A Developer's View of the Shopify Platform

Six Years of Working with Merchants and their Shops

By David Lazar, P.Eng

Table of Contents

Forward

Being a software developer for the Shopify platform is a great way to work with computing skills that are currently in demand and to pay the bills. Ecommerce is seeing explosive growth and there are new markets constantly emerging. Working on Shopify merchant sites and App projects, your email inbox fills with letters like: “Just thought you’d like to know Shopify sent you $148.00 USD” or “FizzBuzz.com just sent you $2622.50 USD with PayPal”. That should inspire the effort to learn the API and programming patterns for ecommerce! It is a great way to earn *passive* income that you can combine with working for merchants to make *active* income. It’s a career, can be the basis for a successful business, and offers up a wide range of interesting opportunities. Specializing on theme development is financially rewarding for those with design flair and skills. App development and making custom scripts for merchant shops is my favourite. It offers the challenges of using cloud computing, networking, database development, connecting to API driven services like social media and many others.

It is great when the code you’ve written helps a merchant increase their sales by 20% overnight! I sincerely hope that developers who are considering to work on Shopify projects read this and are inspired to work on making the next great merchant shop or App. I have worked with and for hundreds of happy merchants and they continue to tell me how happy they are with the Shopify platform.

Preface

I like to think that I don’t really mark time with much enthusiasm or rigour. When I began my professional career developing code to non-destructively test steel bridges for structural integrity, I marked off eight years before I sensed I had done enough crazy climbing and driving, and that I needed a break. I spent a good five years around the dotcom bubble renting a nice office with good friends while trying to make web apps. Then I spent five more years taking on a variety of consulting gigs that, in the end, never amounted to a hill of beans. But it did teach me how to wear various kinds of business hats.

I ended up catching the Shopify bug and tried working with a boutique web agency. As five more years passed, I realized that I knew a lot about running an independent computer business and that I knew a lot about tricking out Shopify for an ever growing list of clients. I have yet to give myself an official work title, nor have I declared my intentions or loyalty to any one type of computing. My corporation is pretty much a sad but expensive accounting exercise that I leave to those who get a kick out that kind of thing. I am sure my corporation is going to reach a decade in age as neglected as my vinyl

collection. But one needs a corporation to run off invoices and interact with clients in foreign nations, so I go along with that. Dividends anyone?

One beautiful autumn day in Montreal, Shopify came for a visit with their CTO Cody Fauser, Joey Devilla on the Accordion, and Edward Ocampo-Gooding, the King of the API, with much swag and the company credit cards to cover the beer bill. We owe that to Meeech who organized it all and attracted a small but curious crowd of locals to hear more about Shopify. I decided at that time to pitch this effort – an eBook written from the perspective of a developer who has a lot of experience working on but not for Shopify. Over the years my nom de plume of Hunkybill on the Shopify public forums, wrote many words on many subjects and I endured a gamut of possible responses. My all time favourite is still *Plonk* because that word continues to tickle my funny bone. If a couple of thousand Hunkybill posts survive on some hard drive in some data centre for a few more years that is one thing, but I was pretty sure I wanted to create something a little more structured and formal.

Acknowledgements

First and foremost, I want to say a big thank you to Shopify for supporting this effort through the Shopify fund. Over the years I have expended energy and time recommending Shopify to all comers. At the same time, I can see that I was also a critical and not always a perfect gentleman in my writing. Instead of ignoring me and my pitch, Shopify chose to sweep aside some of the lower points from the past and take the high road. For that, I am grateful.

To all the developers and designers that I have interacted with, I owe a hearty thanks to as well. Some fine people have come and gone from the Shopify community over the years and I respect the fact that a lot of their contributions remain useful and just as valid today, as they were five years ago. It is very inspiring to have witnessed how one Caroline Schnapp jumped into the Shopify community, starting off with some simple scripting work to make shops better, until ultimately she was so valuable to Shopify that they hired her! My thanks to Caroline for sharing with me her wisdom about how Shopify really works and for always correcting me when my forum posts are clearly sour or just plain wrong. I mean no harm to anyone and am trying hard to avoid controversy these days.

I also want to thank Tobi Lütke from Shopify for his numerous great contributions to open source software like Liquid and Delayed Job. Programmers always learn by studying what other successful programmers have done and I must admit that I have learned many good things from his code.

Thanks to Jesse Storimer for sharing his eBook tools with me and his code. His original Shopify API Sinatra application was the inspiration that moved me to develop an App for myself and today I have over 500 installed Apps all based of off Sinatra running well in the cloud thanks to Heroku! The best developer friendly cloud platform has to be Heroku.

Thanks also to Cody, Edward, Joey and Harley for their hospitality when I trucked on down the 417 to Ottawa for a visit to the new office. The new office is a very swell place to work and I am sure the entire gang that works there have one of the better workplaces in all of Canada to go to on a daily basis.

Thanks to my wife and kids for putting up with a guy who likes to sit in the corner all day and poke away at a keyboard instead of doing something heroic like flying space shuttles or putting out forest fires or saving baby seals from being clubbed. I try to be exciting for you but I just can't seem to find the right chords.

Who is this eBook For?

If you know nothing about Shopify, this book will provide some information about how it all works. Perhaps enough to inspire someone to go the next step and do some real research.

If you know a little something about Shopify, this book will teach you at least one thing you did not know.

If you want to develop Apps for Shopify or build a theme for someone, this book will shed some light on the particulars of doing that.

What is Assumed by the Author?

Nothing. Assumptions are almost always wrong. This is no written attempt to coddle anyone on the path to mastering some specific aspect of Shopify. This book is *not* a “Learn Shopify in 7 minutes, 7 hours or 7 days” recipe book. There is no attempt to put into words any of the magic spells, incantations or dangerous chemistry that goes into the work-a-day world of a professional Shopify coder.

About the Author

My engineering career started professionally in 1990 at about the same time the World Wide Web was invented, with the introduction of HTTP protocol for the Internet. Before the use of HTTP was common, the Internet was mainly used to exchange email using

SMTP, to post messages and have discussions on Usenet using NNTP, or to transfer binary and text files between clients and servers using FTP. Archie, Veronica and Gopher were other interesting choices to use for searching documents. Once the Internet concept jumped across the early adopter chasm to be embraced by the dotcom businesses and general population, a flurry of changes ensued. Nortel had ramped up to make the Internet as fast and capable as it is today (and ironically Nortel crashed in the resulting process) and throngs of business people sitting on the sidelines with few clues about what to do with this vast network poured money into the foundations of what are today some of the most valuable business properties in the world.

1998 was the year I transitioned my software development focus from R&D, C and C++ development to Internet Computing. I recognized that one could craft very efficient programs with much less code using dynamic scripting languages like Ruby, Python and PHP. If it took 25 lines of Java or C++ to put a button on the screen, Ruby or Tcl/Tk did it in one line. I was hooked on that concept.

The early commercial Internet was all about established software titans showing off their chops. Coding for the web involved Microsoft ASP, Visual Studio and SQL Server to deliver web applications that ran on hugely energy inefficient Compaq Proliant servers. Other quirky technologies like Cold Fusion were often used too. Many friends and colleagues have done what I have and hung out a shingle as an Internet computing engineer. Need a web enabled CRM? I can write you one. Need to manage your freight forwarding logistics company on the web? I can make you a web app for that. Need a website to sell flowers online? I can make you one. Need to send 50,000 emails to your world-wide employees? Got that covered. Have to interface to an EDI system with an AS/400 legacy database and a transactional website? Let's do it. Ariba XML catalog to a SMB web site with an XML catalog of business pens and office equipment? Sure, why not.

In the early years of running your own business as a software engineering consultant or freelance programmer, you like me, will inevitably experience the pain of working capital shortages. You'll collect a cadre of loyal clients with small or tiny budgets. You cannot easily sell them anything from Microsoft, Oracle or most other enterprise technology companies. Try presenting a license acquisition cost of $22,000 for a database server to your SMB client. The world of proprietary and expensive software pushed me to drop all that and adopt Open Source Software as my toolkit. I started developing using Linux as my operating system, PHP as my server-side scripting language and PostgreSQL as my database.

Ignoring Perl as a web app scripting language and adopting the new kid on the block, PHP, meant building out a CMS was best done with Drupal and ecommerce for the masses with osCommerce. There was very little Ruby or Python code for these endeavours so I hunkered down to try and fill my toolbox with the best tools that did not totally suck.

Yahoo came out with their brilliant YUI framework and I jumped on that immediately for the nice documentation and functionalities it provided. Soon after, a software developer named Jack Slocum came along and decided to extend YUI into his own vision of a framework. He called his version YUI-ext and quickly attracted a small but loyal following. His vision was to provide Grids (like Excel), and Trees (like Windows Explorer or Mac OS Finder), dialog boxes, toolbars and most of the widgets that we take for granted today but that were rare back then. Eventually that code matured into its own product called ExtJS that today is perhaps better known as Sencha. Shopify was born at about this period of time in my timeline. Just as Ruby on Rails was becoming the new kid on the block.

PHP and Javascript were constantly evolving, as were web browsers. As Microsoft oddly chose to stagnate with their now infamously bad IE web browser, Mozilla, Safari and Opera continued to extend web browser capabilities. They did enough so that we could realize the vision of web applications with enough sophistication to justify comparing them to native applications. Tools like Firebug and the Firefox browser, combined with AJAX (ironically a huge thanks to Microsoft for asynchronous Javascript) and the Ruby language for offering introspection into a running web app, developers now had something really hot to work with.

Finding Shopify, What a Relief!

Shopify was being developed and released as a beta service. I signed up as soon as the beta was available, knowing I would be able to offer ecommerce to my clients without having to hack osCommerce PHP again. If you have never examined osCommerce code, you are somewhat lucky. It was at best a spaghetti western in terms of code. Shopify offered a hosted service, eliminating worries about security, backups and system administration tasks. Having to maintain my own Linux and Windows servers in a co-location facility over many years provided me with ample opportunities to freak out with the responsibilities of system administration tasks. How to SSH tunnel between boxes? How to best freak out when hard drives fail? How can RAID can be a source of mirror images of garbage? How come hackers keep trying for years to break into my puny systems? How lousy is it to have to patch or upgrade services like Apache, PHP, Postfix etc? Answer: very much a lousy thing, at least for me. There is nothing like finding out a client's SSL certificate has expired and I cannot exchange it for a new one without the password for the old one which has been long forgotten or lost. There is nothing quite like finding out that the database has failed to record entries for a few weeks because of a memory corruption bug.

Shopify to the rescue! Hosted ecommerce! No more headaches! Simple templating with Liquid! The ability to use and write any Javascript I want! That is the pattern that works for me and should work for almost anyone.

Chapter 1

The Shopify Platform

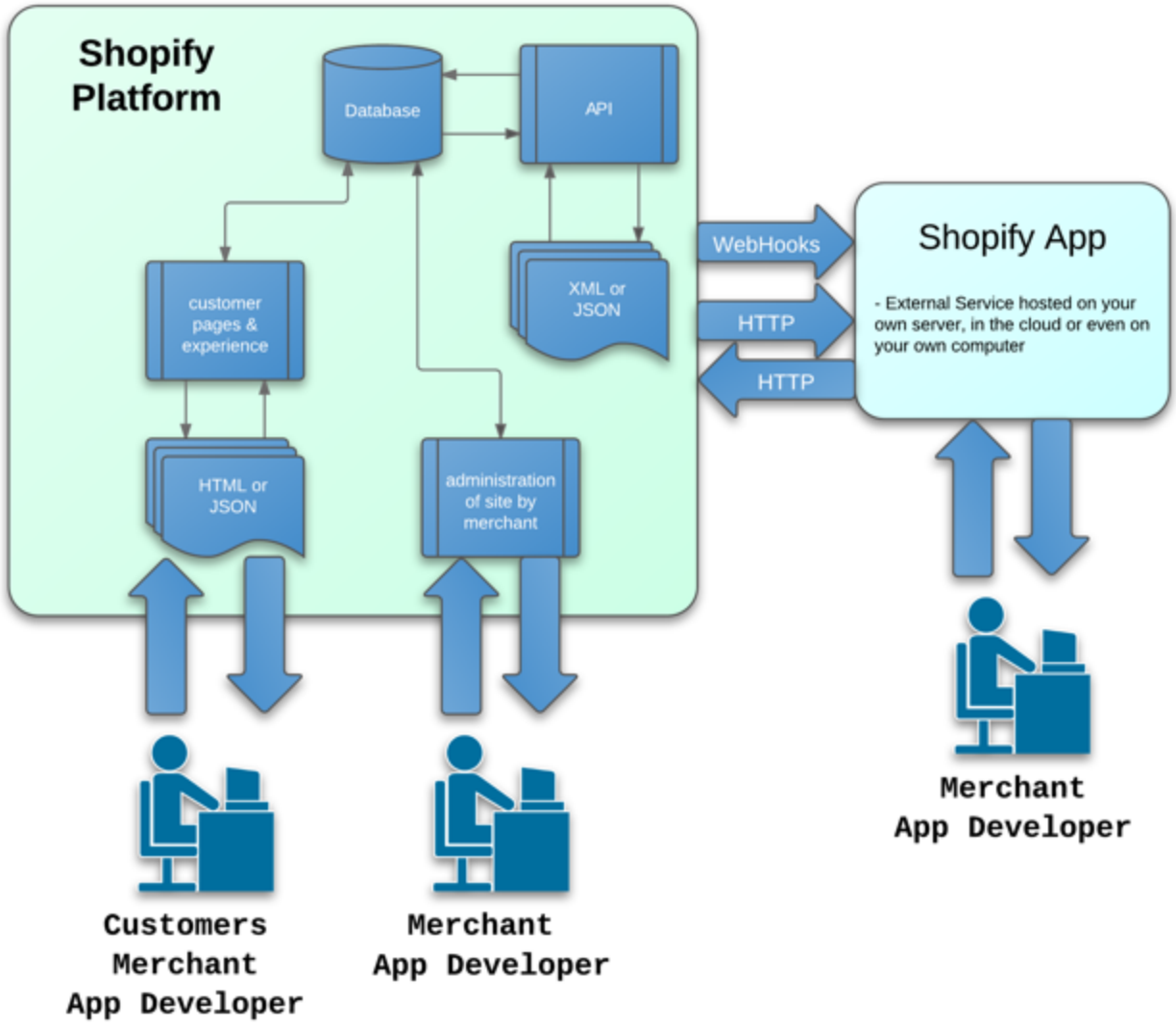
Shopify's API allows external computer programs to interact with a shop according to certain rules. Almost anything the merchant can do to their own shop from their admin can be done from the API.

If an attempt is made to communicate via the API to a shop and that request is authenticated, it will go through and probably return some data as XML or JSON. Maybe that request will return HTML in a separate website. As long as the request is formatted correctly and has the correct authentication token, it will work.

A Shopify App is an externally hosted web application that uses this API to connect to a shop and interact with it programmatically. Shopify merchants can give external Apps permission to access their shops to provide additional functionality – this act is synonymous with "installing an app". App installation is usually initiated by a visit to the Shopify App Store.

Some apps are free, others have a monthly recurring fee and some offer a one-time fee as well. It is entirely possible for a merchant to develop or pay to have an app developed, that will only ever be installed and used in his or her shop. There is no real limit to the number of uses for apps as far as Shopify is concerned. This is a very powerful argument for why Shopify is a leader in hosted ecommerce since the API and platform are very well done for this kind of computing. apps can be hosted in the cloud, on a personal server and even run off of laptops in a coffee shop. They consume resources but are entirely outside the scope of Shopify's data centre and Shopify's technical support.

The diagram presented here illustrates the relationships that three specific types of people will have with Shopify: the customer, the merchant and the Shopify App developer.



The Shopify Platform and Users

A customer will typically only interact with a shop. A shop may interact with Shopify using the Javascript API to do Ajax. Both the merchant and the app developer also interact with the shop to ensure that it is rendering properly. The merchant runs the shop and deals with orders. Merchants may also log into their apps to control things like fulfillments, inventory management, communications with customers, status updates of orders and any number of other things. App developers will also likely be logging into the app to ensure that operations are optimal. It is possible for both the merchant, the app developer and the customer to use an app that presents a public-facing web presence. For example, the customer could log in at *someapp.fizzbuzz.com* to check on their order status. The merchant might very well log in at *someapp.fizzbuzz.com/admin/* to deal with setting up special aspects of their orders. Finally, all shop webhooks and app links added to the shop admin by the app, would be accessed at *someapp.fizzbuzz.com/shopify/* through the Shopify authentication mechanism.

Shop Administration

One of the most important aspects of setting up and managing a shop for online sales, has to be the web based user interface for controlling products, orders, themes, navigation, collections and all the many aspects that make up a Shopify shop. Originally, Shopify provided some very basic templates to build a shop and the support functions for setting up an inventory of products and collections. The limited degree of functionality made it possible to quickly open up a shop that looked good. As the number of merchants expanded from hundreds to tens of thousands, the administrative interface became the focus of much attention. Every button or link click is under constant scrutiny, with lots of public discussion on how the changes affected merchants. The administration interface operates as an example of a well thought-out design that relies on incremental change, as opposed to radical makeovers. Considering the backlash that a lot of web applications generate when changing their look and feel, Shopify has done a great job of delivering improvements while maintaining consistency. Merchants have to invest in learning how to use Shopify. So it would seem problematic to be constantly changing the way they interact with their shop. The early version of product tags was a text-based interface. Adding or removing tags was neither intuitive nor awkward, but it was not clear how tags worked to improve a shop for customers. By providing examples and guidance on tags and using them with collections to filter products, they became a much more useful part of administration. Tags are now managed with a nicer interface of clickable pills.

Liquid Templating

Web application developers have adopted numerous patterns and techniques for delivering the HTML that makes up the *look* and *feel* of a web application. The first popular web application platforms usually rendered information through a combination of what is commonly referred to as spaghetti code. The mixing of scripting code with output code. PHP was and still is one of the best sources to find examples of this pattern. There, you can be sure developers will be mixing authentication code, SQL database code, form code and other HTML elements directly in the same file. The obvious problem with this mess is that programmers, designers and integrators are all working on the same code. In actuality, this meant that only programmers were able to perform all of those functions. A great innovation was to separate the two camps. This was done by keeping the responsibility of programming for the programmers and letting designers take care of the views. The introduction of templating for web applications meant a template could render the look, while the application's scripting provided the data. This is a pretty good pattern for systems where the end-user is also the owner of the data. Most Wordpress sites function this way as well.

With a hosted platform like Shopify, there is simply no chance that a merchant can directly script their shop. This is because there is a high probability that a merchant would destroy valuable data with a single erroneous command. Instead, Shopify provides merchants with secure and non-destructive access to all of the shop's data, while providing sophistication in rendering that data nicely. The templating project was created by Tobias Lütke and is called Liquid. It is this open-source templating language that is the real star of the Shopify platform for merchants and theme designers. Liquid provides access to shop data without requiring merchants to have a secret database password nor requiring them to create and run complex SQL database queries. Liquid provides common programming idioms like *looping* over sets of data, making *conditional branches* to do one thing or another, and has also been set up by Shopify to provide a large set of very useful *filters* that can be applied to data.

The Liquid processing phase is crucial to a merchant's solid understanding of Shopify. For example, say a request arrives from an Internet customer for a shop at *someshop.myshopify.com*. Shopify locates the shop and loads up their *theme.liquid* file. By processing the *theme.liquid* code line-by-line, a series of assets get pulled into the theme. Examples are the Cascading Style Sheets (CSS) and the Javascript. Each *theme.liquid* file has to specify where to render or draw the content of the current request. This could be the homepage, a product, a blog, a collection or even the cart. Shopify determines the correct Liquid file to pull in and adds it to the whole page once it has been processed. A theme file like *product.liquid* can specify snippets of Liquid code to include in the processing chain. Sometimes it is beneficial to take a special aspect of a shop, like a homepage carousel showing off products, and keep all of that code together in a snippet. Snippets can be re-used on other parts of the site by including the snippet in the Liquid files that need them. Once all of the code is assembled, processed and added together, the Liquid is quickly converted into HTML and receives output as a complete web page for the customer.

The implication for merchants and developers is that they cannot inject code into Liquid. You cannot use Javascript to alter Liquid. It must be understood that Liquid is going to completely process all of the shop data and provide the HTML Document Object Model (DOM) that is rendering in a web browser. Shopify provides an opportunity to take advantage of this Liquid rendering stage to make theme rendering somewhat programmable. Any settings that a theme provides as Liquid gets processed too. One example of where this comes in handy, is in the style sheets for a site. You can name a CSS file *uncle-bob.css.liquid* and this file will be processed with any Liquid being turned into actual CSS directives. A merchant that wants to make their text *bold* and *red* instead of *italic* and *blue*, could use the shop's theme settings link. There is a huge amount of creativity available with this system. It also keeps merchants and developers on track since they cannot break their shop. Totally illegal and bad Liquid will usually generate an error and not be saved or used. Shopify is also all about speed and is one of the *fastest* hosted

platforms available, thanks in large part to Liquid that has been compiled into HTML and aggressively cached. If five, ten or even hundreds of people all request the same shop at the same time, they will all get the same copy of the HTML in their browser quickly. Shopify will not be compiling the Liquid five, ten or hundreds times, but instead will deliver the shop from fast cache nearest the customer. This is why it is important for merchants and developers to write efficient Liquid whenever possible. It is very easy to set up loops within loops within loops to provide a clever shopping experience. However, when analyzed with an algorithmic eye, while these kinds of Liquid constructs do work, they are inefficient. Shopify will not usually interfere with how people use Liquid.

Possible Distributions of Shops and Their Uses of the Shopify Platform

Javascript API

Javascript is usually used inside web browsers on HTML documents. Web applications use HTTP GET and POST to process requests and generate responses for those browsers. Every submit button or link click can result in a whole new web page being sent to the browser. Microsoft developed Asynchronous HTTP that came to be widely known as AJAX (or XHR). AJAX allowed a website to send a request to a server and then listen for the reply, which could come later using a callback function. This radically changed the entire web since it meant that a website could replace a small part of itself with new

information, rather than the entire page. Shopify fully embraces this with the Javascript API. Any shop can use Javascript to send AJAX requests. An example would be to load a cart with 16 products, each with a unique quantity. Once that completes, another request could request the contents of the cart in order to get the total price of all the cart items. Javascript can set a cart note from the homepage. It can delete a product in the cart from the product page. There is a whole world of very useful scripting that can be done with basic Javascript and the Javascript API. The Javascript API truly is a crucial and wonderful aspect of developing shops on the Shopify platform.

There are some interesting Javascript files, apart from the Shopify Javascript API, that are available to help make shops more sophisticated in their product presentation. The most common is probably the *option selector* code. This is available for shops that want to present products with various options like colour, size or material. The natural HTML element to use when presenting options is the *select* element. The use of options can increase the number of variants dramatically. A product that has three colours, four sizes and six different materials generates seventy-two variants! Shopify allows up to one hundred variants per product. The Javascript option selector code presents each option in its own select element making it pretty easy for a customer to select from the available colours, sizes and materials to quickly find the variant they want to purchase. It also allows a merchant to present when variants are sold out or even hide unavailable products. Shopify offers a small Javascript file that will hide any options that are sold out, allowing a shop to present only variants that are in stock. This is a very handy extension to the usual option selector code.

There is also free Javascript that can present estimated shipping costs in the cart, so they can be seen before checkout. The customer sees a form that collects the general delivery location via a zip or postal code, and then uses the Shopify API to query the shipping services set up for the shop. For shops that offer high-priced small-sized items (eg. jewelry) or ones that offer cheaper large-sized items (eg. stuffed animals), shipping can be a real problem to accommodate well because freight is calculated based on dimensional packaging and not necessarily weight. The Shopify shipping estimator can really help customers see a close approximation of the true price of a purchase before any shocks occur at checkout. This can help convert carts into sales.

Shopify also provides some Javascript called *Customizr*. If a merchant has to record a name for an engraving on a glass baby bottle or collect the initials to be monogrammed on a leather handbag or any other number of customization tasks, it is almost certain that collecting this information will be done using *cart attributes*. Cart attributes are passed through checkout and are available in the *Order Notes* section of the order's details. Cart attributes can be rendered in the order confirmation email, in order to inform the customer of their purchase details. Further, there are opportunities to help a shop since orders can be sent to a custom app using WebHooks. A custom app can also perform

additional business logic based on the cart note and cart attributes. This is described in more detail when talking of App development itself elsewhere in this guide. Using cart attributes to collect information is *complex*. This is because a merchant has to collect data at the product presentation level while recognizing previous customization efforts. When switching to the cart view, shoppers will often want more than one product. The customization code has to manage quantity as well. When shoppers remove an item from their cart, any corresponding cart attributes have to be dealt with. Checkout is separate from the shop and if a shopper starts checkout but then decides to revisit the shop, all the customization has to be available. Hence, any code to handle customization has to be rock solid and demonstrate a mastery of browser features like localStorage, cookies and the ability to serialize data as JSON. Ultimately, a merchant has to be able to know if a glass baby bottle gets etched with the name "Eddy" or a handbag gets the initials "G.B.H" or "S.N.F.U". Because Shopify provides it, Customizr goes a long way to making this possible, without a major investment in custom Javascript code.

Shopify API

For merchants and developers, the Shopify API is the most important and best feature of Shopify as an ecommerce platform. For those that like convenience (and really who doesn't), the various available API wrappers offer excellent opportunities to work with shops and their data. The Ruby version provides a Command Line Interface (CLI) that can query a shop using the API with just a few lines of code. It is possible to alter every product in a shop by changing the product's *type* or *vendor*. This beats downloading the inventory as CSV, editing those values, importing that CSV and waiting many minutes.

Developers should join the Shopify Partner program to begin working with Shopify merchants on both their shops and on apps. The partner application allows developers to create Test shops and apps. To get started when creating an app, it needs a name. Once created, it generates an API token and a shared secret. Then, it can be installed into a shop. An app can issue HTTP requests using XML or JSON formatted data. An app that formulates requests, as per the Shopify API can be used to provide additional functionality to a shop. Apps can be created to add features to just one shop as a private exercise, or created for installation in thousands of shops. Hosting an app in the cloud makes it easy to ensure that the app has the resources to handle hundreds or thousands of installations. Some developers prefer to run their own servers and that works equally well. To test apps out, they can even be on a laptop located in a coffee shop.

The API itself is well described by Shopify. However, it may still be opaque to merchants and developers exactly what this API can be used for. One example to illustrate the use of a custom app and the API, is a shop that sells works of art, an online art gallery. Art is a unique domain with specific features that make it interesting and non-traditional for

ecommerce. In an online art gallery, a work of art like a painting is for sale as the product. Paintings can be organized into collections, just as they would in a gallery or museum. For example, paintings may be categorized as being abstract, contemporary, mixed media or print. The challenge for a gallery is to present all of the unique attributes of a piece of art. A painting also belongs to an artist, who almost certainly has a biography. Additionally, a painting has a history. It may have been exhibited in other galleries and museums. Each exhibition adds value to the painting's narrative. The history, artist's information and other metadata is best captured as resources that are attached to the painting, rather than as something stuffed into the painting's description. Using the API, it is easy to capture this metadata and create resources that can be attached to each product. Thus, a shop can manage products for sale so that they render their own history and without it being tightly coupled to the product description. Using an app and the API, it is possible to capture all of the unique product details as resources and to present those resources when needed.

To make a gallery-style shop easier to manage, these extra resources have to be available to the merchant and designer so that they can be presented in the shop's theme as Liquid. An app has an option to add an *App Proxy*. An App Proxy serves as a *special page* in a shop. A online art gallery shop can navigate clients to the special page and render all of the custom resources. An example is when showing the special page, there is a Liquid template that contains custom Liquid tags.

```
<div class="artist">
  <h2 class="artist-name">{{ artist.name }}</h2>
  <p>{{ artist.country }}</p>
  <p>{{ artist.biography }}</p>
</div>
```

When a customer visits this page, there is a brief request made to the App Proxy providing the app with the artist's identification such as name. The app looks for the Liquid template it installed in the shop and it reads that template since it could have been altered by the merchant or theme designer. The App Proxy then searches the app for the artist's information based on the name provided. Once found, it replaces the Liquid tags with the appropriate information. The app then sends the Liquid template back for processing by Shopify. Customers will see the artist's information like their name, country and biography, along with a collection of the artist's paintings, all rendered from a standard Shopify collection. This is one example of the endless possibilities and combinations which can be created with apps and the App Proxy.

The combination of the Shopify API, externally hosted apps and their App Proxies, provide merchants and developers with an amazing degree of latitude to make shops that go far beyond the basics of selling snowboards or t-shirts. Constant development and

improvement to these platform features ensure that we will be seeing incredibly slick and interesting shops in the future.

Theme Store

When Shopify went public, there were just a handful of themes available. Merchants could download these themes as zip files and tinker with the HTML, Javascript, CSS and Liquid constructs using a text editor. There were three ways to see the effect of editing a theme. One was to edit the theme on your computer, zip compress the edited files and upload them to the shop. A second way was to edit the theme directly inside the shop administration screens. This was not a great option at that time, since Javascript editors were primitive and error-prone. The third was provided by a web application called Vision, which you installed on your computer. Vision provided a small inventory of snowboards for sale and the shop templates showed off any theme changes the merchant made. This approach was panned and ultimately abandoned as it failed to keep up with Shopify itself. A lot of merchants had trouble with theme changes that always showed snowboards for sale. The Vision inventory of products, pages and collections could be hacked because it was just a text-based YAML file. But this was also tedious and prone to error. Most merchants were happy when Shopify abandoned Vision and decided to provide Test shops to work off of. As mentioned, when you become a Shopify Partner, you can create as many Test shops as you need and assign as many themes as you want to them. You can also load real inventory into a Test shop. This provides a great way to develop a theme without the risk of borking a live production shop from making sales. At the same time as the Partner program was initiated, Shopify presented a Theme store where merchants could download free or paid themes. Here, paid themes usually come with features that might otherwise cost money for merchants to develop themselves.

For talented web designers that understand the nuances of cross-browser issues with respect to Javascript and Cascading Style Sheets as well as Liquid scripting, having a theme in the Theme store is a great way to make money. It is also likely that the first thing a new Shopify merchant will invest in is the look of their shop and hence, a visit to the Theme store.

App Store

Like the Theme store, when Shopify released the API and paved the way for apps to be developed and integrated into shops, they created the App store. Shopify's mantra of keeping their core set of features simple ensures there is a useful place for custom apps. Custom apps can deliver any extras merchants may demand. There are hundreds of apps

that can be installed in a shop and this number is sure to continue climbing as merchants continue to express their desire for additional features.

For Shopify developers, the App store presents their app to merchants as well as anyone researching the Shopify platform. To make the financial aspect of developing and servicing apps easier, Shopify makes their billing API available to developers. In exchange for 20% of app profits, Shopify collects the money from merchants directly and deposits it in the developer's Partner account. This is great for developers that do not have access to online credit card processing. It is possible for developers to set up one-time charges, recurring monthly charges and forget worrying about payments. As of June 2012, Shopify is paying developers on a bi-weekly basis, making it very similar to working for a company.

Shopify Experts

As the number of merchants has grown, it is only natural that some of these merchants will have needs that exceed the core features provided by Shopify. These merchants need to know that they can have their unique processing demands taken care of, without having to launch a search for talent. Shopify Experts was launched to showcase the talent who work on the Shopify platform. Theme designers, app developers, photographers and even accountants can join to show off their portfolio and capabilities. The slots available in the Experts Showcase for accountants and photographers show that these skills are also in demand by merchants. Merchants use photographers to present the best possible look for their products. They also use accountants in order to create reports on what is flowing through their online store.

Chapter 2

Shopify Developer Tools

As an engineer and software developer with some experience, over the years I have coded for many processors in many languages. From Z-80 Assembly language on the TRS-80's Zilog Z-80 8-bit CPU, through 6502, 6800 and 68000 chips, to Amiga 500, Sun Workstation C++ and then to PHP, Ruby and Javascript on homebrew Pentiums. The day-to-day code editing tools have not varied much. Coding complex web applications on a laptop in a coffee shop is also normal. For developers that want to create an app or build out a nice theme for a merchant, all you need is a laptop with Internet connectivity and a desire to learn.

The Text Editor

Vim, Emacs, TextMate, UltraEdit, Sublime Text 2, Coda, Eclipse, Visual Studio, Notepad and the list goes on. Choosing a text editor is a very personal choice. I cannot touch type so in this regard, I can offer no wisdom. I get by with my horrid typing skills, in the sense that I type at about the same speed that I can think of code constructs. As long as my typing matches that speed, I am happy. It is amazing to me when I see people who write code as they stare out of a coffee shop window. If I was hiring a programmer, I would use a typing skills test in the interview. Unsure of where that Esc key is? Are you sure you're a programmer?

Back to the text editor itself, a text editor has to *syntax highlight* Ruby, Javascript, Liquid, Haml and Sass, and other languages. It also has to *autosave* any edits whenever typing stops or focus switches to another task.

The Web Browser

Obviously, every developer needs a good web browser to work with. A good web browser provides decent tools for examining the results of any Shopify theme tweaks or web application interactions. Today, all the major browsers are suitable. However, each browser brings certain quirks to the table. Currently, Chrome, Safari and Firefox offer decent developer tools. While Internet Explorer remains a poor choice because their developer tools are second rate, at best.

Versioning Your Work

Shopify will version templates as you change them. This built-in versioning system is not terribly useful for a team and is not something you want to rely on as your only mechanism to backup your theme. Learning a Distributed Version Control System (DVCS) like *git* is highly recommended. The git versioning system is a basic skill every developer should have. It will also payoff in spades. With git, you can version *everything you work on* – every line of code, every proposal, even binary work like images, and even assets that you might not ordinarily think of as something that should be under version control. Almost all the best open source projects are available using git. There is also a serious community of developers all working and sharing code with git. Github offers free accounts and comes highly recommended as one of the best places to host your codebase.

Dropbox

Dropbox is great way to cheaply share files and serves well for a workflow between small teams and clients. You can toss files into Dropbox to speed communications along. It also beats managing large attachments in email and has reasonable security, most of the time. More and more applications are being released with Dropbox connectivity, so it makes sense to adopt this into any workflow that needs it.

Skitch

Screenshots remain a great way to start conversations about design and functionality. Skitch makes it easy to add notations to screenshots and then share the resulting image with a client. It can also be useful to Skitch 2 month old client invoices, with the time stamp showing them logging in and viewing the invoice. Fire that image off to them and enjoy the mumbled excuses and apologies from their embarrassment. Remarkably, some clients are oblivious to their obligations and these are the ones to watch out for. It's best to keep notes, print and date copies of agreements. Also, watch out for constantly changing requests. Some clients will accept an estimate for work but once the work is delivered, they will suddenly change their requirements and expect the developer to go along with those changes.

Instant Messaging

Using Skype, Adium, Pidgin, iChat or another messaging service can really make it easier to work with clients. Email does not cut it when you want to really rip through a work session with a client and bounce ideas off of them. Screen sharing is one of the quickest

ways to teach a client about what your app does, what the shiny buttons do and to show off the overlooked luxuries you've provided them. Writing a user manual for an app is fine too but that takes many hours. Also, the second you finish writing, the manual is likely outdated with defunct screen shots, descriptions and information because web apps can evolve in near real-time, even after they have been "delivered". Re-factoring code is a constant process and that makes documentation development tough.

Terminal Mode or Command-Line Thinking

Shopify makes a Command-Line (CLI) utility available that can be installed on any computer with Ruby scripting language on it. The utility provides commands that allow a developer or designer to download a shop's theme for editing. Once downloaded, the entire theme's codebase can be checked into a version control system like git. There is also a command that tells the computer to watch for any changes in the theme files. If a change is detected in any file or if new files are added, those changes are automatically updated in Shopify. This lets you refresh your browser and see the changes you've just made. Even better, there is a development tool called Live Reload that will auto-refresh your browser whenever changes are detected to the code. This means you can edit a theme or app and then switch your focus to your browser to see the results.

Localhost Development on a Laptop, Desktop or Other Devices

With text editing, version control and a web browser, a developer is ready to tackle almost any kind of Shopify project. To develop an application that can be hooked up to a merchant's shop, it is imperative to be able to develop the application on your local machine. Testing a script out on a new concept or running an entire app, should not be tied to a server on the Internet. It is crucial to be able to develop localhost when offline. Most scripting languages that are used to develop web applications like Ruby and Python, have nifty web servers for use on a local machine. Other languages get by with programs like Apache and Nginx. With this technique, if you lose Internet connectivity at least you can keep developing and testing an app.

Pre-Compiled CSS

A final tool for the Shopify developer (and one that deserves more attention) is the use of compiled CSS through the use of Less or Sass. Developing style sheets by hand is clearly the least efficient method, especially when a developer does not have a good understanding of how CSS works. Less can be compiled with Javascript. Sass is compiled

with Ruby. The advantages are somewhat spectacular. A developer can build a complex theme using these tools and gain a lot of very important flexibility. By changing one value in Sass or Less, the change propagates throughout the CSS.

Chapter 3

Common Questions

There are plenty of reasons to choose a *hosted* platform like Shopify. One reason is that the implementation responsibility for ensuring that your shop's site is reachable by the public, is purely Shopify's. By offering a platform that provides most of the basic features needed to run an ecommerce website, Shopify has created a thriving community of shops that sell everything from soup to nuts. There is decent community support for most merchants that are new to the platform. It can be comforting to know that when an issue arises with respect to how to configure a shop, there are other people who have faced the same issue and have likely resolved it.

How to Capture that Extra Information

One of the earliest and still most common questions is how to capture custom information for products purchased in an order. There are thousands of shops that need to collect custom information. From glass baby bottles etched with a monogram, handbags with initials stitched into the leather, to silver pendant jewelry with the name of the family dog or the newest twins on the block, the presentation of a form to collect this information is a source of endless discussion by merchants.

With the introduction of *cart attributes*, it became possible (but not necessarily simple) to pass extra data through checkout with an order. The cart attributes are a simple key:value pair where a key is used to refer to some value. An order can have as many of these cart attributes as they need to fully define product customization. A typical key might be the identification number of a product or variant. The value that can be stored with the key is usually a string of text. The following code presents a simple key and its value.

```
attribute['I_am_a_key'] = "welcome to outer space astronaut!"

attribute[12345678] = "David Bowie"

attribute[44556677] = '[{name: "qbf", action: "jtld"}, {name: "lh", action: "lnot"}]
```

Those are three examples of setting a key and its value. Using Javascript, it is possible to experiment by setting a cart attribute and then checking whether it was set correctly. Web browsers all provide Javascript. Javascript represents data and objects in a format called JSON. One excellent fact about this is that JSON can be stored as a text string and that means merchants and developers can create complex data structures and save them as cart attributes with any order! The third example shows this off. The cart attribute for ID

44556677 has been assigned a string of JSON. The value could be read as "There are TWO 44556677 variants in the cart. One is named *qbf* and the other *jtld*". When rendering the cart to customers, it's possible to show these values in the appropriate line item along with the variants. The checked out order will display the same keys and values.

```
44556677: '[{name: "qbf", action: "jtld"}, {name: "lh", action: "lneg"}]
```

Not only is the above script is a little confusing, but some merchants may balk at having the collected custom information look like that. It is possible to deal with this by using more sophisticated code. Instead of directly storing the customization data in cart attributes, it can be stored in a cookie or the web browser's built-in localStorage. During the creation and subsequent editing of customized information, it is handled as JSON. But before submitting the order to checkout, the code can translate the JSON into plain English. That would then be set as the value in the cart attributes. As an example, if a variant represented a type of farm animal for a selling price of $24.00, the attribute could be represented as:

```
attribute["farm_animal"] = "Name: QBF, Action: JTLD, Name: LH, Action: LNOT"
```

This is more readable than the previous example and can be easily interpreted by the merchant.

One important aspect of customization that should be addressed, is that flexibility like this comes with a price. While it does imply that you collect extra information for a product, you can lose certain inventory functionality when you use it. If you need to customize a product with four options, Shopify only has three. So you could decide to collect the fourth option with a form field and use the built-in options for the other three. When you make this choice, you lose the ability to keep track of inventory based on that fourth option. If price changes are involved, you have no choice but to use the built-in options to customize variants. This presents costs in terms of SKUs. There is also a limit of up to 100 customized variants. A virtually unlimited amount of customization is possible, but it should be applied to options that have no inherent cost or effect on inventory management.

Image Switching

Shopify organizes a product by assigning it attributes like vendor, type and description. A product has no price but it does have variants. It has at least one default variant. Each variant has a price and can be setup within inventory to have options. A product also has

images. You can upload multiple images for a product but there is no connection for those images directly to the variants!

If a merchant wants to present a product that comes in five colours or perhaps seven different kinds of fabric, it is likely that they will want to change the main image in order to match the selected variant. For example, when looking at a t-shirt and selecting the colour blue, a customer expects the t-shirt image to change to blue. A common customization job concerns providing this to merchants. Wall Glamour in the UK is a simple example of this. When choosing any kind of wall stickers, you can click a colour palette and the main image changes to match. Another example would be Polka Dot What, where clicks on the left or right leg change the available thumbnail images and the main image.

The heart of the issue is that when there are twenty variants and twenty product images uploaded, how do you connect them together? Recently, Shopify introduced *alt* tag editing for product images. This assigns text to the images. Images with alt attribute text are great for search engine optimization (SEO). They can also be used as a hook so that when variants are selected, an image swap can occur. For novice theme developers, the Shopify alt tag approach is pretty good. You might sacrifice SEO results for image swapping to present a nicer images when customers select different variants.

One cool aspect of swapping images is that all the images are readily available to Javascript from Liquid when you pass the product through the built-in Liquid *JSON* filter. Javascript code can access any image and use it as needed. With Javascript, you can easily substitute different product image sizes and place them anywhere on the screen. Once the Liquid phase of rendering a product is done, it can be left entirely to Javascript to make image swaps with nice effects like fades and other easing motions.

Cross-Domain Support with JSONP, CORS or IFrame Elements

Shopify is a hosted platform and all merchant shops are known by their myshopify_domain name combined with the *myshopify.com* root domain. You can use the domain name system (DNS) to help customers find your shop at any domain you want, such as *www.mygreatshop.com*. Despite this, it is still and forever will be *myshop.myshopify.com* as well.

Since most apps will be hosted in the cloud, you may have to do an AJAX call to an app on a different domain. How can one do this? Cross-domain AJAX is possible using both CORS and JSONP. You can also use an HTML *iframe* element to embed a form in a shop. The cross-domain AJAX request is common and can be quite useful. It can serve as a

support mechanism for sites that need it. To avoid JSONP and just use straight up Ajax, an app needs to be created on a subdomain of the main shop, like *app.mygreatshop.com*.

Special Invites

Before the App store and lockdown apps like Gatekeeper were available, it was really difficult to keep a shop locked down so that only registered customers could access it. It is possible to build apps that present a form to capture a customer's email address and perhaps a secret code. The code and the URL of the app can be presented to the customers you want to be able to access the shop. Each customer that types in the correct secret code and their email address can trigger the app to unlock the password protected shop, using credentials only known to the app. If the app is on a subdomain of the shop, it can set the needed session cookie for a customer and seamlessly transfer them into an otherwise locked shop.

The customer account scripts provided with Shopify can also be setup to provide a modicum of security. There are a few supported patterns for customer account creation with Shopify. It remains to be seen if this has resulted in a better customer experience.

Chapter 4

So You're Wondering About Clients?

It is a good thing to wonder about your clients. The amazing thing about hanging out your shingle as a Shopify developer is the sheer unbridled variety of clients that you will meet. If you are like me and enjoy sticking pins in maps, working with Shopify merchants will probably push you to buy a Costco-size box of pins for your world map. Like the Ham Radio operators of yesteryear, collecting merchant shops from far off countries and cities to work on, is neat. Though I still have no hits from Easter Island, I am holding out hope.

Before corporations caught on and heard all about the Internet, there was little to no online commerce. Book or cake decorating tool orders from Amazon with next day delivery to your door? Not much chance of that, a mere decade or so ago. As Nortel sacrificed itself on the alter of corporate greed (after laying fibre optic networks for everyone), high-speed Internet bootstrapped a new age of electronic goldrush and now *everyone* wants in on that action.

Some merchants understand that while Shopify does not try to offer everything to everybody, what it does offer is sufficient for most of their immediate needs. There are also plenty of merchants that approach Shopify with notions of what they *think* they need in order to succeed at ecommerce. So when you tell them that Shopify does not support their needs directly nor for free, they pop-off like bottle rockets. These little conflicts are great and provide an opportunity to argue from both sides of the fence. Some merchants make a good case but they cannot realistically use Shopify without sacrificing something fundamental in their business objectives. Internationalization is one of those issues. Many merchants need to serve their customers in more than one language. But Shopify does not operate in more than one language. Although these merchants are in the minority, the only resolution for them is to open one shop per language and try to strike a deal with Shopify in the process. Luckily, since templates and inventory can be bulk exported and imported, this is less of a concern than it seems at first.

Other common problems that merchants point out: - Shopify can't do one page checkout, oh my god that sucks! - I can't pre-program the Discount Code! Gah! That's terrible! - I can't just change the price by 10% when they order 2 or more? That is just wrong!

After six years or so, we can be sure that Shopify's support staff are also pretty tired of hearing the same complaints. As time has passed, the Shopify platform has grown and extended its capabilities but it has shown a certain amount of resilience too. Faced with

the choice of trying to do too much or just doing what it does do well, Shopify has embraced the philosophy of doing what it does well.

Merchants Respond to Clear Explanations

A good way of explaining to merchants why Shopify cannot simply just add a feature that they want, is the story about Javascript. Javascript is the lingua franca of all web browsers today. It was created in about 10 days by a programmer with lots of experience at developing languages. Today, it is part of every smartphone, every iPad tablet and all desktop web browsers. For all the flaws Javascript has, it remains much of the same today as when it was created. It turns out that you cannot put the genie back in the bottle once it has been released. Shopify cannot undo the patterns laid down six years ago either. Software is probably mankind's most fiendishly difficult invention and so we have little choice but to go with what works for as long as possible. We do not want to tinker with a working formula. Right Coca-Cola? In 1992, when Coke changed their original formula to New Coke, they quickly learned the meaning of a marketing disaster.

If a merchant complains about checkout, it is fair to ask them about the data that supports that one page checkout converts sales better than multipage checkouts do. Are there enough Ph.D dissertations awarded that clearly show this with certainty? Discount codes are another interesting trend these days. As Groupon and Living Social have demonstrated, handing out coupons to people can be really good for increasing business. Shopify has one text input on the second page of checkout and it cannot be scripted. Merchants have complained that customers are overlooking that field and making mistakes. It is hard to see how to really improve this process without a lot more experimentation and tweaking. Soon enough, it will come to pass that checkout will be made even easier.

Merchants Want Variable Pricing

In Shopify, a product has no price. Only a product's variants have a price. A price on a variant cannot change at any point in a customer's shopping session. You cannot make a price higher or lower through any kind of scripting or fiddling. You just cannot do that. It often comes up that merchants will want to bundle products together. If a customer buys an entire bundle, it should result in cheaper prices for the products. Shopify does not work that way. If you bundle three products that cost $10 each, when they go in the cart they still cost $10 each. The bundle costs $30. It cannot be $25, if it is composed of three $10 products. The way to bundle products for a discount would be to make a product with all three products and then price that at a discount. This can be a difficult process for some merchants.

The *ideal merchant* will understand that opening an ecommerce shop is a tough business. Working with merchants, you want to get them online and making sales. They need to be found by search engines and they need customers. As customers find their shop and the shop makes sales, it naturally becomes easier to show merchants that they can spend a little money on their theme to make their shop beautiful. They can then be convinced to take the next step and perhaps add some custom coding to make product customizations a reality. They can also use apps to assist with their order processing. It is rarely a one-size-fits-all strategy. Often, there are more than a couple of ways to use Shopify to make sales.

Chapter 5

Shop Customizations

Here are some examples of Shopify customizations that have stood the test of time. They underline the flexibility of the Shopify platform.

For example, a merchant launched his shop and was met with initial success. In fact, so many orders came in that the merchant was up against a wall with the shop administration interface. He had one thousand orders that were all paid up and needed to be fulfilled. Those orders were from a short period of time, under a week. He wanted to know how to convert the task of individually fulfilling one thousand orders (which was totally stressing out his wife, who was further stressing him out) into a single click.

Turns out, you can use the Shopify API to inject a link into the Orders Overview screen. So, an app was quickly built that would respond to a click-and-fulfill all the selected orders. His wife was no doubt pleased to be relieved of the burden of thousand-click order fulfillment sessions.

Fulfilling orders in bulk is one small but handy use for the Shopify API. Speaking of one thousand orders, the Shopify API has limits on how much any one app can use in a short period of time. An app has to stay within its limits. Currently, that limit is that an app can make 500 API calls in 300 seconds. If an app fulfilled 250 orders and then a second request came in requesting 250 more orders, followed shortly thereafter with another 250 orders, the API limits would certainly be exceeded. The app would be blocked from using the API. The important task a developer has to think of is ensuring that these limits do not cause problems for merchants.

Using Delayed Jobs to Manage API Limits

Early attempts to work within the API limits resulted in code that would sleep for 300 seconds when the API limit was reached. This turned out to be awkward for many reasons. Keeping track of the number of app operations that are completed without failure, and the number of API calls used, is prone to error. A better solution is to set up a delayed job and process API operations in the background. This turns out to be true of most apps. You never want to lock up the user interface of an app, while doing ten seconds or thirty or more of processing. That is not acceptable. Any created delayed jobs are run by a cheap worker process that just waits around for jobs. Each API call to Shopify either succeeds or fails. If it fails because the API limits have been reached, the delayed job can react to this exception and spawn a new delayed job,in the future, with the remaining work. Using this technique, it is possible to process as many API calls as

needed without limit problems and without slow locked up interfaces. It's a very elegant and robust system proven to work well by hundreds of thousands of successful API calls. Shopify itself performs all of the inventory import and export operations using delayed jobs.

Back to our example, the merchant has solved the fulfillment issue and has now turned to other special issues. For example, his store products are perishable and they always get delivered on a Monday or Tuesday. When ordering the products, customers wanted to order products for more than one week and were comfortable paying for an entire month of products. Shopify does not currently support recurring orders or a subscription service. So, the solution was to provide a quantity field for the product in the cart for each week a person wanted the product. A person could order one, two, three or up to four weeks of products and pay just once for all the deliveries. By recording the delivery dates and quantities of each ordered product, it is possible to know exactly how many products get delivered per week per customer. Some customers order two or more per week so this had an immediate positive impact on the bottom line. There is even a button providing up to 3 months worth of future dates. With tweaks like this, watching a merchant nail 15% higher in sales is rewarding.

Using Cart Attributes and WebHooks Together

Using a WebHook to capture paid orders, an app can be used to inspect the line items and the cart attributes for quantities and dates per product. Setting up a small data structure to record the dates and quantities means the merchant can generate a nice Excel style grid of weekly deliveries, with the ability to plan ahead. Once that step proved successful and many thousands of orders were being booked, it turned out that the ability to automatically fulfill orders using the API was crucial. The reason was that since you could fulfill an order as many times as you wanted, an order that delivered in the future could be fulfilled *each* time a delivery comes up. When you use the API to fulfill an already fulfilled but still open order, the Shipping Confirmation email alerts the customer that their delivery is on the way. Only when all deliveries are completed is an order closed. At that point, it can even be removed from the app.

Adding Upsells to Boost Sales

At this point, the merchant's shop was running smoothly and booking many thousands of orders per week. The merchant wanted to add a new feature to the shop. He wanted to *upsell* special products with the existing products.

For example, when Valentine's Day rolls around, it would be nice to offer a box of chocolates as an additional product. By creating a new product in Shopify and setting its type to *upsell*, the shop could offer this special product alongside its regular products. The app also allows the shop to assign any products of type upsell to regular products. Using Liquid, if the regular product has been assigned any upsell products, we can render them as well. By presenting upsell products, the shop was able to sell a huge amount of additional products per order. Upsells were an immediate hit. Using the API to customize the operation of a shop can really boost sales. One particular day saw an upsell convert on 1449 of 1450 carts. That is pretty impressive.

The pattern of upsells eventually moved back into a pure Shopify solution. Collections were created to hold the products of type upsell. Using a special *product.upsell.liquid* template when rendering products, it became possible to sell all the products in the collection with one 'add to cart' button. This improvement did away with a lot of AJAX code, complexity and demonstrated that shops can truly go through periods of experimentation and evolution, while still recording sales.

Adding SMS Notifications

With so many people using smartphones and SMS services, it made sense to add this to the shop fulfillment operations. Emails can sometimes be blocked by corporate firewalls and can be less reliable. The app sends an SMS to each customer when their order is fulfilled. It was also easy enough to add a form to the shop's *Thank You* page asking the customer if they wanted to receive an SMS when their order is fulfilled. Remarkably, a huge number of people have provided their SMS numbers.

This kind of customization illustrates that you can use the Shopify platform with confidence in knowing that it can handle the twists that get thrown into the typical mix of customer-driven business cases and scenarios.

Other recent surprising experiences came from integrating Shopify with the well-known Salesforce CRM system. It turns out that when you subscribe to Salesforce and want to send orders, they do not process the Shopify WebHook XML properly. So, a quick bridge was built by deploying an app to the cloud to accept WebHooks from Shopify. Then the app forwards the order to Salesforce, using XML formatted for Salesforce. Apps as a Proxy! Additionally, Salesforce comes with some pretty severe limits on what you can do with an entry level plan. Sometimes, it turns out that you can do better with an app running in the cloud. Bridging Shopify to an app and Salesforce has shown itself to be a pretty powerful but cost effective system as well. Shopify is soon to release an approved SalesForce app that will likely appeal to the CRM crowd.

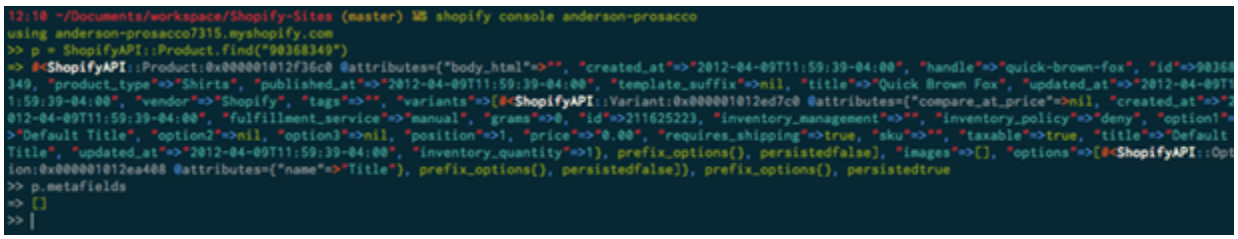
Chapter 6

Shopify Inventory

When working out a development plan for a merchant's shop, the inventory setup should come first and the theme should come last. Unfortunately, many merchants jump into a theme only to realize that while it looked good in theory, the reality is much different with their actual inventory. To achieve success, a shop needs to present inventory in a way that complements a smooth shopping experience. It is not gimmicks that sell. Sometimes deeper thinking about a product, its variants, pricing, images and options, will dictate the best presentation and process. If the inventory is not organized correctly, a lot of effort will be wasted trying to fix up a theme.

At a minimum, a product requires only a title. The title will then be turned into a unique *handle* that is a reference to the product in the shop. A quick example is a product with the title Quick Brown Fox.

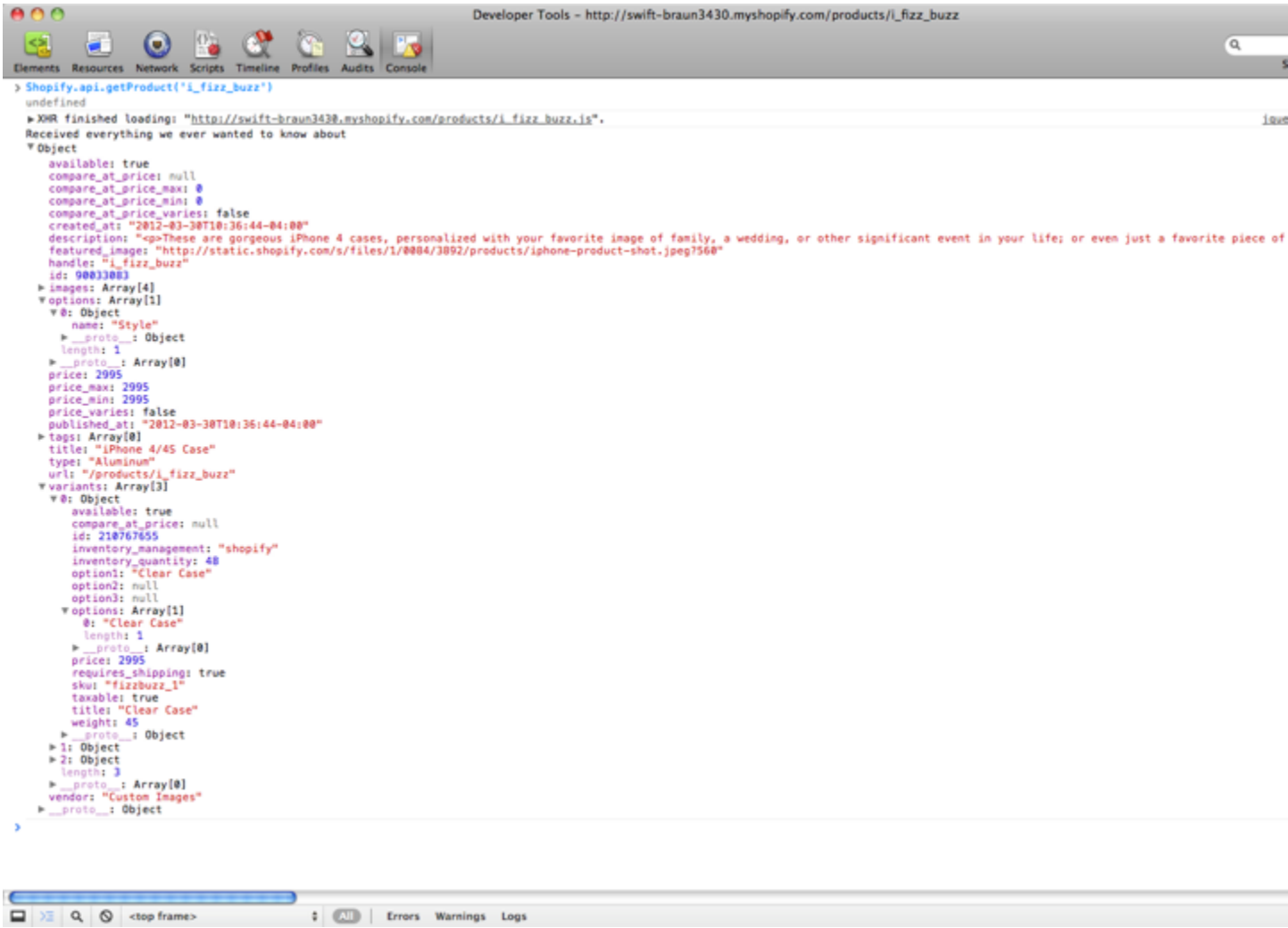
Shopify will set a handle to this product as quick-brown-fox and the product will then be found online as the shop's URL appended with `/products/quick-brown-fox`. With no further action from a merchant, a new product has been created and added to inventory with a zero dollar price, infinite inventory for sale, no tags or images, no description nor other useful attributes. It exists for the purposes of showing up on the site and can be accessed with an API call using the handle or assigned ID. You can *always* get the ID of a product by looking at the URL in the shop admin, when editing it. An example is `/admin/products/90368349` where `90368349` is the unique ID for the product. As long as that product exists in the shop, that number will never change. The number is used during import and export operations in order to identify products that need to be updated. It is also handy to use that number when quickly checking out a product with the API. For example, to quickly see if a product has any metafields, a very quick but effective command-line effort would be:



Example API Call

With just a couple of keystrokes, the product is shown. Also, it has no metafield resources attached to it.

When setting up a product in inventory, the variant can be assigned a title as well as a price, SKU and inventory management policies. Shopify also provides three options that can be assigned to a variant. If a product has a colour, size and material, they are supported as variant options. These options can be anything, so there is a fair amount of flexibility in using them. A product with attributes like density, plasticity, taste, style or even chemical composition can be setup too. Assuming that all three options are used, Shopify allows a product to have up to 100 variants. This means a product with five colours, four sizes and three materials would require sixty variants (5 x 4 x 3 = 60). Each one counts as an SKU in Shopify for the selected pricing plan. It is not necessary to assign an SKU to each variant. You can also assign the same SKU to as many variants as you need. Shopify simply treats the SKU as an attribute with no special meaning. Using the Shopify AJAX API, it is very simple to use the handle of a product to get all of its attributes. If we open a store that has a version of the API code rendered as part of the theme, we can use the developer tools of our browser to inspect the details.



Getting Product Info using AJAX

An examination of the object representing a Shopify product shows us the structure of a product's options and how they are responsible for the attributes generated for each

variant. We can use this to completely customize the way Shopify renders products and variants. Many shops use the code that Shopify provides in order to render each option as a separate HTML select element, instead of one select element with each element separated with slashes. When a selection is made, a callback is triggered with the variant. That allows a shop to update things like pricing and availability. It is simple, reliable and it works. It can be relied on by developers to take front-end shop development to the next level beyond the basics.

If there are images uploaded for a product, it is possible to detect images that might match the SKU assigned to the variant or variant title. It is easy to make up some rules dictating how these images are coded, so that code can be used in the callback logic.

Summary of Developing Code with Products

Knowing how Shopify inventory works, with respect to variants and options, is crucial to building shops with advanced capabilities. To take a shop to a level where customers can easily add the product or products of their choice to the cart and buy it without too much effort or guesswork is essential. The first thing to keep in mind is that the product has options. There can be three. They have simple names like Colour, Size or Title. The product object you get from Liquid allows an inspection of the options array to see which ones are present. Every product can have different options, so it makes sense to always have some code that checks the ones assigned to the product.

```
// save product into a Javascript variable we can inspect.
var product = {{ product | json}};
console.log("Product %o has Options %o", product, product.options);
```

When rendering with the Shopify *option_selectors* code, there is a callback function that receives the selected variant. Logging that to the console shows the option values of the selected variant. A variant will always provide three attributes: option1, option2 and option3. If the value of the first product option was set to colour, the value of variant attribute option1 might be "red". At the same time, there is a variant attribute called options. Options is an array that would have an entry where the first slot would be occupied by the value "red" as well. So there are two ways to determine a variant's current options.

While the options are determined, the variant ID is also known. Therefore, it makes sense if a shop needs to present clickable images, buttons or swatches. All the elements are present to support code development for this. Hiding the usual dropdowns and rendering a clickable swatch that provides the variant options or even the ID can thus be used to

ensure that the correct item ends up in the cart. At some point, it is likely to become a common option for shops to present clickable images or swatches.

Chapter 7

The Shopify API

One of the most interesting features of Internet computing that has evolved since the 1990's has been the introduction and growth of service-oriented companies that work strictly using Internet protocols. YouTube, Twitter, Facebook, Shopify and multitudes of other companies that are run from brick and mortar headquarters, have relatively small employee headcounts but count millions and perhaps billions of users. Little of their success can be attributed to knocking from door-to-door or through TV advertising. Service-oriented companies leverage the Internet itself for their growth. One of the underlying reasons for their rapid growth, adoption and success is due the concept of the public Application Programming Interface (API).

What better way to introduce a service to the public than to allow them to build it, populate it and enjoy it by using their own labour and tools. YouTube, Facebook and Twitter would not exist without user generated content. But how do you ensure that everyone can contribute to your service without being overly technical or specialized? How do you accept a video from an iPhone or Android phone, an iPad or Galaxy Tablet, a Mac or a PC? The key is the use and promotion of an API. An API provides a simple mechanism that everyone can take advantage of.

Shopify released their API platform after some years of processing a steadily growing (or perhaps explosive) number of transactions. During this period, Shopify learned not only the intimate details of what happens when millions of dollars flow through Internet cash registers but also the huge number of possibilities and limitations that can be faced by a single codebase. Once the early adopters crossed the chasm and became profitable enterprises, there was a corresponding demand and need for introducing better order processing.

The API platform uses software architecture known as REpresentational State Transfer (REST). REST is accepted by a majority of the Internet community. The API provides support for merchants to create, read, update or delete all of the resources of their shop, like products. One current limitation is that you cannot programmatically create an order because of security concerns around credit card numbers. Almost all ecommerce companies offer some sort of API but there is clearly a difference in the degree of maturity and the amount of thought that has gone into some of them. The Shopify API is proving to be very helpful for merchants that need advanced capabilities. Whenever the general question is asked, "Can Shopify do such and such?", the right answer to the question is usually, "Yes, it can. You need to use the API for that custom feature."

Shopify has established a hosted ecommerce platform, transacting hundreds of millions of dollars per year for tens of thousands of merchants. If all of the features that merchants demanded were added to the core of Shopify, the platform would become unstable and prone to outages, breakdowns and a destroyed reputation for reliability. Developers would have to become familiar with a constantly changing system. By offering an API built around a core set of resources, Shopify has enabled an ecosystem of app developers to come together and create unique and necessary offerings, which make running an ecommerce shop on the Shopify platform easier for merchants. Developers need only learn the API and then they can apply their knowledge to any shop, for any merchant.

A good example of extending a shop for a merchant, is when fulfilling an order using the API and an app. Shopify has an option allowing a merchant to select from a few fulfillment companies. What happens when you check out that short list and you do not see an acceptable option? Your supply chain beginning from manufacturer DrubbleZook to warehouse Zingoblork, with the US Postal Service or Royal Mail, is simply not there. But you know every single order can be sent to an app. You know you can select up to 250 orders at once and send them all to an app. So surely an app can be built to automatically handle the fulfillments.

An app running in the cloud is listening for incoming orders, 24/7. When it gets an order, it robotically follows a set of instructions that ensures that the merchant is happy. The app takes the order apart and inspects it. The app knows where each line item is to be sent. It knows the ID of every variant and whether a discount code was used. It knows the credit card issuer. The app can take the order details and format it for Zingoblork and their special needs in a special XML format just for them. It is 2012 and the Zingoblork warehouse system runs off of any of the following data exchange mechanisms:

- A Microsoft DOS server connected to the Internet by FTP. They only accept CSV files via FTP.
- A Microsoft NT Server connected to the Internet by sFTP. They only accept CSV files via sFTP.
- Email. The company can only deal with email. They have been around forty years and it's all email, all the time.
- HTTP POST. A modern miracle! A warehouse fulfillment company that actually has modern infrastructure!
- EDI which we won't even bother to describe, but suffice it to say, it is the Chevy Vega of exchanges.
- SOAP protocol for those who would rather wash the car with a toothbrush.

The app receives fresh orders thanks to the API and WebHooks. It processes them and sends them off to the fulfillment company. Once the fulfillment company has accepted the order(s) and sent them off to their final destination as a delivery, the customer needs

to know. Some companies will collect all the orders they process for a shop and create a daily manifest of tracking codes assigned to the orders. Then, they will place those in a holding pen accessible only by a special FTP account. Others will send the tracking numbers to the merchant via email, completely destroying the merchant's email inbox and sanity in the process. The best fulfillment companies will send the tracking numbers and order ID directly back to the app. This allows the app to use the tracking number to automatically create a fulfillment using the API. Shopify automatically detects the creation of a fulfillment and sends the appropriate email. This is wonderful since the app and the API together can close the loop automatically. Once fulfilled, the app closes the order and the merchant has not lifted a finger.

Without an API, it would be difficult to offer this level of customization to a merchant. Another interesting use of Shopify is when a merchant sells products on a consignment basis for external vendors. If a merchant accepts 1000 widgets to sell for $20 each, the vendor wants to know if the shop made $20, $400 or even $4000 in sales. If the merchant accepts 1000 widgets to sell on behalf of 20 or 30 different vendors, his merchant life will almost certainly be miserable unless the API is used to facilitate the necessary reporting. An app records each and every line item sold as a Sale that is assigned to the vendor. Every product has a selling price and a cost price. The difference between those is the profit that will be split between the merchant and the vendor(s). The deal with the vendors is to give them an 80/20 or perhaps 60/40 split on profits. That percentage can be assigned to each sale so that different deals can be made depending on the day of the week, the colour of the sky or the popularity of the product. This is an app that can be plugged into a shop today. None of this would be possible without the API.

The presence and functionality of the Shopify API is a crucial decision to consider before going into business with the Shopify platform. If a merchant knows that their annual fixed costs are $1000 and that their theme will cost $5000, the API is the only other option that deserves attention. Adding an app can add to the cost of running a shop, but can also save hundreds of hours in manual labour. While paying for the development of a custom app can cost a few thousand dollars, it can result in measurable sales growth. Many companies are looking for a reliable hosted ecommerce platform, as well as partners to help bring their ecommerce vision to life. The presence and capabilities of the Shopify API is enough incentive to choose Shopify over competing platforms.

Chapter 8

How to Handle WebHook Requests

This is a big important topic. If Shopify doesn't receive a 200 OK status response within 10 seconds of sending a WebHook, Shopify will assume that there is a problem and will mark it as a failed attempt. Repeated attempts by Shopify will be made for up to 48 hours. Too many failures and the WebHook is deleted. A common cause for failure is by an app that performs too much complex processing when it receives requests, before responding. When processing WebHooks, a quick 200 OK response that acknowledges receipt of the data is essential.

Here's some pseudocode to demonstrate what I mean:

```
def handle_webhook request
  # Note that the process_data call could take a while, e.g. 2-6 seconds
  process_data request.data

  status 200
end
```

To make sure that you don't take too long before responding, you should defer processing the WebHook data until *after* the response has been sent. Delayed or background jobs are perfect for this.

Here's how your code could look:

```
def handle_webhook request
  # This should take no time, so the overall response is quick
  schedule_processing request.data

  status 200
end
```

Even if you're only doing a small amount of processing, there are other factors to take into account. On-demand cloud services, such as Heroku or PHPFog, will need to spin up a new processing node to handle sporadic requests. This is because they often put apps to sleep when they are not busy. Though this can only take several seconds, if your app is only spending five seconds processing data, it will still *fail* if the underlying server took five seconds to start up.

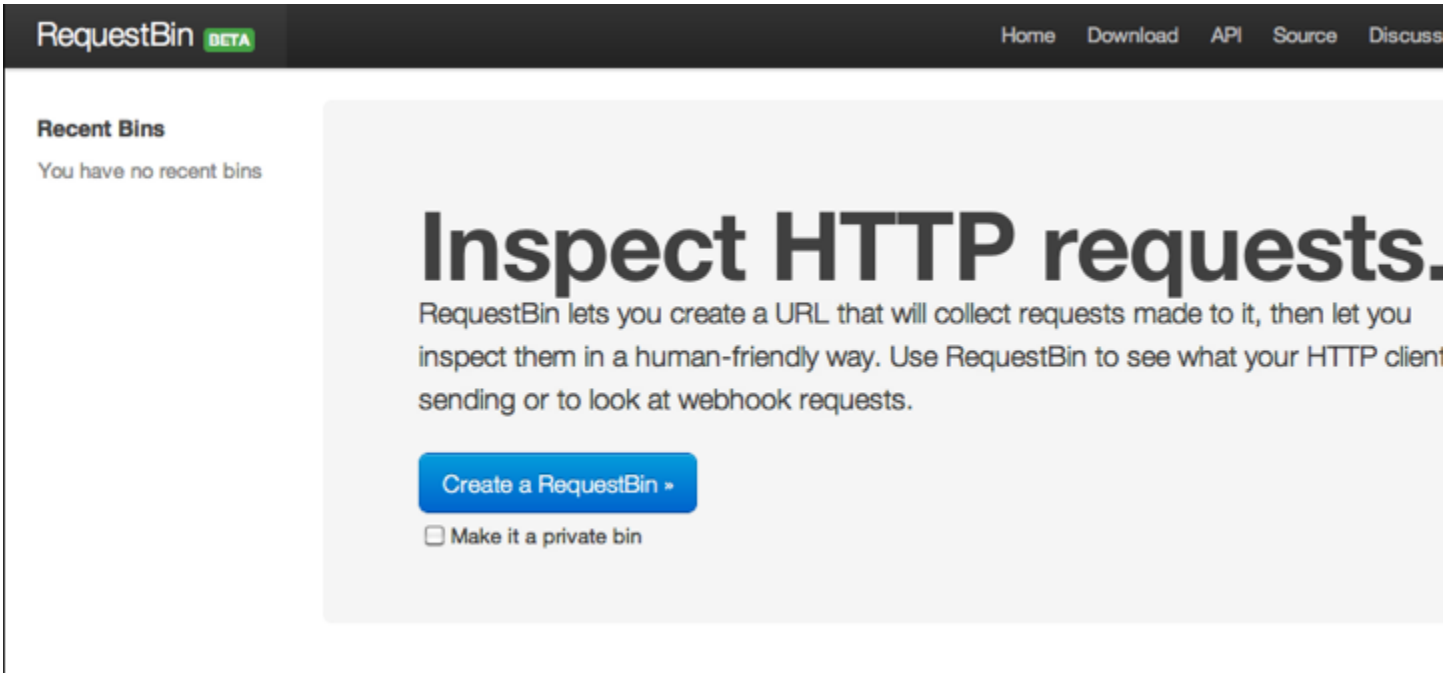
The Interesting World of WebHooks

Shopify does a fine job of introducing and explaining WebHooks on the wiki and there are some pretty nifty use cases provided. The *best practices* are essential readings and should be thoroughly understood, in order to get the most out of using WebHooks. There are all sorts of interesting issues with WebHooks.

Shopify WebHooks Documentation

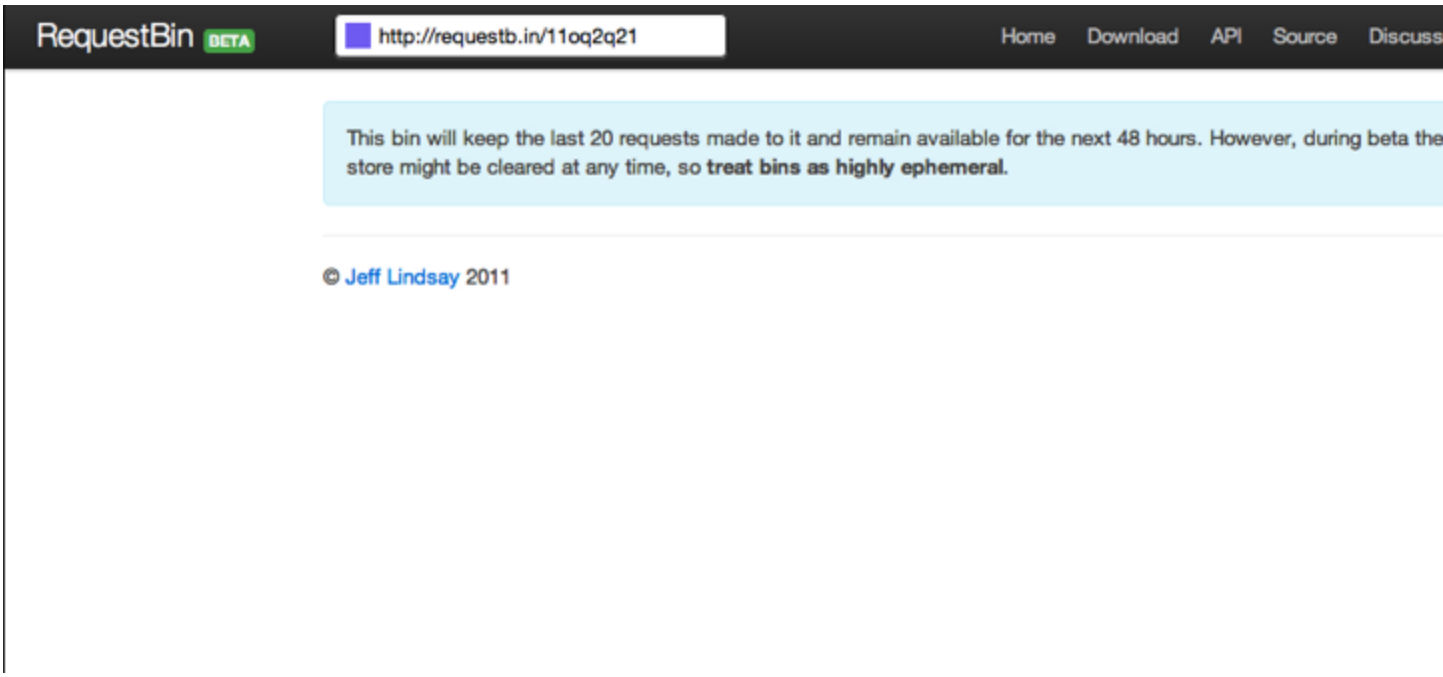
When you are dealing with Shopify WebHooks, you are in the Email & Preferences section of a shop. You can setup a WebHook using the web interface. Pick the type of WebHook you want to use and provide a URL that will be receiving the data. For those without an app to hook up to a shop, there are some nifty WebHook testing sites available which are free.

Let's take one quick example and use RequestBin. The first thing to do is create a WebHook listener at the Request Bin website.



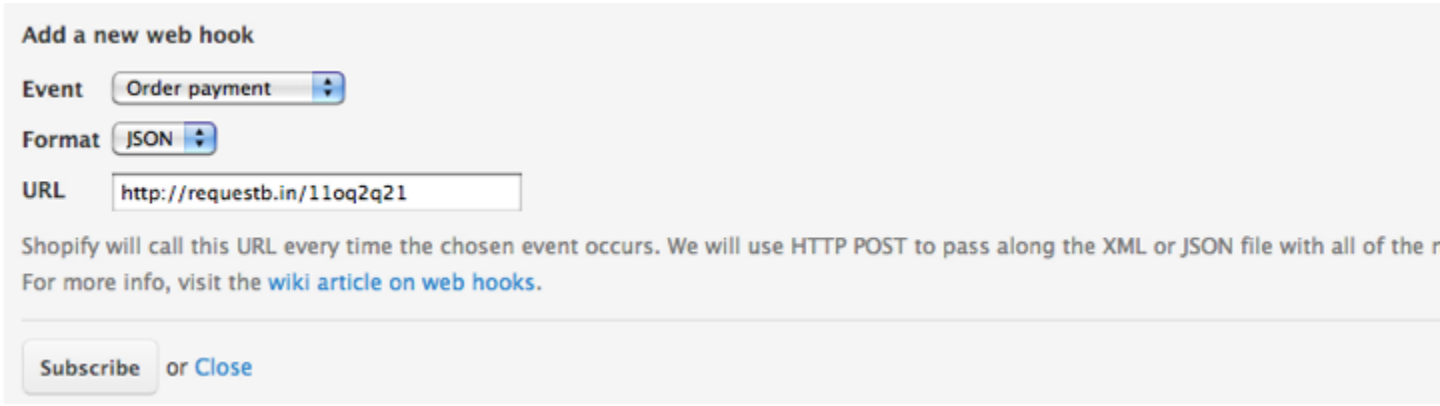
Create a new RequestBin for your WebHook

Pressing the *Create a RequestBin* button creates a new WebHook listener. A URL is generated, which can be used for testing. Note that one can also make this test private so that you are the only one who can see the WebHook results being sent to the RequestBin.



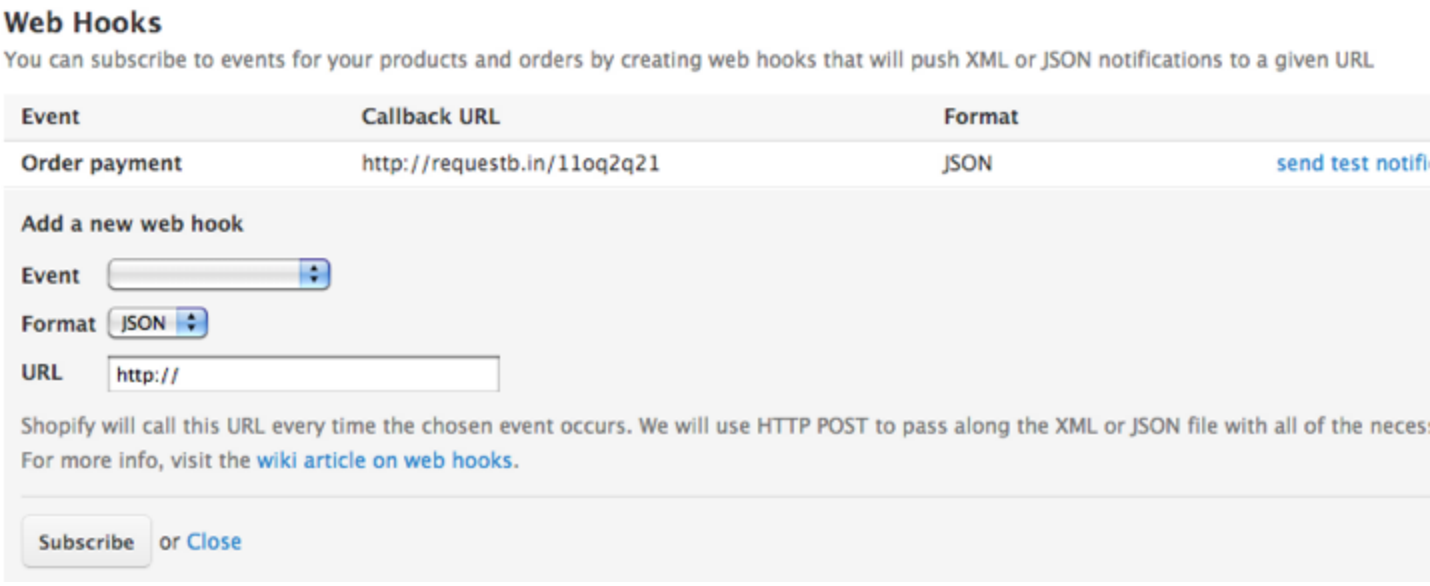
Newly Created RequestBin

The RequestBin listener is the URL that can be copied into the Shopify WebHook creation form at the shop's Email & Preferences administration section. For example, under http://www.postbin.org/155tzv2, the code 155tzv2 was generated specifically for this test. Using the WebHook Create form, one can pick what type of WebHook to test and specify where to send it.



WebHook Created in Shopify Email and Preferences

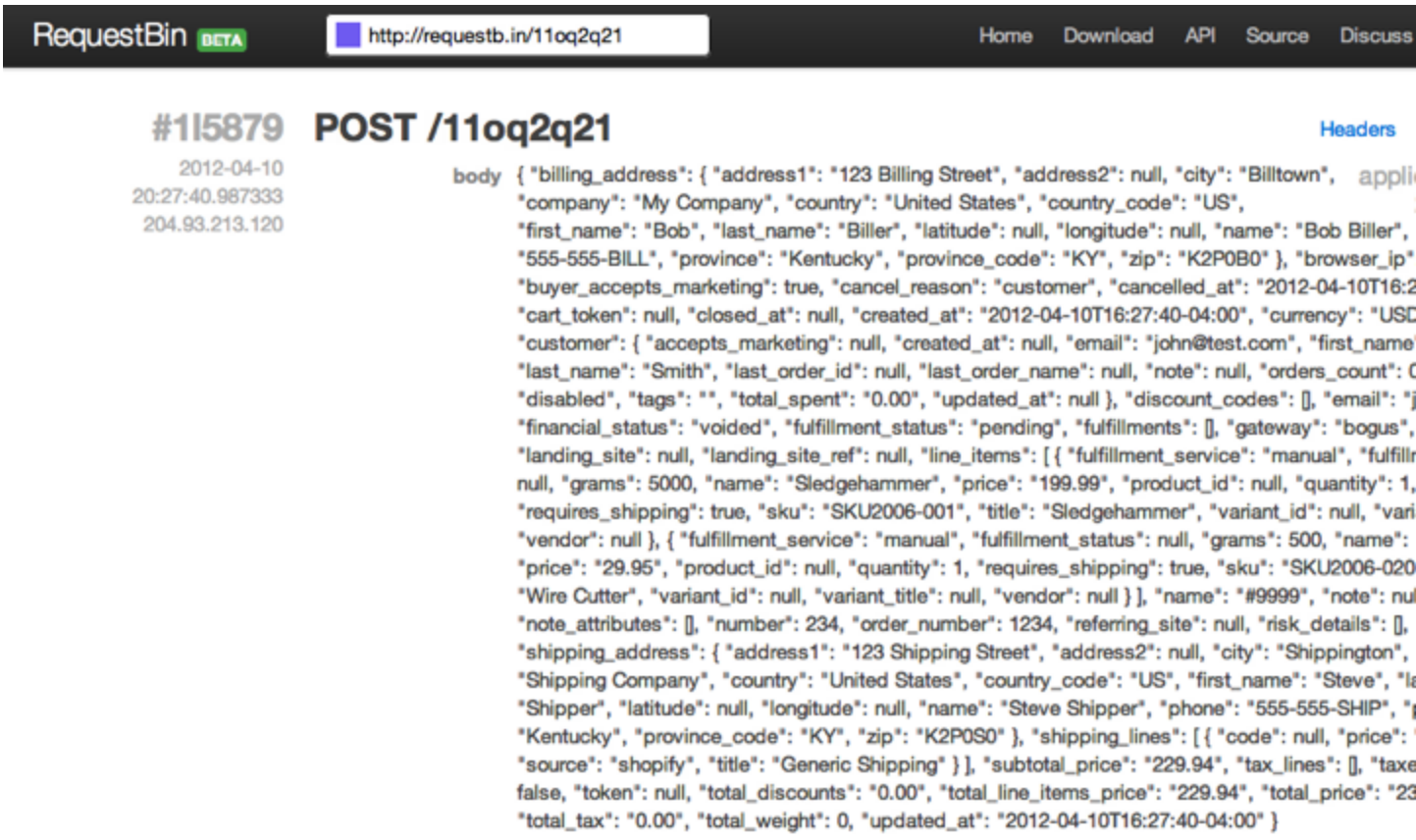
When the WebHook has been created, you can send it to the RequestBin service at any time by clicking on the send test notification link and standing by for a confirmation that it was indeed sent.



Testable WebHook

The links to delete and test a WebHook are beside each other. Exercise some caution when clicking in this neighbourhood! It is easy to accidentally press the trashcan icon and remove a WebHook that should *never be removed*. Oops! It can only take seconds of carelessness to decouple a shop from a crucial app.

Sending a test is easy, and the result should be immediately available in RequestBin. The example shows a test order in JSON format.



WebHook Results

Looking closely at the sample order data which is in JSON format, there is a complete order to work with. The loop is closed on the concept of creating, testing and capturing WebHooks. The listener at RequestBin is a surrogate for a real one that would exist in an app but it can prove useful as a development tool.

For the discussion of WebHook testing, note that the sample data from Shopify is good for testing connectivity but not for testing an app. Upon close examination, the data shows a lot of the fields are empty or null. It would be nice to be able send real data to an app, without the hassle of actually using the shop and booking orders. A real-life scenario might be to test and

- Ensure that the WebHook order data actually came from Shopify and that the source shop is correctly identified.
- Ensure that there is not already an identical order, as it makes no sense to process a PAID order two or more times.
- Parse out the credit card, the shipping charges and the discount codes, if any.
- Parse out any product customization data in the cart note or cart attributes.

This small list introduces some issues that may not be obvious to new developers to the Shopify platform. Addressing each one will provide some useful insight into how to structure an app in order to deal with WebHooks from Shopify.

WebHook Validation

When setting up an app in the Shopify Partner web application, the key attributes generated by Shopify is the authentication data. Every app has an API key to identify it, as well as a shared secret. These are unique tokens and are critical in providing a secure exchange of data between apps and Shopify. In the case of validating the source of a WebHook, both Shopify and the app can use the shared secret. When you use the API to install a WebHook into a shop, Shopify knows the identity of the app that is requesting the creation of a WebHook. Shopify uses the shared secret that is associated with the app and makes it part of the WebHook itself. Before Shopify sends off an app-created WebHook, it will use the shared secret to compute a Hash of the WebHook payload and embed this in the WebHook's HTTP headers. Any WebHook from Shopify that has been set up with the API will have HTTP_X_SHOPIFY_HMAC_SHA256 in the HTTP request header. Since the app has access to the shared secret, it can use that to decode the incoming request. The Shopify team provides some working code for this.

```
SHARED_SECRET = "f10ad00c67b72550854a34dc166d9161"
def verify_webhook(data, hmac_header)
  digest = OpenSSL::Digest::Digest.new('sha256')
  hmac = Base64.encode64(OpenSSL::HMAC.digest(digest, SHARED_SECRET, data)).strip

  hmac == hmac_header
end
```

The app calculates a value that only it could know. The header provides a value from Shopify. If the two computed values match, it is assured that the WebHook is valid and came from Shopify. This is why it is important to ensure that the shared secret is not widely distributed on the Internet.

Looking out for Duplicate WebHooks

As explained in the WebHook best practices guide, Shopify will send out a WebHook and then wait for up to ten seconds for a response status. If that response is not received, the WebHook will be re-sent. This continues until a 200 OK status is received, ensuring that even if a network connection is down, Shopify will keep trying to get the WebHook to the app. The initial interval between re-tries of ten seconds is not practical for a large number of re-tries. Thus, the time interval between requests is constantly extended, until the WebHook is re-tried every hour or more. If nothing changes within 48 hours, an email is sent to the app owner warning them that their WebHook receiver is not working and will be deleted from the shop. Though this has harsh consequences, mitigated by the fact that the email should be sufficient to alert the app owner to the existence of a problem.

Assuming that all is well with the network and the app is receiving WebHooks, it is entirely possible that an app will receive the odd duplicate WebHook. Shopify is originating WebHook requests in a Data Centre. There are certainly going to be hops through various Internet routers as the WebHook traverses various links to your app. If you use the *tracert* command to examine these hops, you can see the latency or time it takes for each hop. Sometimes, an overloaded router in the path will take a long time to forward the needed data. This extends the time it takes for a complete exchange to happen between Shopify and an app. Sometimes, the app itself can take a long time to process and respond to a WebHook. In any case, a duplicate is possible and the app might have a problem, unless it deals with the possibility of duplicates.

A simple way to deal with this might be to have the app record the ID of the incoming WebHook resource. For example, on a paid order, if the app knows a priori, that order 123456 is already processed, any further orders detected with the ID 123456 can be ignored. Turns out, in practice this is not a robust solution. A busy shop can inundate an app with orders or paid WebHooks and at any moment no matter how efficient the app is at processing those incoming WebHooks. There can be enough latency to ensure that Shopify sends a duplicate order out.

A robust way to handle WebHooks is to put in place a Message Queue (MQ) service. All incoming WebHooks should be directed to a message queue. Once an incoming WebHook is successfully added to the queue, the app simply returns the 200 Status OK response to Shopify and the WebHook is completed. If that process is subject to network latency or other issues, it makes no difference because the queue welcomes any and all WebHooks, duplicates or not.

The app has a queue worker process which is used to *pop* WebHooks from the queue, for processing. That way, there is no longer a concern over processing speed and the app can do all of the sophisticated processing it needs to do.

Also, it is possible to be certain whether a WebHook has been processed already or not. Duplicated WebHooks are best taken care of with this kind of architecture.

Parsing WebHooks

Shopify provides WebHook requests as XML or JSON. Most scripting languages have XML and JSON parsers to make request processing easy. With the advent of NoSQL databases, storing JSON as documents in CouchDB or MongoDB is possible. It is also easy to use Node.js on the server to process WebHooks where JSON is a natural fit. Since the logic of searching a request for a specific field is the same for both formats, it is up to the app developer to choose the format they prefer.

Cart Customization

Without a doubt, one of the most useful but also more difficult aspects of front-end Shopify development, is in the use of the cart note and cart attribute features. They are the only way a shop can collect non-standard information from a customer. Any monogrammed handbags, initialed wedding invitations or engraved glass baby bottles will have used the cart note or cart attributes to capture and pass this information through the order process. Since a cart note or cart attribute is just a key and value, the value is restricted to a string. A string could be a simple name like "Bob" or it could conceivably be a sophisticated Javascript Object like

```
[{"name": "Joe Blow", "age" : "29", "dob": "1958-01-29"},
 {"name": "Henrietta Booger", "age" : "19", "dob": "1978-05-21"},
 ...
 {"name": "Psilly Psilon", "age" : "39", "dob": "1968-06-03"}]
```

In the app, when parsing cart attributes with JSON, it is possible to reconstitute the original object embedded there. This pattern of augmenting orders with cart attributes and passing them to apps by WebHook for processing, has made it possible for the Shopify platform to deliver a wide variety of sites with unique features.

Chapter 9

Command Line Shopify

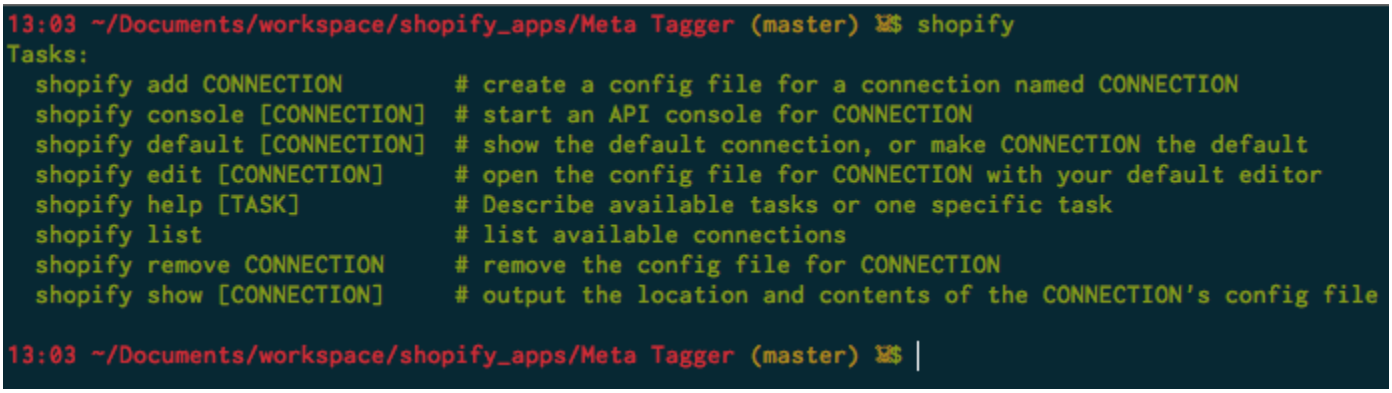
Development of apps requires the ability to fire off requests to a shop and process the responses. WebHooks are a source of incoming data from a shop, but there are many interesting possibilities available (using a console terminal) to access the API at the command line.

Firing off a request for a resource to a shop, uses JSON or XML and it occurs with a standard HTTP verb like GET. To send a GET request to a shop with authentication, is an involved process using the command line. A curl command to send off a request to a shop for a product or collection resource is somewhat complex. Who memorizes curl commands? It is tedious. There are better ways to do this.

Shopify has a super application which they bundle with their Ruby gem, for the Shopify API. Before Rails merged with Merb, the Merbists advanced the concept of command line interface (CLI) development for Ruby, with the introduction of Thor. Thor is Ruby code that generates command-line interfaces and is a nice replacement for rake tasks. The new command is simple:

```
$ shopify
```

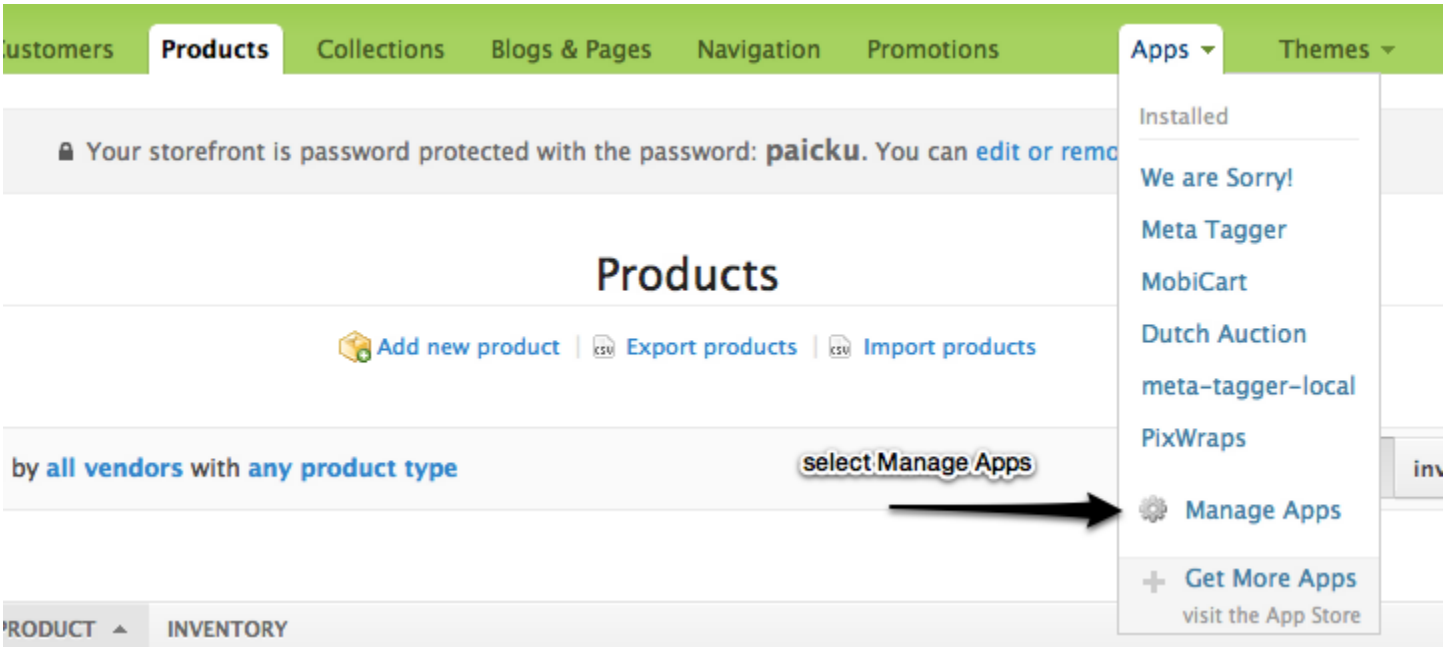
Start a terminal session on your computer and you will be able to test this command out. You want to know the console and also want to have the Ruby scripting language and Shopify API gem installed.



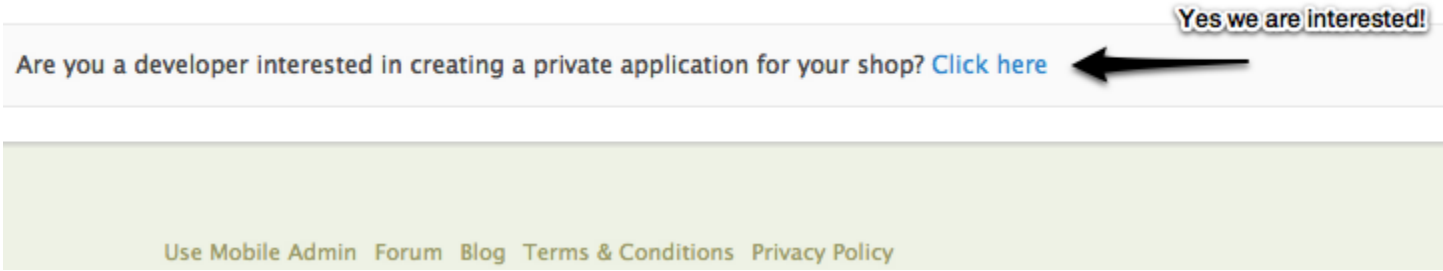
Available Options for the shopify Command Line Interface

The Shopify Command Line Console

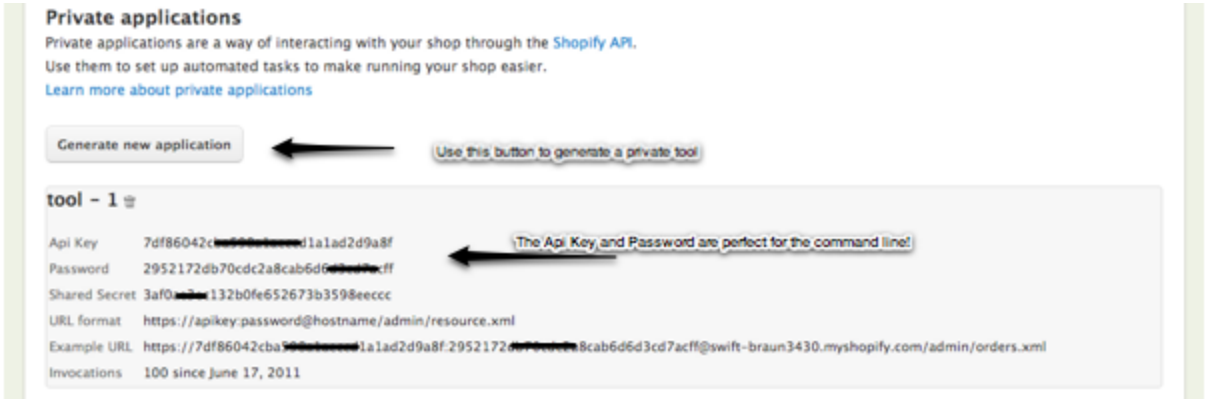
There are just a few options listed. In order to really cook up some interesting examples, use the configure option that comes with the command. The arguments required to configure a Shopify session are the shop's API key and a password. To get those values, use the shop itself. For some developers, this will mean using their development shop. For others, their clients have provided access to their shop. So, a private app can be created. Or they have created the Private app for the developer and have passed on the credentials. The following screen shots show the exact sequence.



Menu Options to Setup Shop Access for the Command Line



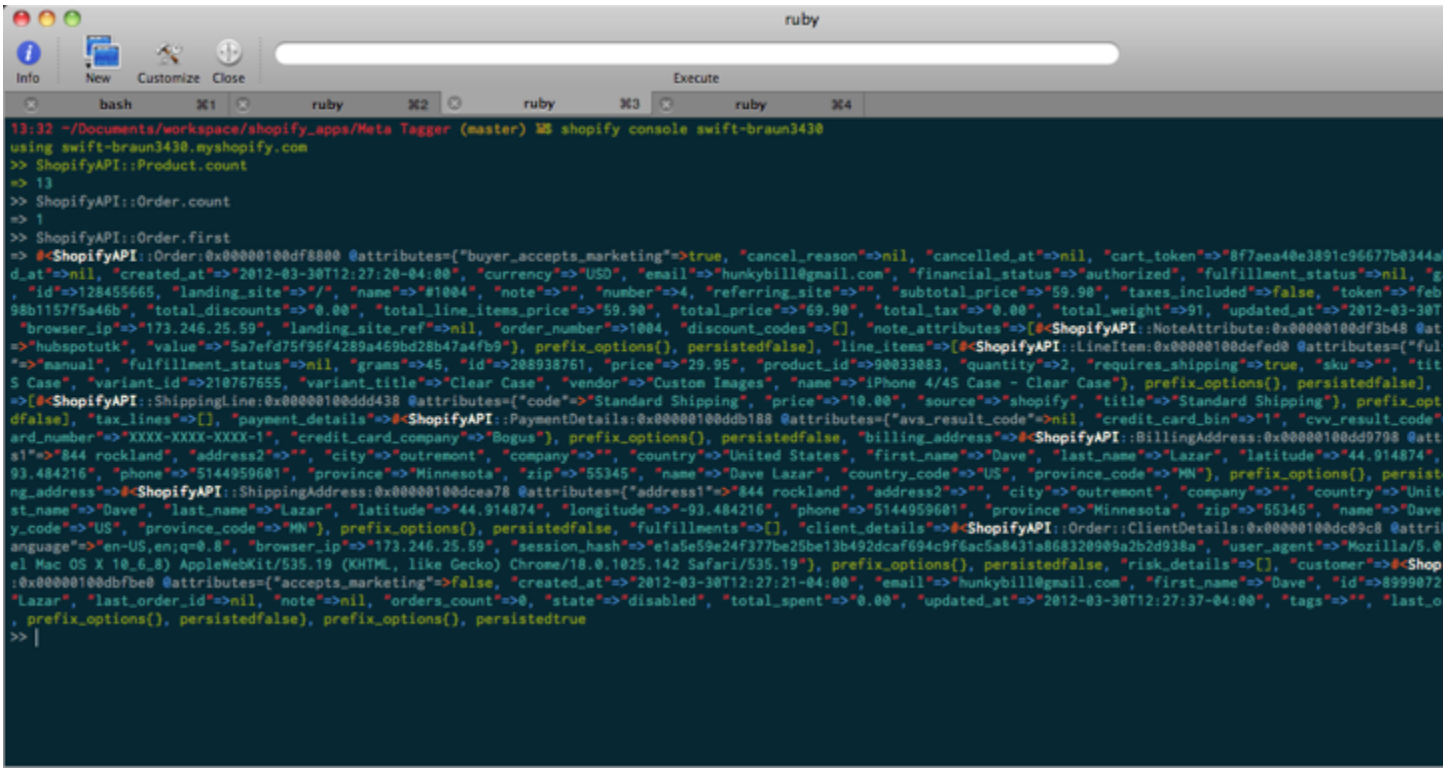
Setting Up Access for a Private Application



Final Step Reveals the Needed Access Credentials

When examining the credentials provided for a Private app tool, use the API key and password along with the shop name. Once the configuration is completed, accessing the shop using a console is easy.

The following figure is a console session for a development shop that shows a query for the shop's count of products, orders and the details of a single order.



Output of Calling a Shop Using the API

Quickly testing out potential code snippets without incurring a lot of overhead, is a huge benefit. API code can be tested with the fewest possible keystrokes and minimum effort with this handy console. It is possible to access metafield resources by providing an ID for the resource. The syntax of a call with Active Resource can be a challenge to keep in your head. With the console, it's possible to try out command syntax to see what the exact

response will be. With the console, it is easier to try a few different call patterns before stumbling upon the desired one.

You can add connections to as many shops as needed and you can list them all. When working on a client shop, one of the first things to do is to use the shop admin interface and the app tab to create a private tool. The console is super useful to quickly test out any custom queries that are build into an app for the merchant.

The Shopify Theme Command Line Console

A second useful command line tool that Shopify provides is not in the shopify_api gem but a separate gem called shopify_theme. With this gem installed on a system, you can use the command theme to develop the shop theme. Theme offers a simple workflow. The first thing to do with a shop development project is to create a directory to hold all the files that make up the merchant's theme.

```
$ mkdir fizz-buzz.com
$ cd fizz-buzz.com
$ theme configuration
```

```
13:46 ~/Documents/workspace/shopify_apps/Meta Tagger (master) 🖼$ shopify
Tasks:
  shopify add CONNECTION         # create a config file for a connection named CONNECTION
  shopify console [CONNECTION]   # start an API console for CONNECTION
  shopify default [CONNECTION]   # show the default connection, or make CONNECTION the default
  shopify edit [CONNECTION]      # open the config file for CONNECTION with your default editor
  shopify help [TASK]            # Describe available tasks or one specific task
  shopify list                   # list available connections
  shopify remove CONNECTION      # remove the config file for CONNECTION
  shopify show [CONNECTION]      # output the location and contents of the CONNECTION's config file

13:46 ~/Documents/workspace/shopify_apps/Meta Tagger (master) 🖼$ theme
Tasks:
  theme configure API_KEY PASSWORD STORE  # generate a config file for the store to connect to
  theme download FILE                     # download the shops current theme assets
  theme help [TASK]                       # Describe available tasks or one specific task
  theme remove FILE                       # remove theme asset
  theme replace FILE                      # completely replace shop theme assets with local theme asse
  theme upload FILE                       # upload all theme assets to shop
  theme watch                             # upload and delete individual theme assets as they change,

13:46 ~/Documents/workspace/shopify_apps/Meta Tagger (master) 🖼$
```

Output of the Theme Command Line Console

The listing of available options and arguments to the *theme* commmand is slightly different from the *shopify* command. It does, however, use the same API key and password to access shop resources. The configuration is saved as a file config.yml. The first useful step, once configured, is to download the complete theme.

```
$ theme download
```

Once the assets are downloaded, set up the files under version control. Make a local repository with git and store the client's code under version control for safety. Advanced developers might even add a git remote pointing at github, so that all the work is safely stored in a private repository there too.

```
$ git init
$ git add .
$ git commit -m 'initial commit of Shopify code for client XYZ'
```

With the code in git, it is a great time to start working on the theme. Use the *theme watch* command to *watch* the directory for any changes. As soon as a change is registered to a file, the theme watcher will transmit the changed file to the client site. Any changes can be stored in git. This ensures a smooth and hassle free development experience.

```
$ git commit -am 'Fixed that pesky jQuery error the client had from blindly copy and pasting some
```

This is a very productive and safe workflow. Because the code is under version control, it is easy to edit theme code using your favourite tools. In the future, if the client wants more work done, you just use the *theme download* command to grab new files or changes, even if they are initiated by the merchant or others. Git will keep track of all changes. You have a fighting chance when you are not the only person touching or editing a merchant site. Squashing some other designer's work or other person's files is rarely an acceptable practice!

A Developer's View of the Shopify Platform

David Lazar, P.Eng