REPORT ON OpenDreamKit DELIVERABLE D3.2
Understand and document SAGEMATHCLOUD backend code.

ERIK BRAY

Due on	02/28/2017 (Month 18)
Delivered on	02/28/2017
Lead	Université Paris-Sud (UPSud)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/61`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #61 ON 2017-02-28

- **WP3:** Component Architecture
- **Lead Institution:** Université Paris-Sud
- **Due:** 2016-02-28 (month 18; originally month 6)
- **Nature:** Report
- **Task:** T3.6 (#55): Document and modularise SageMathCloud's codebase
- **Proposal:** p. 43
- **Final report**

SageMathCloud (SMC) is both an open source software project (`https://github.com/sagemathinc/smc`) and an online instance of that software (hosted at `https://cloud.sagemath.com/`) that provides an interactive, collaborative environment for teaching and research in science, technology, engineering, and mathematics. See e.g. D2.3 (#43) for a description of this emergent technology.

SMC predates OpenDreamKit (ODK) and acts as prototypical example of VRE that can be built from the ecosystem OpenDreamKit aims at fostering. In particular, SMC is one of the main channels through which some of the most important technologies of ODK, such as Jupyter and SageMath, are distributed on-line. It makes it a good mean to distribute some of the newly developed ODK features, like the new Jupyter kernels of D4.4 (#93). It is very probable that many users will benefit from some of the ODK new developments through SMC. Reciprocally, the inner technologies of SMC are of special interest to ODK developers: they show advanced uses of cutting-edge web technologies and explore new leads that could inspire the work we do in ODK.

For all these reasons, is has been planned since the beginning for ODK to collaborate actively with SMC. For example, in D2.4 (#44), we have developed a short course for educators who wish to adopt SMC and related ODK technology in order to enhance their teaching.

In this deliverable, we start exploring the main layers of SMC's backend code and give a general overview of its functioning. The material we have produced can directly help the platform attract more developers. One of the expected follow-up is an easy install for a local version of SMC especially designed for development which could be part of upcoming D3.5 (#63). The long term goal however is to understand the extent of a full install of a SMC instance on a server or cluster: How hard is it? What is the total cost of ownership? Is it a viable solution for institutions of various scales to deploy and run a local SMC instance?

CONTENTS

1. INTRODUCTION

SAGEMATHCLOUD (SMC) was presented as an emerging technology in D2.3. It is an on-line open source platform which allows the creation of collaborative scientific projects. Each user can create *projects* within the platform. Each project is hosted on a LINUX virtual machine accessed through a web interface and provides access to many scientific tools. It is thus an important working example of a Virtual Research Environment (VRE) as described in Section 1.3.2 of the Proposal, with capabilities suitable for reasearchers, software developers, and teachers. Launched in 2013, SMC presently hosts over 400,000 projects and nearly 20,000 weekly active users. This fast adoption by a wide variety of users demonstrates the relevance and the long term impact this kind of collaborative environment can have. Although it is just one example of a VRE, it serves as an important prototype for OpenDreamKit (ODK).

Because there is no one VRE that suits all needs, a framework for VREs must be highly flexible, allowing researchers to choose from the tools that best suit the task and combine them in arbitrary, yet interoperable ways. SMC has already taken steps in this direction by providing a unified interface to software systems such as SAGE, GAP, and other ODK components, as well as a full LINUX shell with common software development tools, LaTeX document editing and display, JUPYTER notebooks, computing resources, and other capabilities that one would need for a VRE with unlimited possibilities. While there is still work to be done on further integrating the components of SMC, this is work that ODK can feed back into SMC while simultaneously using SMC as an example VRE.

Another major effort in building a VRE, beyond innovations on the environment itself and development of effective workflows within a VRE, is the underlying software and computing infrastructure needed to make VREs accessible and widely available. SMC has given us a real working example, with open source code, of how to build a cloud-based VRE, accessible from a web browser from anywhere in the world, with a consistent user interface around its components. There are many technical details we can learn from it, such as what software technologies its framework is built on, and how its underlying computing resources are organized and administered.

The purpose of this deliverable is to better understand the technical details of how SMC works, so that the ODK members can learn from it, and also eventually contribute innovations from ODK project back into SMC.

2. HISTORY OF SMC'S DEVELOPMENT

William Stein, the creator of SAGE and SMC, started working on SMC in 2012 as a way to make SAGE– notoriously non-trivial to install – more accessible to users via a cloud service. Around the same time, it became apparent that, in order to survive and to compete with commercial mathematical software systems, it would be extremely helpful to have a self-sustained income stream based around SAGE. Stein realized that, with enough value-added on top of SAGE itself, users might be willing to pay for access (and in particular computing and network resources) for this service. So SMC became more than just SAGE, but rather a cloud-based research and teaching environment built in large part around but not solely focused on SAGE.

For most of its development history, Stein has been SMC's sole developer, working in his spare time while being a professor at the University of Washington at Seattle. It was not until September 2013 when its second most extensive contributor, Harald Schilly, made his first commit to the SMC source code repository. Since then, especially once the code was made open source in Fall 2015, a little more than two dozen people have made developments. Nevertheless Stein still has done more than ten times as much work (roughly, in terms of number of repository commits) than anyone else on the project, and as such is the only person who truly understands its full design.

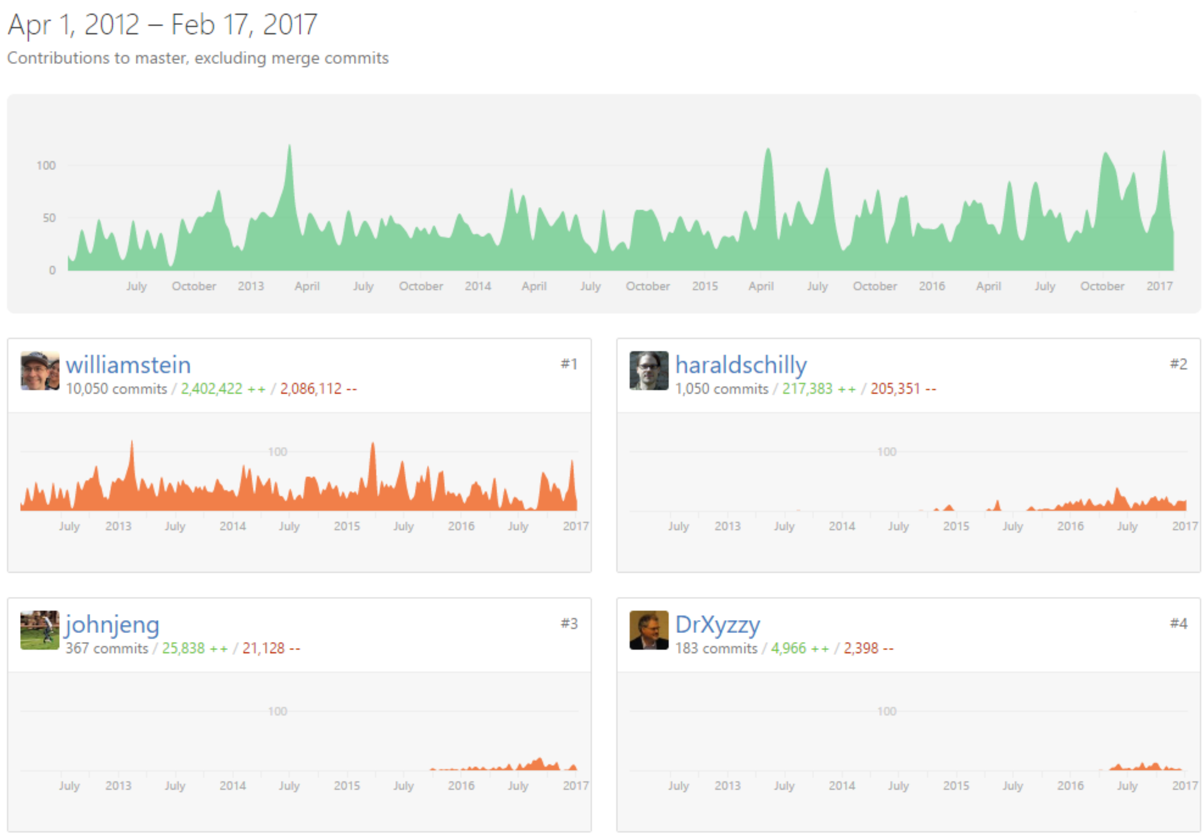


FIGURE 1. top SMC source contributors from the project's beginning in 2012 through February 2017

Because it has had effectively one developer, and because of the breakneck pace at which it was built, very little of SMC's internal design–both overall and the lower-level details– is documented in an accessible manner. In order to understand SMC's design, one currently has to read the code, and get inside Stein's head a little bit to understand how he might have been thinking. Further complicating matters is that SMC has gone through multiple partial rewrites (most notably a rewrite of the server code, originally in PYTHON, to JAVASCRIPT). Because these rewrites have been only partial (due to Stein's limited time and developer resources), this has resulted in what amounts to layers of digital sediment that must be sifted through carefully in order to understand how and why some design decisions came about.

Because of the fast pace of development, documentation on how to help with development of SMC itself–something of interest if ODK is to contribute back to SMC–has also fallen behind. Even the talk How to contribute to SageMathCloud given by William Stein in late 2015 and mentioned in the report for D2.2 is no longer *as* useful for getting started on full stack development of SMC as it was a year before this report (though it still contains some helpful information).

None of this should be read as criticism of Stein or SMC–the reasons for the relative opacity of SMC's design are understandable. It is just helpful to understand why it is particularly challenging to understand and document this already enormously complex piece of software. An additional challenge of this task is to write documentation and development procedures that will be maintainable through future development without always becoming immediately obsolete.

3. CURRENT STATE OF SMC'S SOURCE CODE AND DOCUMENTATION

SMC's source code currently resides in a single GIT repository hosted on GitHub. This repository contains all code written uniquely for SMC–web-based frontend and all backend servers–as well as scripts and tools for development of SMC.

It also has a collection of many small scripts and utilities used for managing the live SMC site and the many servers that comprise it (hubs, compute nodes, etc.) Many of these are just single command-line commands for what might be common tasks. However, there is little to no documentation for most of these tools or when and how they should be used. Most of them are not directly relevant to doing development of SMC.

Most of the main SMC source code is split across two PYTHON packages, five JAVASCRIPT packages, and one directory containing additional web resources such as fonts and images, as well as "vendored" JAVASCRIPT–third-party source code that may require special handling to integrate into the SMC web app.

There is also a configuration script for Webpack, a tool used to bundle JAVASCRIPT sources, HTML, images, and other resources into a working website. This file serves as something of a map to how all of SMC's frontend code fits together. Because there is no equivalent for the backend, it is a little more difficult at first to understand how the server-side code works. That said, the general layout is clear enough that an experienced developer, equipped with high-level documentation of SMC's infrastructure, can gain insight into what features might be implemented where. More such high-level documentation is needed, however.

Beyond a few brief "README" files in the main source packages, and scattered in-line documentation in some of the source files, there is very little documentation of what each source file is or what internal or external APIs might exist. So while the high-level structure is not difficult to understand, it is somewhat more difficult to understand the flow of data and program control.

4. DOCUMENTING SMC'S INTERNAL DESIGN

In order to help newcomers better undertand SMC's internal design, we have begun documenting some aspects of it in better detail. The first version of this document is included as Appendix A of this report. It has also been accepted into SMC's source code repository for easier discovery by anyone interested in how SMC works.

This document covers two topics in particular: It goes into greater detail than any previous documentation on the core JAVASCRIPT libraries and tools that SMC is built on, including a brief introduction to those tools, and some discussion of how and why they are used by SMC. This list is still not exhaustive, but it covers most of the key technologies one needs to have understanding of in order to understand SMC's code base.

Our documentation also provides a brief tour through SMC's code base– specifically the client-side code–explaining step by step how a page is rendered by SMC's client-side code. This explains specific examples from SMC's code base of how its dependencies are used, and how the source code is organized internally. Some of the exact details may change in the future, but the basic design (at least on the client side) is likely to remain stable for some time. This is due in part to the mostly finished work of rewriting SMC's web client on top of React–a JAVASCRIPT UI framework created and heavily supported by Facebook. This React-based design is not likely to change in the forseeable future.

The walkthrough we have written on SMC's code base is for a very simple example. For future work, it would be instructive to include a more complex example, such as a walkthrough of how SMC's advanced real-time documentation collaboration features work. A detailed walkthrough of the server-side code-base is still needed as well, but Stein and Schilly are in the process of significantly reorganizing much of that code. Thus it would be advisable to wait until more of that work is complete before writing further detailed documentation.

5. Overview of SMC's software development process

A large, complex web application like SMC is non-trival to do development on compared to more self-contained software projects. A full SMC deployment consists of multiple server processes that must work in concert, and other moving parts such as Webpack builds for assembling the web client. SMC's source code documents a few different ways to work on SMC:

- The simplest is to run a development SMC server directly on one's personal computer, reloading the server as necessary while making changes to the source code. SMC in fact has a "development" mode wherein all of SMC's server components run on one's local machine, which also acts as the sole compute node. Any projects on the development server are created and run on the local machine, under the developer's login account. This is certainly the simplest way to work, but the code currently has some (known) bugs such that a SMC server run in this way is not actually fully functioning. Thus, while this is feasible for some development tasks, it is not currently possible to develop the full SMC stack in this way. We recommend trying to fix that if at all possible. Another downside to this method is that the developer must install all of SMC's requirements manually, and this is not well documented at the moment. As much of SMC is UNIX-oriented this also creates a barrier for would-be developers working on Windows.
- Another compelling way to develop on SMC is referred to as "SMC-in-SMC". It is possible, on an existing SMC server (particularly the main one hosted at `cloud.sagemath.com`) to download the SMC source code into a project hosted on that server and run a full SMC server from within the project. This is somewhat similar to the previous method, but takes advantage of the fact that all of SMC's runtime dependencies are already available inside an SMC project. It is easier to get SMC-in-SMC up and running than most other methods, and it works impressively well considering its recursive nature. While one is restricted to working within an SMC project this is not *much* of a downside considering the flexibility afforded to SMC users). It does, however, require having a non-free SMC account (the default free accounts for SMC do not allow internet access from within projects), though Stein is generous in giving less restricted free accounts to users who wish to help with SMC development.
- The easiest way to get a fully functioning SMC server up and running on one's personal computer is with the official Docker container for SMC (see the report for D3.1 for more information on Docker containers). The Docker container image is well-maintained to work with the latest SMC source code, and provides a turn-key solution for a full-stack single-server SMC deployment. There is a slight downside, however, that its current design is not very amenable to development. For example, the SMC source code is contained in the container itself, which means any and all development, including editing the source code, must be done within the Docker container. This may require a would-be developer to manually reproduce their preferred software development environment (tools and settings) and edit all source code directly inside the Docker container. A better approach is to design a Docker container for development that allows the developer to keep the SMC source code on their local machine. The SMC Docker container would then mount the source code as an external "volume" (this is a feature of containers similar to a shared folder between the container and its host). The developer can than do all development work on their local machine, but *execute* the source code inside the Docker container, freeing them of the need to worry about setting up SMC's dependencies. We have begun work on a development-friendly Docker container for SMC, but more work

is required to make this fully viable. When completed, this will likely be the easiest way for anyone to get quickly up and running with SMC development on their personal computer.

6. Conclusion and future work

Prior to work on this deliverable there was scant documentation on SMC, its overall design, its dependencies, or its internals. There was some documentation on how to do development on it, but not enough, and sometimes out of date. With the documentation we have added it should require less effort for a motivated developer to gain the necessary background knowledge (such as knowledge of the core technologies SMC is built on) to do development on SMC. Our documentation should also help lead to a faster understanding of the general flow of SMC's design, though it would not have been feasible to document the implementations of all of its features. Some of its more advanced features–especially key features such as its real-time collaborative document editing–do require further documentation. More documentation is needed on the design of SMC's server components, though that work should wait until their implementation details have stabilized more.

We have also gained experience within the ODK team on setting up a SMC server and the process of doing development on SMC. Gaining this shared knowledge is a prerequisite for future work on determining what aspects of SMC–at its core a framework for cloud-based hosting of complex mathematical software, and thus a powerful tool for building VREs–can be integrated into ODK. However, further work is needed on improving the development tools for SMC–in particular we recommend improving the existing Docker container for SMC for development on one's personal computer.

Appendix A. SMC documentation overview

Please find attached the document that was produced as part as the deliverable objective. It is also available on-line at this address: `https://github.com/sagemathinc/smc/blob/master/src/doc/design_overview/overview.rst`, that is, inside the source code repository of SMC.

SageMathCloud Design Overview

SMC's README gives a very high level overview of its architecture. However, it doesn't really explain much about the implementation details--what technologies are used, how, and why. This is in part because a lot of it has been developed by the seat of their pants (understandably; this is not meant as a criticism) and a lot of the details have been in flux, and not worth documenting. After all, the details should only really be of interest of the SMC developers themselves. The details even of setting up a personal SMC server have mostly been squirreled away into install scripts and Dockerfiles such that it doesn't require one to have a deep understanding of how it all works.

Technologies used in SMC

This is a partial list of some of the major components that are used to make SMC--especially the parts that make up SMC's own source code (as opposed to external components such as the database and HAproxy, though those will be mentioned as well).

This will also attempt to give a brief overview of what those technologies are used for in general, and how they are used by SMC. The finer details of the architecture will be discussed further in later sections.

You can either read all these words first, or skip to the diagram where I've tried to show how everything fits together. But you won't understand the diagram unless you know what most of these terms are.

CoffeeScript

CoffeeScript is one of many languages that transcompile to JavaScript (a.k.a. compile-to-JavaScript), and one of the most popular. Wikipedia gives an easy to digest overview. It's basically just syntactic sugar for JavaScript with inspiration from other languages. It's somewhat more Python-like (no braces or semicolons) with some syntax borrowed from functional languages like Haskell; some of it is also resembles YAML. Code written in CoffeeScript can be translated directly into equivalent, and not too hard to read JavaScript. Its use is not essential to SMC's design, but William prefers it over plain JS (and is in good company) so it's used virtually everywhere (along with a variant known as coffee-react or CJSX to be explained later).

Because of a number of factors, such as the prevalence of Node.js (see below), and also the differences across web browsers in what variants of JavaScript they support, as well as browser-specific quirks in their JavaScript implementations, it has become quite common in the web development community to use projects like Babel to write JavaScript in some meta-language, and then compile it to JavaScript that will actually work on different browsers (including "polyfills"--essentially backports--of features that are not available yet on all browser).

In the bad old days we used to clutter our JavaScript with browser and/or feature checks. But the movement in the web community has been toward writing JavaScript to some standard, and then using build tools (such as webpack discussed later) to run Babel and other tools to prepare the "real" JavaScript that will run in specific browsers. It's basically just a repeat of high level languages like C superseding the need to write assembly language for different hardware platforms. But in this case the "assembly languages" are browser-specific JavaScript implementations. This has led to an explosion in "high-level" languages over native JavaScript, as well as associated compilers and build tools.

Node.js

Although SMC's source tree contains a smorgasbord of (mostly small) Python scripts for various management tasks, most of SMC's backend is built not with Python, but with Node.js (the most notable exception is the server backend for Sage worksheets which is still in Python, as it has to interface closely with Sage).

Most software developers are, at this point, at least passingly familiar Node.js (often referred to as just "Node"). But in short, it is a stand-alone JavaScript runtime that can run on a server. It includes an HTTP server and various I/O primitives that have been designed from the groud up for event-driven asynchronous

I/O. Because it uses JavaScript it has been meteorically popular in the web development world because it allows developers to write both their front-end and back-end code in JavaScript, and even share library code between the front- and back-ends. This also made it easier for developers who otherwise specialize in web front-ends to cross into "full-stack" development.

None of the technologies SMC uses to drive its front-end necessarily require Node.js on the back-end--naturally; any server providing the same interfaces will work. But using Node.js makes sense due in part to the ecosystem around it--server and client technologies that were designed to work together--and because being able to write back- and front-end code in the same language reduces cognitive overhead as well.

Express

While Node.js has a decent standard library--mostly focused, as is its purpose, on network I/O--it is mostly relatively low-level and doesn't make too many impositions on how to build an application on top of it. So for example, while it includes a web server, it does not include a web *framework* for handling the kinds of things web frameworks typically do like URL routing and request handling.

Express is one such framework, built on top of Node.js. It is a fairly minimalistic and flexible framework, similar in concept to the popular Flask framework for Python.

SMC uses Express as the basic HTTP request handling framework for the majority of its web server components (particularly the Hub). That said, much of the communication that goes on between the SMC client side and the server happens (where available, which is most places these days) over WebSockets. But more on that later.

The main entry point to Express is an 'app'--an instance of its Application class. The app serves as a request handler for Node's HTTP server. Every time the server begins handling a request, it fires off a `'request'` event for which the app is a listener, and the app handles the request and returns a response to the client. The Express documentation on routing should give a basic idea of how it works. If you've used any web framework before it should be immediately familiar. If web development is new to you it might take a little more thinking, but the idea is basically simple: The app receives a request, looks at the URL and/or HTTP method of the request, and maps that information to a specific function registered to handle that request. The handler function is passed a request (representing the incoming request), and a response object that can be used for building up and sending an HTTP response to the client. Typical handlers will be the end-point for a request, and so will end with a call like `res.send(...)` which will send the response to the client and end the connection. However, there may also be middleware handlers that do something with the request (such as check authentication) before handing it off to other handlers.

Primus

Another important feature of modern web applications is real-time, bidirectional communication between the client (or clients) and the server. Because of the importance of this, and the many different ways to implement it all with different advantages and disadvantages, there has been an explosion of different real-time communication frameworks. As with other aspects of web development, the techniques to implement this functionality ("ajax", long-polling ("comet"), WebSockets, etc.) have changed over the years and have varying degrees of support on different clients. There are also higher-level connection features one needs such as multiplexing, heartbeats, automatic disconnects/reconnects, etc. and these are the kinds of features provided by these real-time frameworks.

As is typical, the plethora of frameworks has led to a meta-framework, and that's where Primus comes in. It abstracts out an API common to many / most real-time frameworks, and allows swapping out the underlying implementation. It's fair to say that different real-time frameworks have different advantages and disadvantages, and some are better than others for specific applications depending on their implementation details. Primus allows you to write your application once in its common API and then swap out the lower-level core implementation.

Engine.IO

Engine.IO is one such real-time communication framework, designed to be event-driven and asynchronous for integration into and compatability with Node.js. In fact, it attaches itself to Node.js's HTTP server, which it uses initially to negotiate connections with the client via traditional HTTP methods like long-polling. It then works with the client (with help of a client-side library) to discover support for other transport methods (like WebSockets) and transparently switches to those channels as they become available. The overall design and advantage of using Engine.IO is best explained by its documentation.

Engine.IO is actually the core to another higher-level real-time framework you will see reference to called Socket.IO. But SMC does not use Socket.IO directly, opting instead to use Primus as its "high-level" real-time framework, with Engine.IO being one of Primus's supported underlying transport layers (whereas Socket.IO is designed to work only with Engine.IO).

pug

pug (formerly named "Jade" but recently renamed for trademark reasons) is a template engine for templated HTML in particular, written for Node.js. Although it has its own particular syntax, the concept should be familiar to anyone who's written a web template before. Pug/Jade is the *default* template engine used by Express, though one can easily substitute it for any other template engine (after all, at the end of the day all a template engine is doing is returning an HTML string to be sent in the HTTP response). If you've used Flask, this is just like how Flask uses Jinja2 by default, but by no means enforces its use.

We won't go much more into pug as SMC barely uses it. In fact there is currently only one pug template in SMC (`webapp-lib/index.jade`) for the main index page to SMC. Mostly all this page does is provide some metadata and favicons, and display the big "Loading" banner you see when you first load SMC. All the rest of the front-end is loaded in via React which we'll discuss next.

React

React, also often referred to as React.js, ReactJS, etc. is a powerful toolkit for web UIs, developed by Facebook. Although one still uses HTML+CSS to specifiy the look and feel of a UI component, React allows one to manipulate components of a UI in an object-oriented manner, not unlikely traditional desktop GUI toolkits.

The example on their front page gives a great introductory example of a little "TODO list" widget. It's implemented as a class, which has a `render()` method used to display the widget in its initial state, a few internal attributes for managing its state (such as the list items), and some methods for handling different events on the widget. There's also a very nice tutorial for building a tic-tac-toe game. If you can grok that then you'll have the hang of React.

If you've ever used a GUI toolkit like wx or Swing it shouldn't be too hard to pick up on what it's doing.

Using React is quite a bit different from the old-fashioned way of making reactive web UIs with JavaScript. What I'm calling the "old-fashioned" way is a couple things. For one, the server might serve up a bunch of HTML containing all the elements in your page, many of which might by "hidden" using CSS, and the JavaScript would hide and unhide elements on the page. Or the JavaScript might generate some elements and insert them directly into the DOM and remove them as needed, either using the DOM API directly or, somewhat later, tools like jQuery (note: jQuery still has a role to play even in conjunction with React though).

In other words, gone are the days of servers rendering and returning HTML to the browser. All the rendering is pushed entirely to the client, with the client-server communication focused on as light-weight as possible message passing. This potentially frees up enormous resources for the server, while pushing much more work to the client (which is why so many of your browser tabs are using over 100 MB of memory, among other reasons).

The way React works, in short, is this: It maintains its own "virtual DOM" separate from the actual DOM of the browser document, with the same API as the real DOM. Whenever you show, hide, or otherwise update the contents of a UI element in the application, it uses a copy of its virtual DOM to figure out exactly what needs to change in order for that to happen, and generates (and subsequently applies) a stream of operations to perform on the actual DOM in order to enact those changes. The result is that there's nothing in the real DOM except for what's actually displayed on the page, which is convenient for debugging and inspection via your browser's development tools. There's a simple demonstration of this aspect in the docs.

Another nice aspect of React is its JSX domain-specific language which I'll discuss more next.

As mentioned in the section on pug, essentially all of SMC's web frontend is built using React. Almost no HTML is ever sent from the server. Instead the frontend is built up by React. When user interactions with the UI need to be persisted, those are sent as event messages (typically over WebSockets) to the server, which may in turn respond with events that result in updating the UI in some appropriate way (the event messages are usually a JSON object of some kind). This is still an over-simplification (see for example the section on Redux later), but that's the basic idea.

JSX

JSX is a language that comes as part of React. It's a superset of JavaScript that allows embedding templated HTML. In some ways this resembles the bad-old-days of mixing code with HTML à la PHP. But it does have some advantages too, described in the linked docs. It's actually a very convenient way to use markup to describe how a UI element should be rendered. It's also a convenient way to nest UI components. For example, one might define some UI component as a class that extends `React.Component`:

```
class MyWidget extends React.Component { ... }
```

This now lets you use `MyWidget` in JSX as though it were any other HTML element like:

```
<div id="widgets">
  <MyWidget name="foo" />
  <MyWidget name="bar" />
  <MyWidget name="baz" />
</div>
```

and so on.

React can be used without JSX, but it saves a lot of verbosity and is probably a bit clearer, especially to anyone with HTML template experience.

CJSX

If JSX is the preferred way to write React components, this presents a challenge for integrating JSX with codebases that otherwise use CoffeeScript. One could write everything in CoffeeScript *except* for the code for React components (which would have to go in separate JSX files), but that introduces another difficult cognitive overhead.

To solve that, the CJSX language is just a simple superset of CoffeeScript to support JSX-like syntax. In other words, CJSX is to CoffeeScript as JSX is to vanilla JavaScript.

So this is what all the `.cjsx` sources (something that might be new to most readers) are in SMC. If you see a file in SMC with the `.cjsx` extension you can bet there's probably a React component defined in there somewhere.

Redux

It's a little tricky to explain exactly what Redux 'is' without specific examples. According to its docs:

> Redux is a predictable state container for JavaScript apps.

It's really little more than a simple protocol for application state updates by way of immutable state containers and pure functions that return an updated state based on some action performed on it (where the action can be any abstract operation that results in an updated application state). These functions are called reducers.

There's very little else in Redux--it's mostly convenience functions for managing a state object, and combining reducers to produce new states from state changing actions.

The purpose, all in all, is to provide a sane, predictable, reproducible way to manage and track (using middleware that logs actions) the live state of a complex application. We'll come back to this later with some specific examples. SMC wraps most of its use of Redux into its own abstractions that are implemented in `smc-webapp/smc-react.coffee`.

React-Redux

SMC's `smc-react.coffee` modules also makes use of the React Redux JavaScript module to tie Redux state objects to React containers (i.e. update displays when the state changes--abstracting the state itself from any given view of the state). This is just a package for making it convenient to implement model / view separation in React components. The idea is to design React components that are stateless and just display a "snapshot" of some data that might be in the state, and then wrap the stateless views in "container components" that do have state, and contain one or more stateless presentational compoenents. These then handle updating the view upon state changes.

React Redux makes it easy to auto-generate these "container components", connect them to a Redux state and its reducers, and re-render the underlying view every time the state changes. This includes defining a function called `mapStateToProps` which, given any application state, specifies which "props" (variable data) of the view are associated with the given state. So when the application state changes, it can check which "props" in the view have changed, and determine whether or not the view needs to be re-rendered, including exactly which sub-components need to be re-drawn.

If this is unclear, probably the best way to understand quickly is to read the example in the Redux docs. We will go more into exactly how SMC uses React-Redux later.

webpack

Preparing a large, multiple-file web application consisting of specialized JavaScript dialects like CoffeeScript and JSX with many interdependencies, as well as external dependencies, and getting everything to load in the correct order is tricky.

For one, the modern ECMAScript supports features not supported by the JavaScript on browsers, such as the `import` statement for loading variables, classes, and functions into other files (without polluting the global namespace, as was necessary to share between JavaScript files in the bad old days). Unfortunately, most (in fact no) browsers support this feature. One also needs ways to find static resources relative to JavaScript modules, transform the development dialect into JavaScript that can run in the browser, minify and/or obfuscate the code, and put it all together in a big bundle that loads everything in the correct order.

webpack is one of a number of popular build tools that serve this purpose. The entry-point to a webpack project is a file called `webpack.config.js` (or in SMC's case `webpack.config.coffee` since it uses CoffeeScript just about everywhere). You can think of `webpack.config` a little bit like the `setup.py` in a Python project, but don't take the analogy too far--it doesn't work the same way (a larger part of this purpose is also served by the `package.json` file that defines npm packages).

The `webpack` CLI then reads in this `webpack.config` and outputs a single file containing all your Javascript. This is of course the most basic usage--SMC currently actually generates three JS files (from three separate "entry points" to the dependency graph webpack generates). It also generates the `index.html` file that is served at the root of the website (from the aforementioned `index.jade` template) into which webpack inserts `<script>` tags that load its generated JS files. It also does some other tricks, such as appending a hash to the JS filenames so that they can replace cached versions whenever the source changes.

In practice it's less convenient to run `webpack` over and over again; instead one can run `webpack --watch` which watches all files in the project for changes and rebuilds continuously.

Conclusion

As previously stated, this is only a partial list of the tools that go into building SMC--particularly the core backend and client code. It doesn't even discuss the many dependencies that go into its various features, such as Jupyter, browser-based editors and terminals, chat clients, etc. Later I may include an update to list more of those.

How it all works

High level view

The high level architecture diagram from the Readme in SMC's source is accurate:

```
  Client    Client    Client   Client  ...
    /|\
     |
  https://cloud.sagemath.com (primus)
     |
     |
    \|/
HAproxy (load balancing...)HAproxy                  Admin     (monitor and control system)
/|\       /|\      /|\      /|\
 |         |        |        |
 |http1.1  |        |        |
 |         |        |        |
\|/       \|/      \|/      \|/
Hub<----> Hub<---->Hub<---> Hub <------> PostgreSQL
          /|\      /|\      /|\
           |        |        |
  ---------|        |        | (tcp)
  |                 |        |
  |                 |        |
 \|/               \|/      \|/
Compute<-------->Compute<-->Compute <--- rsync replication  to Storage Server, which has BTRFS s
```

It may be helpful to explain some of the entities in this diagram a bit more.

Client

This is the SMC client interface, built primarily with React and bundled together webpack as described previously. When a user goes to the root of SMC in their web browser, the HAproxy configuration serves it the `index.html` from its default backend, which happens to be a simple nginx server dedicated to static files. It also gets images, and the client JavaScript from the static server. Once the JavaScript takes over everything else happens in the browser including setting up the appropriate view for the client (whether or not they're logged in, etc.) and communicating with the hub using Primus/Engine.IO (through the HAproxy--more on that next). The majority of the client is implemented in the code in `smc-webapp` and `webapp-lib`, with some bits from `smc-util`.

HAproxy

HAproxy serves as the front line to all connections from clients to SMC. It routes all connections to different backends depending, primarily, on the URL (and port). The main frontend is of course HTTPS over port 443. By default requests are sent to the static file server (nginx) as mentioned above. Most other requests are sent to the 'hub' backend, which may be running any number of the hub servers, one of which is selected using the currently configured load balancing scheme (it also uses a session cookie to keep individual clients connected to the same hub instance).

Hub

The "Hub" is the primary server backend for SMC, built on Node.js as described previously. It consists of an HTTP server with Primus + Engine.IO attached to handle real-time bidirectional client/hub communication. Most communication between the Client and the backend happens through the Hub, whose HTTP server uses Express to route requests to different services (account management, project management, payment, etc.). Each Hub also sets up a `ComputeServerClient` which gives it access to all the running compute servers (discussed next). The names and URLs of all the available compute servers live in a system table in the database.

It also uses node-http-proxy to create an HTTP proxy server associated with each Hub (on port number one higher than the Hub's HTTP port). The proxy handles all requests that are to be forwarded to individual compute nodes (such as requesting files, or resources on web servers belonging to a specific project, including the Jupyter notebook server's websocket). HAproxy doesn't know anything about the compute nodes themseves--it just sees URLs that look like they belong to a project (they begin with a project UUID) and forwards those requests to the Hub's proxy, which in turn checks that the requester is authenticated and has permissions to access that project's resources. The proxy then forwards the request to the appropriate port on that project's compute node:

```
Client <--> HAProxy <--> Hub Proxy <--> Compute
```

Compute

Compute servers are where the real work gets done in SMC projects. Every project is associated with a specific compute server where all their data is stored (by way of storage servers mounted on the compute node) and where all process and computation tasks done by the project are performed. This includes running Sage. The Compute servers are Linux VMs with varying degrees of hardware capacity, depending on how much you're willing to pay. In most cases the servers are shared between projects (you don't have admin on the servers) though in principle one could pay for one's own compute server as well.

Otherwise, one can do quite a bit of different things on their compute node, including log in to the shell directly (you log in as a user named after your project's UUID). This can be done either through the web terminal in SMC, or one can SSH in directly.

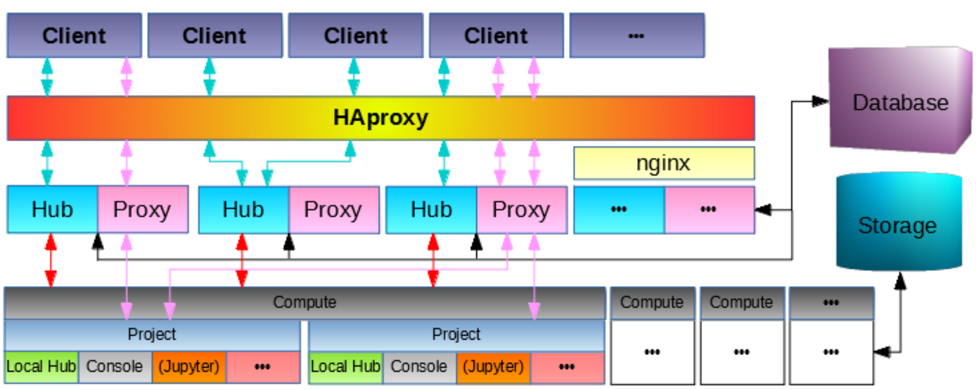
Each compute node also runs a simple socket server `smc-hub/compute-server.coffee` that is used by the Hub to communicate with the compute node (using simple JSON messages). For example, one can make status inquiries on the node, or send commands to run a command in a project.

Additionally, each *project* runs a couple per-project daemons when the project is created and running. These include the the console server (`smc-project/console_server.coffee`) which provides the backend for the web terminal, and the "local hub" (`smc-project/local_hub.coffee`). We haven't yet detailed everything that the "local hub" does, but whereas the "compute-server" manages the entire compute node, the "local hub" runs per-project (as that project's user) and helps coordinate connections between software running in the project and the "global hub" (i.e. the Hub, through which the client is communicating).

Whereas the compute server is used to issue commands to the compute node on behalf of a project, such as starting the project's local hub, the local hub then takes care further actions on behalf of a specific project. This helps to further logically separate projects from each other.

Deeper view

With all that said, let's consider a more complete picture of the current architecture (which still leaves a lot out, but incorporates some of the additional elements discussed above):



A few explanations about this diagram:

- The cyan arrows represent communication between the client and the hub. All client communication goes first through HAProxy, and then interactions--particularly those that don't directly involve projects--are otherwise between the client and the hub. This includes authentication and user settings, billing, and some things involving projects like project creation.
- The pink arrows are communications between the client and a project, and/or processes (such as the Jupyter server) running within a project. Again, this goes through HAproxy, but then passes through the hub proxy which forwards the request directly to the relevant project.
- The red arrows are communication just between a hub and a compute node, such as creating and destroying projects on behalf of a user, or getting status information about the compute node.

This diagram also demonstrates a few possible use cases for clients (certainly not exhaustively). The left-most client has connections both to a hub, and to the proxy associated with that hub, through which it is directly accessing resources on some project.

The second and third clients are both connected to the same hub, and are not connected to any projects (perhaps they're just setting up their accounts, or doing other administrative tasks not related to a project).

The fourth client is connected to the third hub, and is connected to resources on two different projects (albeit on the same compute node) through the same proxy.

This is of course still leaving out a lot of details that would be hard to fit on a single page diagram.

How the Client works (an example)

There are of course many different aspects to the question of how the SMC web client works. The easiest way to explain might be go step by step through what happens when a user points their browser to SMC and a page loads. Obviously this assumes we're observing at some particular scale where details like transport protocols are assumed. However, if we just gave a bullet list many points may be unclear, so what follows is a lengthy narrative of what happens.

Let's also assume, for this particular example, that the user has already created and logged into their account, and has at least one project already. So when they first load SMC in their browser, what (currently) happens is they land on the `/projects` page that shows the list of projects they have access to.

Initial page load and connection

When the user first goes to `https://cloud.sagemath.com/` the request is handled by HAproxy which routes it based primarily on the path. In this case the path is just `/` so it is routed to the default backend, which is the nginx server, and is served the default page--`index.html`. This is a static file generated the last time the administrator ran the webpack build. As previously mentioned there's very little on this page except the blinking "Loading SageMathCloud..." banner you first see (which works entirely in HTML + CSS), followed by some script tags that load the React site and related libraries.

In particular, the last script it loads is called `smc.js`, and this is where everything happens. To understand what's in this script, recall that it was generated by webpack, from one of the webpack build's three entry-points. In this case it's the entry-point named `webapp-smc.coffee`. This in turn "requires" three files in the following order: `smc-webapp/indes.sass` (this is compiled into a CSS stylesheet), `smc-webapp/client_browser.coffee`, and `smc-webapp/entry-point.coffee`). The end result of this you can think of almost as though each of these files were loaded one by one in the browser with `<style>` and `<script>` tags, but in reality they're all glommed together into a single file (sort of like building a single `.a` archive from multiple `.o` object files). When you run in development mode you can see quite explicitly how this works, but this is a detail about webpack and not particular to SMC, so I'll leave it as an excercise.

We'll look first at `client_browser.coffee` because some important things happen here as soon as it loads. This module defines a class called `Connection` (itself a subclass of a more generic class of the same name in `smc-util/client.coffee`). It immediately creates a single instance of this class as a global variable in `client_browser.coffee` named `connection`. It's this `Connection` object that sets up the Primus client and begins setting up communication with a hub as quickly as possible. The Primus client is responsible for the details of setting up WebSockets where available, or falling back on long-polling techniques when not. It's worth noting here that Primus is configured with an HTTP path that it can own for its own protocol communications with the Primus server. By default this path is `/primus`, but SMC has it configured (see `webapp-lib/primus/update_primus.coffee`) to `/hub`.

Assuming one or more hubs are already running (the full server-side story should be described in another chapter), HAproxy recognizes the path `/hub` and forwards Primus's connection to start talking to one of the hubs. Each connection Primus makes is handled by an object that Primus calls a "spark". (This name is used so as to not be confused with an actual "socket" or something like that, since Primus is abstracting out the details of the underlying I/O method). Most of SMC's code doesn't use the word "spark" and just uses "conn" or "connection".) Each spark is given a unique ID, which may be reused in some cases for example when restablishing a previously established connection. However, let's assume this is a brand new connection. Each hub maintains a hash table mapping from this connection ID to an instance of a `Client` class (`smc-hub/hub.coffee`) that is used to manage the hub's connection to each client. Since this is a brand new connection the ID is not yet in the hub's table, so it creates a new `Client` from this connection and writes the client's ID to the socket so that the client can know it too. After the client `Connection` receives its ID, it installs its default "ondata" handler--a callback function that serves as an entry point to the handlers for all subsequent data it receives from the hub.

Redux setup

So far all we've described is what happens when `client_browser.coffee` runs. Next in the list is a module called `smc-webapp/entry-point.coffee`. This is where we actually set up the user interface (note that that doesn't happen at all if we can't at least establish a connection to a hub first--there are also fallbacks for displaying messages to the user in case there are delays in making that connection). The first module loaded from `entry-point.coffee` with any notable side-effect `smc-react.coffee`. This initializes a single instance of a class called `AppRedux` which it exports to other modules with the variable name `redux`.

`AppRedux` is the driver for SMC's own very-high-level wrapper around Redux. Explaining this is difficult unless you've read at least the introduction to Redux earlier in this document, if not read and understood the full documentation for Redux. `AppRedux` maintains a sort of Redux meta-store. It contains only a single actual Redux store (as created with `redux.createStore`), but this is used to manage any number of sub-stores represented by key/value pairs at the top level of the main store's state. The reason for this is that each page in SMC's UI might have its own state that is mostly independent of the state of other pages. For example the "account" page may have state that is mostly independent of the "project" page's state, so the full state of the application looks something like:

```
{
    "account": { <...account page state...> },
    "projects": { <...projects page state...>},
    ...
}
```

There is even a sub-store called "page" for managing the overall current state of the CLI, such as what the currently active tab is.

There are a few reasons for organizing things this way:

- It keeps the application state fairly sanely organized, with sub-states for each page, and easy routing of actions to the sub-states the affect.
- However, since the entire state is stored in a single Redux store (as opposed to, say, having separate stores for each page) it is also possible to produce actions that affect multiple pages, or even other parts of the application state that are not tied to a particular page or view. For example, the <Page> component, which we'll look at later, connects to several different stores.
- The `AppRedux` class makes it possible for each page/view to independently and dynamically register a sub-store for itself. The `AppRedux` instance that is passed throughout the application then serves as a sane way to manage all the known state stores.

  In fact, much of `AppRedux`'s API mimics the Redux library's own top-level API. So instead of calling `redux.createStore()` for each sub-store, one actually calls `AppRedux.createStore()` (the latter has some important differences from the former, however, which we'll come to later). In fact, since SMC names the `AppRedux` singleton `redux`, one *does* in fact literally call `redux.createStore()`, but it's important to be clear that here `redux` is an instance of `AppRedux`, not the Redux library itself.

The whole thing is fairly smart, and almost nothing about this framework is particular to SageMathCloud--it could (and probably should) be factored out into a stand-alone package at some point. We haven't explained everything about it yet either but will add more details soon.

Anyways, all that's happened so far is the `AppRedux` singleton has been created. No stores have been added to it yet. But it's important to explain what it is before moving forward.

Finally, this also creates a [React component] called `Redux` that encapsulates the `AppRedux` instance as its sole property. This is just a thin wrapper around React-Redux's top-level `<Provider>` [component] which is used to pass the Redux store down to all elements of the view.

Server stats

Continuing to follow `entry-point.coffee`, the next module that's loaded is one called `smc-webapp/redux_server_stats.coffee`. This actually sets up a "synchronized table"--a client side view of one of the database tables--and attaches this to the `AppRedux`, which also carries around a collection of synchronized tables which are instances of a class called `Table` defined in `smc-redux.coffee`. The tables are actually *not* part of the Redux store, and are probably just attached here for convenience's sake, though this may seem a little confusing at first. We will come back to this later.

Next, an area for system notifications is set up similarly. This isn't immediately visible so it's not particularly interesting to the story.

Page actions

The next module of interest imported from `entry-point.coffee` is called `smc-webapp/init_app.coffee`. This adds a store to the `AppRedux` for the overall application page. This includes information like the currently active tab, as well as the "ping" status one sees at the top-right corner, among other things. The `redux.createStore` call adds the `"page"` store. The different state variables associated with that store are in the `stateTypes` (currently it seems that not much is done with the types themselves). You can also see that it sets the initial default active page to the `'account'` page. This `active_top_tab` will be used later when we finally render the page.

It also uses `redux.createActions` to instantiate an instance of the `PageActions` class. Most of the methods on this class combine what Redux calls "action creators"--functions that return a new Redux action--with dispatching of that action. For example `set_active_tab` dispatches a page state change which sets the new `active_top_tab` value--it then also performs any side-effects associated with that state change, such as setting the window title, or loading projects. (Note: This method hasn't called anywhere yet, as it depends on other application state stores being set up first.)

Finally, this module installs some event handlers that impact the page state (now that the `"page"` store has been set up), on an object named `salvus_client`. This is actually the same `Connection` object that was instantiated back in [Initial page load and connection]. "Salvus" is the working name for earlier versions of SMC, and there are still references to it throughout the sources. Here, the use of `salvus_client`, is probably just code smell that hasn't been cleaned up yet.

For example, it calls `salvus_client.on("ping")` to set a handler to update the page's ping time display (actually, just the underlying state is updated here--we haven't attached a display to it yet) every time the connection receives a ping back from the server.

Rendering the app

Finally, after all this, we're ready to actually display the app. SMC has both mobile and desktop client UIs, each of which have separate entry-points to their top-level views. Let's say we want to display the desktop client, so it calls `desktop.render()`.

In short, `render()` calls [`ReactDOM.render()`] to create the page from the aforementioned [`<Redux>` component] wrapping another React component called simply `<Page>`. This `Page` component goes along with the `PageActions` and `"page"` store created previously. It also has a little bit of redundancy with the information captured by those classes, some of which could probably be eliminated. But it does make sense to keep separate--whereas the definitions in `init_app.coffee` represent application state independent of the view, the `<Page>` component implements a specific view of the state.

`<Page>` is created using a helper function called `rclass` which is defined in `smc-react.coffee`. This is a wrapper around both React and React-Redux for creating the component class, and hooking different props on the component up to the Redux state. The component can have props that are filled/updated from any number of sub-stores in the `AppRedux` instance. These are given by the `reduxProps` argument passed to `rclass`. So you can see that the `<Page>` component uses all values stored in the `"page"` store as props, as well as a few others (such as `"account"`, for displaying your login status and/or username on the page).

The `<Page>` component's `render()` method contains some JSX-style markup for everything you see on most pages, such as the top nav bar with project tabs, the notification bell, ping status, etc. Most of these components are defined in other modules. At the very bottom it contains an `<ActiveAppContent>` component that is responsible for displaying the rest of the page depending on what the current view is (whether it's the projects page, the settings page, and individual file in a project, etc.)

Routing to the projects page

At this point the page has been rendered. But one thing you'll notice is that if you have an account and are logged in, you actually get redirected immediately to `https://cloud.sagemath.org/projects`. But we said earlier that the currently displayed page is stored in the `page.active_top_tab` state variable which defaults to `'account'`. At what point does this change from `'account'` to `'projects'`?

In fact it does start out on the account page. If your browser is running slowly enough you can catch this briefly, while the page displays the "Signing you in" message. This is part of a component called `<LandingPage>` defined in `smc-webapp/landing_page.cjsx`. In fact this is the page that is displayed when you're not logged in (i.e. the `"remember_me"` cookie is not set), with all the marketing content and account creation box. But if your cookie is set, then it just shows the "Signing you in message".

Meanwhile, on the server side the server checks the cookie, and if it's valid it sends a "signed_in" message over the Primus socket. Meanwhile, there is an event handler registered in `smc-webapp/account.coffee` for the "signed_in" event. This checks a module global variable called `first_login` (which defaults to true). It subsequently sets `first_login` to false, and calls `history.load_target('projects')`. This in turn calls `redux.getActions('page').set_active_tab('projects')` to update the page's state so that the current view is the "projects" page. This in turn also calls the `set_url` helper function which manipulates the browser's history to set the new URL to `/projects`.

Finally, due to the React-Redux connection between the page state, and the `<Page>` component (particularly, the connection between `active_top_tab` and the `<ActiveAppContent>` sub-component of `<Page>`) the `<ActiveAppContent>` automatically gets re-rendered, this time with the projects page as its contents (which is implemented in `smc-webapp/projects.cjsx` as a React component called `<ProjectsPage>`).

Rendering the projects page

There's something that has been happening in the background all this time that we have not talked about yet. When the client UI was first loaded (i.e. `desktop_app.cjsx`), the sources for several other pages (such as the projects page, the account page, etc.) were also loaded. In particular, the projects page is implemented in `smc-webapp/projects.cjsx`. This has some non-trivial side-effects.

One of these side-effects is that the `ProjectsTable` class is registered with `AppRedux` via the `AppRedux.createTable()` method. This class is a client-side front-end to the `"projects"` table in the DB backend. This creates an instance of a fairly complex object called `SyncTable` defined in `smc-util/synctable.coffee`. We won't go into detail of how this works, but basically it keeps a client-side copy of the results of queries to the Hub's database, and sets up change listeners that synchronizes those cached query results every time the real database changes (via a "changefeed", which asynchronously pushes an update to the client every time the query result changes).

In effect, while the page has been loading, in the the client app has been sending a database query to the server for all the user's projects, and receiving the result. The `ProjectTable` has a change handler that updates a prop called `project_map` which contains the result of the query for all the user's projects. When the `<ProjectsPage>` component renders, it checks to see if `project_map` is undefined. If it *is* undefined this means the database query hasn't completed yet (if the user has no projects the result of the query would be an empty list, but not `undefined`). In this case a "Loading..." spinner is rendered. The React-Redux connection ensures that the page is re-rendered whenever `project_map` changes, so as soon as the database query is completed the page will be re-rendered.

The rest is fairly straightforward, given an understanding of React. The `<ProjectsPage>` component consists of a number of sub-components, including one called `<ProjectList>` where the all the projects are looped over and displayed in a table. There are some additional complications related to the ability to filter projects in various ways, but there's nothing special to this.

Conclusion

And that's it! We got from an empty browser window to the user's projects listing. Many aspects were still simplified, as this was a long enough journey as it is. But understanding this process should give a basic understanding about how most other pages in the SMC client are displayed.

When interacting with an actual project things are a bit more complicated, but the basic principles are the same. In this case, many requests are routed by HAproxy not to the Hub, but to an associated Hub Proxy that redirects the requests straight to services running on the project's compute node. For example, when running the Jupyter Notebook in an SMC project, the Notebook server is configured so that all its websocket requests go through a URL that starts with `<project_id>/port/jupyter`. This way the Jupyter Notebook can set up its own websockets as normal (without any special SMC-specific modifications) and the requests are proxied directly to the project's Notebook server.

In principle the scheme supports other web services embedded in SMC as well.

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.