# DIY Grip Tape Website

By

Jason Krein

Software Engineering Department

College of Engineering

California Polytechnic State University

June 2017

# Abstract

The goal of this senior project was to create an easy to use website that will allow people to design and order their own custom skateboard grip tape. Custom skateboard decks are a large business, however the top, more visible, part of the board is largely ignored by custom shops. With the help of a local entrepreneur, I aim to enable skateboarders the ability to get custom grip tape shapes that they can stick on their board to spice it up from the normal black tape that everyone uses. The website was completed over the course of 4 months, in about 50 hours of work.

# Contents

Abstract ..... 2

1. Intro ..... 5

1.1 What I aimed to accomplish ..... 5

2. Website Overview ..... 6

2.1 User flow ..... 6

2.1.1 Initial Visit ..... 6

2.1.2 Sign In ..... 8

2.1.3 Create Grip Tape Design ..... 9

2.1.4 Saving a Design ..... 10

2.1.5 Loading a Saved Design ..... 11

2.1.6 Uploading an SVG ..... 11

2.1.7 Creating an Order ..... 12

2.1.8 Viewing Order Status ..... 13

2.2 Administration flow ..... 13

2.2.1 View Orders ..... 14

2.2.2 Alter Status ..... 14

2.2.3 Security ..... 15

3. Website Architecture Choices and Comparisons ..... 15

3.1 Database backend ..... 15

3.2 Drawing Libraries ..... 16

3.3 Stripe ..... 17

3.4 SVG image format ..... 17

3.5 PHP serverside ..... 18

3.6 Login options ..... 18
4. Implementation ..... 19
4.1 Database ..... 19
4.2 Login ..... 20
4.3 LiterallyCanvas ..... 22
4.4 Stripe + PHP ..... 23
4.5 Saving Drawings ..... 25
4.6 Summary ..... 26
5. Lessons Learned and What to do Differently ..... 26
5.1 Clarify Expectations ..... 26
5.2 What Users Expect ..... 27
6. Next steps ..... 27
7. References ..... 28
1. LiterallyCanvas ..... 28
2. Firebase ..... 28
3. Stripe ..... 28
4. Custom Grip Tap ..... 28

# 1. Intro

There are several websites that currently exist that will let users buy custom skateboard grip tape. For example, zazzle.com and boardpusher.com are some of the first results on Google. However, none of them let you draw directly in the website. They will let you upload an image from your computer, which is nice for people with some graphic design experience, but having an easy drawing interface on the website is an untapped market. My goal is to make a website that will add that option, in addition the allowing users to upload their own photos or choose from premade designs. I had hoped to enable users to share designs as well, but unfortunately that turned out to be a much bigger challenge than anticipated, and I was unable to complete it.

While I am not a skateboarder, my business partner, Matthew Jensen, is. He saw potential for a product like this to potentially take off and be successful, and he had the mechanical expertise to make it happen, so I jumped on board to manage the online portion of the project.

## 1.1 What I aimed to accomplish

There were two main components involved in the project – the website and the physical manufacturing of the grip tape. For my project, I had to account for both the website, obviously, and the manufacturing process. While the drawing interface was an important part of the project, I felt that the administration tools would turn out to be equally as important. The aftermath of an order being placed – confirmation and tracking on the customer side, and easy modification and access on the administrator's side, was something I had to take into consideration.

While there were many technical tasks I aimed to face, some were out of scope of the project. I did focus on creating a "beautiful" website, only on creating something that worked

well, and did the job. This was due to my limited timeframe, as well as Matthew's talents being more in line with making the website look good. I was also not responsible for the technology behind the actual cutting of the grip tape – that would all be done by Matthew, about seven or eight months after the end of my senior project. The delay is because he needs to purchase a special laser cutter to start production. With the scope out of the way, I'd like to discuss the flow of the website from both the user and administrator's point of view.

# 2. Website Overview

This section will be about the experience of using the website. Many of the UI elements are not final, and the general look of the website is likely to change after being handed off to Matthew. However, the general flow of actions and responses will stay largely the same, and that is what I will be showing. The goal behind the design is to allow any user, no matter if they're advanced or a beginner, to create whatever design they desire, and be aware of roughly what it will look like when they receive their order. In addition to the user experience, administration was also considered, and will be talked about in this section.

## 2.1 User flow

First, I would like to go over the use of the website from a typical person's point of view, since that will be the most common, and most important, scenario. The typical order of action would be to visit the home page, create a tape shape and color, save the drawing for later or order the drawing, and then view the progress of your order completion.

### 2.1.1 Initial Visit

If a user has never logged in to the website, the first time they visit they are greeted with the screen shown in **Error! Reference source not found.**. The paragraph is a description of a ll the different features of the website. This is likely to change to be a bit more prominent, as well as shorter, and a lot of the information may move to either the "pricing" or "about us" pages. When a user clicks on the "Begin Drawing" button, the text is replaced with a canvas, and the page will look like **Error! Reference source not found.**. The reason I decided to add the i nitial text is that pretty much everyone I showed the site to was understandably confused about what it was for. Adding some text was the easiest solution, however it is not the most

effective. One alternative I considered was having some real-life previews of boards that have the custom grip tape on them, but unfortunately Matthew had no good promotional material that I could work with. One of the downsides of working on a website for a product that hasn't been produced yet is that it's very hard to envision the product, both as the designer and as the customer. Due to the lack of promotional imagery, the screen does feel relatively plain, but there is plenty of space for icons and images to be added in the future.

*Figure 1*

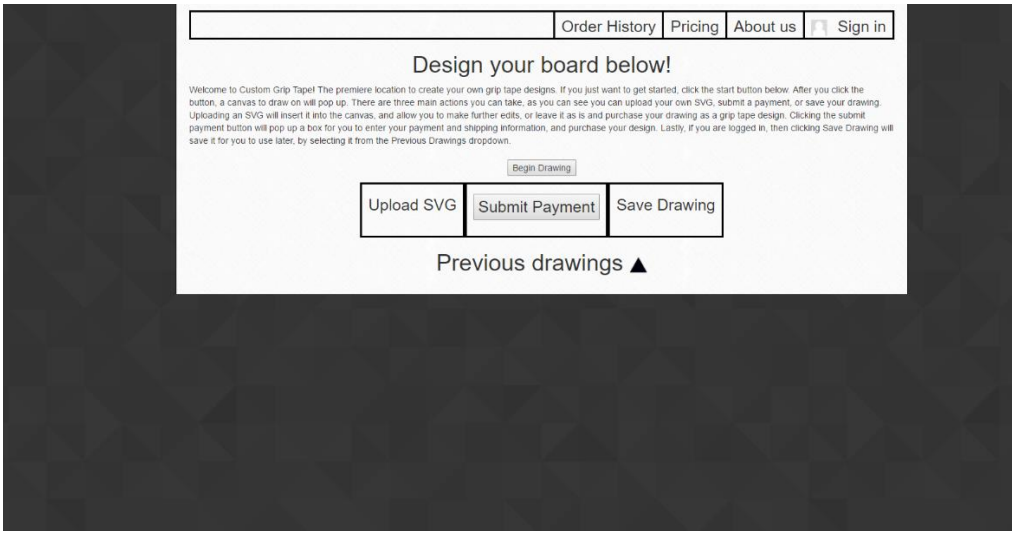
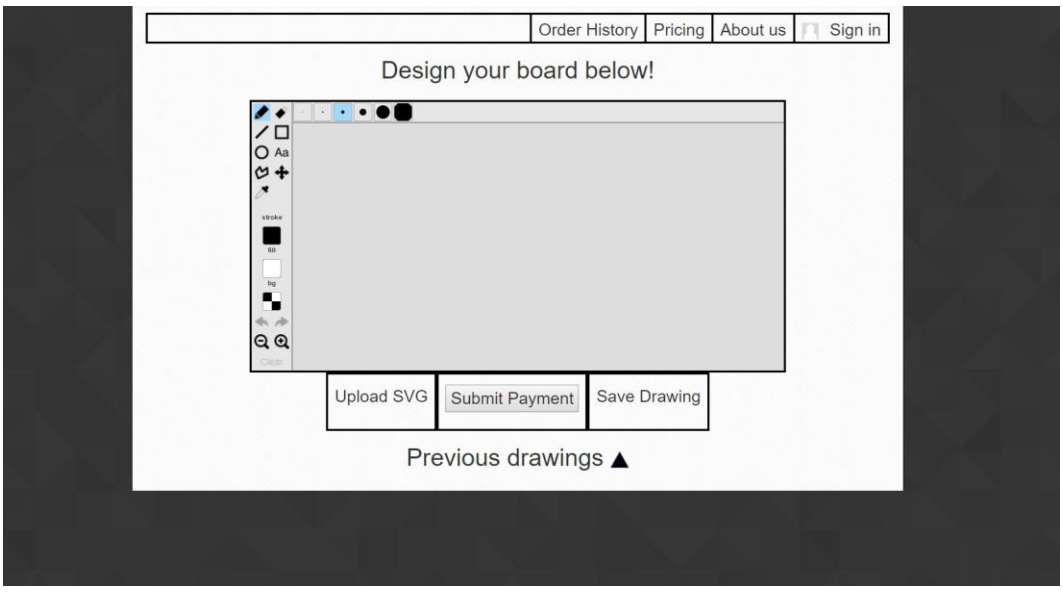



*Figure 2*



### 2.1.2 Sign In

Clicking the “Sign in” button on the top right pops up a dialog box which allows users to sign in with their choice of Facebook, Google +, or to create an account with an email address as seen in **Error! Reference source not found.**. As mentioned, I was more focused on f unctionality than looks, so the custom email button is not very pretty. The Google and Facebook logins are self-explanatory, and clicking on them brings a popup window with the appropriate Google or Facebook OAuth pages. Clicking “Use Custom Email” replaces the login buttons with email and password fields, and the option to log in, create an account with the given email and password, or reset their password.

If the user chooses to log in with Google or Facebook, the website is then able to pull their profile photo to customize the page a little more by displaying the user’s first name and profile picture, shown in **Error! Reference source not found.** in the top right corner. Once a u ser has logged in, they will stay logged in even after leaving the site, unless they explicitly log out.

*Figure 3*

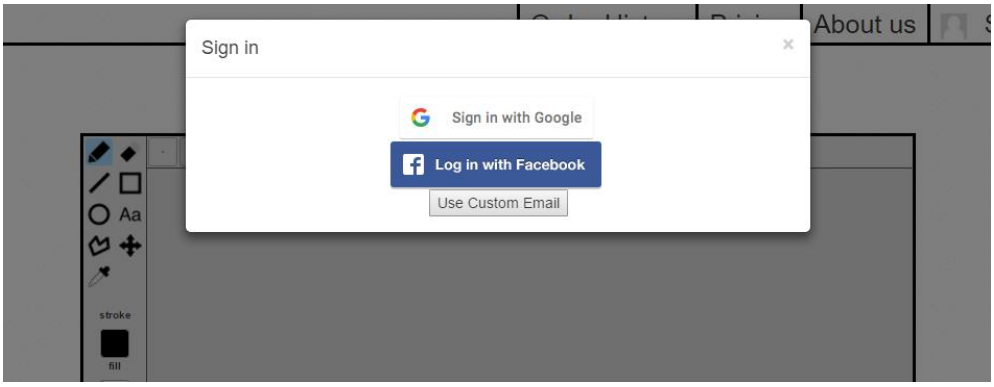
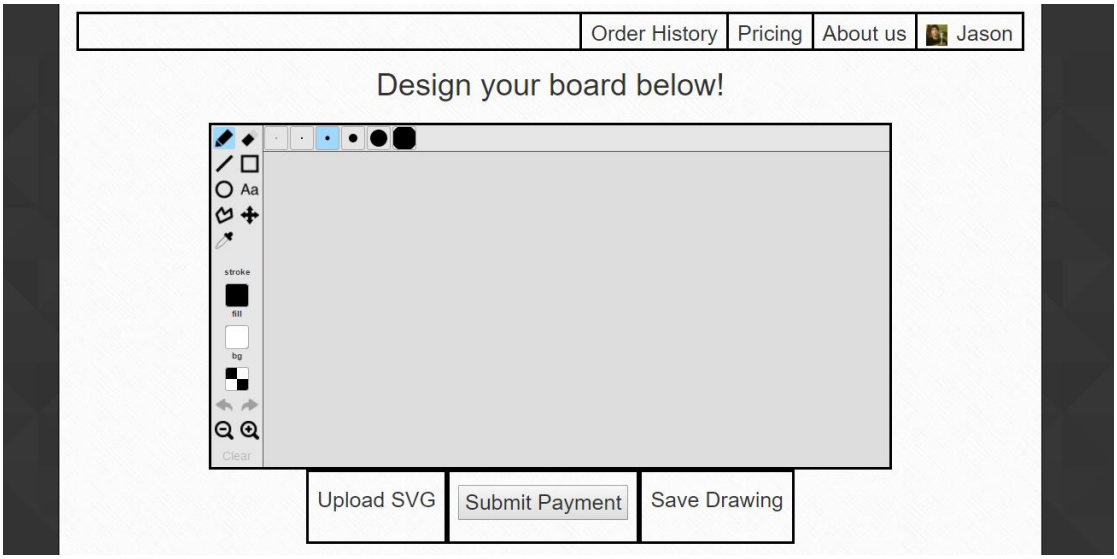


*Figure 4*


*Figure 5*



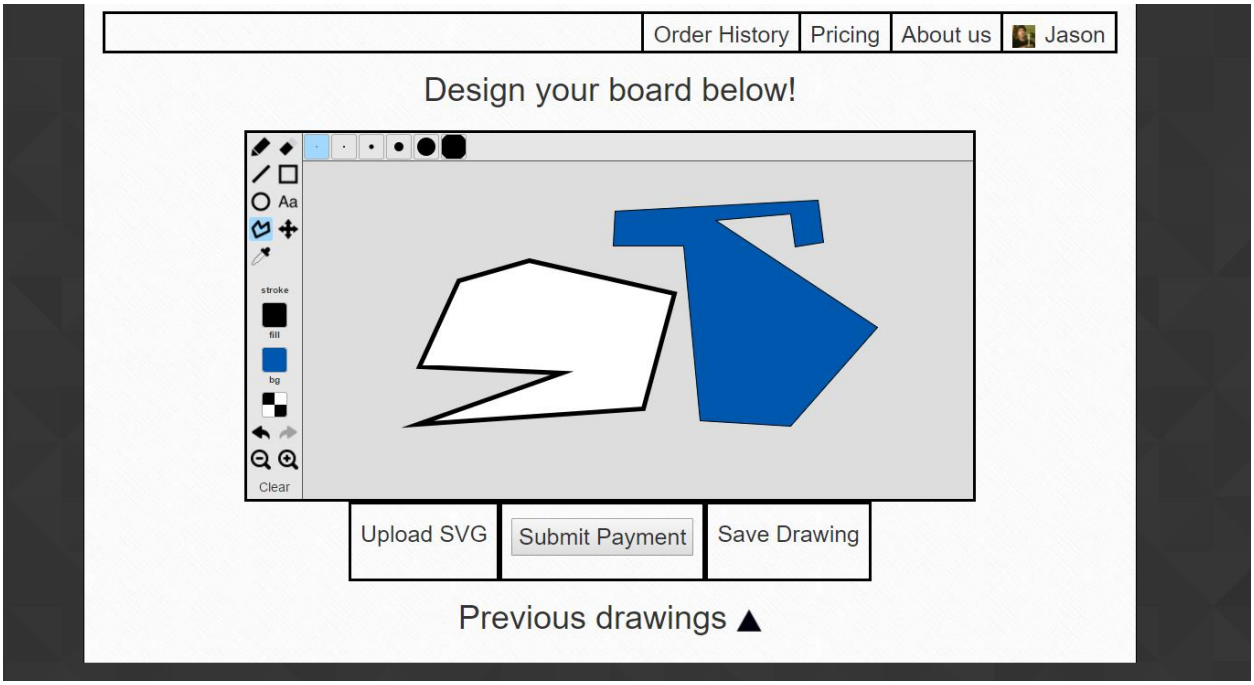

### 2.1.3 Create Grip Tape Design

Creating the design was one of the key requested features from Matthew. I decided to use an open source library, LiterallyCanvas, to facilitate drawing creation. The technologies and reasons behind choosing LiterallyCanvas are discussed later, in section 3. Website Architecture. Because the physical good being delivered is going to be laser cut out of grip tape, the tool only allows users to create closed shapes, such as the one shown in **Error! Reference source not f**

**ound.**. Users can also use multiple colors, allowing them to create a variety of designs, as also demonstrated in Figure 6.

I had planned to add a skateboard outline for users to draw on in the actual canvas, but have not yet received the SVG required to do this. It is an easy add in the future though, and all the template code is there, just waiting to be given the SVG file name. This will allow users to more accurately place their designs in the preview, and get a sense of how big/small their drawings will look relative to the real board.

*Figure 6*

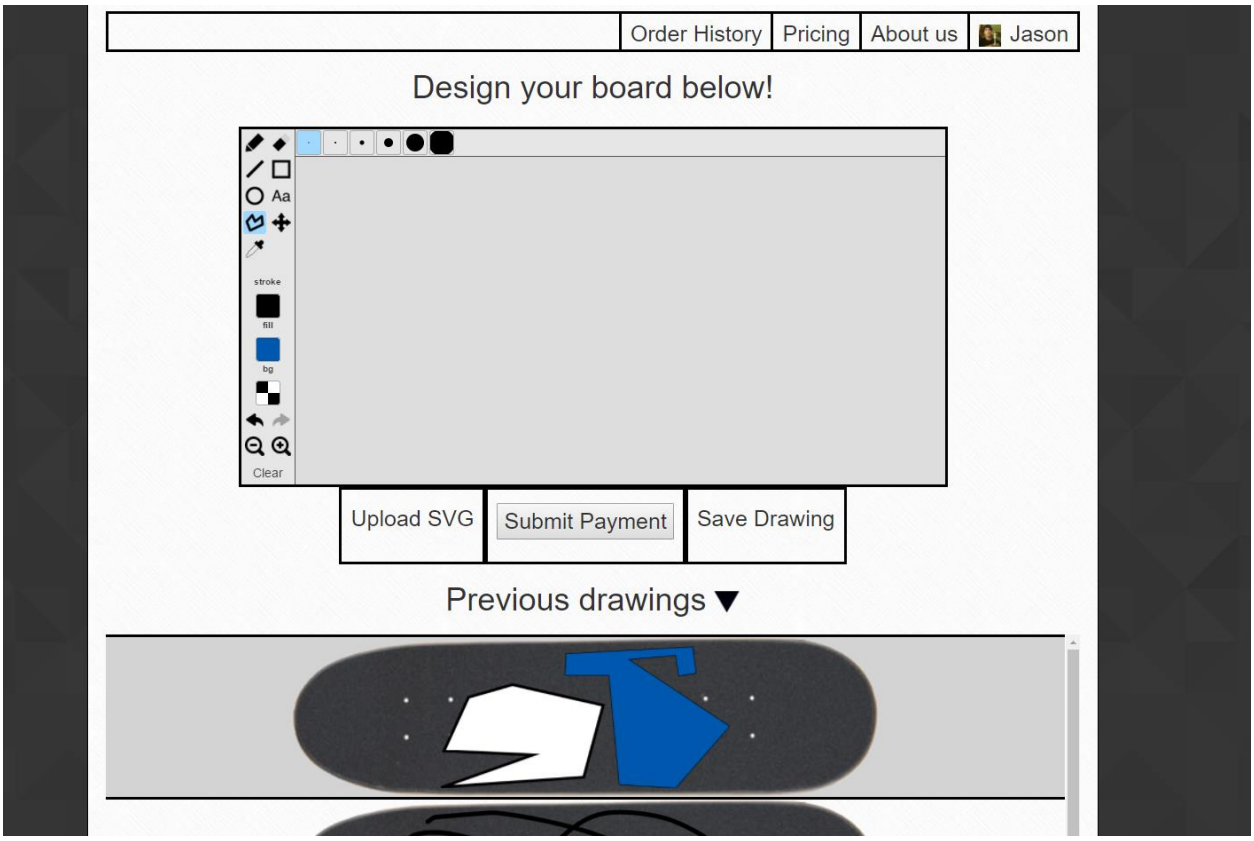



### 2.1.4 Saving a Design

This is a feature that is made very easy by the LiterallyCanvas library. When a user clicks the “Save Drawing” button, the drawing object is written to the database, and stored under the “previous drawings” dropdown, shown in Figure 7. Drawings are ordered chronologically by when they were last saved. They are loaded as soon as the page is opened, rather than when the dropdown is revealed. I chose not to do lazy loading because it can be quite jarring to see all the photos pop into place, and loading them when the dropdown is not revealed leads to a much smoother-feeling user experience.

In the saved design dropdown, notice that the images are overlaid on a picture of a skateboard. This allows users to more easily imagine what the grip tape will end up looking like, and reiterates that the drawings will be going onto the skateboard.

*Figure 7*




### 2.1.5 Loading a Saved Design

Loading a saved design is very easy, thanks to the LiterallyCanvas library. When the drawing is saved by the user, the whole history of the drawing is uploaded. What this means is that when a user loads a saved design by simply clicking on it in the previous drawings dropdown, it is loaded in to the canvas, and they are right where they left off. The background and stroke color choices are the same, and they can use the back and forward buttons as if they had never stopped drawing.

### 2.1.6 Uploading an SVG

Uploading an SVG is one of the features that will likely only be used by more advanced users who have experience with some external software like Inkscape or InDesign. It allows

users to upload an SVG into the canvas from their computer, and then edit it if they'd like, or go straight to ordering.

### 2.1.7 Creating an Order

To receive custom grip tape, users need to create an order by clicking on the "submit payment" button. This requires some drawing to be in the canvas already, if the canvas is blank a popup appears that informs the user they must create or select a drawing before making a purchase. I decided to go with Stripe as the payment processor, and they provide a nice customizable UI to use for ordering, seen in Figure 8.

After the user clicks the "checkout" button, they are brought to an order confirmation screen, shown in **Error! Reference source not found.**. This screen confirms the user's name, a ddress, and grip tape design, and for now just provides an email to send any problems to. Because the site is expected to have relatively low traffic at the beginning, automating address changes seemed unnecessary and open to exploitation.

Order creation is also reflected in the Stripe console, so administrators have easy access to make sure users have paid for their orders.

*Figure 8*

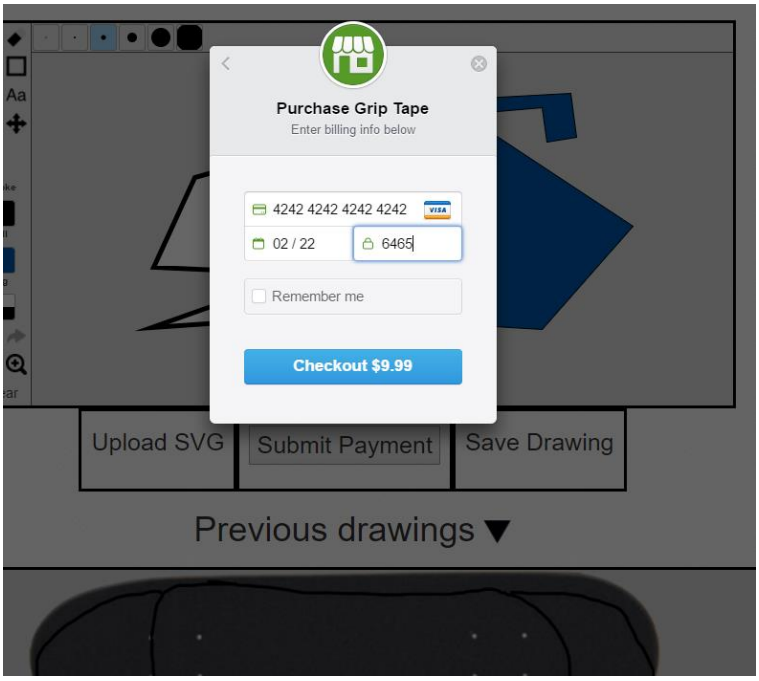
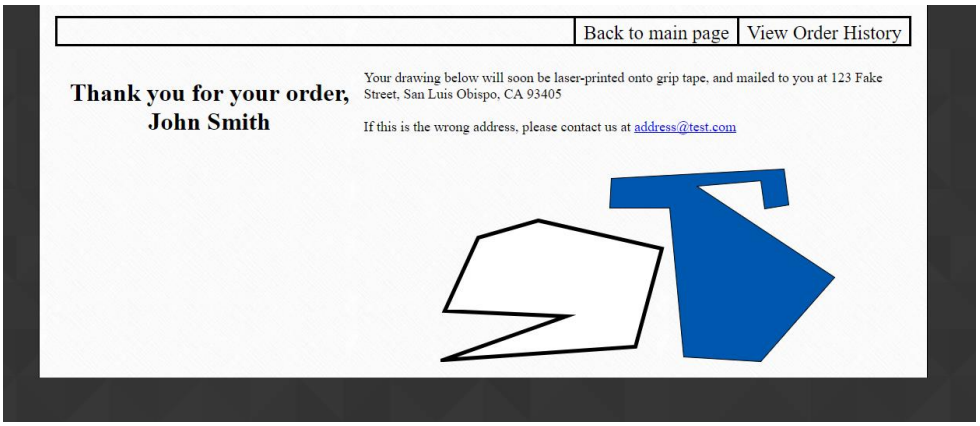


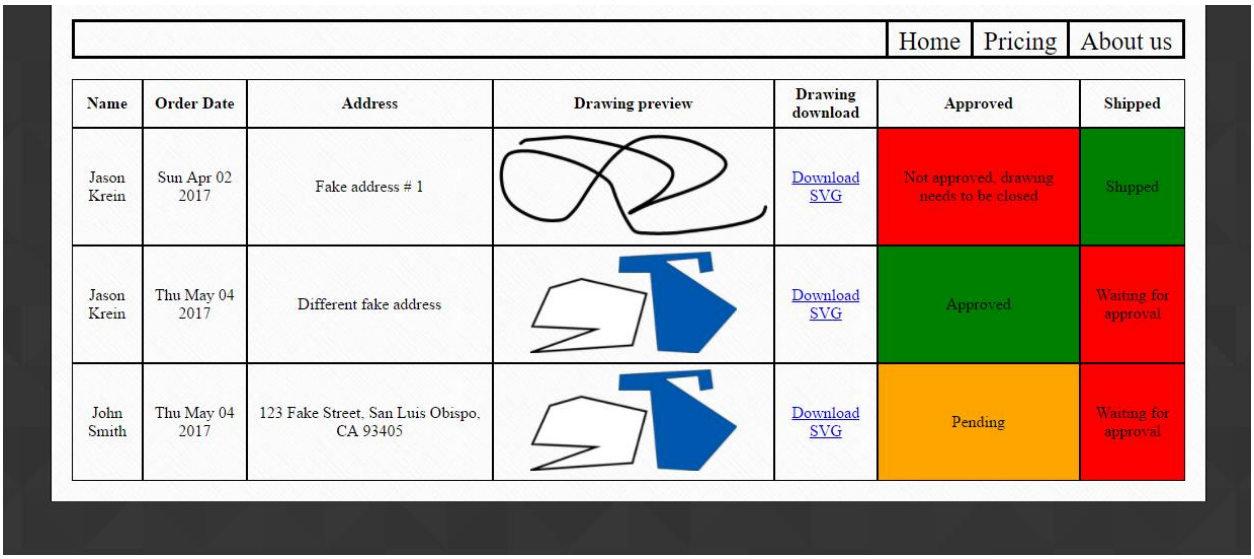
*Figure 9*




### 2.1.8 Viewing Order Status

The last page that users can access is their order history page. This contains the name that will be on the shipping label, the order date, address to ship to, drawing preview, SVG download, and approval/shipping status as seen in Figure 10. This allows users to easily see when they can expect orders to arrive, and if there was anything wrong with their designs. It also allows them to share their designs with their friends if they want to, as they can send the SVG of their drawing and anyone can upload and order the same one.

*Figure 10*



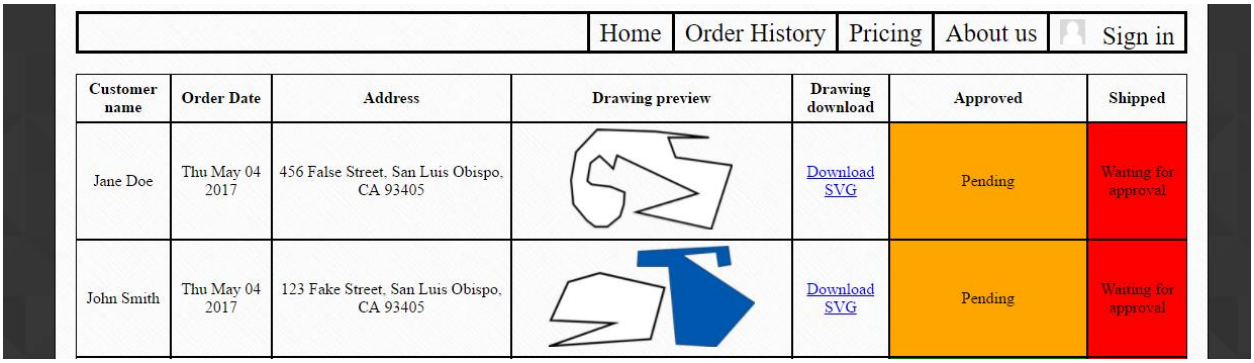

## 2.2 Administration flow

The administration flow is arguably just as important as the user flow. If the laser cutting engineers can't efficiently fill orders, there is potential for wait times to begin to stack up, and users could be waiting months to receive their orders. Because of this, I tried to keep all the information in one spot, and easily viewable and editable. The admin screen is essentially the same as the user's order history screen, with a few small differences.

### 2.2.1 View Orders

The administrator can see all user's orders, as seen in Figure 11. They are displayed chronologically, so that the newest orders are at the top and the admin shouldn't have to scroll far to see unfulfilled orders. The drawing preview is also important, as it allows admins to see designs with just a glance, without having to download the SVG and view it in a separate program.

*Figure 11*




### 2.2.2 Alter Status

Once an administrator decides to approve or deny a drawing, they simply click on the box next to the drawing they want to change. This causes a dialog to pop up, and the admin can enter anything they'd like. If the enter "Approved" or "Pending," the box automatically changes

to green or orange, respectively. If anything besides “Approved” or “Pending” is entered, the box turns red. The updates are automatically propagated in real time, so other administrators will immediately see the change reflected on their admin panel, if they are viewing it from another computer, and users also see the change reflected on their order status page if they happen to be viewing it when their order is approved or denied.

### 2.2.3 Security

Since this is a web application, security is something I had to be extra careful about. Because Firebase is used as the backend, users can potentially send any query they want. I’ll discuss the solutions for this further in section 3. Website Architecture, but in summary, only admins can change order information after it is first created. Admins are added as a separate table in the database, which can only be changed by manually logging in to the Firebase dashboard, and adding unique IDs to the table. This ensures that users can’t mess around with their order status, and administrators will always be able to trust the data.

## 3. Website Architecture Choices and Comparisons

### 3.1 Database backend

One of the main decisions that had to be made was how to store all the drawing and user data. My two choices for database designs were between NoSQL and MySQL/MariaDB. I ultimately went with a NoSQL database hosted with Firebase.

First, I’ll discuss what using MySQL would have involved. MySQL is a relational database, meaning that at the basic level, you will have a bunch of tables that relate to each other, usually through IDs. This would have been hosted by the web host, which in my case was NearlyFreeSpeech. In addition to the SQL data storage, I would need to create REST endpoint and server for my client-side JS to interact with. While this would have been a good learning experience, I ultimately decided I didn’t have the time or prior experience to undertake that by myself. The data I needed to store is also not heavily relational, so using NoSQL was not a huge inconvenience.

NoSQL is the type of database type is that Firebase uses it. Firebase is a Google product, advertised as "Backend as a Service." Essentially, it allows me to have server side storage and a database, without having to create REST endpoints or run my own server. Firebase provides a JavaScript library that allows users to query the database directly in JavaScript. The main drawback of Firebase is that it isn't relational – where clauses and joins aren't a thing in Firebase queries. This means that data duplication is needed to make efficient queries, which is discussed further in the implementation section. Firebase also has a nice "rules" feature, to allow or prevent data from being written to and read from based on anything from authentication token to what the content of the data is.

Ultimately, I decided to use Firebase's NoSQL database, mainly because it cuts out the middleman server, and is easier to validate read and write permissions than making my own server would be. Using firebase allowed me to focus more on getting all the client side code working, while also having the ability to keep information on a server.

## 3.2 Drawing Libraries

I knew from the start that I did not want to create my own custom implementation for web browser drawing, as that would likely have been enough work to be a project all by itself. I spent a lot of time searching for a good library to build off. Some of my options were drawer.js (https://www.drawerjs.com/), zwibbler (http://zwibbler.com/), and LiterallyCanvas (http://literallycanvas.com/). The first two options were paid, and free libraries that allow for user created drawings in the browser are surprisingly sparse. However, despite essentially being forced into using LiterallyCanvas due to lack of other options, it worked out great, and LiterallyCanvas has lots of key features that made it useful to me.

The main features of LiterallyCanvas that I utilized were the easy setup, and the nice saving features. Inserting the canvas into the page only takes a couple of lines, and is covered in the implementation 4.3 LiterallyCanvas section. This made it very easy to get started with the project, as I didn't have to do a bunch of frustrating installation steps. I was then able to quickly use the most helpful features, for me, of LiterallyCanvas, saving. The library enables saving to either SVG or PNG formats, and I use both. SVG is the format Matthew requested, as it allows

easy scaling and conversion to DXF format, which is what the laser cutter uses. However, saving and subsequently displaying SVG files in Firebase storage was not trivial. This is where the PNG saving feature came in handy – PNGs are much easier to work with, both in Firebase and display client side, so they are used for all the preview images that are shown to the user.

LiterallyCanvas also allows me to save and load the state of the canvas, meaning users can load not just the image they made back into the canvas, but also the whole history of the image. While this is a pretty trivial feature, I think it's nifty and adds to the usability of the drawing tool.

LiterallyCanvas also provides an API for adding your own tools. Unfortunately, it's a little sparse and hard to figure out, so I had to take a different approach to creating a custom tool, which will be further discussed in the Implementation section on LiterallyCanvas.

### 3.3 Stripe

The main goal of the website is to get users to order the custom designs they make. To facilitate orders, I had to decide on a payment processor to use. Some choices here were Stripe, Braintree, and Authorize.net. I ended up choosing Stripe because they seemed to be the most geared towards being plugged in to a custom site, instead of being part of a site-builder application like Braintree. Stripe also has a nice provided UI, which made it so that I didn't have to worry about making a pretty popup window and doing all the customer info validation, which didn't seem to be part of the offering of Authorize.net. Lastly, when I visited the Stripe website I had a much easier time figuring out how to get started with integration, while Braintree and Authorize had more hidden documentation pages, which made Stripe seem like the most developer friendly option.

### 3.4 SVG image format

When deciding on the format to save the images, I didn't have much of a choice. The laser cutter that Matthew plans to use needs .DXF files, which aren't possible to create on the web. However, SVG is convertible to DXF, making SVG the next best option. Luckily LiterallyCanvas has built-in functionality to save to SVG, so that wasn't a problem. Something that may be nice to add in the future is to use CloudConvert APIs to convert the SVG to a DXF

before uploading it to the server, instead of needing to convert the files locally before cutting the grip tape. Because that conversion is a paid feature of CloudConvert and wouldn't be needed for a few months after my part of the project was completed, I decided it was out of scope.

### 3.5 PHP serverside

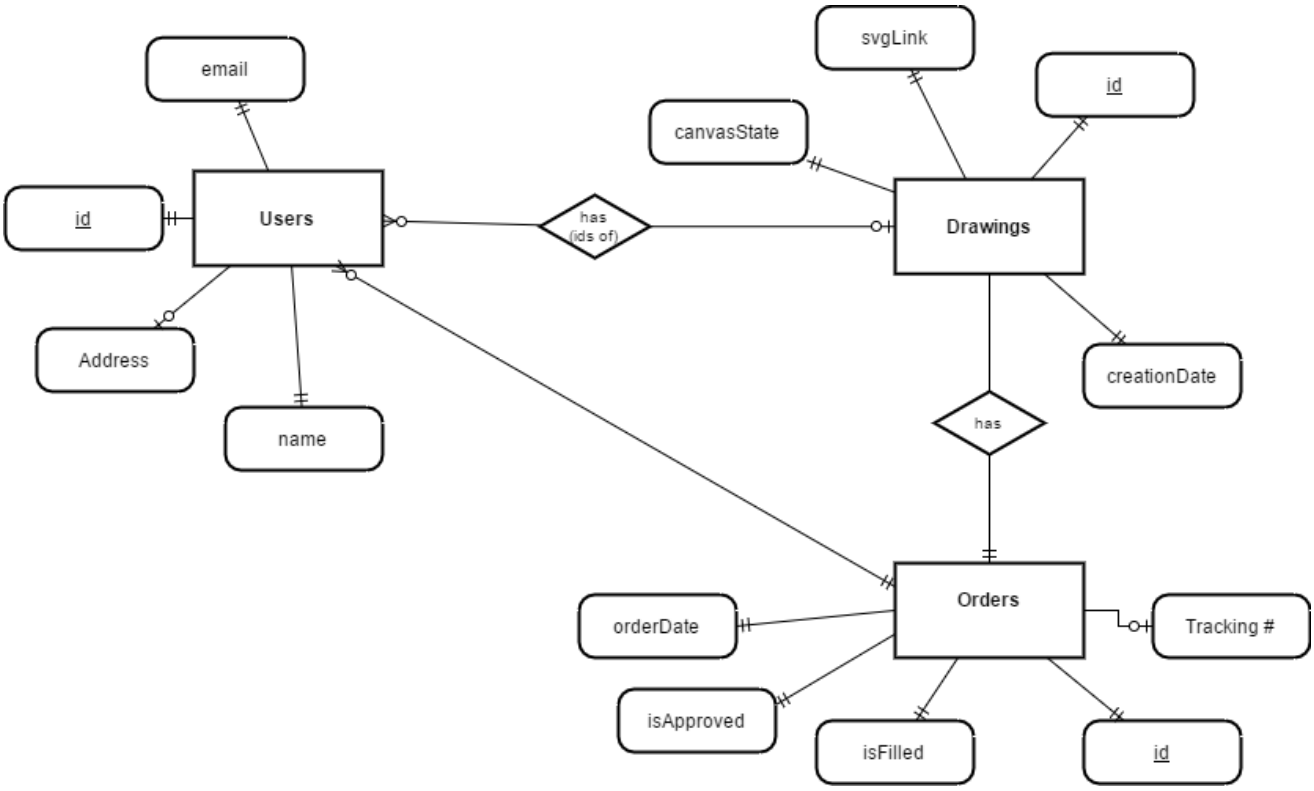
There was a minimal amount of serverside code needed for this project thanks to Firebase. Stripe did require some serverside logic though, due to how payments are processed and security issues with leaving everything on the client. Stripe has the option for many different serverside architectures such as Rails, Flask, Express, ASP.NET, and PHP. Because I am already familiar with PHP and it requires the least setup to get working on the webhost I am using, I decided to use PHP for Stripe server integration. While PHP is generally disliked due to some language quirks and messiness, it isn't a huge part of the website. PHP would only be used to process payments, something that requires very little logic, as shown in the Stripe Implementation section. Due to the minimal usage PHP would get, the pros of small setup time outweighed any cons of the language.

### 3.6 Login options

Firebase makes login very easy, as it keeps track of everything for you. I just had to setup the UI elements and link them to a firebase call, and sessions would be persisted across website visits, and all the OAuth requirements are taken care of. I also decided that adding an option to sign in with a custom email and password would be a good option. Many people I showed the site to indicated they weren't very comfortable logging in with their Facebook or Google accounts, which is understandable. Luckily Firebase makes custom email auth easy, so I added that as another option for people who are unwilling to sign in with an account linked to social media.

# 4. Implementation

## 4.1 Database




The above is the entity relationship diagram for the custom grip tap database. It will consist of 3 tables; Users, Drawings, and Orders. Users can have lists of 0 or more Drawings or Orders. Drawings are the representation of inputs the user makes to the canvas and then saves. A drawing has a creation date, a link to the filename of its svg or png in storage, and the state of the canvas, which allows the drawing to be loaded in to the LiterallyCanvas portion of the site. Drawings store a creationDate so that they can be displayed in chronological order, allowing users to easily find drawings they were recently working on.

A drawing can be linked to 0 or 1 users. If they aren't linked to a user, they must be linked to by an Order. This will mean that the person who made the order did not log in. We do

allow non-logged-in users to make orders, with the condition that they will be unable to view their order status on the site.

In addition to having a list of drawings they've made; users can also have a list of orders. The orders have a date, and two Strings indicating if they've been approved for production, and if they've been fulfilled (laser-etched into the tape, and shipped). If they have been fulfilled, the tracking # should be filled in, which will allow the user to track their order in transit. The reason for making the status be a String, is that if an order is rejected, we would like to leave a note saying why. In code, these are represented as Javascript objects which can be put into Firebase, for example, Order is shown below.

```
function Order(userId, drawingId, shippingAddress) {
    this.drawingId = drawingId;
    this.userId = userId;
    this.shippingAddress = shippingAddress;
    this.orderId = drawingId;
    this.approved = "Pending";
    this.shipped = "Waiting for approval";
    this.orderTime = new Date().getTime() * -1;
}
```

## 4.2 Login

Firebase allows the site to listen to authentication state changes, so I have a callback for when a user logs in or out. Upon login, there are several things that need to happen. First, if this is the user's first time logging in, their information should be written to the database. While this is not strictly necessary due to the authentication object storing all the necessary information, I find it easier to keep all the information together in the database, and put the general user information like name and email in the same spot as the more specific information, like orders and created drawings. Below is the code for detecting an auth state change, then checking if the user's information is saved in the database already, and if not, uploading it.

```
firebase.auth().onAuthStateChanged(function(user) {
    if (user) {
        // User is signed in.
        saveNewLoginToFirebase(user);
        // Initialize the canvas if it hasn't already been, IE they log in
        // after beginning to draw
        // ... page view manipulation removed from sample
    } else {
        // No user is signed in.
        // ... Code to execute when user logs out. Removed from snippet
    }
  });
```

```
// Returns true if this was a new login, false if no info needed to be saved.
function saveNewLoginToFirebase(userData) {
    firebase.database().ref(USER_PATH + userData.uid).once("value")
            .then(function(firebaseSnapshot) {
        if(firebaseSnapshot.val() === null) {
            //If they've never logged in before, create the initial entry based
            //on facebook or g+ data
            var uid = userData.uid;
            var email = userData.email;
            var name = userData.displayName;
            if(name === null) {
                name = email;
            }
            var photoLink = userData.photoURL;
            newUser = new User(uid, name, null, email, photoLink, null, null);
            firebase.database().ref(USER_PATH + uid).set(newUser);
            return true;
        }
        return false;
    });
}
```

For this code to get called, the user needs to log in. For Facebook, this is accomplished with the code below, with Google being similar and using a different provider.

```
function facebookLogin() {
    var provider = new firebase.auth.FacebookAuthProvider();
    loginWithProvider(provider);
}

function loginWithProvider(provider) {
    try {
        bootbox.hideAll();
    } catch (error) {
        // Just means we tried to login from a page without bootbox
    }

    firebase.auth().signInWithPopup(provider).then(function(result) {
        // This gives you a Google Access Token. Used to access the Google API
        var token = result.credential.accessToken;
        // The signed-in user info.
        var user = result.user;
        return user;
    }).catch(function(error) {
        // Handle Errors here.
        var errorMessage = error.message;
        console.log(errorMessage);
    });
}
```

Signing in and creating a custom account are a little different, since the email and password aren't necessarily known by the provider already. The code for custom account creation and login is below.

```
function customLogin(button) {
    var email = document.getElementById(emailId).value;
    var password = document.getElementById(passwordId).value;
```

```
    firebase.auth().signInWithEmailAndPassword(email, password).then(function() {
        bootbox.hideAll();
    }).catch(function(error) {
        // Handle Errors here.
        var errorCode = error.code;
        document.getElementById("error_text").innerHTML = error.message;
        // ...
    });
}

function createAccount(button) {
    var email = document.getElementById(emailId).value;
    var password = document.getElementById(passwordId).value;
    firebase.auth().createUserWithEmailAndPassword(email,
password).then(function(result) {
        bootbox.hideAll();
    }).catch(function(error) {
        // Handle Errors here.
        document.getElementById("error_text").innerHTML = error.message;
    });
}
```

That about sums up the login code. Having Firebase is especially helpful here because as you can see, there's no need for any serverside code or cookie storage on my part. The state listener is also helpful, because it will always be called when a user logs in or out, and I don't have to remember to put callbacks into all the login and logout calls.

## 4.3 LiterallyCanvas

LiterallyCanvas was very easy to set up, and required only a single method for initialization, in addition to an div with the class "user-drawing," seen below.

```
function initCanvas() {
    //Start Literally Canvas
    var img = new Image();
    img.src = "skateboard.svg";
    var backgroundShape = LC.createShape('Image', {x: 0, y: 0, image: img});
    document.getElementById("intro_text").style.display = "none";
    document.getElementsByClassName('user-drawing')[0].style.display = "block";
    canvas = LC.init(
        document.getElementsByClassName('user-drawing')[0],
        {
            imageURLPrefix: './img',
            strokeWidths: [1],
            defaultStrokeWidth: [1],
            tools: [
                // Don't use pencil, doesn't close automatically
                LC.tools.Eraser,
                LC.tools.Line,
                LC.tools.Rectangle,
                LC.tools.Pan,
                LC.tools.Eyedropper,
                LC.tools.Polygon,
                LC.tools.Ellipse,
            ],
            // Add skateboard shape to the background
            // and then uncomment the below line.
```

```
            backgroundShapes: [backgroundShape]
        }
    );
}
```

I also did some modifications to the LiterallyCanvas source code, to make the Polygon tool always create a closed polygon. Unfortunately, I was unable to use their provided API to create a neat isolated tool that does what I wanted. Because of this, the actual code for creating the polygons isn't very readable, and I don't fully understand how it works, I just added some extra calls to existing methods in the library. Because of this, adding a snippet in here would not be very instructional. Essentially, there was already code that would complete the shape if the beginning and end of the polygon was close enough, so I just made that length longer.

## 4.4 Stripe + PHP

There are two components to the Stripe flow. First, the submission client side. This is a bit complex, because the drawing's key needs to be submitted along with the payment information, meaning the key must be created after the user presses submit, but before they leave the page. Luckily Stripe provides a "token" property in their modal API, which allows a callback to be made before form submission. This way after a user presses the "submit" button, the image is uploaded and the drawing saved, and the resultant key is tacked on to the form's submission, seen below.

```
$('#stripeButton').click(function(){
    if(!hasUserDrawn()) {
        alert("Please create or select a drawing before purchase");
        return false;
    }
    // Token is the function that will be called after user enters all
    // info. Arugments is the shipping and billing addresses.
    // for token and arguemnts keys, see StripeKeys.txt
    var token = function(userInput, args){
        var $input = $('<input type=hidden name=stripeToken />').val(userInput.id);
        // Create new form, put in custom attrs, then submit
        var form = $('form');
        for(var value in args) {
            var addrField = value;
            var addrFieldVal = args[value];
            var inputField = $('<input type=hidden />').attr('name',
                addrField).val(addrFieldVal);
            form.append(inputField);
        }
        saveDrawingAndClear(false).then(function(uploadedKey) {
            var $keyInput = $('<input type=hidden' +
                ' name=orderedDrawingKey />').val(uploadedKey);
            $('form').append($input).append($keyInput).submit();
        })
    };
```

```
    StripeCheckout.open({
        key:                '<?php echo $stripe['public_key']; ?>',
        billingAddress:     true,
        shippingAddress:    true,
        amount:             <?php echo $popup_vars['tape_cost']; ?>,
        currency:           'usd',
        name:               '<?php echo $popup_vars['popup_title'] ?>',
        description:        '<?php echo $popup_vars['popup_desc'] ?>',
        panelLabel:         'Checkout',
        image:              '<?php echo $popup_vars['popup_logo'] ?>', // small logo
        token:              token
    });

    return false;
});
```

Once the form is submitted, Stripe generates a one-time token that can be used to charge the user. This is POSTed to whatever page is specified in the form submission, in my case the page is order_received.php, specified in the form tag below

```
<form action="./order_received.php" method="POST" id="payment-form">
```

In order_received.php, all the relevant information is stripped from the POST variable using PHP, and the user is then charged for their order.

```
<?php
    require_once('./stripe-php-4.4.0/init.php');
    require_once('./config.php');

    $token = $_POST['stripeToken'];
    $drawingKey = $_POST['orderedDrawingKey'];
    $shipping_array = array(
        'name' => $_POST['shipping_name'],
        'street' => $_POST['shipping_address_line1'],
        'city' => $_POST['shipping_address_city'],
        'zip' => $_POST['shipping_address_zip'],
        'state' => $_POST['shipping_address_state'],
        'country' => $_POST['shipping_address_country']
    );

    $cost = $popup_vars['tape_cost'];

    // Charge the user's card:
    $charge = \Stripe\Charge::create(array(
      "amount" => $cost,
      "currency" => "usd",
      "description" => "Grip tape. DrawingId:".$drawingKey,
      "source" => $token,
    ));
?>
```

Notice that the amount charged is the $cost PHP variable. This is set based on a value in the config file included at the top, and not based on any POST data. This is because the POST can potentially be manipulated by the customer. If they manipulate any of the other variables, it's

more of a loss to them than it is to us, but if they changed the cost before POSTing the form, we would be charging them an amount different than what is advertised.

## 4.5 Saving Drawings

Saving drawings is an important part of the app, and is mostly handled by LiterallyCanvas. However, I also need to upload the saved canvas snapshot to the database, which is done with the code below. I save both an SVG and PNG, because PNGs are easier to display in the browser as a preview. The uploading is done one after the other to make sure both are uploaded before leaving the page.

```
function uploadDrawingPngAndSvg(drawingKey, doneUpload) {
    var snapshot = canvas.getSnapshot(['shapes','imageSize','colors',
        'position','scale' ]);
    var image = LC.renderSnapshotToImage(snapshot).toDataURL().split(',')[1];
    var svgString = LC.renderSnapshotToSVG(snapshot);

    //Save PNG for preview files - easier to display in browser.
    //The PNG preview should have a download link to the SVG
    firebase.storage().ref(DRAWING_PATH + drawingKey + ".png")
            .putString(image, 'base64').then(function (snapshot) {
                //Save SVG for use in conversion to DXF and production
                var svgUpload = firebase.storage().ref(DRAWING_PATH +
                    drawingKey + ".svg")
                        .putString(svgString);
                svgUpload.on(firebase.storage.TaskEvent.STATE_CHANGED, {
                    'complete' : function() {
                        doneUpload();
                    }
                });
            });
}
```

The other important part of saving images, is being able to load them. Below is the code that fetches the PNG previews from firebase, and displays them in a div passed as a parameter.

```
function addPreviewImage(png, drawingId, domLoc) {
    domLoc.insertAdjacentHTML('afterbegin', "<div class='drawing-container'>" +
        " <img class=\"drawingPreviews\" " +
        "onclick=\"loadDrawing('" + drawingId + "');\"" +
        "src=\"" + png + "\"/> </div>");
}
function getPreviousUserDrawings(putDrawings) {
    // Make sure to order by key so the newest one is always at the top(ish)
    // Sometimes they'll be a little out of order due to load times.
    var userDrawingsRef = firebase.database().ref(USER_PATH + globalId + "/"
            + DRAWING_ID_PATH).orderByKey();
    putDrawings.innerHTML = "";
    userDrawingsRef.on('child_added', function(snapshot) {
        var drawingId = snapshot.key;
        var thisDrawing = firebase.database().ref(DRAWING_PATH + drawingId);
        thisDrawing.once('value').then(function(snapshot) {
            var snapVal = snapshot.val();
            var pngStorage = snapVal.finalDrawingLink + ".png";
```

```
            // This is the part that slows it down.
            firebase.storage().ref(pngStorage).getDownloadURL().then(function(url) {
                addPreviewImage(url, snapVal.id, putDrawings);
            }).catch(function(error) {
                console.log(error);
            });
        });
    });
    return userDrawingsRef;
}
```

### 4.6 Summary

Those are the main code points of the app. I left out most of the formatting because as mentioned earlier, it will likely be changed a lot in the future. Overall I think that aside from the LiterallyCanvas changes, my code is relatively easy to navigate and make changes to. Everything is nicely contained in its own file, and I tried to keep the actual logic separate from the display code. One thing I wish I had done is written more tests, but unfortunately I've never done testing in JavaScript before, and it does take significantly more time to write tests, and I did not feel like I had the time.

## 5. Lessons Learned and What to do Differently

### 5.1 Clarify Expectations

One thing that this project taught me was that even if you think you know what the customer is expecting, you should still double check. When I first started working on the project based on Matthew's description of what he wanted, I thought it was going to be a design etched onto the bottom of the board. He clarified this early on, when I showed him my Software Requirement Specification. After that, for the first two thirds of the project, I thought that the design was being etched onto a single piece of grip tape. So, it would be effectively the same as normal grip tape, except with cool colors instead of being all black.

It wasn't until I demoed the nearly complete website that he further explained the design would not be drawn onto the grip tape, but that the user was essentially creating shapes that would be cut out of the tape. Luckily this misunderstanding didn't cause a large redesign, but it taught me a valuable lesson. Throughout the development process I was convinced that the user was just drawing onto a normal strip of grip tape, but that was just wrong. Because of this, I had to develop some extra functionality for the drawing tool, namely automatic shape

closing, since pieces can't be cut from the grip tape if the ends of a user's design aren't connected.

To fix this in the future, I plan to go over the SRS more carefully with customers. For this project, I just emailed it to him, and expected that he would carefully read it, but that was not the case. Meeting in person would be the better call, and allows easier communication to make sure everyone is on the same page.

### 5.2 What Users Expect

I also learned a couple of interesting things about typical users. First, they aren't very good at experimentation. Many times, I would show the site to my friends and they might start drawing something, but wouldn't try to use the save button. It also isn't clear what they are supposed to be creating, but once the skateboard background is added, I think that will be resolved. The other step I took to fix the clarity issue is adding a bit of introductory text if the user is not logged in when they load up the site.

Speaking of logging in, I also learned that nobody wants to log in with their social media accounts. Everyone was hesitant to login with Facebook or Google when I told them about it, and said they would prefer to create a new account instead with their email. I thought this was interesting because I feel like a lot of places are incorporating Facebook and Google login, but it turns out there isn't much demand for that. Luckily for me, adding custom email login is not very difficult, and I was able to incorporate it into the website.

## 6. Next steps

Although a significant amount of work has been put in to the front end of the site to allow users to create designs, there are still some features missing, and the physical infrastructure needs to be finished. Some features that I would like to add given more time are better drawing tools – perhaps a pencil tool that will autofill, like the polygon tool. I would also like to add the ability to see designs that other users have made, and can order those so that even the artistically challenged will be able to find value in the site. Lastly, a feature that Matthew requested and I was unable to get working was the ability to choose different colors

for the same shape. Current functionality allows the user to specify the fill color while they are creating the drawing, but once they make the shape, it is stuck as whatever color they chose. I think some sort of paint bucket tool would be nice to have.

Overall, the site now has a solid base to build from, and could potentially start receiving orders today. Although there will always be more features to add, as is true for any software project, I'm satisfied that I could complete everything I originally set out to do, and am confident that Matthew will be able to start the custom grip tape company that began this whole project.

# 7. References


## 1. LiterallyCanvas

Available at http://literallycanvas.com/

## 2. Firebase

Available at https://firebase.google.com/

## 3. Stripe

Available at https://stripe.com/

## 4. Custom Grip Tap

Available at https://www.sonkrein.com/seniorproj/