SketchWorld

A Mobile-Friendly Sharing-Focused Creative Coding Platform for Schools, Students, Artists, and Enthusiasts

Julien Blanchet

Devin Balkcom

Dartmouth College

Department of Computer Science

Spring, 2015

1 TABLE OF CONTENTS

2 MOTIVATION & BACKGROUND

2.1 AREA OF OPPORTUNITY

In recent years, as smartphones have become ubiquitous and people's time spent interacting with the digital world has expanded, Science, Technology, Engineering and Mathematics (STEM) has come to be seen as increasingly important to the lives of ordinary people. A subset of STEM, Computer Science has emerged from being seen as an area confined within the realm of specialists and hackers to a widely relevant technological skill. Public support has exploded for the expansion of computer science programs in both K-12 and higher education,[1] and enrollment in college level computer science programs has reached an all-time high.[2]

As computer science programs are introduced to younger grades and as a wider swath of the population enrolls in existing courses, teachers face a challenge in finding effective ways to teach a highly technical subject to younger students and students with less background experience in the field. Enabling students to create highly tangible and engaging works early in the learning process can be highly motivational, making programming fun and giving students the sense "I can do this." Those enrolled in traditional college and post-graduate level computer science courses would also benefit from such an innovation, as would established tinkerers and creative coding enthusiasts – who could make use of such a platform for rapid development and sharing of prototypes.

These days, many children learn how to operate a smartphone or tablet before even starting school. Many teenagers use their devices almost constantly throughout the day to play games and socialize with their peers. Even adults have been known to be prone to the occasional smartphone game addiction (as my mother can attest to with the game 2048).[3] As educators of computer science, we have an opportunity to leverage students' attachment to these devices to motivate students by showcasing the potential of programming and instilling students with a sense of pride over their achievements.

[1] https://news.cs.washington.edu/2015/02/23/new-poll-reveals-washington-citizens-support-for-computer-science-education/

[2] http://now.dartmouth.edu/2014/04/enrollment-soars-computer-science-courses

[3] https://itunes.apple.com/us/app/2048/id840919914

2.2 STATE OF THE ART

Numerous technologies and products have been released recently which aim to lessen the learning curve in creating a work that a student would be proud of. Perhaps the most well-known of these is called Scratch, with over 9½ million projects shared on the site as of May 2015.[4] Started by the MIT media lab in 2003 with the goal of teaching children basic programming concepts, Scratch replaces the traditional text-driven programming paradigm with a Lego-like interface in which students "snap" together blocks (logical units) of code. Every unit of code is associated with a character (a *sprite*, in Scratch parlance), and might contain a sequence of blocks representing commands such as "move 10 steps forward" or "go to x, y position 100, 200." Although these commands are easy to learn and may seem quite simple, teachers have shared student-created works of remarkable sophistication, such as a multi-episode animation about warrior cats, [5] a rope-sliding rock-dodging game,[6] and an intense scenario based memory tester.[7]

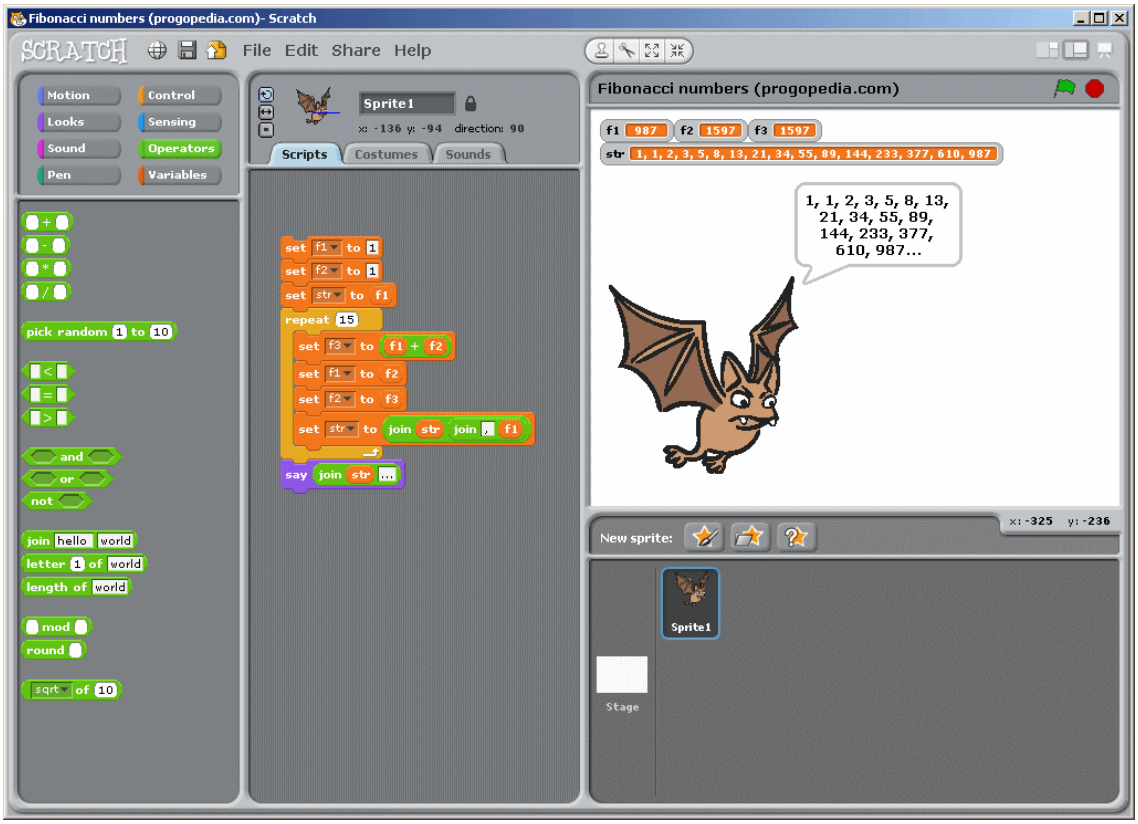


Figure 1: Scratch offers a drag and drop interface perfect for learners

---

[4] https://scratch.mit.edu/about/
[5] https://scratch.mit.edu/projects/62747672/
[6] https://scratch.mit.edu/projects/28478704/
[7] https://scratch.mit.edu/projects/57765552/

Scratch is an awesome tool for introducing people (especially children) to computer science in a fun and motivating project-oriented way. However, sketches created through Scratch are not designed for mobile consumption and are not natively sharable on these devices. Furthermore, although the drag and drop interface is genius for its removal of some mundane frustrations often experienced by beginning coders (such as syntax errors) and although this interface does a commendable job of representing important computer science concepts such as program flow, variables, and object orientation, Scratch is inherently more limited than standard programming languages used in the industry. No matter how advanced a student becomes in Scratch, he/she will be limited to sketches involving a set of sprites doing a (potentially complex and really interesting) sequence of actions predefined by Scratch's developers. Therefore, Scratch cannot fully instruct a student in the wider range of programming concepts employed in general purpose text based computer languages that are used professionally. Scratch is only the first step in becoming a truly proficient programmer.

The Processing programming language is another recent development that makes creating pride-worthy creations easier for beginners. Unlike Scratch, Processing is text based and borrows its syntax from Java, one of the most prevalent programming languages in the world (in fact, the central Processing project is actually implemented as a library running Java code behind the scenes). Instead, Processing exposes a simple, well-documented API[8] which is oriented around drawing shapes and objects onto a canvas. Creating a Processing sketch equivalent to a Scratch sketch requires significantly more effort and a greater understanding of how different parts of code work together to achieve a desired effect. Whereas a sprite in Scratch has a built-in location, rotation, and size, creating the equivalent in Processing requires manually storing and updating values for x and y position, rotation, dimensions, as well as explicit instructions on how to render the object. However, learning and implementing these concepts in Processing is a practical and foundational achievement in understanding computer science more generally. Furthermore, many interesting Processing sketches can be created without a notion of a "character".

```
void mouseDragged() {
  line(pmouseX, pmouseY, mouseX, mouseY);
}
```

Code Snippet 1: One useful, yet barebones Processing sketch that lacks a notion of a "character": a simple drawing program

Processing was originally implemented as strictly a Java based language and coded using a Java-based IDE. Since its inception in 2001, multiple sister projects have attempted to bring the simple Processing API to new platforms. The most popular of these by far is Processing.js[9], which compiles Processing code in to JavaScript targeting the HTML5 canvas element. Since all the leading mobile operating systems offer HTML5 browsers and support web-technology-based apps, this library enables mobile use of Processing code. Although the

[8] https://processing.org/reference/
[9] https://processingjs.org/

Processing.js project doesn't offer its own online editor, 3rd party developers have published their own, called Open Processing[10] and Sketchpad.cc.[11]

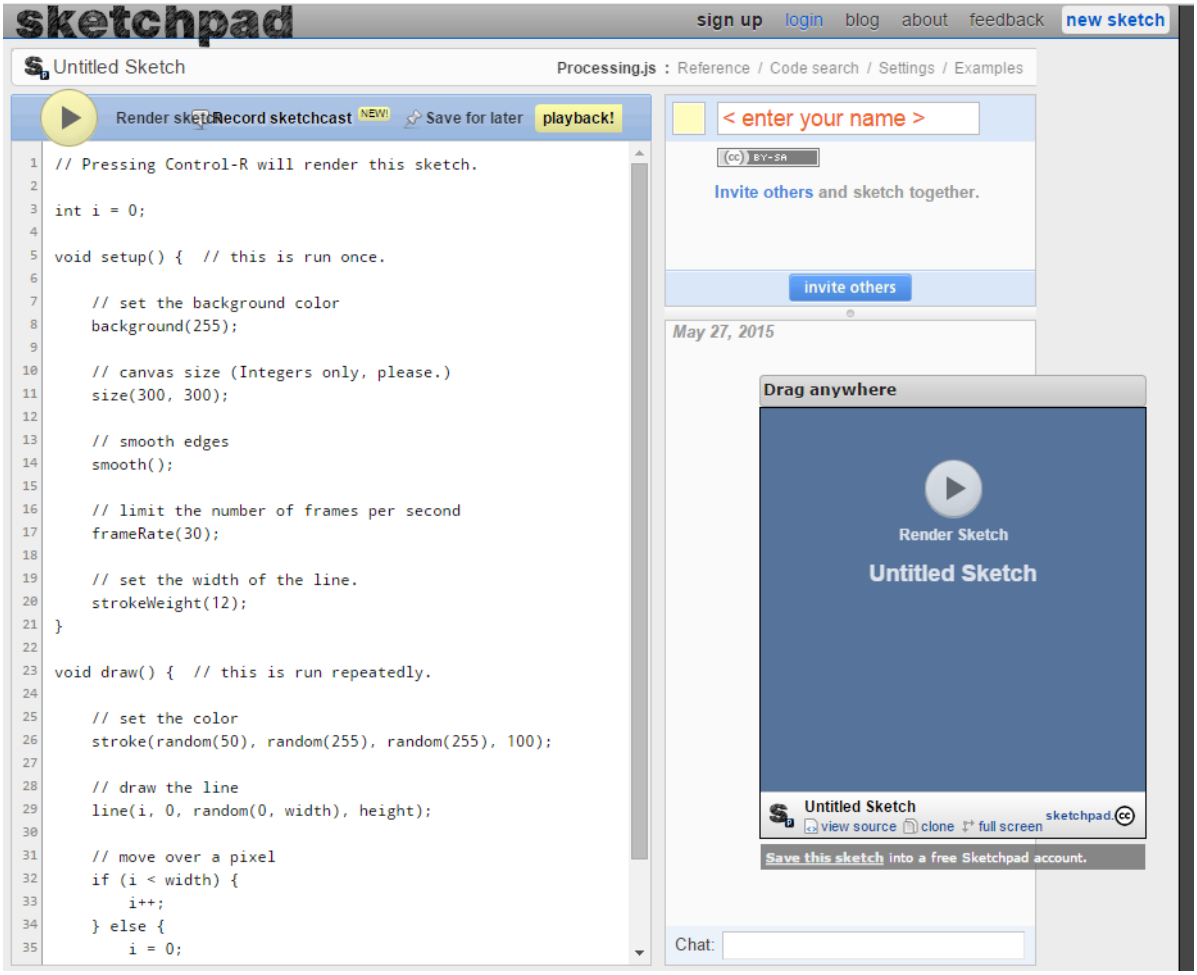


Figure 2: Sketchpad.cc offers a web-based Processing.js experience, but the editor is frustrating and the interface is not mobile friendly

Open Processing, Sketchpad.cc, and scratch.mit.edu all offer collaborative, community-focused editing experiences that promise to make learning to code a fun and exploratory process. However, none of these websites is well suited for playing (much less editing) sketches on a mobile device. Furthermore, the two Processing sites offer subpar text editing experiences that lack live syntax highlighting, proper indentation support, code completion, and syntax verification. This makes for a frustrating editing experience in which minor syntax errors can block the entire sketch from running without feedback as to the nature of the error. I believe offering a high quality editing experience for Processing is important and that providing sketch execution

[10] http://www.openprocessing.org/
[11] http://sketchpad.cc/

capabilities on mobile devices will interest and motivate students as well as offer some unique creative opportunities.

2.3 VISION

I addressed the above limitations through the creation of a mobile friendly web application for developing and running sketches that offers a rich text-editor, a fast code/test iteration cycle, and a mobile friendly design. With further development, the site will merge these strengths with the strengths of Scratch and sketchpad.cc to offer seamless sharing and collaboration functionality as well as community pages containing featured sketch galleries and user-contributed tutorials. This app works online and offline on whatever device the student has available to them, synchronizing their sketch portfolio wherever they sign in.

I envision a student opening the app through the encouragement of a teacher or after reading about it online and immediately being presented with a world brimming with creativity and energy and a sense of possibilities which revolves around the creation and sharing and consumption of sketches. The student can start a sketch of their own from a blank slate, or based off something they saw in the community. With a rich text editor (that may one day evolve to incorporate drag and drop features inspired from Scratch) and the guidance from well-designed tutorials, the student can learn industrially applicable programming concepts while creating an exciting game, animation, or code-driven piece of art. Having accomplished this, the student can publish their work to the community and create a shortcut on the home screen of their mobile device that, when clicked, opens their sketch in full screen glory as if it were a full-screen app.

The student can then pass their phone to friends, inviting them to check out the sketch. The friends, impressed with the student's ability to create such a cool piece of work, realize that they too would be interested in making sketches. The friends then download and log into the app, create their own sketches, learn programming concepts for themselves, and show *their* friends what they made. Eventually, entire schools and entire countries full of kids realize the programming is fun and doable and go on to incorporate what they now understand about computer science into forward-thinking ideas that change the world.

3 FUNCTIONAL DESIGN

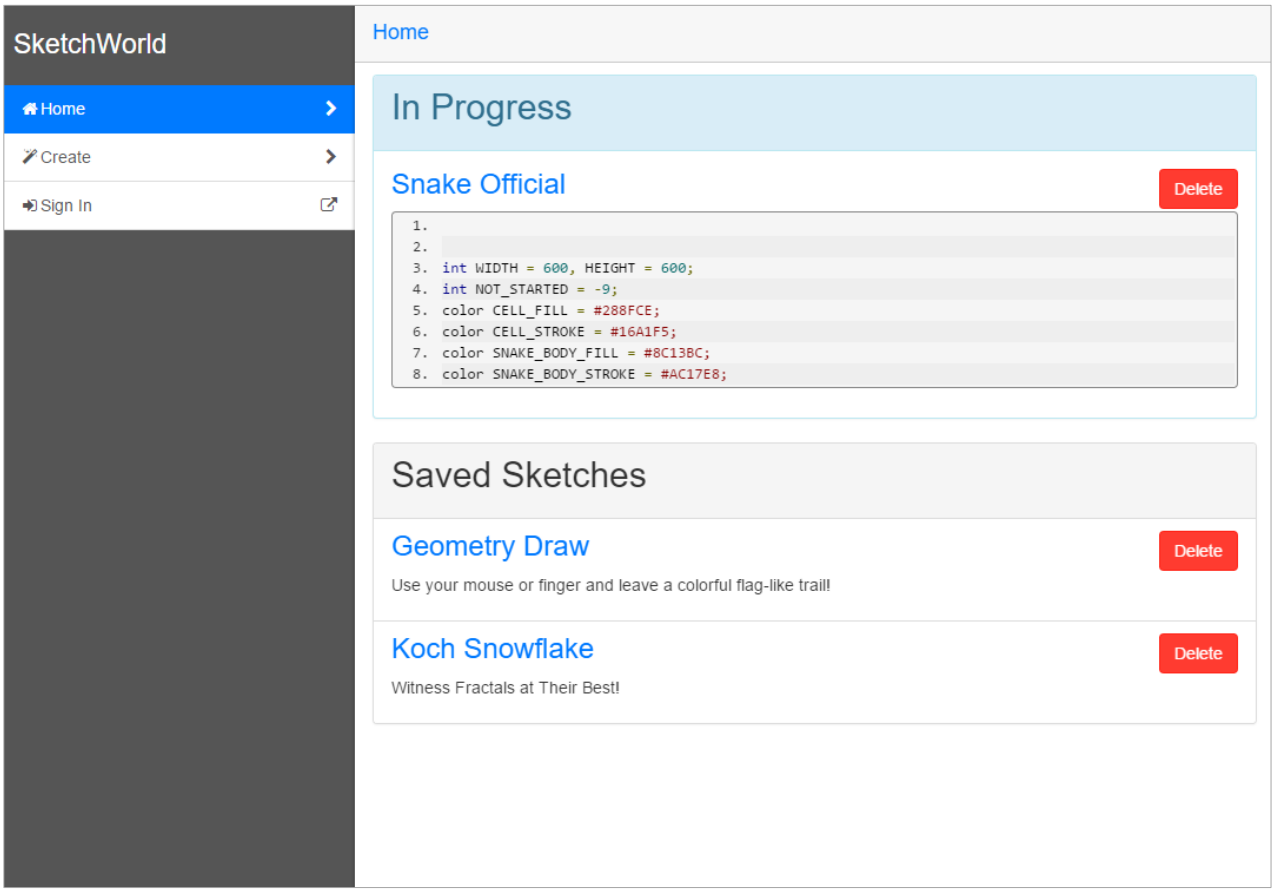
3.1 GENERAL LAYOUT

The layout and navigational structure of any app must be comprehensible if it's to attract usage. SketchWorld utilizes a main content area in conjunction with an omnipresent side menu that gives users a persistent sense of location within the app. Each of the apps *views* (to be explained later) is present in the left sidebar and is marked by a right chevron, giving the user a visual indicator that the selection of these menu items determine what is presented in the main content area. Simple, identifiable, yet visually pleasing icons accompany each menu option. Additional globally applicable functionality is exposed in the left area, such as the sign-in button. For these buttons, an external link symbol takes the place of the right chevron, indicating that these options do not alter the main content window but work with popups and dialogs. Page-specific functionality (such as the button to save a revision in the editing view, or the sync now button in the home view) is not exposed on the left sidebar. Rather, these interface elements appear in the top title bar of the app when

contextually appropriate, allowing for quicker access. To conserve screen space, the left side menu auto-hides on small devices and is available and dismissible with a clearly labeled button. Contextual top bar interface elements collapse into a submenu on smaller devices – less accessible, but necessary to conserve screen space.

3.2 HOME (DASHBOARD)

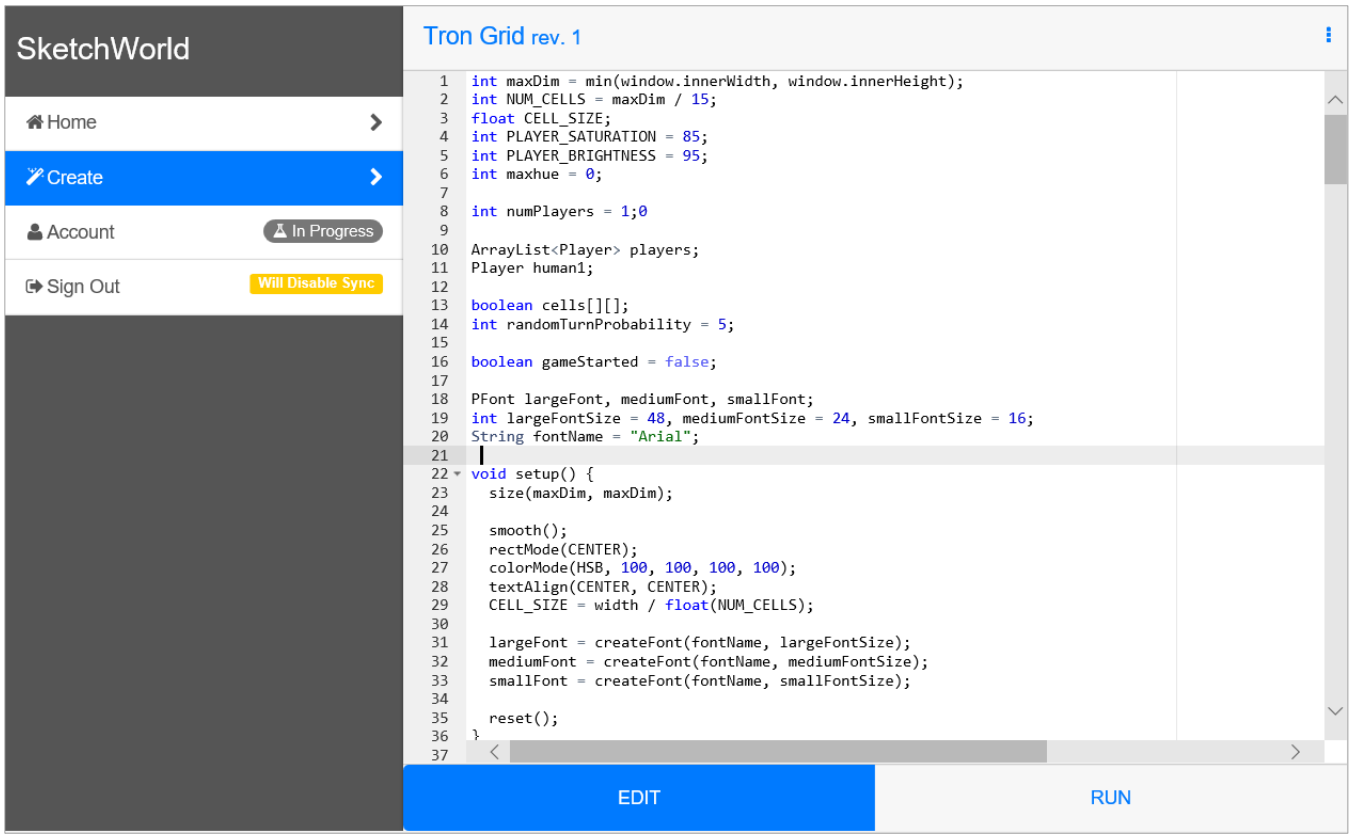
When opening SketchWorld, the user is brought to a home view that displays the titles and descriptions of sketches that he/she has recently worked on. The most recent sketch is listed at the top, enabling the user to quickly resume previous work. In the future this home view will also display community sketches and tutorials as well as updates from the development of SketchWorld (such as new features and events). This view is essentially a dashboard allowing the user to see an overview of recent activity, both their own and in the community.



3.3 CREATE (EDIT AND RUN)

The create view is where the user actually develops sketches. The create view has two modes, edit and run, that the user can rapidly toggle between using a bottom tab bar. Each time the user selects the run mode, the view is updated with the latest code from the edit tab. The text-editor embedded within the edit mode is

derived from the Ace editor project,[12] designed to yield a first class editing experience, involving full support for syntax highlighting, line formatting (such as indentation and spacing), bracket matching, code folding (hiding units of code such as a function body), and comment toggling. These features, especially syntax highlighting, improve the coding experience for learners and experienced coders alike by reducing frustration and calling attention to the syntax errors that bog down and frustrate new users. The user can edit properties of the sketch (such as the title and description) and can save a revision for later reference using the dropdown menu on the top bar. Future improvements may include the addition of Processing tailored code completion and multiple tabs of code, allowing for better code organization.



3.4 Accounts, Synchronization and Sharing

Signing in to SketchWorld is a completely optional feature, but enables the user to take advantage of sketch syncing, sharing, and publishing functionality. Rather than create new account credentials, SketchWorld leverages the OAuth 2.0[13] standard to allow users to login with either a Google or Facebook account. Once the user signs in, all sketches he/she makes or has made are synchronized with the server. This sync occurs quite

[12] http://ace.c9.io/
[13] http://oauth.net/2/

frequently. Every time the user visits the home view, the app checks with the server for sketch updates and loads any updated sketches it finds, and every time the user makes a change to the code or other sketch information in the create view this change is pushed to the server. If the same sketch is modified from two places before synchronization has occurred, the modification that was performed more recently will be propagated. In the future, there will be an account overview page where a user can see a complete view of their activity and sketch portfolio. At this point, sharing of sketches is not built into the system. However, the SketchWorld backend is setup to allow this in the future. Once this is implemented, a user on a mobile device will be able to pin a shortcut to their device's home screen that opens directly to a running version of their sketch with minimal UI – acting as if their sketch were a full-blown app. A user will also be able to share a sketch though a button which will invoke the platform's native sharing functionality to send a link to their sketch which, when clicked by the recipient, will open a device-appropriate landing page displaying the title and author of the work, as well as a description and a window running the sketch's code.

4 Technical Design

4.1 Challenges & Technical Overview

SketchWorld is designed to meet students where they are, which means it must target as many platforms as possible. For the time being, I have chosen to focus on Chrome Packaged Apps (which allow web-technology based applications to run offline on desktop operating systems); iOS, Android and Windows Apps (mobile apps which can run offline); and a website (accessible to any internet connected device, but which must function on Chrome, Firefox, Safari, and Internet Explorer). Although some large tech companies have the resources to build separate apps using the native technologies for each platform, I do not. A single page application built on web technologies (HTML, CSS, and JavaScript) is supported by all these targets and therefore was chosen as the most suitable technology stack for this project.

Even with support for the same general tech stack, these platforms vary in their support for newer features of the HTML and JavaScript standards and impose different security restrictions on the application package. Of most significance for the purpose of this thesis was Chrome packaged apps' restrictions on scripting. Only sandboxed pages are allowed to use the `eval()` method or `function()` constructor, both of which are necessary for proper operation of the Processing.js library. This restriction prompted me to design a solution in which all user sketches are sandboxed inside an iframe that communicates with the main page solely through inter-window POST requests. Because of these limitations and restrictions, I have coded to the lowest denominator in terms of platform features and the highest in terms of security restrictions.

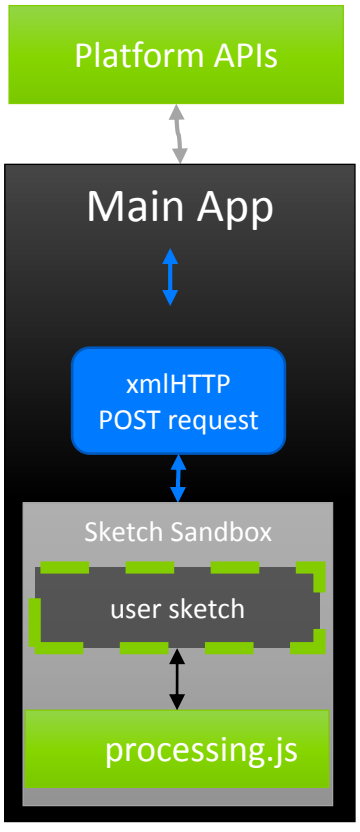


Figure 3: Solution for running Processing.js sketches in Chrome Apps

4.2 CLIENT DESIGN

4.2.1 Backing Frameworks: Angular.js, Mobile-Angular-UI

Although it is possible to build a single-page web application like SketchWorld from scratch using pure JavaScript, using a well-designed framework leads to less boilerplate code (especially around such mundane details as DOM ↔ JavaScript data-binding and URL routing) and enforces, to a certain degree, good programming practices such as modularity. Angular.js[14] is one of several frameworks designed to facilitate building web-apps, is supported by Google, has fantastic free tutorials,[15] and has an active community on Stack Overflow with over 96,000 questions as of May 2015.[16]

4.2.1.1 Decoupling DOM Manipulation from App Logic: Two Way Data Binding

Angular structures the underlying logic of the SketchWorld app. The developers behind Angular have three core design goals for the framework,[17] each of which has propagated to the design of SketchWorld. The first is to decouple of the HTML DOM manipulation from application logic. Modifying the HTML document from JavaScript is a cornerstone of any web app, yet built-in methods for performing this are verbose and prone to error, relying on hardcoded HTML strings within the `.js` file and a coordination of DOM element specifiers (such as IDs or class names) between HTML and JavaScript. Angular addresses this by offering automatic two-way data binding between fields within an HTML document and JavaScript objects. One does this simply by setting properties of a special `$scope` object in the JavaScript file, then using a curly bracket notation within HTML to refer to those values. Angular will automatically substitute the curly bracket expression with the computed value. HTML elements such as `input` that modify a value will automatically update associated JavaScript variables in `$scope`.

```
$scope.sketch = {
    title: "Demo Sketch",
    desc: "A basic sketch for demonstration purposes",
    revNum: 1
};
```

Code Snippet 2: Data Binding in Angular.js - JavaScript Side

```
<div class="sketch-list-item">
    <h2>{{sketch.title}} <span class="small">{{sketch.revNum}}</small></h2>
    <p>{{sketch.desc}}</p>
</div>
```

Code Snippet 3: Data binding in Angular.js - HTML side

---

[14] https://angularjs.org/
[15] http://campus.codeschool.com/courses/shaping-up-with-angular-js/
[16] http://stackoverflow.com/questions/tagged/angularjs
[17] https://docs.angularjs.org/guide/introduction

4.2.1.2 Decoupling Client Side of Web App from Server Wide: Client-Side Routing

The second goal of the Angular.js framework is to decouple the client side of a web application from the server side, allowing development of each side to progress in parallel and allowing for reuse on both sides (thus, the server that backends the SketchWorld web app may also serve as an API server for the SketchWorld mobile apps). Angular offers multiple features that support this separation, but the most important one is URL routing, which allows the web app to display different views associated with different URLs *without* loading a new page from the server – as opposed to a typical website which loads a new page with each address change. Angular does this by monitoring for changes in the anchor part of the URL (the part after the # symbol), which is defined in the HTML standard as a way to navigate to specific parts of a webpage (without triggering a reload) but which does not normally trigger any modifications to the DOM. This has the effect of giving the app a sense of location independent of the server – for example, http://sketch.world/app/#/create/sw-1432389605334 links to a specific sketch, whereas http://sketch.world/app/#/ links to the homepage, yet they both load the exact same HTML and JavaScript resources from the browser (Angular handles the navigation).

```
$routeProvider.when('/', {
    templateUrl: 'pages/home/home.html',
    controller: 'HomeController'
});
$routeProvider.when('/create', {
    templateUrl: 'pages/create/create.html',
    controller: 'CreationController'
});
$routeProvider.when('/create/:sketchId', {
    templateUrl: 'pages/create/create.html',
    controller: 'CreationController'
});
```

Code Snippet 4: URL routing in Angular using the ng-route module

4.2.1.3 Providing Structure to Web Apps: Templates, Directives, Controllers, and Services

The third goal of Angular is simply to provide a structure for a web app that promotes modularity and organizes code, thereby facilitating the entire process of building a web app ranging from designing the UI to writing business logic to testing. Angular enforces a model-view-controller (MVC) separation and provides four components to help achieve this. The first is the templating system, mentioned in 4.2.1.1, that offers automatic data binding to JavaScript variables, therefore enabling the view to be factored out of application logic. Yet some UI elements require most custom logic than simple data binding – a need that directives, the Angular's second MVC feature, help fulfill. Directives allow the developer to attach JavaScript defined behavior to a HTML element declaratively, without cluttering the app logic code. SketchWorld uses many built-in directives to control visibility of different parts of the app and a custom defined directive to instantiate the code editor.

```
app.directive('aceEditor', function(sketchStorage){
    return {
        restrict: 'E',
        templateUrl: 'components/ace-editor/ace-editor.html',
        link: function($scope, $elem, attrs){
            angular.element(document).ready(function(){
             window.editor = ace.edit('ace-editor');
             window.editor.getSession().setMode("ace/mode/java");
            });
        }
    }
});
```

Code Snippet 7: A simplified version of the ace-editor directive declaration

```
<ace-editor>
</ace-editor>
```

Code Snippet 5: Sample use of a directive

```
<div id="ace-editor">
Sketch Code Loading...
</div>
```

Code Snippet 6: A sample directive template

Together, directives and templates define the view of an Angular web app. Controllers, quite appropriately, define the controller (central business logic) of the app. Controllers are JavaScript functions that are run when the element they are attached to is created. They are in charge of initializing `$scope` (template-visible) variables and defining functions to be run whenever specific events occur (for example, define a function to be run when the user clicks the *create sketch* button). Controllers can be attached to any HTML element and are often attached to directives. In SketchWorld, I've explicitly attached controllers to the `ng-view` directive at various states (as shown in Code Snippet 4) as well as the <`ace-editor`> directive (see Code Snippet 6) and the sidebar template (see Code Snippet 8).

```
<div class="scrollable" ng-controller="SidebarController">
    <h1 class="scrollable-header app-name">SketchWorld</h1>
    <div class="scrollable-content">
        <div class="list-group" ui-turn-off='sidebarLeft' u
            <a class="list-group-item" href="#/">
                <i class="fa fa-fw fa-home"></i>Home <i cla
            </a>
```

Code Snippet 8: Controller being attached to sidebar.html template

The last component defined by Angular is a service, which is a controller-independent object that can be accessed from any controller in the app and is used to share common code (like a library). One very common use of services is as a component for communicating with a back end server. As mentioned in 4.2.5, SketchWorld defines a `SketchStorageService` which is used across all controllers to access, change, and synchronize sketch objects.

4.2.1.4 UI Components for Angular Apps: Mobile Angular UI

Angular.js supports the logic of building a web-app, but does not come with a set of user interface components – the developer is responsible for creating these using HTML and CSS. However, building a nice-looking and functional interface completely from scratch is repetitive

and time-consuming. For web-pages, I've grown familiar with the Bootstrap CSS framework,[18] which provides a responsive layout system (allowing the web page to adapt to differing screen sizes) and a collection of styles that makes it easy to develop a visually appealing interface. Bootstrap is one of the most widely used styling frameworks. However, this framework lacks a number of components that would make it suitable for a mobile application, such as sidebar menus and app bars. Rather than recreate these from scratch, I opted to use the Mobile Angular UI project,[19] which builds off of bootstrap to offer these components packaged as angular directives.

4.2.2 Overall Structure

The high level document structure of SketchWorld is defined in an index.html file. The `head` element pulls in the Angular and Mobile Angular UI libraries along with SketchWorld's custom Angular components (templates, directives, controllers, and services). The `body` element lays out four core sections of the app, enumerated below:

1. The sidebar. The sidebar.html template is pulled in using a `ng-include`[20] directive.
2. The top app bar. This bar holds the toggle button for the sidebar and has two placeholder areas meant to be populated by views using `ui-yield-to`[21] directives: the title area, meant to display the current sketch or the 'Home' label and the right action button area, which is used for contextual actions (the create and home templates can insert elements into these areas using the `ui-content-for`[21] directive).
3. The app content area. This is a `ng-view`[22] directive which is populated by the templates and attached controllers defined using the `$routeProvider` service, as shown in Code Snippet 4.
4. The modal and page-specific navigation bar area. This area is populated by view templates with elements that must lie outside of the main app-body area.

[18] http://getbootstrap.com/
[19] http://mobileangularui.com/
[20] https://docs.angularjs.org/api/ng/directive/ngInclude
[21] http://mobileangularui.com/docs/#capture
[22] https://docs.angularjs.org/api/ngRoute/directive/ngView#!

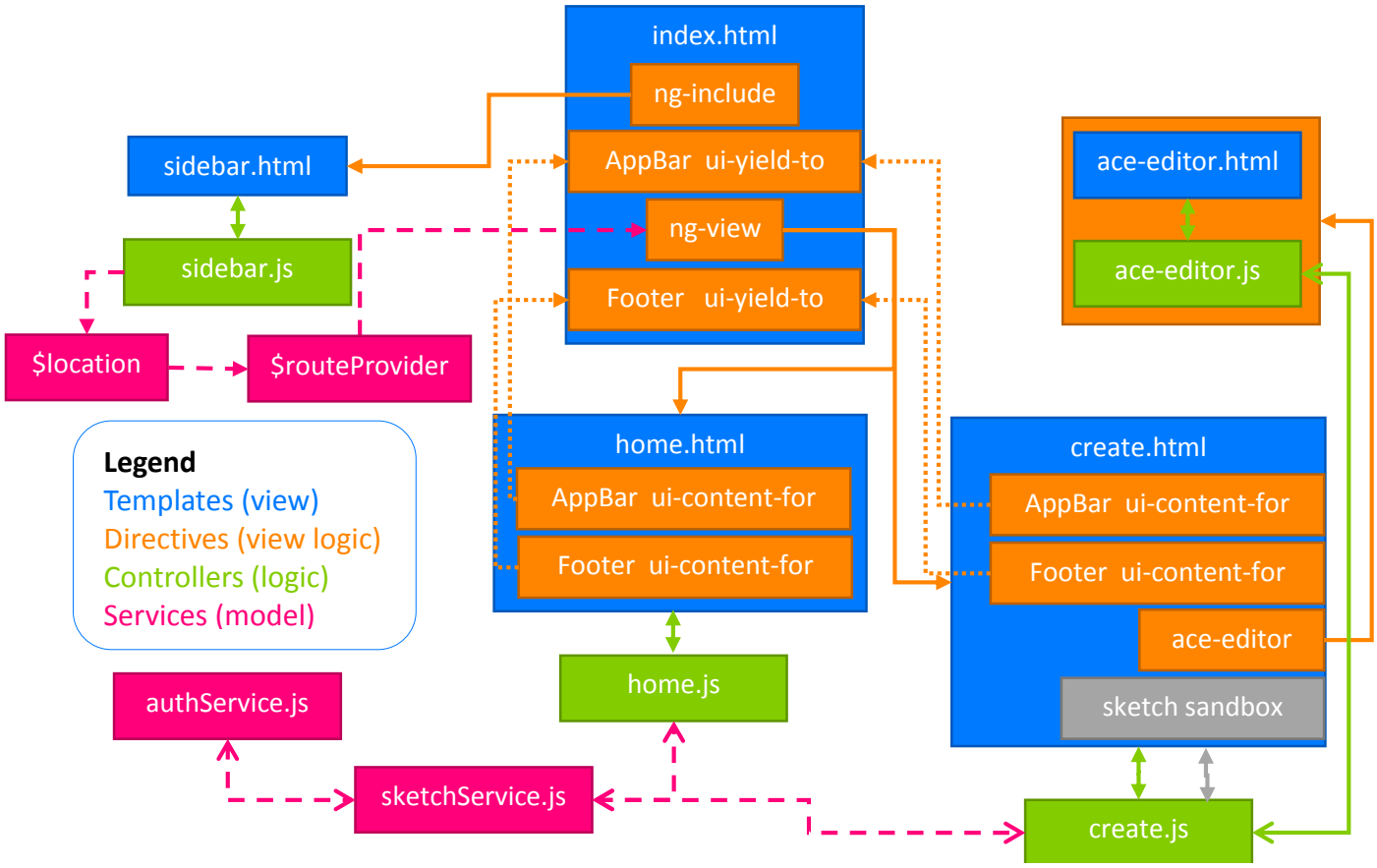


Figure 4: SketchWorld Client Structure

4.2.3 Sidebar & Home View

The left panel, opened and closed through the hamburger menu, functions as the central navigational concept in my application. All the main pages of my application are accessible as links in this area. The sidebar is implemented as simply a list of link elements styled by Mobile Angular UI, which also handles the opening and closing transition. Signing in is a globally applicable action and is therefore available in the left sidebar. The functionality is described in 4.2.6.

The home view is a straightforward template + controller combination responsible for giving the user an overview of his/her sketches. Most of the work in this view is performed in the template, which formats sketch information in a comprehensible, visually pleasing way (see Code Snippet 9). The home.js controller simply exposes sketch objects retrieved from `sketchStorage` (see 4.2.5) to the template `$scope`.

```
<li ng-repeat="sketch in sketchList | filter:isnotLastSketchTester()" class="list-group-item">
    <h3>
        <a ng-href="#/create/{{sketch.sw_id}}">{{sketch.title}}</a>
        <small>rev. {{sketch.getCurrentRev().revNum}}</small>
        <a class="btn btn-danger pull-right" ng-click="deleteSketch(sketch)">Delete</a>
    </h3>
    <p ng-if="sketch.desc !== undefined && sketch.desc.length > 0">{{sketch.desc}}</p>
</li>
```

Code Snippet 9: Displaying user sketches in the home view

4.2.4 Create (Editing & Running Sketches)

The create view has two distinct sub-modes; one for editing a sketch and the other for running it. Navigation between these sub-modes is accomplished with a bottom tab bar, which is styled by Mobile Angular UI and inserted into the footer `ui-yield-to` directive in index.html.

When edit mode is active, a custom `<ace-editor>` directive fills the entire visible area. This directive simply wraps a call to the Ace library to instantiate a feature-rich text editor within a div element (see Code Snippet 6). The Ace editor does not include built-in support for touch scrolling, rendering it unusable on mobile devices. Although I don't expect many students will find coding on their smartphones terribly productive, many might use tablets or other devices (especially with a Bluetooth keyboard) do accomplish their work. To enable this use case, I implemented a basic touch scrolling solution that listens for touch events and calls the editor's scroll method. This implementation lacks the inertia and edge-bouncing that mobile devices use natively, but it functions acceptably (Code Snippet 6 omits this feature).

While in edit mode, the create page uses an auto-save feature. Whenever the user types into the editor, a two second timeout is initiated. If the user makes a further change, this timeout is reset to two seconds. When this timeout completes (if the user pauses typing), the current state of the code is saved to sketch storage and a synchronization with the server is initiated (described in 4.2.5).

```
window.editor.env.document.on('change', function(){
    $timeout.cancel(pendingSync);
    pendingSync = $timeout(function(){
        $scope.saveProgress();
    }, syncDelay);
});
```

Code Snippet 10: SketchWorld auto-save feature (simplified)

The preview mode within the create view runs the user's sketch in the main app content area. Running user-generated code (even if within Processing) in the same context as the main SketchWorld app is bad security practice and (as mentioned in 4.1) is not permitted by some of the target platforms' security policies. Thus, when the run mode is activated, a sandboxed iframe element fills the app content area and communicates with the main app via POST requests. This iframe loads a nearly empty page containing only a reference to the processing.js library and code to listen to POST requests for sketch injections. Communication via POST requests will be expanded in future iterations of SketchWorld to report sketch compilation failures and possibly to offer additional APIs to sketches.

```
$scope.injectSketch = function () {
    var pFrame = document.getElementById('preview-iframe');
    pFrame.contentWindow.postMessage({
        'code': editor.getValue(),
        'lang': 'processing'
    }, '*');
};
```

Code Snippet 11: Injecting a user's Processing.js code into the sandboxed preview environment (within create.js)

```
window.addEventListener('message', function (e) {
    if (e.data && e.data['code']) {
        replaceCode(e.data['code']);
    }
});
```

Code Snippet 12: Receiving the user's code from within the sandboxed preview environment (simplified, from iframe-responder.js)

4.2.5 Sketch Storage Architecture

Managing user sketches is at the core of SketchWorld, and the `sketchStorageService` is responsible for this. The service defines the structure and properties of `Sketch` objects (see Code Snippet 13) and provides methods for manipulating them, including saving, deleting, updating, and synchronizing. This service stores local (browser-based) copies of user's sketches that serve as caches when the user is signed in with synchronization enabled and as the sole repository of the user's sketches in offline (not signed in) mode. In most target platforms, this local storage is achieved using the HTML `localStorage`[23] specification – however this API is not available to Chrome Apps due to platform limitations. When targeting a Chrome App, SketchWorld instead uses Google's proprietary `chrome.storage` API.[24] Considering the substantial differences between the `chrome.storage` and `localStorage` APIs and the complexities of performing synchronization operations with an API server, `sketchStorageService` is easily the most complex component of SketchWorld.

```
// Sketch Data
this.sw_id = 'sw-' + Date.now();
this.title = 'Untitled ' + sketchNumber;
this.desc = '';
this.last_modified = Date.now();
this.last_synced = 0;
this.revs = [{
    revNum : 1,
    code   : '\nvoid setup(){\n\n    //K
}];
this.dirtySinceSync = {
    title: true,
    desc : true,
    code : true,
    revNumber : true,
};
```

Code Snippet 13: Properties of the sketch object defined in sketchStorageService (simplified)

4.2.5.1 Local (Browser Based) Sketch Storage

SketchWorld uses either `localStorage` or `chrome.storage` to store sketches on the user's browser. Although both are essentially key-value dictionaries of high capacity (5MB for `localStorage`)[25] that are maintained by the browser on a per-domain basis, these APIs differ in two substantial ways. The first is synchronicity: retrieval of values from `localStorage` is a blocking operation, which permits a simple programming model suitable for applications that store small values (such as the small sketch objects stored by

[23] http://www.w3schools.com/HTML/html5_webstorage.asp
[24] https://developer.chrome.com/extensions/storage
[25] http://dev.w3.org/html5/webstorage/#disk-space

SketchWorld); retrieval of values from `chrome.storage` is asynchronous and are reported using callbacks. The second difference is in the types of the values stored: `localStorage` only stores strings, while `chrome.storage` stores objects.

`SketchStorageService` abstracts away these differences using two steps. First, the full contents of the key-value dictionary are loaded into memory at the instantiation of `sketchStorageService`. After this initial load, all retrieval and setting operations are performed through wrapper methods which accept JavaScript objects and automatically `JSON.stringify` and `JSON.parse` them when using `localStorage`. Thus, to the rest of the code within `sketchStorageService`, the local storage of sketches is synchronous and operated on objects.

```
function storeKeyValue(key, value) {
    if (isChromeApp) {
        if (value === undefined || value === null)
            chrome.storage.local.remove(key);
        else {
            var storeObj = {};
            storeObj[key] = value;
            chrome.storage.local.set(storeObj);
        }
    } else {
        if (value === undefined || value === null) {
            localStorage.removeItem(key);
        } else {
            localStorage[key] = JSON.stringify(value);
        }
    }
}
```

Code Snippet 14: Abstraction of differences between `localStorage` and `chrome.storage.local`

4.2.5.2 Sketch Synchronization Routines

When the user is signed into SketchWorld, the local cache has stored credentials that allow the web app to authenticate API calls with the SketchWorld backend server (as described in 4.2.6). The backend server returns JSON responses to most requests, as described in 4.3.4. `SketchStorageService` uses Angular's $http[26] module (which in turn wraps the browsers XMLHttpRequest[27] calls) to perform AJAX calls to this server. These calls are stateless, not relying on any concept of session between the client side Angular app and the API server (all required credentials are transmitted with each request).

To determine if a synchronization with the server is necessary, the web app first sends a `sketchStatus` request to the server, which returns date and time of the most recent version on the server. If this does not match what is stored as the last modification time for the sketch in localStorage, a sync is initiated. This

[26] https://docs.angularjs.org/api/ng/service/$http
[27] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

`sketchStatus` request also serves to notify the web-app if the user has deleted a sketch from another device.

```
$http.get(getUserApiUrl() + "sketchStatus/" + this.sw_id)
    .success(continueSynchronization)
    .error(function(data, status, statusText){

        // Sketch wasn't found. Try initial upload
        if(status === 404 && data !== undefined && data.indexOf("Sketch not found") != -1){
            curSketch.uploadInitialToServer();
        }
        // Sketch was deleted. Propogate deletion here
        else if (status === 404 && data !== undefined && data.indexOf("Sketch was deleted") != -1){
            curSketch.delete(true);
        }
        // Undefined error
        else{
            reportSynchronizationFailure("server sketch status retrieval of " + this.sw_id, data, st
        }
    });
```

Code Snippet 15: The first step of synchronization: a `sketchStatus` request

If the `sketchStatus` call returns a positive (found and not deleted) response from the server, the program simply compares the modification times of different parts of the sketches attributes – code, title, and description – to the server modification times reported in `sketchStatus` and keeps the attributes with the most recent modification time. The logic flow is illustrated in Figure 5.

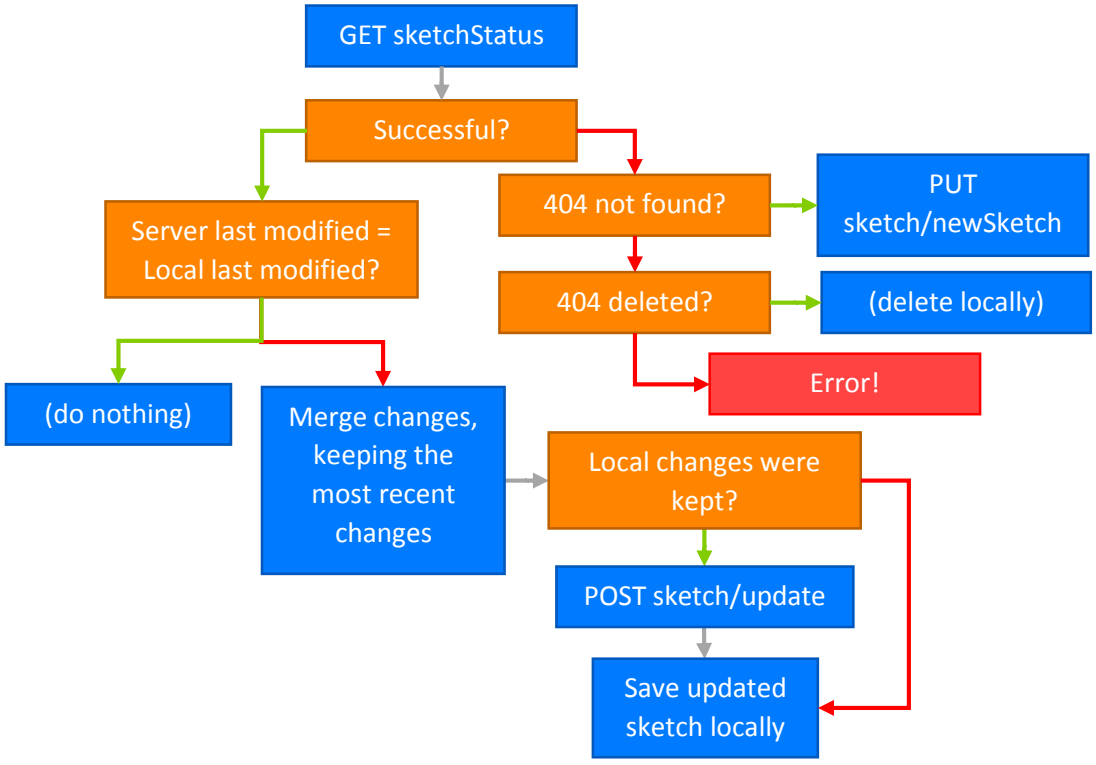


Figure 5: Synchronization Logic Flow

4.2.6 Authentication

Signing in to SketchWorld is a completely optional feature, but enables the user to take advantage of sketch syncing, sharing, and publishing functionality. Rather than create new account credentials, SketchWorld

leverages the OAuth 2.0[28] standard to allow users to login with either a Facebook or Google account. When the sign-in button is clicked, the web-app opens a pop-up at a location on the sketch.world website (even if the user is using the packaged mobile app) that allows the user to choose which service to login with and then redirects the user to the registered SketchWorld OAuth landing page of the appropriate website. After entering credentials for that website and granting permissions to SketchWorld, the user is redirected back to a Sketch.World callback page, which uses the authentication information returned from OAuth to generate a userID and token for use with the Sketch.World server API. The app then reads this information from the popup window, saves it into localStorage, and finalizes sign in of the user, kicking off a round of synchronization. Every API request sent now includes the userID and token as authentication parameters.

4.3 Backend (Server) Design

4.3.1 Backing Frameworks: Node.js, Express,

Although a single page web app can deliver remarkably full-featured functionality by itself (as indeed, SketchWorld is designed to do when working offline), a backend is necessary to power the account-based synchronization and sharing features. A number of technology stacks could have been used to power this backend, but I ended up choosing a Node.js[29] driven webserver running the Express[30] server middleware. Node.js allows programmers to develop applications in JavaScript with full access to the underlying operating system (basically, it allows developers to use JavaScript outside of the context of a web page). Node.js runs on Google's V8 engine and is very fast (3x-6x faster than Apache with PHP).[31] Using Node.js has the additional benefits of consolidating the languages used in SketchWorld's technology stack – leaving JavaScript as the only language in the project.

Express is a minimal web-framework built on top of Node.js. It offers a lightweight layer of features considered fundamental for web apps and is highly extensible. In SketchWorld, Express provides backend URL routing analogous to the client side ng-route module described in 4.2.1.2. Using Express to define API endpoints is as simple as registering a route using the router object, plugging in any middleware that is desired (for authentication or request pre-processing purposes), and providing a function to handle the request. Code Snippet 16 illustrates the process from start to finish:

Line 244: Define an endpoint that handles DELETE requests at the given URL (using Express)
Line 245: Plug in a custom built middleware (a request-processing function than runs before the main handler function) to ensure that the HTTP authentication credentials sent with the request (see 4.2.6) match those of an authorized user in the database (described in 4.3.2).
Line 246: Start definition of the handler function

---

[28] http://oauth.net/2/
[29] https://nodejs.org/
[30] http://expressjs.com/
[31] http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php

Line 248: Find the user from the database whose `oauthID` matches the `oauth_id` in the URL
Line 253: If user is not found, report a 404 not found error
Line 257: Acquiesce the request. Send a deletion successful response
Line 258: Mark the sketch as deleted within the Mongoose-templated user object
Line 259: Commit the changes to the database (Mongoose performs it's magic here).

```
244  router.delete('/user/:oauth_id/sketch/:sw_id/',
245      ensureAuthorizedUser,
246      function (req, res) {
247          console.log("DELETE user/sketch " + req.params.oauth_id + "/" + req.params.sw_id);
248          User.findOneAndUpdate(
249              { oauthID: req.params.oauth_id },
250              { $pull: { sketches : { sw_id: req.params.sw_id}} },
251              function (err, user) {
252                  if (err)
253                      res.status(500).send(err);
254                  else if (!user)
255                      res.status(404).send();
256                  else{
257                      res.status(204).send();
258                      user.deletedSketchIds.push(req.params.sw_id);
259                      user.save();
260                  }
261              }
262          );
263      }
264  );
```

Code Snippet 16: Defining an API endpoint using Express and Mongoose

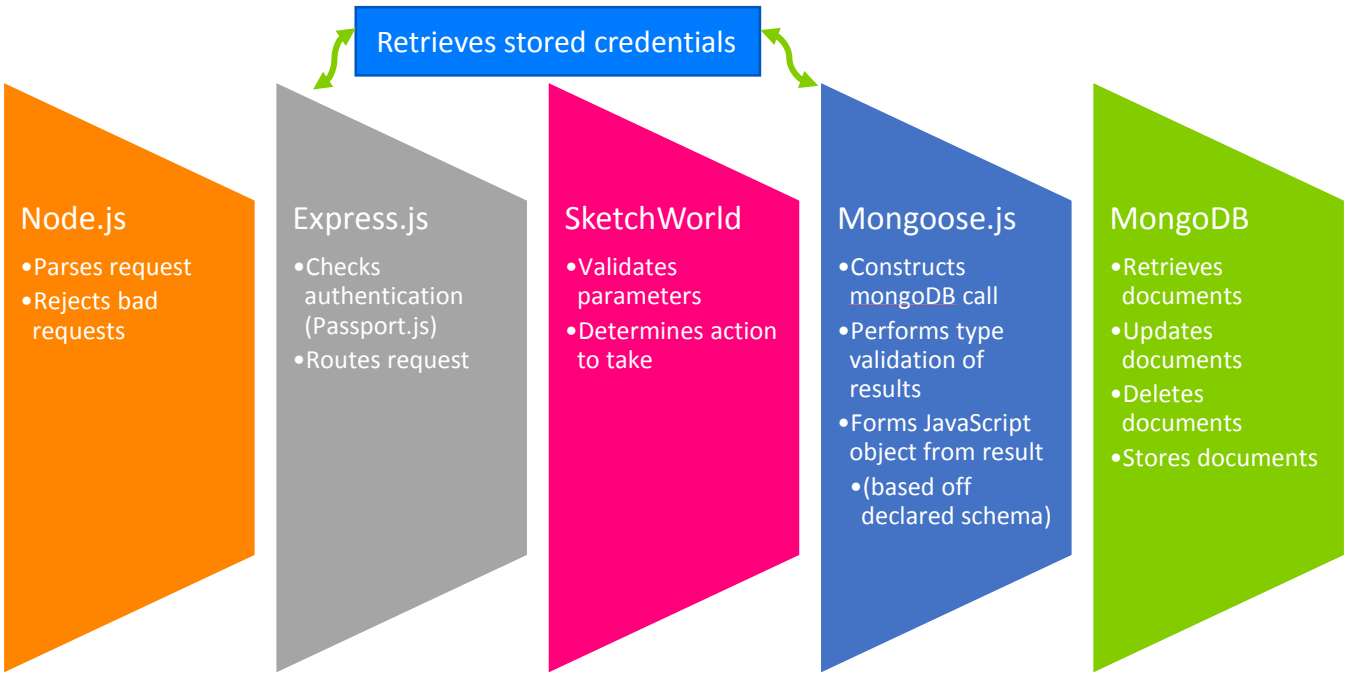


Figure 6: Layers on the Backend: Program Flow of an API Request

4.3.2 Database Design: MongoDB, Mongoose

SketchWorld uses MongoDB[32] as the database backend. MongoDB is a document (not SQL) based database that allows for flexible schema design and efficient document retrieval. This is ideal for SketchWorld, as we expect mostly document-based IO needs with minimal relational algebraic manipulation. We just need to store, update and retrieve documents.

Mongoose[33] is a wrapper for Mongo that allows developers to describe a desired object structure and then easily create, update, and retrieve these objects from Mongo. Because JavaScript is a dynamically typed language and Mongo has no fixed document schemas, there is a potential for mismatching objects to be stored or retrieved from the database. Mongoose takes care of this validation, and allows us to interact and modify the object in a native JavaScript way (such as settings properties of the object using the = operator), then commit the changes with a single `foo.save()` call). The only prerequisite to taking advantage of this convenience is the definition of a schema for each document type. This is a straightforward process, illustrated in Code Snippet 17.

```
var sketchSchema = new Schema({
    sw_id: String,
    title: String,
    last_modified : Number,
    revs: [{
      revNum: Number,
      code: String
    }],
    desc: String,
    thumbUrl: String,
    forkedFrom : {
      user: {type: Schema.Types.ObjectId, ref:'User'},
      sketchId: String,
      revNum: Number,
    }
});
```

Code Snippet 17: Defining a schema for interacting with MongoDB using Mongoose

4.3.3 OAuth authorization flow: Passport.js

As described in 4.2.6, SketchWorld uses the OAuth protocol to allow users to use preexisting Google and Facebook accounts to sign in. Enabling this is highly desirable from a user perspective, as it reduces the number of accounts to keep track of, as well as from a security perspective, as it eliminates the need to store user passwords. Yet, managing the authorization codes, API tokens, refresh tokens, secret keys, etc. is a complicated and error prone process. Passport.js[34] is a middleware layer for Express that automates the process of generating OAuth URLs and

[32] https://www.mongodb.org/
[33] http://mongoosejs.com/
[34] http://passportjs.org/

handling authorization tokens. To use Passport, we must define procedures for storing and retrieving user information in the database and configure the library on startup with client IDs, client secret keys, and callback URLs as registered on Facebook's and Google's developers' consoles.

4.3.4 API Endpoints

Using the process illustrated in Code Snippet 16, SketchWorld defines the following public API endpoints:

Method	URL	Requires Auth?	Description	Returns
GET	/user/:user_id	No	Get information about one user	JSON: User's name, date of registration, and sw_id and title of user's sketches
GET	/user/:user_id/sketch/:sketch_id	No	Retrieve a specific sketch	JSON: All stored properties of a sketch (see Code Snippet 17)
GET	/user/:user_id/sketchStatus/:sketch_id	No	Retrieve synchronization status of a sketch	JSON: last modified date of a sketch as well as latest title and description.
GET	/user/:user_id/sketch /revnum/:num	No	Retrieve the code of a specific revision number of a specific sketch (useful for	JSON: code string belonging to that revision
POST	/user/:user_id/sketch/:sketch_id /update	Yes	Update a previously created sketch	String indicating what was updated
DELETE	/user/:user_id/sketch/:sketch_id /	Yes	Delete a sketch	204 (on success)
PUT	/user/:user_id/newSketch	Yes	Create a new sketch	201 and JSON containing sketch

5 TESTING & ITERATION

5.1 CLIENT SIDE TESTING

Testing on the SketchWorld client-side app has been performed manually. Besides manually verifying intended functionality during development work, I have published an alpha release to http://sketch.world and posted an announcement on Processing.org's forum[35] in an attempt to gather feedback for this project. Although no improvement suggestions have been sent to the email address listed on the sketch.world site, Google Analytics has reported substantially increased activity since the announcement was posted on the forum (see Figure 7). This activity is focused on the United States, but originates worldwide (see Figure 8). It's unknown what proportion of this activity represent real users versus one-time visitors versus bots. Regardless, I *have* received feedback from individuals in CoderDojo, an after-school programming club for middle and high schools, as well as from fellow Dartmouth students and my family, and have a backlog of improvement ideas to capitalize on. Despite this activity, there is clearly further work to be done in the area of client side testing. Angular.js is designed to facilitate unit tests using mock services and the like, so this is a route I plan to pursue soon.

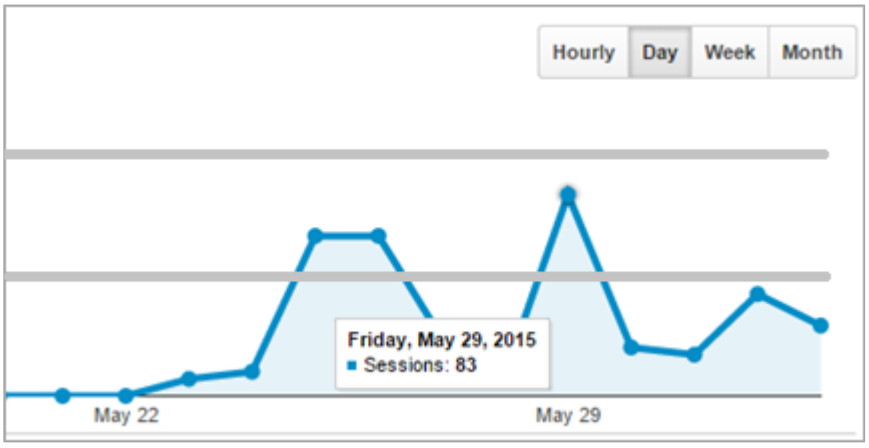


Figure 7: Sketch.World activity since 5/22/2015. Each line represents 50 sessions

[35] http://forum.processing.org/two/discussion/comment/43155

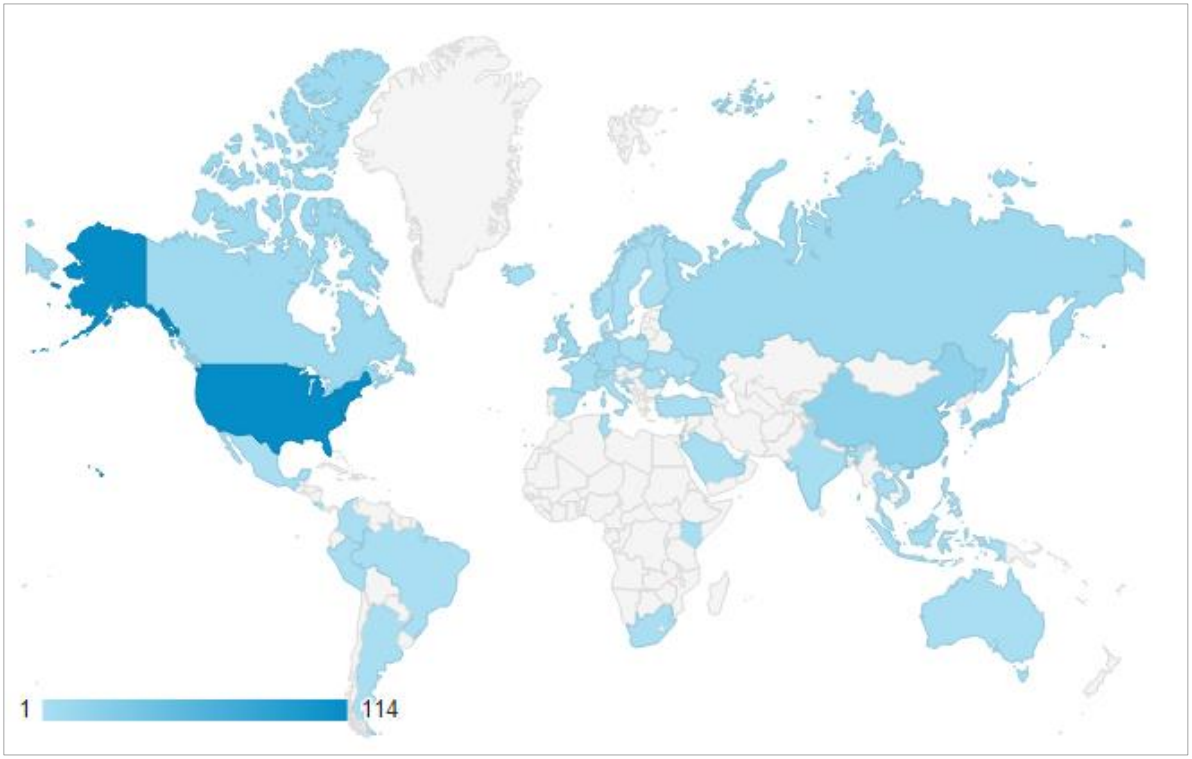


Figure 8: SketchWorld audience locations

5.2 SERVER SIDE TESTING

In contrast to the client side, the backend of SketchWorld has fairly comprehensive unit tests written using software called PostMan.[36] This app enables developers to design custom HTTP requests (to satisfy an API call) including authentication credentials. With each API request, developers can write tests to verify that the server returned the expected response. These requests and associated tests can then be grouped into collection suites to be run at any point, typically after changes have been made to the backend configuration.

Using this tool, I've created a suite of tests for all the SketchWorld API endpoints that test not only expected behavior under valid request conditions, but also verify proper error codes for invalid requests. These tools are useful for preventing regressions in backend code, an essential assurance for a web-app destined for wide usage. Of course, the backend also benefits from manual testing of the front end, ensuring the client and server interface correctly.

[36] https://www.getpostman.com/

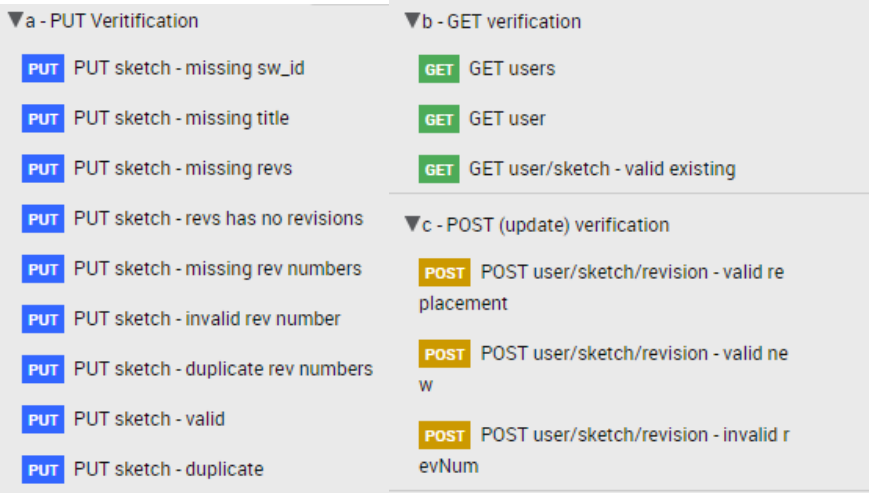


Figure 9: Collections of API Unit Tests in PostMan

Test editor Help

```
1  tests["Status code is 200"] = responseCode.code === 201;
2
3  var sketch = JSON.parse(responseBody);
4  tests["Sketch has id"] = !(!sketch.sw_id);
5  tests["Sketch has revs"] = sketch.revs.length > 0;
6  tests["Sketch has proper revNum"] = sketch.revNum === sketch.revs.length;
```

Code Snippet 18: Example of PostMan API tests

6 CHALLENGES & LESSONS LEARNED

Developing SketchWorld has been a challenge. Not only is the world of web app development incredibly fast moving, it's also an area in which I had little prior experience. Besides Bootstrap and raw HTML and CSS, every technology and library used in SketchWorld was new to me (including, for the most part, JavaScript). I found myself spending weeks making very little visible progress on the app while doing tutorials on the myriad of different technologies that are meshed together in SketchWorld. Yet I am also convinced that the technology stack I chose was essentially the right one for this application. A single page web app simply provides the greatest flexibility in terms of target platform and the best user experience in terms of app responsiveness and cohesion.

Mentioned in 4.1, it has been a difficult technical challenge to ensure that SketchWorld performs as expected on the variety of different platforms to which it is targeted. My developmental workflow consisted primarily of making changes on a text editor and immediately reloading the page in Google Chrome to see the effect. This created trouble when attempting to package SketchWorld as mobile apps. Despite working in the web browsers of iOS and Android, the app fails to load in Cordova due to platform restrictions (probably security restrictions). It would have been a better idea to configure the app for deployment to all target systems at the

beginning and test on all target systems throughout the iteration process to catch bugs that only appear on certain platforms early, as they are created, rather than after the fact.

7 CONCLUSION & NEXT STEPS

In its current state, SketchWorld is a useful app for those seeking to do Processing.js work on the web. The development experience of sketches on SketchWorld matches or exceeds those on sketchpad.cc or openprocessing.org. The responsive mobile layout enables a completely new application compared to existing platforms, and although coding on iOS isn't possible due to a bug where the entire editor is always selected, running sketches on mobile does work and is far easier on SketchWorld than Processing IDE's Android mode (which uses the Java backend and creates a full Android APK app package).

Nonetheless, development work on SketchWorld is far from finished! There are many improvements and fixes that would greatly further the utility of SketchWorld. Chief among these, SketchWorld should be available as a native app on mobile platforms, and as an offline app on other platforms. Considering the difficulties mentioned in 6, this will require a careful pruning and reconstruction of SketchWorld's code until the offending code is located and modified.

Beyond that, I have several items on the list to be implemented in the medium term. These include improving the editing experience by offering feedback when there are errors in a sketch and offering multiple code tabs. After that, I'll improve sketch capabilities on mobile, ensuring touch events are passed to sketches as expected. Once these are nailed down, I'll add sharing functionality such as standalone links to sketch pages and allowing sketches to be pinned to the home screen of mobile devices that support pinning.

This thesis has given SketchWorld an excellent start in pursuit of its vision of empowering and motivating beginning programmers. But this is just the beginning! With further development, SketchWorld can evolve into the ideal way to learn computer science, giving students a tangible, personal, and sharable reward for their efforts and showing the next wave of beginning programmers that computer science is the coolest, most fun, and most productive outlet for their talents.

7.1 ACKNOWLEDGEMENTS

I'd like to thank

- Devin Balkcom, my thesis advisor, for his support and advice
- Holley Allen, my mother, for proofreading
- The Dartmouth class of '61, for their financial support