →STATE(answer received)

There will be three action creators for every API call.

Ex - LOAD_TARGETS(called during api call),

LOAD_TARGET_SUCCESS(called when the response is received successfully),

LOAD_TARGET_ERROR (called when there is an error while receiving the response).

A basic reducer for an api call will look something like this-

```
function targetsLoadReducer(state = {}, action) {
   switch (action.type) {
     case LOAD_TARGETS:
       // Return default state
     case LOAD_TARGETS_SUCCESS:
       // Return new state (with response received)
     case LOAD_TARGETS_ERROR:
       // Return new state (with error received)
     default:
       return state;
   }
}
```

3. **Enhancing performance using Selectors and Reselect**

   Consider the following **problem**: Suppose the **Targets Page** of our webapp has 3 types of inputs:

   - URLs of the targets added
   - No. of targets
   - Severity of each target

   The problem is that whenever the state of any of the inputs is modified (a new target is added, a target URL is changed, or the selected state is changed), everything will need to be recalculated and rerendered. This would be very problematic if we had hundreds of targets. To avoid such redundancy we use Reselect**.**

   Reselect is a library for building memoized selectors. We define selectors as the functions that retrieve snippets of the Redux state for our React components. Selectors are beneficial because they encapsulate knowledge of where to find that particular subset of data and are also reusable and flexible. Using memoization, we can prevent unnecessary rerenders and recalculations of derived data which in turn will speed up our application.

   Reselect's Syntax: Reselect offers up a function called *createSelector()* that's used to create memoized selectors. Below is an example code that get **load** state from the targets data:-

```
import { createSelector } from 'reselect';

const selectTarget = (state) => state.get('targets');

const makeSelectFetch = createSelector(
 selectTarget,
 (targetState) => targetState.get('load')
);
```

   createSelector is a function that takes two arguments:

   1. Selector(s) — selectTarget in the above example.

2. A transformer function that takes the values of the selectors from the first arguments and uses them to select or derive relevant data

Using Reselect to retrieve state in a component: Reselect offers a *createStructuredSelector()* function that is being passed to connect. It is most helpful to use in components that are pulling in a number of selectors.

```
const mapStateToProps = createStructuredSelector({
 targets: makeSelectFetchTargets,
 fetchLoading: makeSelectFetchLoading,
 fetchError: makeSelectFetchError,
 });
```

IMPLEMENTING AN AUTOMATED TESTING ENVIRONMENT USING JEST AND ENZYME

Jest: Jest is a testing library that can be used to test simple Javascript code or React components. This is done through simple API that Jest provides to users. Using these APIs it is possible to make assertions on how functions should behave and then test our expected outcome against the test outcome. These are some major highlights of Jest:

- It is very easy to set up and is usable right out of the box.
- Ability to run tests in parallel. For a huge app with lot of tests, this feature prove to be very beneficial.
- Snapshot testing is the most powerful feature of Jest. It helps in reducing the number of tests we have to write as it creates a snapshots of our code and if anything changes in the component, It will throw an error when the snapshot is generated the next time.
- Code coverage is available right out of the box. Jest can collect code coverage information from entire projects, including untested files.
- In-built Manual mocking. Jest allows us to mock any object outside of our test’s scope by using a custom resolver for imports.

Enzyme: It is another library commonly used with Jest. With Enzyme we can create a mock DOM to test whether components are rendered correctly, and whether they behave correctly when acted upon. Enzyme's mock rendering can mainly be done in two ways:-

1. *Shallow rendering* is used when doing unit testing of a component. This allows for tests to focus only on that component and how it functions, without caring about other components it might interact with.
2. *Full DOM rendering* involves rendering of component and all children components. The allows for more in-depth testing to see how our components on the DOM interact with each other.

Setting up Jest & Enzyme

The following packages are required to be added as devDependencies in the **package.json** file:

```
$ npm install --save-dev jest
$ npm install --save-dev babel-jest
$ npm install --save-dev enzyme
$ npm install --save-dev enzyme-adapter-react-16
$ npm install --save-dev react-test-renderer
```

* jest — Unit Testing framework for ReactJs developed by Facebook.

* babel-jest — To support ES6 and ES7 for our tests.

* enzyme — JS testing utility developed by Airbnb to make it easier to assert React Components.

* enzyme-adapter-react-16 —To provide compatibility with React 16.x.

* react-test-renderer — Used to grab snapshot of DOM tree rendered by React DOM/ React Native components

The **scripts** in package.json also needs to be modified to run the tests:

```
...
"scripts": {
  ...
  "test": jest,
  "test:watch": "npm test -- --watch"
}
...
```

A **src/setupTests.js** file also needs to be present in order to use Enzyme, which essentially configures Enzyme to run with React using the adaptor we just installed.

```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
```

Writing Tests

There are 3 naming conventions we can adopt in order for Jest to pick up our tests:

- Any file with a **.test.js** suffix or a **.spec.js** suffix. (I am currently using .test.js convention)
- Any .js file within **__tests__** folders throughout your project.

I have written down initial unit tests for some of the very basic components (like NotFoundPage & NavigationBar components) to get a gist of how unit testing actually works. Shown below is a snippet from the **NotFoundPage.test.js** file that tests the correct rendering of the page:-

```
describe('NotFoundPage Component', () => {

   it('Should render without errors', () => {
     const component = shallow(<NotFoundPage />);
     expect(component.find('p').length).toBe(1);
     expect(component.find('p').text()).toBe('Page Not Found');
   });

});
```

Let’s break down the above example to understand the syntax:

- *describe()*: An optional method to wrap a group of tests with describe() allows us to write some text that explains the nature of the group of tests conducted within it.
- *it()*: Similar in nature to describe(), it() allows us to write some text describing what a test should successfully achieve.
- *expect() & .toEqual():* Here we carry out the test itself. The expect() method carries a result of a function, and toEqual(), in this case, carries a value that expect() should match.

Snapshot Testing with Jest

Snapshot testing is a useful feature to make sure that our markup does not unexpectedly change, and equally as true, makes sure that render() outputs what we intended it to.

A snapshot file(**.snap**) is automatically generated in a **__snapshots__** folder when the **toMatchSnapshot()** method is called within our tests, often in conjunction with expect(). The snapshot test for the NotFoundPage is shown below:

```
it('NotFoundPage should render without errors', () => {
     const component = shallow(<NotFoundPage />);
     const tree = toJson(component);
     expect(tree).toMatchSnapshot();
   });
```

The contents of a snapshot file mostly consists of markup that represents the expected output of the NotFoundPage component.

Note: Explained above is just a basic idea on how to start writing unit tests for our app. There is lot more when it come to testing our entire application like testing the connected components, the action creators, the reducers, the sagas etc all of which can not be summarized here. I have gone through a lot of react app testing tutorials most the their links are provided in the **References** below.

REFINING LAYOUTS AND ADDING NEW FEATURES/COMPONENTS

For ideas related to layout, I have gone through various widely used web application testing frameworks like **Arachni**, **Probely** etc, read about design patterns that reduces user efforts. I have also looked at some other commercial websites that provide wonder user experience like **Netflix**, **Dropbox** etc. My primary aim would be to make the UI user friendly as much as possible without introducing a bunch of unnecessary new features. I'll be mainly using **Evergreen-ui** (by segment) to style the individual pages along with **React-Bootstrap.**

Here are the mockups of some important pages of the app along with the details of their UI components:

1) Dashboard

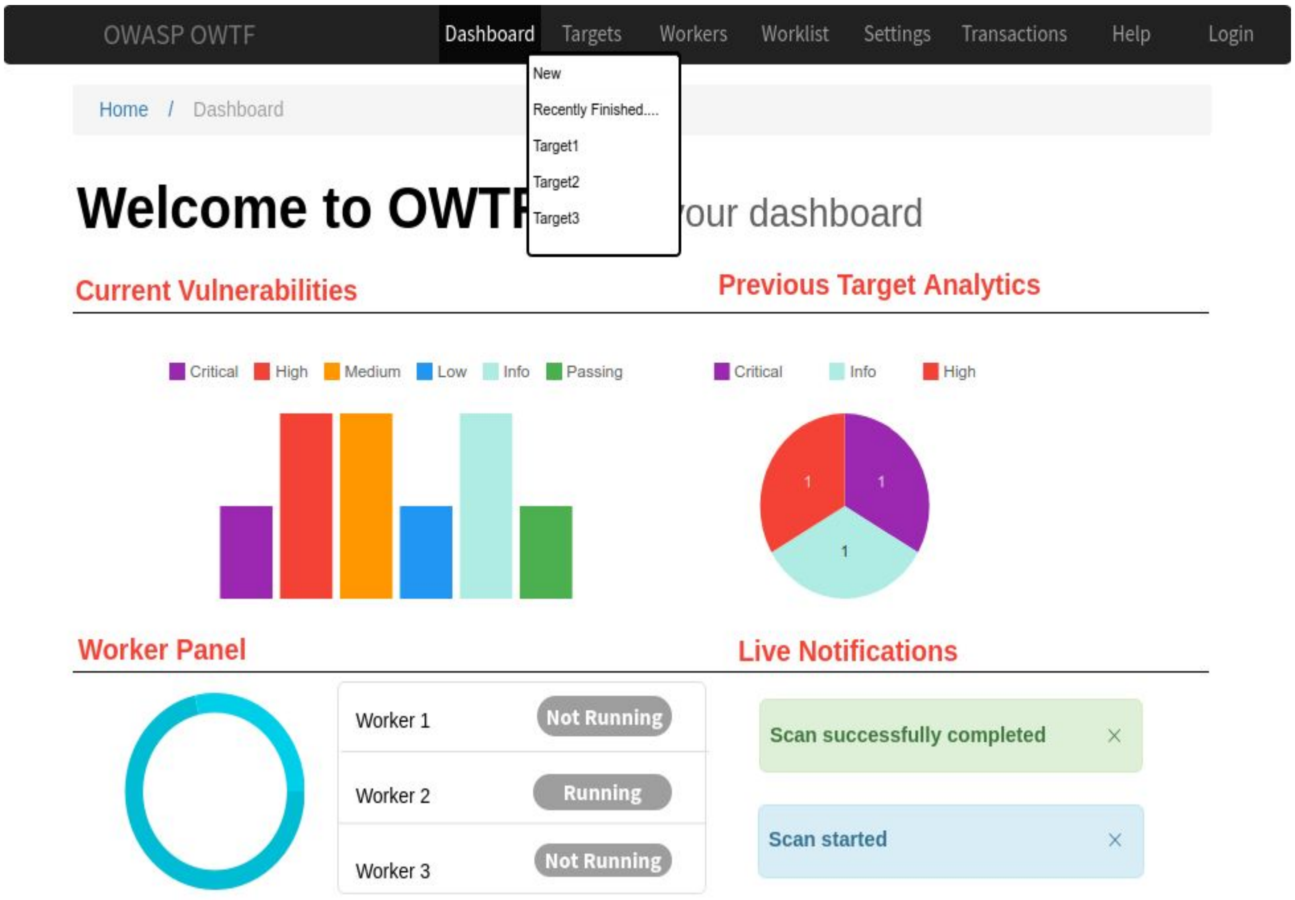
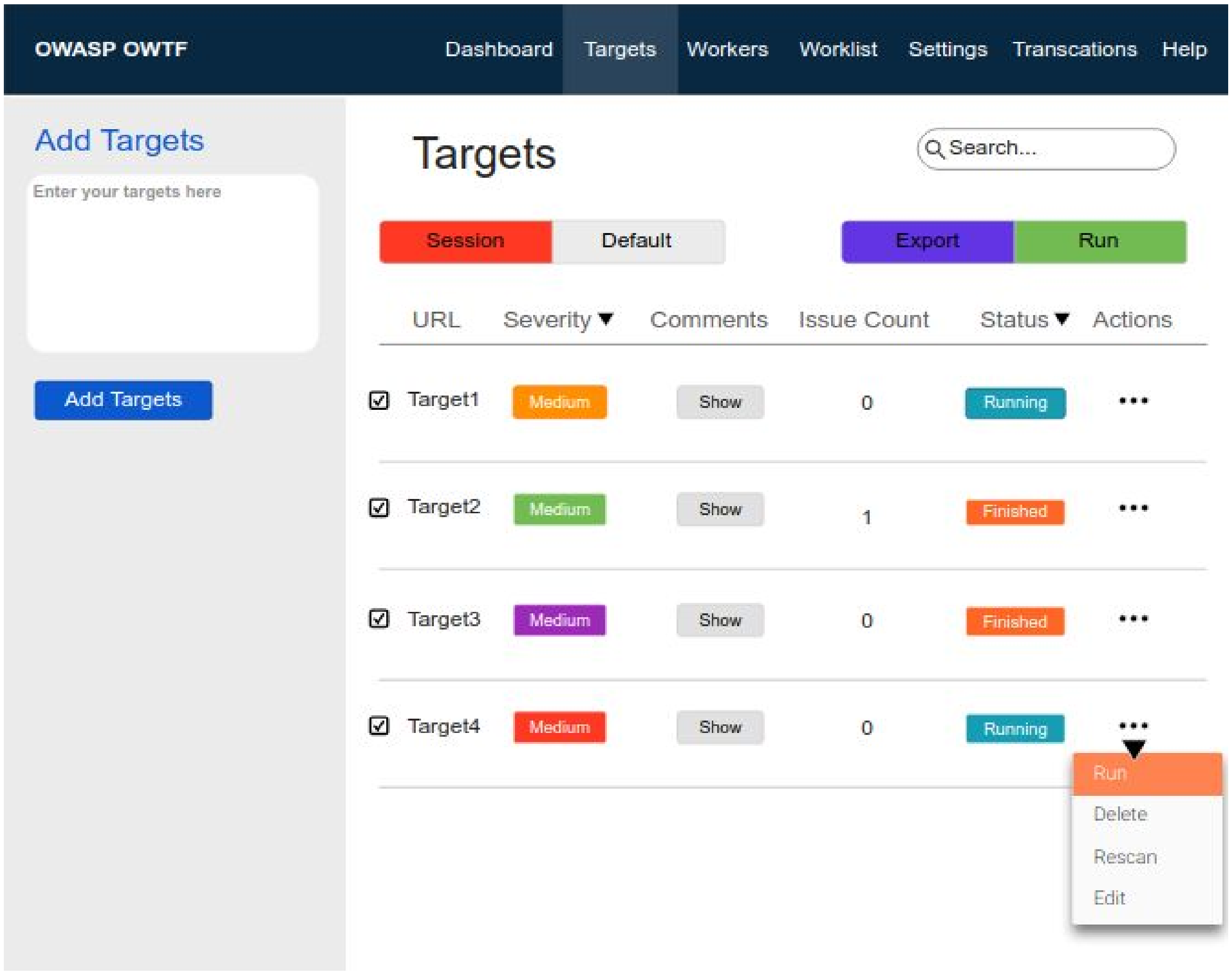


Figure 1

The major components of the Dashboard will include:

- **NavigationBar** which will present on all the pages. The target link will contain a **select menu** (evergreen-ui) which will take the user directly to the recently finished target and allows us to add a new target.
- **Breadcrumb** (react-bootstrap) present on every page for easy navigation between pages.
- **Pie chart** (Pie-react-chart.js) showing the analytics of the previous finished target.
- **Current Vulnerabilities bar chart** (Bar-react-chart.js) showing the count of all target severities**.**
- **Worker Progress bar** (doughnut-react-chart.js) for all workers along with a **table** showing the current state of each worker.
- **Floating live notification panel** (Toaster-evergreen-ui) that notifies the completion/start of target scan or worker.

2) Targets Page



	URL	Severity ▼	Comments	Issue Count	Status ▼	Actions
✓	Target1	Medium	Show	0	Running	•••
✓	Target2	Medium	Show	1	Finished	•••
✓	Target3	Medium	Show	0	Finished	•••
✓	Target4	Medium	Show	0	Running	•••

Figure 2

Components on the target page will include:

- **Add Target Panel** (Collapse-react-bootstrap) which will take a list of target urls and add it in the target list after validation. I wish to make this component collapsible.
- **Targets Table** (Table-evergreen-ui) which contains list of all added targets .All the headers in the tables will be search boxes so that filter the targets according to severity & status.
- Each **Target Row** shows details and actions for all the targets. The details include - **Comments** about the targets (key points to remember), **Status**(Running or completed), **Actions** - Delete, Rescan, Edit, Delete from

session. These details will provide users sufficient information/functions related to the target without navigating to the individual target page.

3) Report Page

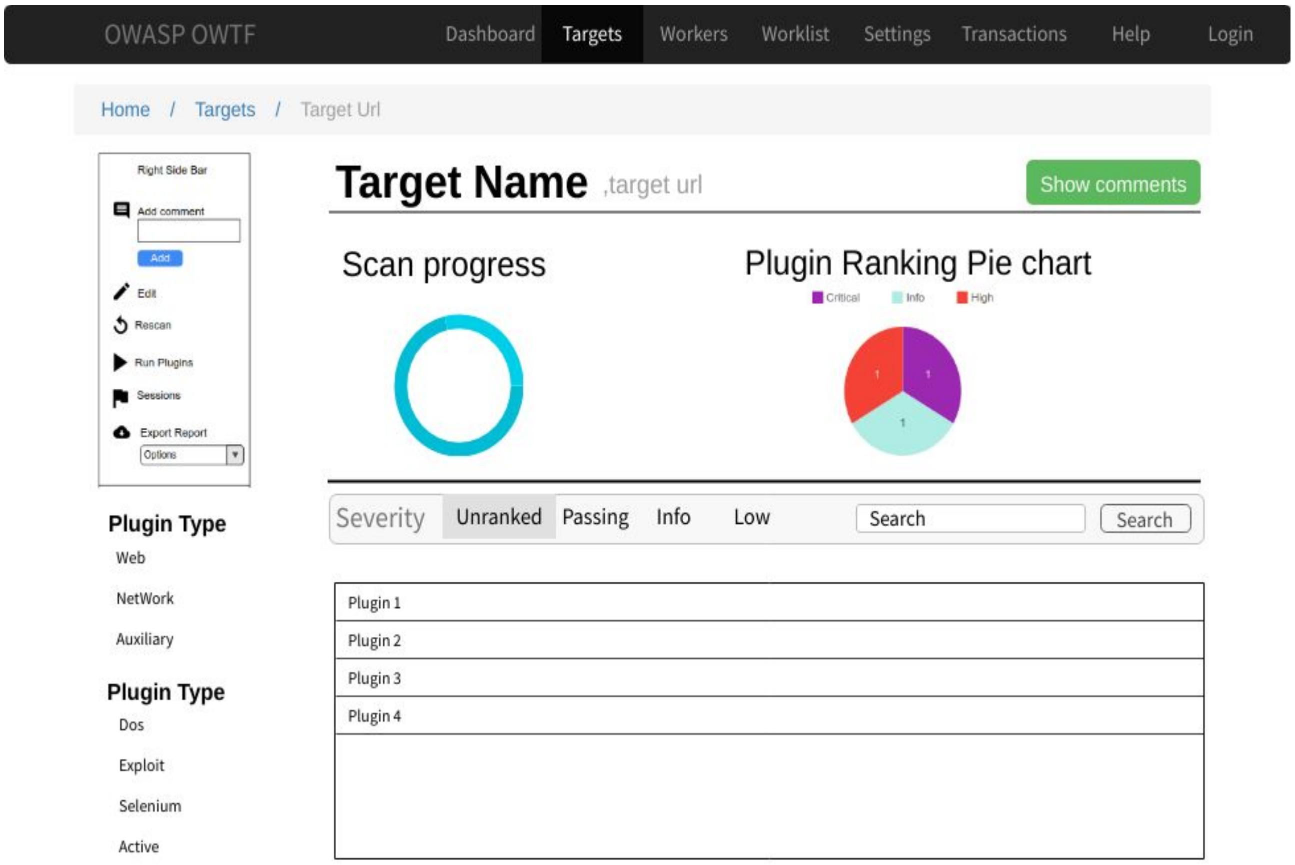


Figure 3

Features/components on the report page include:

- **Pie chart** (Pie-react-chart.js) representing plugin rankings in a chart form help in measuring target severity (low, medium etc).
- **Progress bar** (Doughnut-react-chart.js) showing progress of target scan.
- Plugins section will contain:
    - Various **Tab Navigation Panel** (Tabs-evergreen-ui) for plugins filtering.
    - **Plugin list** component which will show plugin information when clicked in the form of a **side sheet** (evergreen-ui)**.**
- There will be **Sidebar** which contains the following features like add comment, edit, rescan, run plugins, change sessions, export report (in different formats) for targets.

- **Show Comments** button showing all the comments for target using a dialog(evergreen-ui).

4) Plugin Side Sheet

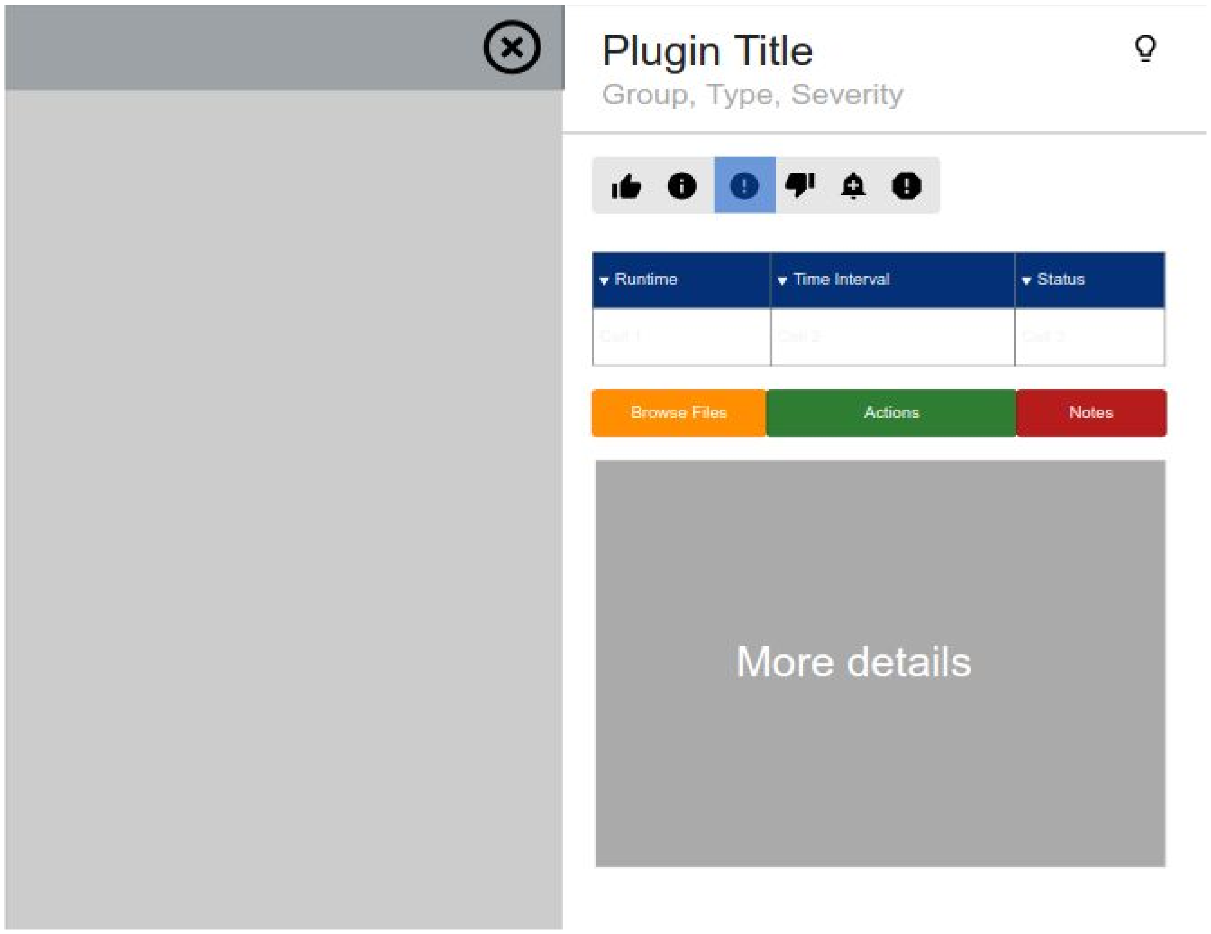
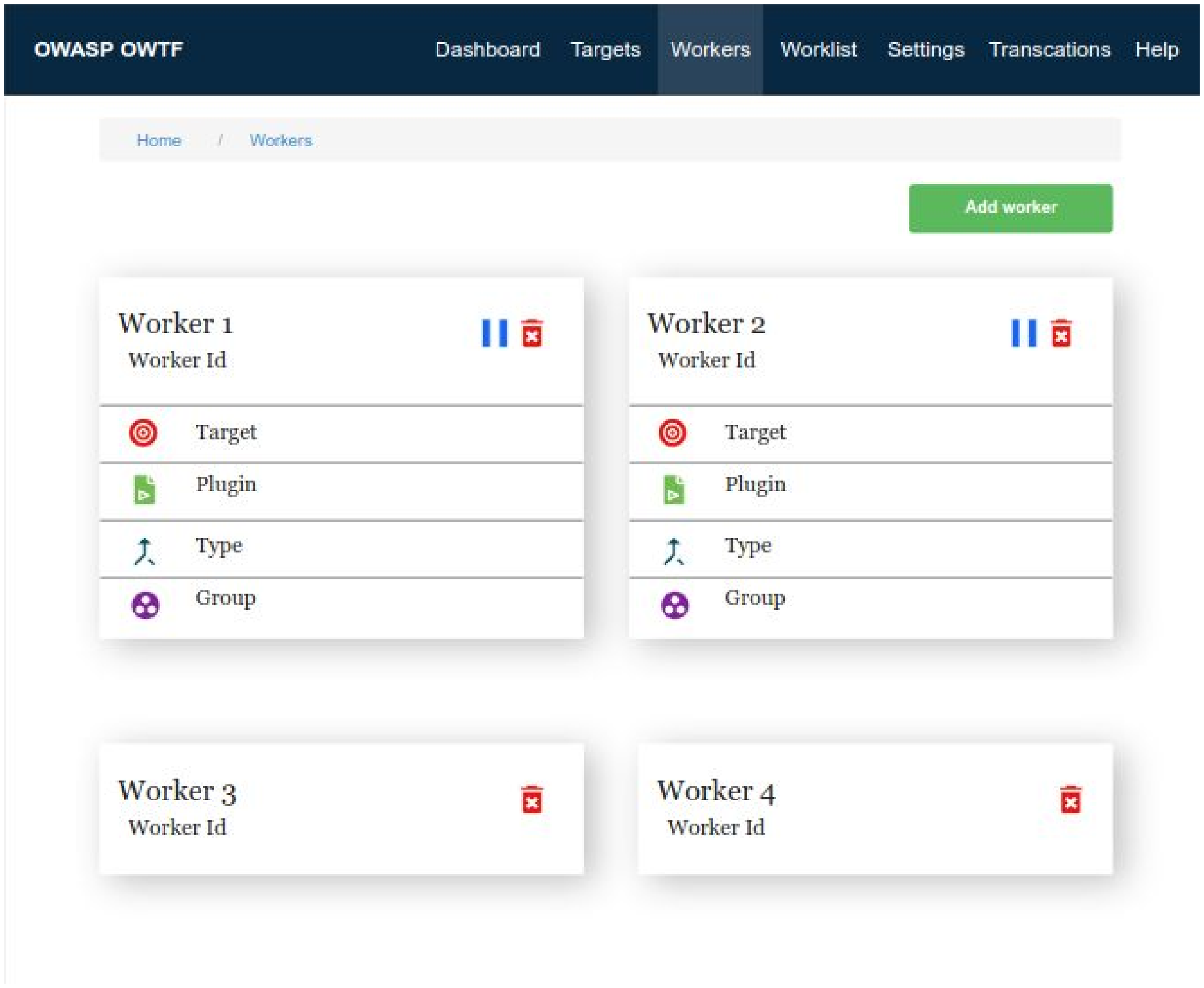


Figure 4

Plugin Side sheets (evergreen-ui) are going to be collapsible components which will appear when a plugin is clicked in the plugins list. In the side sheet we have:

- **Labels** like plugin severity, type, group etc.
- **Ranking Panel button-group** to set the plugin ranking.
- **Plugin details table** (table-evergreen-ui)
- More information about plugin like test commands, output script etc.

5) Workers Page



Workers page will consists of:

- **Panels/Cards** (pane-evergreen-ui) for each worker containing information like Id, Target, Plugin etc.
- **Buttons** (button-evergreen-ui) to pause or delete a worker.

Note: I have few more ideas in my mind which needs some more thinking and discussion with the community.

DELIVERABLES

PRIMARY GOALS

- Implement a full functional and responsive React based web interface with Redux as its state manager while following recommended design patterns.
- Enhancing the performance of the app by introducing reusability/flexibility and removing redundant code (Ex- Using **Reselect** to retrieve data from the store.
- Refining the page layouts in order to reduce user effort and make the tool more user friendly (reducing poor click flow, unnecessary scrolling etc).
- Improving the styling by replacing the old vanilla-bootstrap with the latest and more responsive **Evergreen-UI** with some components written in **react-bootstrap.**
- Introducing new UI components/features in the app while removing the outdated ones along with proper documentation of the code.
- Provide an automated and easy to build testing environment having good test coverage with Unit and Integration tests written in **Jest and Enzyme** for each component of the app.
- Implement automated linting using tools like **ESlint and Prettier** which helps us catch mistakes by enforcing consistent standards and best practices and also maintains the quality of our code.

SECONDARY GOALS

- Adding **TypeScript** in our app to reduce the no. of errors and improve maintainability of our code.
- Performance tuning and benchmarking of the app components using **Perf** (a profiling tool to enhance the performance of a react code).
- Write code implementing the **Login page**.

PROJECT TIMELINE

Span	Task
May 6, 2019 - May 13, 2019	• Bonding with the community. • Start writing tests for the static components.
May 13, 2019 - May 20, 2019	• Clear ambiguities and get a better understanding about the project by discussing it with the community. • Start working on the targets page, fix bugs, complete the session functionality.
May 20, 2019 - May 27, 2019	• Redesigning the page layouts by going through different react UI component libraries.
May 27, 2019 - June 3,2019	• Finish porting the Targets page to Evergreen-ui. • Work on refining and designing page layouts by creating mockups and getting feedback from the community.
June 3,2019 - June 10,2019	• Add tests for the Targets and Transaction pages.
June 10,2019 - June 17,2019	• Refine the page layouts and code by getting feedback from mentors. • Side by Side adding documentation for the written code.
June 17,2019 - June 24,2019	• Make up week. • Start implementing the reports page in react. • Submit work for phase-1 evaluation.
Mid evaluation: June 24 - 28, 2019	
June 28,2019 - July 5,2019	• Complete the remaining reports page(components, reducers, actions, sagas, selectors, documentation, styling).

	• Write test for reports page.
July 5,2019 - July 15,2019	• Start working on Dashboard. • Parallely writing tests and documentation for Dashboard.
July 15,2019 - July 22,2019	• Make-up week. • Finish the remaining Dashboard Page. • Submit work for phase-2 evaluation.
Mid evaluation: July 22 - 26, 2019	
July 26,2019 - August 10,2019	• Implement the workers and worklist page to react with styling using evergreen-ui. • Start writing tests & documentation for these pages.
August 10,2019 - August 20,2019	• Finish implementing workers and worklist pages. • Refine the UI as suggested by the community.
July 20,2019 - August 26,2019	• Review and catch-up weeks. • Fix bugs and wrap up the work. • Start working on the secondary goals. • Work on more features if time permits.
Final evaluation: August 26 - May 2, 2019	

Note: The time given to a specific part of the project is flexible and can be easily scaled up/down as required.

PERSONAL INFO

Name - Mohit Sharma

College/University - International Institute of Information Technology, Hyderabad, India

Degree Program - B.Tech in CSE and M.S in Computational Natural Sciences

Email - ms892075@gmail.com

Github Profile Link - https://github.com/sharmamohit123

AVAILABILITY

I will work 5 - 6 hours on weekdays and at least 7-8 hours on weekends (Saturday and Sunday). So I will be able to devote at least 42 hours per week for the project (5.5 * 5 + 7.5 * 2 = 42.5 hours).

BACKGROUND INFORMATION

I started coding since my 11th standard out of interest, I enjoy developing something that can be of direct use to us. My skills got their required polishing after coming to college. I have been developing various web-apps.
I have also done a project in my second year under a startup **FitAi,** where my task is to develop an android application which works as an online fitness platform. I have also created some webapps using languages like php, ruby on rails. Some of these applications care of various security aspects such as CSRF , SQL Injection and XSS. I have used C++ in my Computer Graphics course to make games using OpenGL3.
I open-source most of my assignments and course projects which can be found on my github profile. The following are some of the relevant courses I have completed:

- Computer Programming
- Data Structures
- Algorithms
- Database Systems
- Structured Systems Analysis and Design
- IT workshop
- Statistical Methods in AI
- Computer Graphics
- Operating Systems
- Principles of Information security

REFERENCES

OWASP OWTF - https://www.owasp.org/index.php/OWASP_OWTF
OWASP OWTF repository: https://github.com/owtf/owtf
Compatibility of py2 and 3: https://docs.python.org/3.4/howto/pyporting.html
PEP-8: http://legacy.python.org/dev/peps/pep-0008/
Arachni: http://www.arachni-scanner.com/
Probely web application vulnerability scanner: https://probely.com/
Mockup projects:

- https://app.moqups.com/ms892075@gmail.com/TtMh5BMCJ6/view/page/abac47c60
-

Webapps in react and information about react-redux framework:

- https://medium.com/@krithix/multi-page-website-with-react-in-2017-f6f2af326526
- https://medium.com/@rajaraodv/step-by-step-guide-to-building-react-redux-apps-using-mocks-48ca0f47f9a
- http://www.thegreatcodeadventure.com/react-redux-tutorial-part-iii-async-redux/
- https://www.sourcetoad.com/app-development/the-benefits-of-using-react/
- https://www.valentinog.com/blog/react-redux-tutorial-beginners/#React_Redux_tutorial_what_problem_does_Redux_solve
-

Best Practices with React-Redux Application:

- https://medium.com/@alexmngn/how-to-better-organize-your-react-applications-2fd3ea1920f1
- https://dzone.com/articles/best-practices-with-react-and-redux-web-applicatio

Redux-Saga:

- https://redux-saga.js.org/docs/api/
- https://medium.com/@lavitr01051977/make-your-first-call-to-api-using-redux-saga-15aa995df5b6

Reselect:

- https://medium.com/@pearlmcphee/selectors-react-redux-reselect-9ab984688dd4
- https://rangle.io/blog/react-and-redux-performance-with-reselect/

Testing with jest and Enzyme:

- https://hackernoon.com/implementing-basic-component-tests-using-jest-and-enzyme-d1d8788d627a

- https://hackernoon.com/unit-testing-redux-connected-components-692fa3c4441c
- https://airbnb.io/enzyme/docs/api/shallow.html
- https://medium.com/netscape/testing-a-react-redux-app-using-jest-and-enzyme-b349324803a9
- https://www.codementor.io/vijayst/unit-testing-react-components-jest-or-enzyme-du1087lh8
- https://www.youtube.com/watch?v=EgJZv9Iyj-E&list=PL-Db3tEF6pB8Am-IhCRgyGSxTalkDpUV_
- https://medium.com/@rossbulat/testing-in-react-with-jest-and-enzyme-an-introduction-99ce047dfcf8

Using react-redux with typescript:
https://blog.usejournal.com/using-react-with-redux-and-typescript-c7ec48c211f6
JS linting with ESlint and Prettier:
https://medium.com/@joshuacrass/javascript-linting-and-formatting-with-eslint-prettier-and-airbnb-30eb746db862