McGill University
School of Computer Science
COMP 400

McLab tools on the web

Comp 400 Project

Deepanjan Roy

January 6, 2016

Contents

List of Figures

Abstract

We discuss the design the implementation of a web application that lets users take advantage of the tools developed under the McLab project. This application is explicitly designed to cater to the needs of scientists and engineers using MATLAB for their work, and therefore makes ease-of-use the highest design priority. At the same time, particular attention has been paid to the architecture of the software so that it remains easy to add new functionality and extend the features. This report gives an overview of the current features of the software, provides some context for some of the design decision made, and documents the architecture of the current implementation. It also gives a concise introduction to React and Flux - two modern influential web technologies that were used to build the web client.

1 Introduction

The McLab research project aims to provide languages, compilers and virtual machines for dynamic scientific computing languages, particularly MATLAB [1]. Over the years, the project has produced several fantastic tools - sophisticated static analysis frameworks, static and JIT compilers, and several toolkits and libraries to bring MATLAB to the world of distributed and parallel computing. However, the current offering of tools almost exclusively target compiler researchers or highly technical programmers. Most MATLAB users, au contraire, are scientists and engineers who are often not expert programmers - they will greatly appreciate software that is simple and easy to use.

McLab Web is an attempt to fill this void. It is a web application that allows anyone to use some of the McLab users with very little cognitive overhead. The current workflow of using the software involves simply dragging and dropping some project files into the interface and get some useful analysis results out with very little additional input. The McLab tools are run on Sable servers on the user uploaded files to produce the results.

2 Why a web application?

There are several key reasons why we decided to build a web application instead of going the desktop app route:

- **Installing software is unpleasant:** Installing software, especially research software is often a very unpleasant experience, and it is not uncommon to run into OS incompatibility issues where the software only installs cleanly under specific distributions of Linux but fails to work under Microsoft Windows operating system. On the flip side, developers need to put in tremendous amount of effort to build and maintain installers for all the different operating systems - time and effort that would be better spent building new features instead. Having a web application instantly makes the software accessible to anyone running a decent browser, and developers have one version to maintain.
- **Instant updates:** In a similar vein, when a desktop application is updated, not everyone instantly updates to the latest version and they miss out on the latest developments. A web application ensures everyone is always on the most updated version. It also frees the developer of having to occasionally support and patch older versions of the same software when a critical bug is discovered.
- **Capturing user input:** An indirect benefit of having a web application is that we can capture the files uploaded by the user[1]. This is very valuable research data, as it gives us a glimpse into the usual structure and properties of MATLAB programs used in the real world, and can guide future research directions.
- **Looking ahead at the future:** There are several project under development in the Sable lab that target the browser as a numeric computing platform. A web based tool can make is much easier to integrate with all these cutting edge tools.

[1] Of course, with the consent of the user

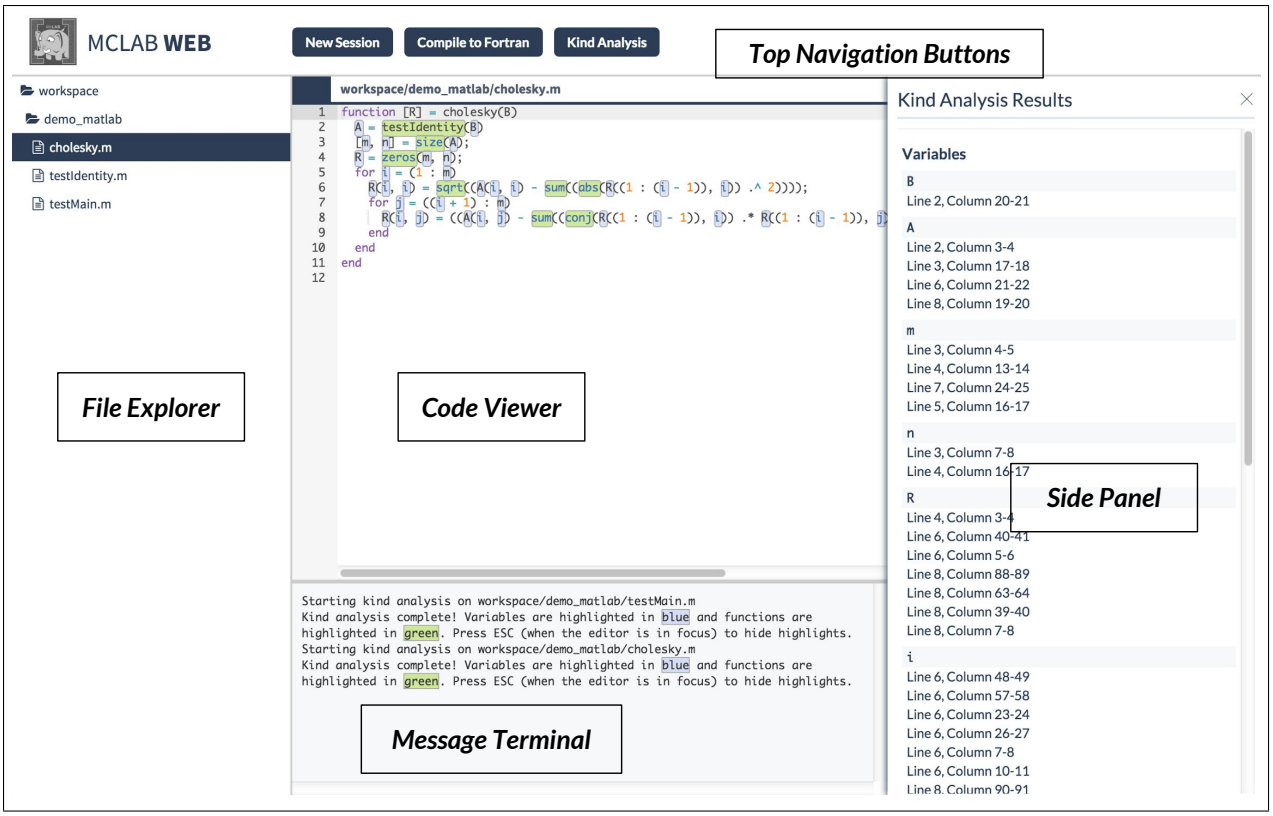

Figure 1: McLab Web Interface

3 Features

McLab Web contains a set of useful core features, and implementation of two McLab features - Kind Analysis and Compilation to Fortran.

3.1 Core Features

File Explorer

The File Explorer lets the user navigate the file tree of the project. The project files can be added by dragging and dropping a zip file into it. The file navigator also has a *Selection Mode*: When the user needs to provide a file as input, the file explorer changes its shade and doubles as a file selector. For example, when compiling to Fortran, the user needs to specify what the main file is - this is done through the file navigator selection mode.

Code Viewer

Clicking a file in the file editor shows its contents in the code editor with MATLAB syntax highlighting. It is also possible to highlight chunks of code with different colors (this highlighted chunks are called *markers*) - this feature used to show Kind Analysis results, as can be seen in Figure 1. The editor is currently set to read-only mode, but since it is based on the Ace code editor[2], editing capabilities can be turned on by switching a flag (and implementing a file save functionality on the server.)

Message Terminal

There is a simple message terminal where you can print any information. This mimics the console that is available to command-line programs. However, this terminal can display any html fragment as opposed to just text - this feature can be used to render links or progress bars in the terminal.

Side Panels

Some actions can open a particular side panel - these can be used to accept user inputs, or to display information that will not be very readable if output to the message terminal. For example, in order to compile to Fortran the user needs to provide the compiler what the main file is, and what argument it takes. The inputs are taken in through a side panel. For kind analysis, a list of all the variables and functions occurring in is shown, and displaying this information in the small terminal would lead to a frustrating scrolling experience. Instead, a side panel is used to show this information. Side panels are quite versatile, and in McLab way they are the primary method of accepting user input.

Lightweight sessions

When a user first visits the site, she is redirected to a url that contains a unique session id; for example, when she visits `mclab.com`, she will be redirected to `mclabweb.com/session/41b1605d-a262-477e-960c-a21366feeaa4/`. Any files uploaded in this session will persist on the server, and as long as the user has the url she can access them. The url is also very easy to bookmark and share.

3.2 Implementation of McLab features

In addition to the core features, two McLab functionality has been implemented so far:

Compilation to Fortran

The user can compile the project code to Fortran. The compilation is carried out by Mc2For[6] on the server side, and a link to download the zipped compiled output shows up in the terminal once the compilation is done.

Kind Analysis

In MATLAB, function calls and array access have the same syntax, so it is not very straightforward to tell whether a variable is a function or not. McLab Static Analysis Framework provides a *Kind Analysis* feature, which can determine this using static analysis techniques[3]. In McLab web, you can run kind analysis on the open file, and it marks the variables and functions in different colors, and shows a list of variables and functions in the file with all the occurrences.

4 Design and Architecture

At the very high level, the application consists of a server that will reside on Sable servers, and a single-page web application that will run on the users browser. After the initial page load, all communication with the server occurs through AJAX calls.

4.1 McLab Web Client

The McLab Web client is the frontend of McLab Web. It is built using React, a modern sophisticated UI library developed by Facebook[5], and the Flux architecture pattern that complements React's composable

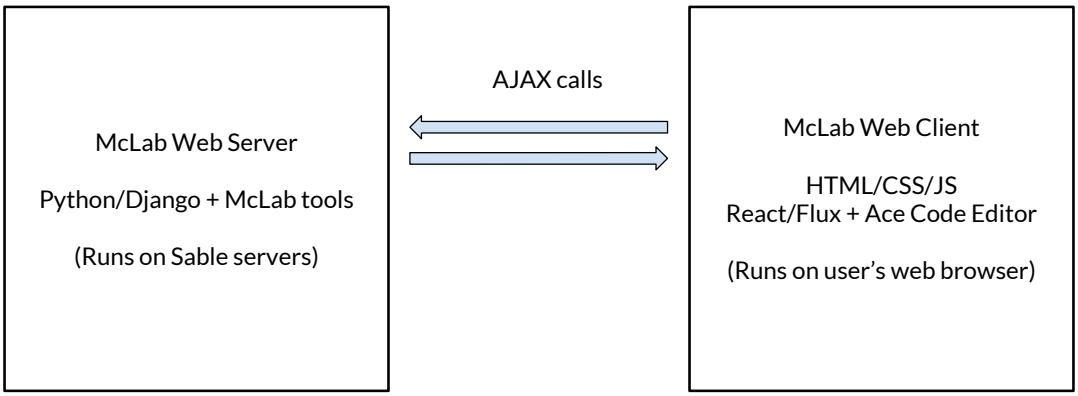

Figure 2: McLab Web Architecture

views, also first introduced by Facebook[4]. React and Flux is particularly well suited for building complex applications while keeping the code easily extensible as the codebase grows to tens of thousands of lines; React/Flux has enjoyed heavy adoption among big industrial players in addition to Facebook. Since the use of React and Flux heavily dictated the architecture of the client, we briefly introduce the core concepts of React and Flux here before talking about the specifics of our design.

4.1.1 Brief introduction to React

Traditional approach to building complex web application is to directly manipulate the DOM in response to user events. However, as the application grows in complexity, this approach quickly becomes unwieldy.

As a small example, assume you would like to display a list of 2000 names, and if any two names are identical, you want them both to be coloured blue, and the other unique names should be coloured black. If you know the 2000 names in advance, it is quite straightforward to first detect which names are duplicates, and then render 2000 `<li>` elements with the non-unique names containing a special CSS class that turns them blue. However, assume now that the user can delete items from this list. Every delete operation will force us to recalculate which items should still be blue, and we will have to maintain a mapping between items in the JavaScript list and the rendered list items in the DOM so that we can change the color of the correct element. It is in fact quite difficult to maintain keep this mapping in sync - there might be other user operations that are also working on the DOM (imagine perhaps the user can temporarily 'hide' an element instead of deleting it.) Handling all this cases will lead to some very complicated and hard-to-follow UI logic.

Another option is to instead of selectively changing the color of some `<li>` elements, we destroy all the `<li>` nodes, and rerender the list from scratch. This of course makes the rendering logic much simpler, but we will (a) have terrible performance - DOM changes are slow (we are already uncomfortable rerendering a 2000 items list - imagine if you are rerendering the whole Facebook news feed any time you click something) and (b) No matter how fast it is, it may still be noticeably jerky to the user - the list will momentarily have zero height and can change the scroll position of the browser.

This is where React comes in and tries to offer the best of both worlds. In React, you write several *Component* classes. Each Component object has some state and a render method. The render method returns a 'UI fragment' - which is a React subtree that looks very much like an HTML subtree, but instead of having just the familiar HTML elements like `<div>` and `<span>` and `<li>`, we can also use any other React component we have defined. For example `<MyReactComponent><div><MyReactChild /></div></MyReactComponent>` is something you could return - thus one way to think of React components is that they expand the set of HTML elements available to you. Whenever the state of a component changes, the render method is called with the new state. Going back to our list example, we would implement a MyList Component class with

the state consisting of the array of names. The render method will consist of the simple logic of creating the the whole list from scratch starting from current list in the state.

React maintains a virtual DOM in parallel to the browser DOM. When the state of any component changes, the render method is called, and React creates a new virtual DOM tree. Since the virtual DOMs are never directly displayed on screen, these operations are very fast - most of the browser DOM's slowness comes from having to repaint the elements on the screen. React then computes a diff of the two trees, and determines the minimal set of changes it needs to make to the browser DOM. In our list example, for a delete operation these minimal changes will involve deleting a single node and changing the color of precisely the nodes that have a different colour after deletion.

To sum up, React lets us have our cake and eat it too - the UI logic remains easy to reason about, and the performance penalty is insignificant for most use cases.

4.1.2 Brief introduction to Flux

React addresses the problem of rendering the application state to the DOM; Flux dictates how the application state should change in response to the user action.

When starting out building a complex application, it is helpful to isolate a few objects that serve as the unique source of truth for the complete application state. A popular design pattern is Model-View-Controller (MVC): The model stores the application state and the controller has the job of syncing the models with the views. When the user interacts with the view, the view communicate this to the controller leading to change in one or more models. The controller then syncs these changes back to the view layer. Flux is somewhat based on the MVC design pattern, so it is important to understand when MVC hits its limitations.

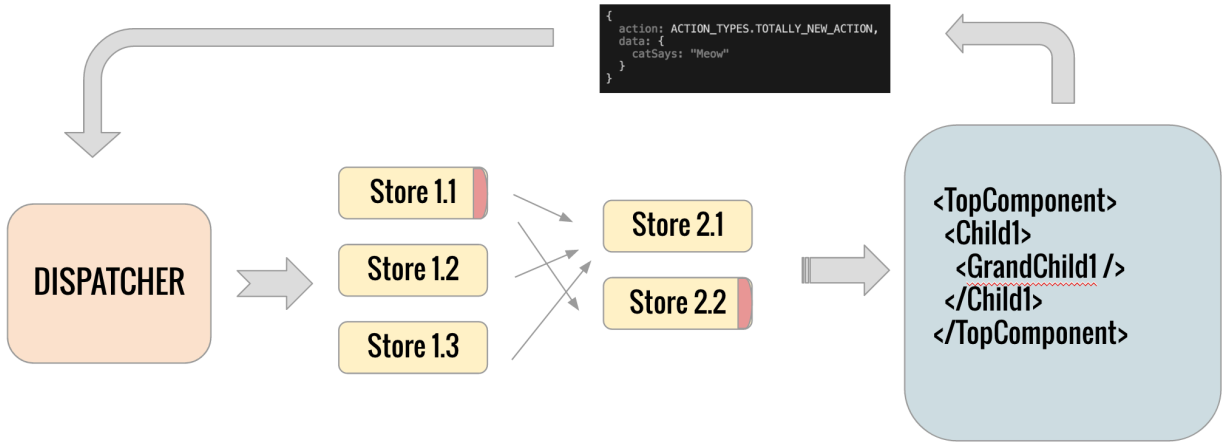

Figure 3: Schematic representation of an application with flux architecture. Red blobs on a store indicates it has emitted a change event

As the number of models increase, it is all too common to create a complicated web of dependencies between the models - change in model A triggers a change in model B, and a change in model B triggers a change in model C, and if we are not careful, one day we can make a modification of model C code so that under rare circumstances, a change in model C triggers a change in model A, and suddenly deciding whether a model change with create a finite chain of changes becomes almost equivalent to deciding the Halting Problem. Another issue with this pattern is that if we refactor our code to delete model A, we will have to hunt down every single piece of code in the application that makes a change in model A and fix it.

The Flux architecture has a concept of *stores* instead of models. Stores are loosely like models in the

sense that they serve as the unique source of truth for the application state; however, they have no setter methods. It is impossible to change data in a store from a random place in the code. Instead the stores listen for actions and changes their internal state when particular actions happen. It is possible to impose an order of reaction on the stores so that some stores will always react to actions first, and then subsequent stores can take the changed state of first store into account when changing their states. Thus we do allow stores to be dependent on each other, but this can never be circular (even accidentally.)

The actions are created when specific events happen (e.g. the user clicks something, the server sends new data etc.) Concretely, actions are simple json objects with two attributes: the action type, and data (can contain any JavaScript object.) The actions are passed on to the dispatcher, which are then passed on to every single store in the application. Each store reacts to a specific set of action types, and ignores actions of other types. The actions are therefore 'fire-and-forget': they are very decoupled from the stores they eventually change. Deleting or adding stores does not require changing the code that fires the actions.

React components, which would be the view layer in the MVC pattern, get all their state from the stores. When a store changes its data after reacting to an action, it emits a change event. React components subscribe to the change events of a specific set of stores, and whenever they detect a change in those stores they recalculate their states and rerender themselves.

To sum up, Flux imposes a *unidirectional data flow* on the system - actions are always first taken to the dispatcher, from where it travels through a layers of stores in a predetermined order, until it hits the view layer with React components.

This is a somewhat simplified description of what React and Flux is, but should be sufficient for our current discussion. Armed with our expanded terminology, we can now describe the specifics of the design of McLab Web client.

4.1.3 Flux Stores in McLab Web

McLab client currently uses ten different stores. We present here a brief description of some of the important ones:

ActiveSidePanelStore: Used in the side panel. Keeps track of which side panel is open. We currently have two side panels: The Fortran Compile Panel, and the Kind Analysis Results Panel.

EditorMarkerStore: For each file, stores which substrings should be marked in the editor, and what CSS class should be applied to the marker. The colour and shape of the marker is controlled with the CSS class.

FileContentsStore: Stores the contents of a file, which is then rendered to the code editor.

FortranCompileConfigStore: Stores the configuration for Fortran compilation. Even if you close the Fortran compile side panel, the settings stay in the store. So, as an example, once you have chosen a main file, you will not have to choose a main file again unless you refresh the browser.

KindAnalysisResultStore: Stores the results of kind analysis for each file.

SelectedFileStore: Keeps track of which file is currently selected in the file explorer.

OpenFileStore Keep track of which file is currently open. This depends on the SelectedFileStore. If the selected item in the sidebar is a directory, the value of the open file does not change. Otherwise it changes to match the selected file.

TerminalBufferStore Stores an array of lines to be rendered that will be rendered to the terminal.

4.1.4 React Components in McLab Web

There are 21 React components in the current version of McLabWeb, and it would be too dreary to describe all of them. We focus here on some of the key ReactComopnents:

CodeContainer This contains the code section, including the title of the open file, and the code viewer. As one would expect, this subscribes to the open file store to know which file is open, to the FileContentsStore to get the file contents, to the EditorMarkerStore to get the markers for the code viewer, and perhaps surprisingly, to the ActiveSidePanelStore to trigger an editor resize when the size panel is opened/closed. The code viewer is implemented using the Ace code editor, a very feature-rich open source web code editor.

FileExplorer This is the root component for the file explorer, and it composes FileTile and FolderTile react components. This is one of the more complex React components of the project, and its state includes a complete file tree of the project that it renders with proper indentation. It uses an open source react component called Dropzone to implement the drag and drop functionality. Clicking any file/folder in the file explorer dispatches a `FILE_EXPLORER.SELECTION_CHANGED` action that updates multiple stores.

FortranCompilePanel This is the side panel for accepting Fortran compiler configuration. It composes FortranCompileArgumentSelector, a separate react component for argument selection. The render logic is written in a way to reveal further configuration options and the compile button only when valid previous configurations are chosen. For example, it does not even display the final compile button until a main file is selected.

Terminal This is the React component for the message terminal. Each item in the list received from TerminalBufferStore is rendered in a separate line. It can render any HTML in as a line, and in fact, it can render any React component as a line. This makes it easy to add a progress bar react component for the terminal.

McLabWeb This root React component that is the ancestor of all other React components. It has no state and only rendered once on initial page load.

4.2 McLab Web Server

The server is built using django - a modern python web framework. The django server contains an url router, that relays requests to particular url to the specific handler function. Url patterns and routing rules are defined using python regular expressions, and it is possible to use part of the url as an argument to the view function. For example, this is an example of a url pattern routing definition:

```
url(r'^session/(?P<sessionid>[\w-]*?)/readfile/(?P<filepath>.*?)$', views.readfile),
```

Now given the url `http://mclabweb.com/session/41b1605d-a262-477e-960c-a21366feeaa4/readfile/demo_matlab/testIdentity.m`, Django will call the `views.readfile` function with two arguments: the session id `41b1605d-a262-477e-960c-a21366feeaa4`, and the file path `demo_matlab/testIdentity.m`.

In django parlance, these handler functions are called views. A particularly interesting view function is the file upload view handler: When the user drops a zip file in the File Explorer, an HTTP multipart post request comes in. The whole request is passed on to the view handler, that takes the attached zip file, looks at the session id, extracts it to the right location, and if everything goes well (i.e. the file is a valid zip file, and there were no unforeseen difficulties in extracting the file like disk space shortage) the the handler returns a simple 200 OK success message. Django takes this HTTP response and returns it to the client.

When requested, the server can run McLab tools like Mc2For and McSAF by spawning a new shell and capturing the output of the command line invocations of those tools. These outputs are then converted to json and sent back to the client.

5 Challenges

Implementing user interfaces presents a lot of quirky challenges that are hard to foresee. The first challenge is in fact deciding how to divide the interface into meaningful components that will make it simple and beautiful. An example of UI design decision is the side panel, which is a uniform component to accept user input and display data - this is an alternative to using many different kinds of dialog boxes. Another subtle design decision is to use the file navigator as also the file selector, and thus reuse onscreen components to reduce clutter. However, this goes without saying that this a rather subjective art as opposed to precise science.

Regarding more concrete technical challenges, a surprising amount of effort went into getting the correct scrolling behavior of html elements. We briefly talk about two examples of scrolling issues and one example of interplaying react with non-react components - they will illustrate the strange perils one can encounter in user interface land.

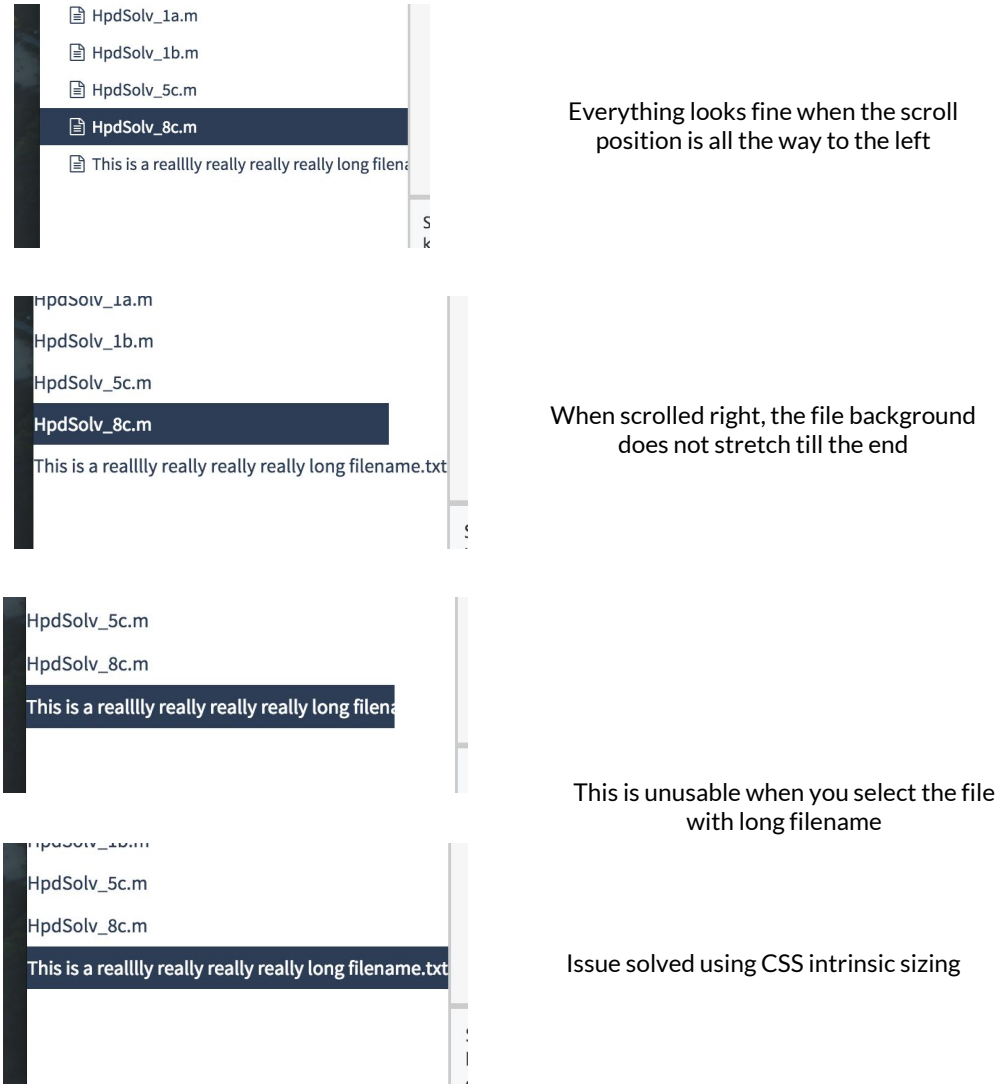

Figure 4: Horizontal scrolling issues in File Explorer

Horizontal scrolling and long filenames

If you have a long filename (or a deeply nested tree), the file navigator allows you to scroll horizontally to reveal the whole contents. However, as seen in figure 4, the naive implementation did not extend the hover highlighting or selection highlighting effect to fill the entire width of the container. This is objectively ugly. Fixing this required using a CSS intrinsic sizing property (namely 'max-content') which was only introduced in a W3C working draft in 2012[7]. Intrinsic sizing properties are supported by Firefox and Chrome, but they still lack support in Microsoft Internet Explorer and Edge.

Horizontal scrolling of the code viewer when side panel is open

Opening the side panel takes up a lot of screen space on the right side, and if the user has a small display, or has code with very long lines, it is important that the panel does not actually cover real code content and let the user scroll to the end of the line. The ace code editor cannot automatically tell when it should resize the code editor because other elements of the html tree changed (as opposed to when you resize the whole window, in which case ace editor *does* automatically resize itself). To remedy this, the CodeEditor react component now actually listens to side panel open/close actions, and manually calls the resize method on the editor when needed.

Figure 5: Making sure it is possible to scroll all the way to the right when side panel is open

Using Ace Editor with React

React expects that any changes to the DOM will be done through the render method of the components (of course, because otherwise the state of the virtual DOM will no longer reflect the state of the browser DOM.) However, the Ace code editor relies heavily on mutating the DOM which goes against the core react assumption, and led to many flakiness issues: the editor sometimes disappeared because of unrelated changes in the react tree, or the editor was completely rerendered in response to actions like displaying the markers in kind analysis, which means the cursor jumped back to the beginning and any text selection would be lost. Reimplementing a code editor to reconcile with the react philosophy would be a monumental undertaking - instead the current solution involves heavy use of react lifecycle methods to override default react rendering behavior.

6 Hindsight

One big learning from the project was that Python/Django, being non-blocking by default, was not the best choice for the backend. Currently any long running operation - unzipping a file, compiling to Fortran, or running kind analysis - blocks the entire server while the operation is happening. This limitation can be overcome in Python itself by using a task queue and a worker pool, at the expense of a more complex backend architecture. Server side JavaScript VMs provide an extensive collection of asynchronous non-blocking functions to deal with these issues in a simpler and more elegant way.

However, the biggest advantage of using JavaScript on the backend will be that it will allow us to very easily move logic between the client and the server. As browsers become faster, and better JS numeric computing libraries become available, it is very possible that we would like to move some of the server side processing to the client to reduce the load on Sable servers.

7 Future Work

The McLab Web project is still at its infancy - there are many McLab tools waiting to be implemented on this platform. In the immediate future, we can think of a few goals:

Linters for McLab: It will be useful to develop linting tools for McLab that can warn users when they are not following best practices for code.

Code profiling tools: Using AspectMATLAB, we could develop code profiling tools that alerts the user if it is possible to attain significant performance improvement by using a different class of functions. In fact, simpler profiling tools that identifies the slowest section of the code will still be immediately useful.

Integrate with McLAB's JS libraries The Sable lab is working on JS numeric computing libraries. They are a natural fit for things that can be integrated into McLab Web.

We hope this project eventually gets widely used, and make the days of many MATLAB users better!

References

[1] Andrew Casey, Jun Li, Jesse Doherty, Maxime Chevalier-Boisvert, Toheed Aslam, Anton Dubrau, Nurudeen Lameed, Amina Aslam, Rahul Garg, Soroush Radpour, et al. Mclab: An extensible compiler toolkit for matlab and related languages. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, pages 114–117. ACM, 2010.

[2] Cloud9 and Mozilla. Ace — high performance code editor for the web. `https://ace.c9.io/#nav=about`, 2015. [Online; accessed 19-December-2015].

[3] Jesse Doherty, Laurie Hendren, and Soroush Radpour. Kind analysis for matlab. *ACM SIGPLAN Notices*, 46(10):99–118, 2011.

[4] Facebook Inc. Flux — application architecture for building user interfaces. `https://facebook.github.io/flux/`, 2013. [Online; accessed 19-December-2015].

[5] Facebook Inc. React — a javascript library for building user interfaces. `https://facebook.github.io/react/index.html`, 2013. [Online; accessed 19-December-2015].

[6] Xu Li and Laurie Hendren. Mc2for: A tool for automatically translating matlab to fortran 95. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 234–243. IEEE, 2014.

[7] W3C. Css intrinsic & extrinsic sizing module level 3. http://www.w3.org/TR/css3-sizing/, 2012. [Online; accessed 19-December-2015].