David Lazar

A Developer's View of the Shopify Platform

A brief history of customizing Shopify

Table of Contents

Forward

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 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 taught me how to wear many various business hats. I picked up the Shopify bug and tried working with a boutique web agency. As five more years passed, I realized that I knew a lot about being 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 and my Corporation is pretty much a sad sack but expensive accounting exercise I leave to those who get a kick out that kind of thing. I am sure it's going to reach a decade in age as neglected as my vinyl collection. One needs a corporation though to run off invoices and interact with clients in foreign nations, so I go along with that. Dividends anyone?

A beautiful autumn day in Montreal and Shopify came for a visit with their CTO Cody, Joey Devilla on the Accordion and Edward 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 with a lot of experience working on Shopify, but not working 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 the gamut possible responses over time. My all time favorite is still Plonk as that one word continues to tickle my funny bone. If a couple of thousand Hunkybill posts survive on some hard drive in some data center for a few more years that is one thing, but I was pretty sure I wanted to create something a little more structered and formal.

Acknowledgements

First an foremost I want to say a big thank you to Shopify for supporting this effort through the Shopify fund. Over the years I expended energy and time recommending Shopify to all comers but at the same time, I can see that I was also critical and not always a perfect gentleman in my writing. Instead of ignoring me and my pitch however, Shopify chose to sweep aside some of the lower points from the past. Taking the high road and 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 do 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 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 plain wrong. I mean no harm to anyone and I am trying hard to avoid controversy these days.

I also with to thank Tobi 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 the hospitality when I trucked on down the 417 to Ottawa for a visit to the new office. 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?

Everyone that is interested in Shopify…

Why?

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 probably teach you at least one thing you probably did not know.

If you want to develop Apps for Shopify, or build a theme for someone, this book may 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 workaday world of a professional Shopify coder.

About the Author

My engineering career started professionally in 1990 at about the same 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 transferring 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 is it today (and ironically 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 running on hugely energy inefficient Compaq Proliant servers. Other quirky technologies like Cold Fusion were often used too. Many friends and collegues 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. You have to interface to an EDI system with an AS/400 legacy database and a transactional web site? 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 for 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 E-Commerce for the masses with OSCommerce. There was very little Ruby or Python code for these endeavors 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), and dialog boxes, and toolbars and most of the widgets that we take for granted today but that were rare indeed back then. Eventually that code matured into it's 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 enough so that we could see realized 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 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 e-commerce to my clients, without having to hack PHP OSCommerce again. If you have never examined OSCommerce code, you are somewhat lucky. It was at best a spaghetti western in terms of code. Plus 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 failed? 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/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 that is long forgotten and lost due to neglect. There is nothing quite like finding out the database has been failing to record certain entries for a few weeks due to a memory corruption bug.

Shopify to the rescue! Hosted E-Commerce! No more headaches! Simple templating with Liquid, ability to do any Javascript I want! That is the pattern that works for me. Should work for almost anyone.

Chapter 1
The Shopify Platform

Since it's release to the public as a hosted e-commerce platform, Shopify has evolved through the adoption of new functionalities, the introduction of new features, as well as the refinement of the interface presented to Shop owners. As much as we all know competition drives companies to improve their products and services, in the niche domain of hosted e-commerce there are enough platforms to choose from that the decision to use one does require some careful study, thought and analysis. My experience over the years as a Shopify developer has informed me on how best to use the platform and when not to shut the door on possibilities.

For the sake of simplicity and consistency there are some definitions that will help in understanding the perspective of this book. The most important definition is the one for Shopify App. Shopify has an API and it allows external computer programs to interact with a shop according to certain rules. 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 right authentication token it will work. A Shopify App is an externally hosted website that has been setup to connect to a Shop. Shopify merchants can plugin external Apps to their shops to provide additional functionality. A Shopify App is like an Android or iPhone App but it is usually found in the Shopify App store. The process of installing an App for a merchant usually involves approving the App for installation and sometimes approving a charge to use the App too. Some Apps are free, others have a monthly recurring fee and some offer a one-time fee as well. The sign-up and installation for Shopify Apps is very much like any other App installation procedure. It is entirely possible for a merchant to develop or pay to have developed an App 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 e-commerce as 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 they are entirely outside the scope of Shopify's data center and Shopify's technical support.

The diagram presented here shows the relationships three specific types of people will have with Shopify, namely the customer, the merchant and the Shopify App developer.
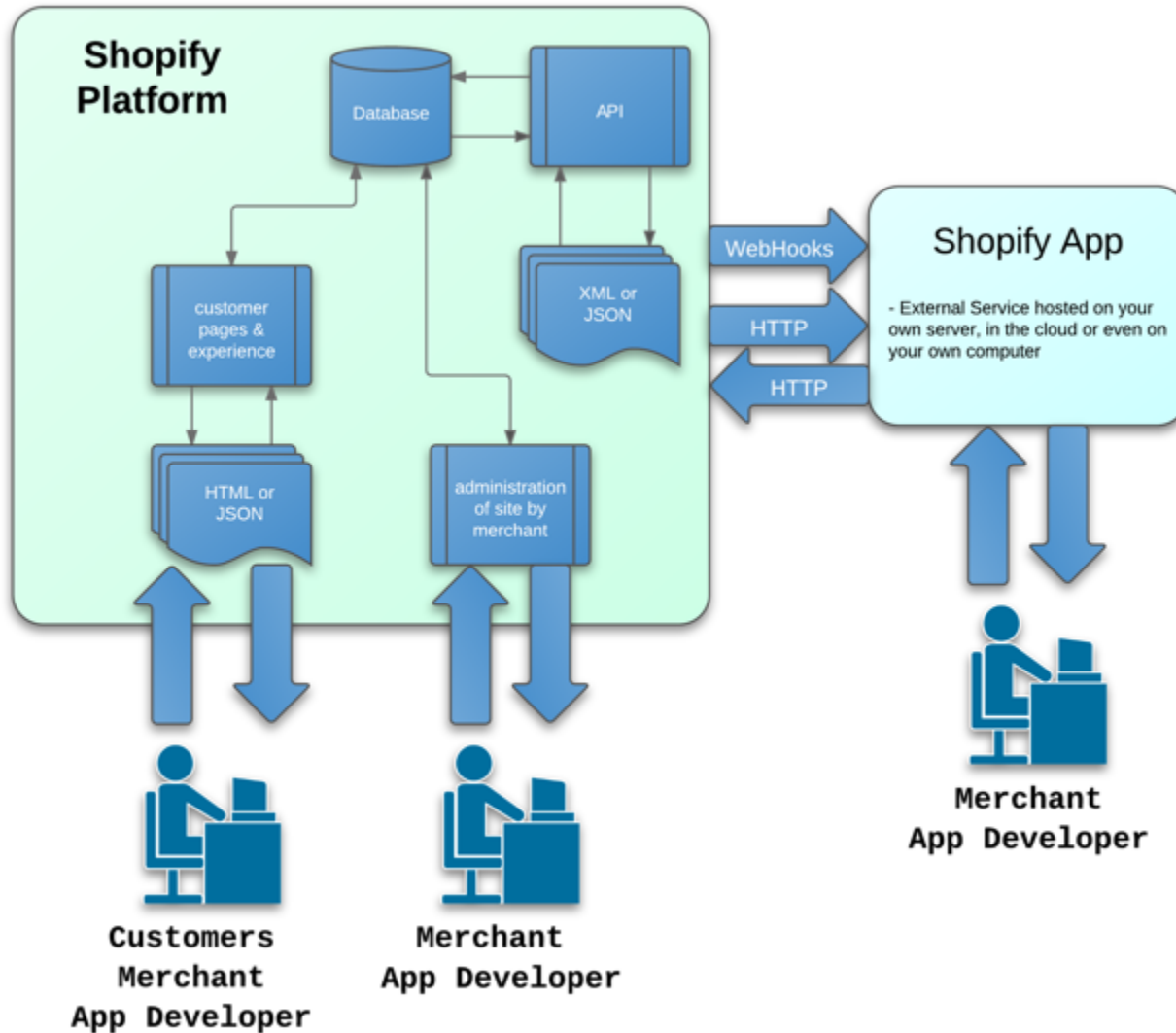
The Shopify Platform and Users

A customer will typically only interact with a shop. A shop may in fact interact with Shopify using the Javascript API to do Ajax. Both the merchant and the App developer also interact with the shop to ensure it is rendering properly. The merchant runs the shop and deals with orders. Merchants may also login to 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 in to the App to ensure operations are optimal. It is possible for both the merchant and the app developer as well as the customer to utilize an App that presents a public facing web presence. For example, the customers could login at *someapp.fizzbuzz.com* to check on their order status. The merchant might very well login at *someapp.fizzbuzz.com/admin/* to deal with setting up special aspects of their orders. Finally, all shop WebHooks and perhaps some internal link added to the shop 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. Originallly 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 amount of functionality made it possible to open up a shop quickly 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 changes affected merchants. The administration interface functions as an example of a well thought out design that relies on incremental change as opposed to radical make-overs. Considering the backlash a lot of web applications generate by changing their look and feel Shopify has done a great job of delivering improvements while maintaining consistentecy. Merchants have to invest in learning Shopify and 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 not intuitive or awkward but it was not clear how tags worked to improve a shop for customers. Providing some 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. New merchants to Shopify do not have the luxury of seeing how Shopify has evolved but that does not mean the history is not evident to merchants with longer term accounts.

Liquid Templating

Web application developers have adapted 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 where 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 it keeps programmers, designers and integrators all working on the same code. This usually resulted in only programmers being able to perform all those functions. A great innovation was to separate the two camps, keeping programming the responsibility of only the programmer, and letting designers look after the views. The introduction of templating for web applications meant a template could render the look when the applications scripting provided the data. That is a pretty good pattern for systems where the end user is also the owner of the data and most Wordpress sites function this way.

With a hosted platform like Shopify there is simply no chance that a merchant can directly script their shop due to the 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 the shop's data, while providing sophistication in terms of rendering that data nicely. The templating project was created by Tobias Luttke and is called Liquid. It's an 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 secret database passwords or 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 also has been setup by Shopify to provide a large set of very useful *filters* that can be applied to data.

The Liquid processing phase is crucial to solid merchant understanding of Shopify. A request from a customer on the Internet for a shop at http://someshop.myshopify.com arrives. Shopify locates the shop and loads up the *theme.liquid* file for the shop. By processing the *theme.liquid* code line by line, a series of assets get pulled into the theme. Examples would be the cascading style sheets (CSS) and Javascript needed by the shop. Each *theme.liquid* file has to specify where to render or draw the content of the current request which 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 process. Sometimes it is beneficial to take special aspects of a shop like a homepage carousel showing off products and keeping all that code together in a snippet. Snippets can be re-used on other parts of the site easier by simply including the snippet in the liquid files that need them. Once all of the code is assembled, processed and added together, all the Liquid templating is converted into HTML that gets output as a web page for the customer.

The implication for merchants and developers to note 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 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 get 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 make the changes using the shops theme settings link where these options are presented. There is a huge amount of creativity available with this system and it 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 when possible. It is very easy to setup loops within loops within loops to provide a clever shopping experience, but when analysed with an

algorithmic eye, these kinds of Liquid constructs are inefficient but they do work. Shopify will not usually interfere with how people use Liquid.
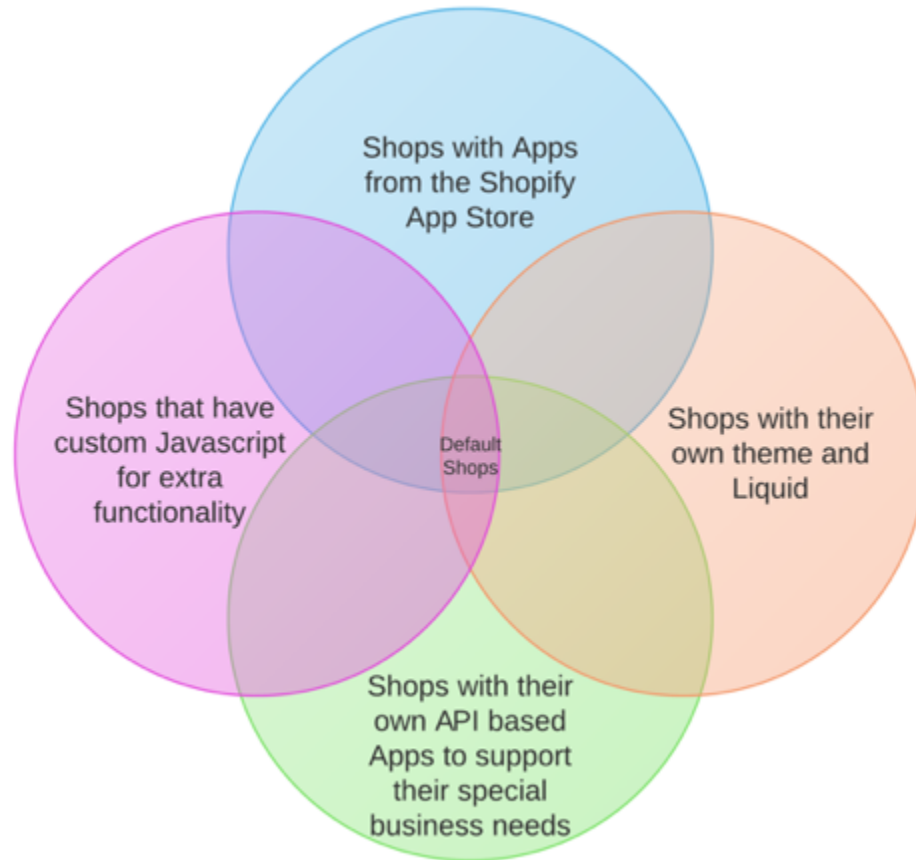
Possible Distributions of Shops and Their Uses of the Shopify Platform

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 web site 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 a website could replace a small part of itself with new information and not 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 then request the contents of the cart to get the total price of all the items in the cart. 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 the presentation of products. The most common is probably the *option selector* code 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 it's 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 to completely 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 actually 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 where they can be seen before checkout. The customer sees a form that collects the general location for delivery via a zip or postal code, and then uses the Shopify API to query the shipping services setup for the shop. For shops that offer expensive small items eg: jewelery, or ones that offer relatively cheap large items eg: stuffed animals, shipping can be a real problem to accommodate well. Freight is calculated 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 and that can really help converting carts into sales.

Shopify also provides some Javascript called *Customizr*. If a merchant has to record a name for 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 right through checkout with the order and are available in the *Order Notes* section of the order details. Cart attributes can be rendered in the order confirmation email to inform the customer of their purchase details. Further excellent opportunities exist to help a shop since orders can be sent to a custom App using WebHooks. A custom App can perform additional business logic based on the cart note and cart attributes as well. 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 of things shoppers will often want more than one of something, and the customization code has to manage quantity too. 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 a glass baby bottle gets etched with the name "Eddy" and a handbag gets the initials "G.B.H" or "S.N.F.U". Customizr can go a long way to making this possible without a major investment in custom Javascript code since Shopify provides it.

Shopify API

For merchants and developers the Shopify API is the most important and best feature of Shopify as an e-commerce 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 is described elsewhere in this book that can be used to 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 and editing those values followed by 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 and once created it generates an API token and a shared secret and can thus be installed in 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 thus 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 they can be created to be installed in thousands of shops, offering their provided functionality to all those 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 but it may still be opaque to merchants and developers exactly what this API can be used for. One example that justifies 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 some specific quirks that make it interesting and perhaps non-traditional for e-commerce. A work of art, say a painting is for sale as a product. Paintings can be organized into collections just as they might be in a gallery or museum. For example paintings for sale might be grouped as abstract or contemporary or mixed media or as a print. The challenge for a gallery is to present all the unique attributes of a piece of art. A painting belongs to an Artist who almost certainly has a biography. A painting has a history in that it has likely been toured around other

galleries and museums. Each such exhibition adds some value to the story of the painting and has to be told. The history, artist information and other meta data is best captured as resources attached to the painting, not as something you just stuff inside the paintings description. Using the API it is easy to capture this meta data 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 without it being tightly coupled to the product description. Using an App and the API it is possible to capture all the unique details supporting a product 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 they can be presented in the shop's theme as Liquid. An App has an option to add an *App Proxy* that serves as a *special page* in a shop. A gallery shop can navigate clients to the special page and render all the custom resources. An example for the concept of an Artist 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 like their name. The App looks for a Liquid template it installed in the shop, and it reads that template since it could've been altered by the merchant or theme designer. The App Proxy then searches the App for the artist's information based on the provided name and once found, it replaces the liquid tags with the appropriate information. The App then sends back the Liquid template 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. There are endless possibilities and combinations that 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 we'll 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 that you installed on your computer. Vision provided a small inventory of snowboards for sale and the shop templates showing off the 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 and pages and collections could be hacked since 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 instead. As mentioned when you become a Shopify Partner you can create as many test shops as you need and you can assign as many themes as you want to them. You can also load an real inventory into a test shop so this provides a great way to develop a theme without the risk of borking a live production shop making sales. At the same time as the Partner program was initiated Shopify presented a Theme store where merchants can download free or paid themes. Themes that cost money usually come with features that might otherwise cost merchants money 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 probable that the first thing a new Shopify merchant will invest in is the look of their shop, 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 too. The Shopify mantra of keeping the core Shopify feature set simple ensures there is a useful place for custom Apps that can deliver any specific needed 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 express their needs.

For Shopify developers the App store presents their App to all the 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 setup one-time charges, recurring monthly charges and to not worry about payments. Shopify pays developers bi-weekly as of June 2012 making it very similar to working for a company.

Shopify Experts

As the years have passed and Shopify has grown and attracted many more merchants it is only natural that a number of merchants will have needs that exceed the typical features provided by Shopify. These merchants need to know that they can have their special processing needs taken care of without having to launch a complex search for talent. The Shopify Experts was launched to showcase talent working on the Shopify platform. Theme designers, App developers, photographers and even accountants can join and they get a chance to show off their portfolio and capabilities. The slots in the Experts showcase for accountants

and photographers show these skills are also very much in demand by merchants that want to present the best possible look for their products and services, and to be able to do accurate reports on just what is flowing through their online universe.

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 tools coders use day to day have not varied much. These days coding complex web applications on a laptop in a cafe is normal and having great code environments and tools is a welcome change. For those that want to try developing a Shopify App or build out a nice theme for a shop all you need is an inexpensive laptop with wireless Internet and the desire to learn the details of Internet computing.

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 and as I cannot touch type I can offer no wisdom of choice. I love Sublime Text 2 though. I merely get by with my horrid typing skills in the sense that I type at about the same speed as I think. Maybe I think slow and my typing matches that speed? I am not sure. I have never been tested. I am amazed when I see people writing code as they stare out the window at the coffee shop and I wish I could do that too. If I was hiring a coder I would judge their typing skills in the sense that if they cannot really type, then they cannot be a real coder. I am guessing that puts me pretty close to the bread line! At a recent Hadoop Hackathon I attended, the local administrative guru literally flew faster through command line and vim commands then I have ever witnessed before or since. Simply amazing the capacity some people have for that. A text editor has to *syntax highlight* Ruby, Javascript, Liquid, Haml and Sass and other languages, has to *autosave* all my code changes whenever I stop coding and switch tasks and it's pretty much a sure thing it will need to be able to accept scripts to speed up development.

The Web Browser

Obviously every developer needs a good web browser to work with, one that provides decent tools for examining the results of Shopify theme tweaks or interactions with web applications. All the major browsers these days are suitable however each brings certain quirks to the table. My favorite is currently Chrome. The developer tools are decent. Safari and Firefox offer decent alternatives to Chrome.

Versioning Your Work

Shopify will version templates as you change them. This built-in versioning system is not terribly useful for a team and certainly is not something you want to rely on for your only theme backup choice. I highly suggest learning a distributed version control system (DVCS) like *git*. Becoming even a beginner with the *git* system is a basic skill every developer should have that will payoff in spades. With git you can version *everything you work on*. Every line of code, every proposal, even binary work like images and assets that you might not ordinarily think of as something that should be under version control. Plus, almost all the coolest open source projects are available using git and there is a serious community of developers all working and sharing code with git.

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 and speed communications along and it beats managing large attachments in email. It has reasonable security most of the time too.

Skitch

Often I will take a screenshot and mark it up with Skitch and then share the resulting image with a client. It is fun to log in to your online accounting package, dig out the invoice you sent to the client 2 months ago, Skitch the timestamped entry showing they logged in and acknowledged the invoice, and fire that image off to them. The mumbled excuses and apologies from their embarassment as you call them out for skipping their obligatory payment for services rendered is a little salve or tonic along the way. Remarkably some clients are oblivious to their obligations and these are the ones to watch out for. The ones that agree to your best estimates to get them most of the way there, and when you finally deliver they come up with a new list of things they insist were there all along and agreed to. Best to keep notes, printed and dates copies of agreements and to watch out for constantly changing requests.

Instant Messaging

You have to use Skype, Adium, Pidgin, iChat or some other service to work with clients. Email does not cut it when you want to really rip through a work session with a client and bounce thoughts and ideas of off them. Screen sharing is one of the quickest ways to teach a client about what your App does, what the shiny buttons they can press do, and the easily overlooked luxuries you've provided them just because you can. Writing a manual for an App is fine too but that takes many hours and in the end, the second you finish that manual it is full of errors or at least erroneous screen shots, descriptions and information as web apps can evolve in near real-time, even after they have been "delivered". I often re-factor my code just because I feel like it (and certainly not because v0.9 is embarrassing, although it might be).

Terminal Mode or Command-Line Thinking

To make working on Shopify themes easier, Shopify has a command-line (CLI) utility available that can be installed on any computer with the Ruby scripting language installed on it. The utility provides a few basic commands that allow you to easily download an entire shop for editing. You can then immediately check the entire shop codebase into a version control system like git. The next step is to tell the computer to watch for any changes in the theme files. If a change is detected like adding a collection title to a template or a for loop is added to render some navigation links in a menu, you want those changes automatically sent to the shop. That way, you can simply refresh your browser and see the changes you 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 that is currently being rendered in the browser, meaning you can edit your Shopify theme or App, and simply switch focus to your browser to see the results without doing much more than a single keystroke combination. As a developer learning to use the command line with skill and knowing how the operating system utilities help is essential.

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 Shopify shop it will be imperative to be able to develop the application on your local machine. In the early days of web application development one had to ensure there was a web server like Apache available in addition to scripting languages like Ruby, Python and PHP. That extra level of hassle we do without since Ruby provides several extremely fast web servers that make local development a breeze. A great example is the *thin* webserver by Marc-Andre Courneyer. Other options exist like Pow from 37Signals, WebBrick, Mongrel, Nginx, etc. Testing a script out on a new concept or idea or running an entire App should not tied to a server on the Internet. Being able to develop localhost when offline is crucial.

Pre-Compiled CSS

A final tool for the Shopify developer that I think deserves more attention is the use of compiled CSS through the use of Less or Sass. Less can be compiled with Javascript and Sass is compiled with Ruby. The advantages are somewhat spectacular in my opinion. You can build a complex Shopify theme using these tools and gain a lot of very important flexibility. A client can change one value in their shop and the change propagates throughout the CSS easily when you use Less or Sass. Working stylesheets by hand is clearly the least efficient methodology especially when a developer does have a good understanding of how CSS works.

Chapter 3
Common Questions

There are plenty of reasons to choose a *hosted* platform like Shopify as the responsibility for the necessary implementation details of ensuring your shop's site is reachable by the public is purely Shopify's responsibility. By offering a platform that provides most of the basic features needed to run an e-commerce website Shopify has created a thriving community of shops that sells soup to nuts. It can be comforting to know that when an issue arises with respect to how to configure a shop that there are other people that have faced the same issue before and likely solved it.

How to Capture that Extra Information

One of the earliest and still most common questions is about how to capture custom information for products. There are thousands of shops that need to collect custom information. Glass baby bottles etched with a monogram, handbags with initials stitched into the leather, 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 seemingly a source of endless discussion.

With the introduction of cart attributes it became possible but not necessarily simple to pass extra data through checkout with the 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 attributes to fully define needed product customization for an order. A typical key might be the identification number of a product. The value that can be stored with the key is usually a string of text. The following code present a simple key and 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 to a value. Using Javascript it is possible to experiment by setting a cart attribute and then checking that it was set correctly. Web browsers all provide Javascript and 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 we can create complex data structures and save them as attributes! The third example I present shows this. The cart attribute for ID 44556677 has been assigned a string of JSON. This is extremely handy in a web application like Shopify. That 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 shoppers, it's possible to show these values in the appropriate line items along with the variants. The checked out order will display the same keys and values.

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

The previous script presented is a little confusing perhaps and some shop keepers might balk at having the customization information collected looking like that. It is possible to deal with this issue by approaching the issue with more sophistication. Instead of directly storing the customization data in cart attributes, they could be stored in a cookie, or in the web browsers built-in localStorage. During the creation and subsequent editing of a specific shopping session the custom code manages the customization information as JSON but before submitting the order to checkout the code translates the JSON to plain english that would be set as the values in the cart attributes. As an example, if my variant represented a Farm Animal, for $24.00, I could rewrite the attribute to be

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

That is a little more readable and could be interpreted by the Shop owner with little difficulty.

One important aspect of customization that should be addressed is that flexibility like this comes with a certain cost. While it does imply that you collect extra information for a Product, you can lose certain Shopify functionality when you use it. For example, if you have the need to customize a product with four options, Shopify only has three. So you 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 that does not matter, then it is obviously not a problem. I often tell people that if price changes are involved, they have no choice but to use the built in options to customize variants. This costs them SKU's and there is a limit of up to 100 customized variants. A virtually unlimited amount of further customization is possible, but it should be applied to options that have no inherent cost or affect on inventory management.

Image Switching

Shopify organizes a product by assigning it attributes like vendor, type and description. A product has no price per se but it does have variants, and each variant of which there has to be at least one has the price. A product also has images. If we think about this knowing just these facts it is clear that if we upload many images for a product then we may have many images uploaded for the product, but there is no connection to the variants!

If a shop is presenting a product that comes in five colours, or perhaps seven differing kinds of fabrics then it is likely they will want to change the main image presented to shopper to match the currently selected option. If I am looking at a t-shirt and I select the colour blue for it, I would expect the t-shirt to change to be blue. Some of my earliest customization jobs were all about providing this to shops. 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.

The vexing issue has always been that if you have 20 variants and 20 images uploaded, how do you connect them together? Recently Shopify introduced *alt* tag editing for uploaded images allowing a simple bit of text to be saved with uploaded images. Not only is this good for SEO but it can be used as a rudimentary hook so that when variants are selected, the image alt tags can be searched for a match, and an image swap can occur. I have never had trouble with more sophisticated approaches involving naming the images according to a variant key like SKU or title and using regular expressions, but for novices the Shopify alt tag approach is pretty good. You sacrifice SEO results for images, but you gain nicer shop presentation when images match the selected variants.

One cool aspect of swapping images is that all the images are readily available from Liquid when you pass the product through the built-in Liquid filter *json*. Any Javascript code can thus grab any image and process it as needed. By that I mean you can tack on an *grande for the Grande size image, or*thumb or any other image size designation and have that image for use on the page. Once the Liquid phase of rendering a product is done it can be left to Javascript to support image swaps with nice effects like fades and other easing motions.

JSONP

Most developers that have done any Javascript have probably done some asynchronous Javascript programming to. With Shopify being a hosted platform, all shops are known by their staked out subdomain name and the *myshopify.com* root domain. Even if I use the DNS to fool the world into thinking my shop is found at http://www.young-marble-giants.com it is still and forever http://no-eat-the-blue-mushrooms.myshopify.com for all intents and purposes. If I have a special question I have to ask of an App how can I do that? I could use an HTML *iframe* element I to embed a form in my shop, or I could use JSONP. That allows me to make a cross-domain Ajax request and so I can render the results of a question asked on the domain http://no-eat-the-blue-mushrooms.myshopify.com but answered by the domain http://veggie-

chef.heroku.com. This can be quite useful and serves as a huge support mechanism on more than a couple of sites I have built out for clients. If I wanted to avoid JSONP and just use straight up Ajax I could always create a subdomain like http://fizzbuzz.young-marble-giants.com and therefore I would be operating on the same domain and Ajax would work.

Special Invites

Before the App store and lockdown Apps like Gatekeeper were available, it was really difficult to open up a Shop and keep it locked down so that only choice guests could shop there. I was asked once to build out a Gatekeeper style solution just around christmas a few years ago. The nervous nelly NYC designer that contacted me was very concerned Shopify could not handle extreme traffic, and he was equally concerned that any App I built for him would crumble under intense load too. I brushed him off since all I knew was that there is no sense is worrying about something that has not happened and for all I knew, his shop would see 20 visits or maybe 200 or maybe even 2000 but all those numbers are small, so why worry.

I built an App that presented a simple form that captured a person's email address and a secret code. The code and the WWW address of the App was presented to all the people watching a popular TV show on D-Day. Each person that typed in the correct code had their email address sent to MailChimp and then the App unlocked the password protected Shopify site using the credentials only the App knew. Since the App was on a subdomain of the Shopify domain, the App could set the needed session cookie Shopify created and thus the TV shopper was seemlessly transferred into a locked shop without having to face the password screen. I paid little attention to this whole setup until D-Day around 3:00pm. I logged into the shop admin and was astonished to see close to 13,000 orders booked for that day alone and sales north of $750K. Considering the items for sale were made in China and cost a buck or two each, this was a real eye opener. I learned the lesson that accepting contracts like that without negotiating for a small % of sales is not great business. Both Shopify and my cloud based App never hiccuped during this event and that really gave me a lot of confidence in the platform.

Chapter 4

And 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 you will meet. If you are like me and enjoy sticking pins in maps, working on Shopify will probably force you to buy a Costco sized box of pins for your world map. Like the Ham Radio operators of yesteryear, collecting shops from far off countries and cities to work on is neat. I still have no hits from Easter Island but I am holding out hope.

Before corporations knew about the Internet, when cocaine and limos still ran the music industry and newspapers and magazines actually made money, there was little or no online e-commerce. Amazon orders for books and cake decorating tools with next-day delivery to your door? Not a chance back then, a mere decade ago. As Nortel sacrificed itself on the alter of greed laying fiber p\ipes and high speed machines for the coming Internet goldrush age, e-commerce was just a pale shade of what it is today. Now, *everyone* wants in and Shopify is a handy platform for many young businesses to try.

Some clients 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 clients that approach Shopify with notions of what they *think* they need and so when you tell them Shopify does not support their needs, they pop off like bottle rockets and rant. I enjoy these little conflicts. It gives me an opportunity to argue from both sides of the fence. Some of them make a good case and I feel bad that they cannot realistically use Shopify without sacrificing something fundamental. Internationalization is one of those issues. Many shops need to serve their public in more than one language and Shopify does not work in more than one language with the same shop. Even if a shop is presented with a deal on a second shop there is a lot of work in trying run two shops instead of just one.

Some other examples of very common rants I hear:

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? What the hell is that?

After six years or so, I am sure Shopify support staff are also pretty tired of hearing the same complaints. As time has passed the platform has not only grown and extended in capabilities but it has shown a certain amount of resilience too. Faced with the choice of trying to do much or just doing what it does do well, Shopify has embraced the idea of just doing what it does well.

Clients Respond to Clear Explanations

One of my favorite ways of explaining to clients why they cannot have Shopify just change on a dime is the story about Javascript, the lingua franca of all web browsers. It was created by a programmer with lots of experience at developing languages in about 10 days. Today it is part of every smartphone, every iPad tablet and all desktop web browsers. For all it's flaws from day one it remains much the same today, a whole decade later. Turns out you cannot put the genie back in the bottle and so 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 Coke? When they changed their formula they had their butts handed to them on a platter.

If a client complains about checkout, I think it is fair to ask them where they get their data supporting their case that one page checkouts convert better than multi-page checkouts. Are there enough Ph.D dissertations awarded that clearly show that as a 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. I am content to ignore discount coupons for the moment although I did come up a great App that makes my clients very happy. When reviewing an order where the client is not a happy camper I

place a link on the order details screen that sends a nice apology letter to the client with a nice discount code attached. Works like a charm.

I Want Variable Pricing

In Shopify, a product has no price. Only a product's variants have a price. If you set a price on a variant you cannot change it willy nilly at any point in a session. You cannot make it bigger or smaller. You may want to but you cannot do it. It often comes up that clients will want to bundle products together and if someone buys entire the bundle, it should be cheaper for the products. And yes I believe that, in fact I am on my knees looking up at the sky right now and I am believing that! The thing is Shopify does not work that way. If I bundle three products that cost $10 each together when they go in the cart they still cost $10 each. The bundle costs $30. It cannot be $25 just because. Some clients shake their heads at this. There are of course ways to sell bundles cheaper, but that involves actually making them separate products.

The *ideal client* will understand that e-commerce and opening a shop is a tough business. My ideal shopper is me. In my shoes I want to buy something off the Internet. I am prepared to part with my hard earned Paypal balance or even add to my Visa debt for what I need. I am willing to wait a week for it to arrive at my door. I start with the search engine. Does what I want even exist for sale? If it is for sale is there an online source for it in my country? It is amazing how Amazon messes this simple need up. I have credits at Amazon.com but I cannot use them with Amazon.ca. I want to buy something from Amazon but only Amazon.com sells what I want. I put it in my cart but at checkout they warn me I cannot buy what I want. No explanation just a No. So now I am browsing for Shopify or other small e-commerce retailers for what I want.

If I find what I want at a smaller shop than Amazon I still expect the buying process to be simple. I want that thing. Please send it to me now! Take my money. Thanks! The point is I like clients that are more concerned with their products being found in the first place than those worried about offering coupons.

Chapter 5
Shop Customizations

I would like to briefly explain some Shopify customizations that have not only stood the test of time but that I think underline the flexibility of the Shopify platform.

One shop started up and met with some initial success. So much success that the shop owner was up against a wall with the Shopify administration Orders 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 phoned me to inform me that the task of individually fulfilling these one thousand orders was stressing his wife out and that that in turn was stressing him out. You can use the Shopify API to inject a link into the Orders Overview screen so I told him that I could hook up an App that would respond to a click and fulfill all the selected orders. His wife was no doubt pleased to be relieved of the click-o-rama sessions.

Fulfilling orders in bulk taught me valuable lessons about using the Shopify API. An App has to stay within the limits that govern how many times you can use the API in a short amount of time. Shops can make 500 API calls in 300 seconds. If an App fulfilled 250 orders and then a second request came in moments later requesting 250 more orders be fulfilled followed shortly thereafter with another 250 orders the API limits would certainly be exceeded and the App would be blocked from using the API.

Using Delayed Jobs to Manage API Limits

Early attempts to work within the API limits resulted in using code that would sleep for 300 seconds when the API limit was reached. This turned out to be awkward for many reasons. I needed certainty that all operations completed without failure without the complexity of counting API calls. My solution was to setup a delayed job to process the fulfillments in the background. The delayed job is run by a worker process. Each API call to Shopify either succeeds or it does not. If it does not succeed because the API limits

have been reached the delayed job spawns a new delayed job with the remaining work to do and it sets it to run 300 seconds in the future. Using this technique it is possible to process as many API calls as possible without API limit problems. It's a very elegant and robust system proven to work well by hundreds of thousands of successful API calls.

Once that task was completed a more interesting issue arose. The products sold are edible and they always get delivered on a Monday or Tuesday. When ordering the product clients wanted to order for more than one week. Shopify does not currently support a recurring order or subscription service so we had to put on our thinking caps. 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 or up to four weeks of the product and pay once for all deliveries. By recording the delivery dates, it is possible to know exactly how many products get delivered per week per person. Some people order 2 or more per week so this had an immediate positive impact on the bottom line. We even added a button providing 3 months worth of future dates. Watching a shop nail 15% or higher sales with tweaks like this is rewarding.

Using Cart Attributes and WebHooks Together

Using a WebHook to capture paid orders we 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 Shop keeper can generate a nice Excel style grid of weekly deliveries with the ability to plan ahead. Once that proved successful, and many thousands of orders were being booked, it turned out that the ability to fulfill orders automatically using the API was crucial. The reason is that since you can fulfill an order as many times as you want, an order that has deliveries in the future can be fulfilled *each* time a delivery comes up. When you use the API to fulfill an already fulfilled but open order, the Shipping Confirmation email goes out, alerting the shopper that their delivery is on the way. Only when all deliveries are completed does an order get closed.

Adding Upsells to Boost Sales

At this point the store was running smooth. The shop owner wanted to add a twist to the cart. He wanted to *upsell* special products with the existing products. As an example, when Valentines 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 it's type to *upsell* we could offer this special product along side the regular products. The App allows the shop to assign any products of type upsell to any other regular products. Using Liquid, if the regular product has been assigned any upsell products we can render them too.

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 in my books.

Adding SMS Notifications

With so many people now using smartphones and SMS services, it made sense to add this to the shop fulfillment operations. It was easy enough to add a form to the shop's *Thank You* page asking the shopper if they wanted an SMS message when their order was fulfilled. Remarkably a huge number of people have provided their SMS numbers. The App now sends an SMS to each person when their order is fulfilled as opposed to an email which can sometimes be blocked by corporate firewalls.

This kind of customization illustrates how you can use the Shopify platform with confidence, knowing that it can handle twists that get thrown into the typical mix of scenarios that are possible. Some 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 there, they did not process the WebHook XML properly. A quick bridge was built by deploying an App to the cloud to accept these WebHooks from Shopify, and having the App then forward the order to Salesforce using XML formatted in a way that Salesforce was capable of accepting. Additionally, Salesforce comes with some pretty severe limits on what
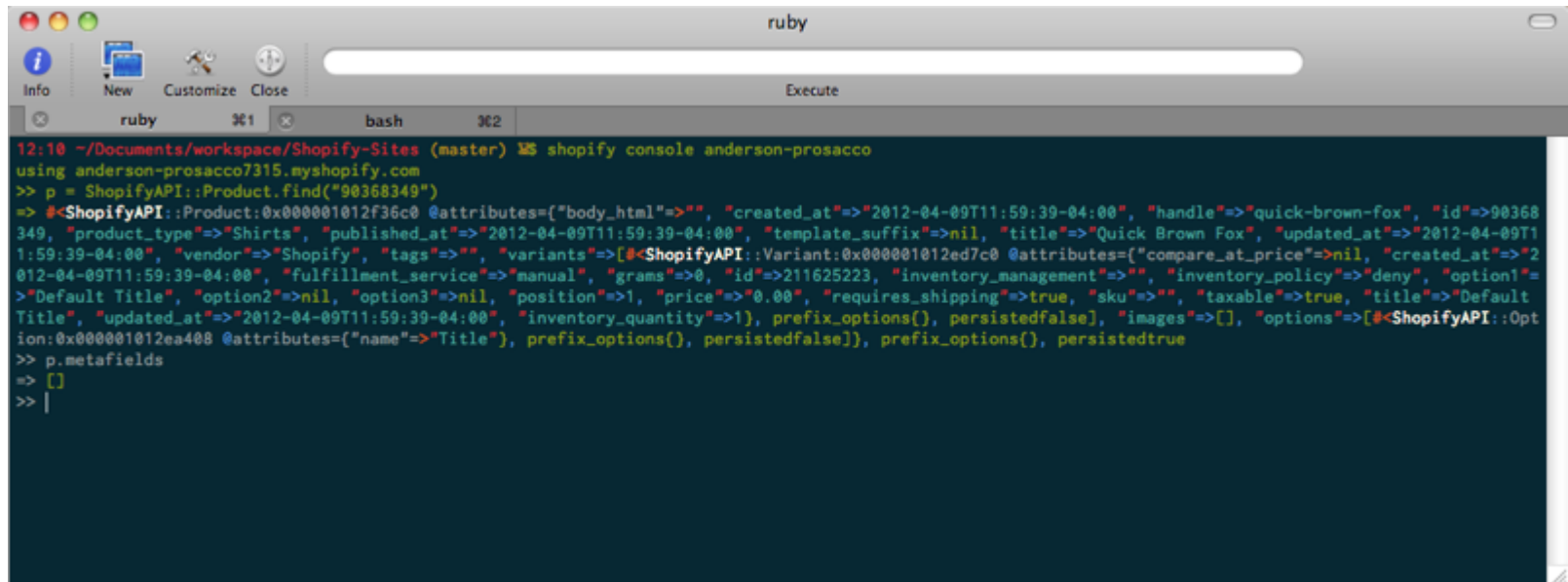
you can do with an entry level plan. Sometimes it turns out that you can do better with an App running in the cloud, so bridging Shopify to an App and Salesforce has shown itself to be a pretty powerful but cost effective system too. Shopify is soon to release an approved SalesForce App so that will likely appeal to the CRM crowd.

Chapter 6
Shopify Inventory

When I am working out a development plan with my engineering hat on, I always put the inventory setup first, and things like the theme last. To achieve success a shop needs to present inventory in a way that complements a smooth shopping experience and that is not necessarily accomplished with a gallery, carousel, slideshow or other front-end gimmick. Sometimes deeper thinking about how a product, it's variants, pricing, images and options will dictate the sales process. If the inventory is not organized correctly too much work can be expended in a fruitless quest for more sales.

A product requires at a minimum 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. Without no further action a product has been created and added to inventory that has a zero dollar price, infinite inventory, no tags, images, description or 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 ID it was assigned. You can *always* get the ID of a product simply by looking at the URL in the shop admin for the product when editing it. And example would be /admin/products/90368349 where the 903368349 is the unique ID for the product. As long as that product exists in the shop, that number will never change. It is used during import and export operations to identify products that will need to be updated. It is also handy to use that number when quickly checking out a product with the API. For example perhaps we want 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 we found a product and we are able to see that it has no metafield resources attached to it.

When setting up a product in inventory we can edit the variant and provide a title for the variant as well as it's price, SKU and inventory management policy. Shopify also provides three options that can be assigned to a variant. If a product has a colour, size and material they can be accommodated as options. These options could 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 could be setup. Assuming all three options are used Shopify allows a product up to 100 variants. This means a product with five colours, four sizes and three materials would require 5 times 4 times 3 or 60 variants. Each one counts as an SKU in Shopify for the pricing plan selected. It is not necessary to assign an SKU to each variant, and you can assign the same SKU to as many variants as needed. 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 it's attributes. If we open

up 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.

Elements Resources Network Scripts Timeline Profiles Audits Console

Search Console

```
> Shopify.api.getProduct('i_fizz_buzz')
  undefined
  ▶ XHR finished loading: "http://swift-braun3430.myshopify.com/products/i_fizz_buzz.js".                                                                          jquery-1.6.2.min.js:18
  Received everything we ever wanted to know about                                                                                                                  api.js:60
  ▼ Object
      available: true
      compare_at_price: null
      compare_at_price_max: 0
      compare_at_price_min: 0
      compare_at_price_varies: false
      created_at: "2012-03-30T10:36:44-04:00"
      description: "<p>These are gorgeous iPhone 4 cases, personalized with your favorite image of family, a wedding, or other significant event in your life; or even just a favorite piece of art from Van Gogh, Ce
      featured_image: "http://static.shopify.com/s/files/1/0084/3092/products/iphone-product-shot.jpeg?560"
      handle: "i_fizz_buzz"
      id: 90033083
    ▶ images: Array[4]
    ▼ options: Array[1]
      ▼ 0: Object
          name: "Style"
        ▶ __proto__: Object
        length: 1
      ▶ __proto__: Array[0]
      price: 2995
      price_max: 2995
      price_min: 2995
      price_varies: false
      published_at: "2012-03-30T10:36:44-04:00"
    ▶ tags: Array[0]
      title: "iPhone 4/4S Case"
      type: "Aluminum"
      url: "/products/i_fizz_buzz"
    ▼ variants: Array[3]
      ▼ 0: Object
          available: true
          compare_at_price: null
          id: 210767655
          inventory_management: "shopify"
          inventory_quantity: 48
          option1: "Clear Case"
          option2: null
          option3: null
        ▼ options: Array[1]
            0: "Clear Case"
          length: 1
          ▶ __proto__: Array[0]
          price: 2995
          requires_shipping: true
          sku: "fizzbuzz_1"
          taxable: true
          title: "Clear Case"
          weight: 45
        ▶ __proto__: Object
      ▶ 1: Object
      ▶ 2: Object
        length: 3
      ▶ __proto__: Array[0]
      vendor: "Custom Images"
    ▶ __proto__: Object
>
```

☐ ⊁≡ 🔍 ⊘ <top frame>   ▼ All | Errors Warnings Logs

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 attributes generated for each variant. We can use this to completely

customize the way Shopify renders products and variants. Many shops use the code Shopify provides 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 and 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 those 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 perhaps the variant title. It is easy to make up some rules dictating how images that are uploaded for a product are coded so that code can be used in the callback logic.

A pattern I have used successfully to bring to life shops with clickable image swatches that change the variant, loading new sets of thumbnails, and allowing a mix of select elements with color swatches is to simply add a small script library to the shop containing code to work with the options. I let the default Shopify code option_selectors.js run knowing it will setup the HTML select elements and wire up whatever callback I want. I hide all the select elements Shopify generated. I modify the callback to run any extra code I may need to ensure smooth operations with the selections. Since I hid the HTML select elements, I run code that renders images or color swatches representing the variants. Each set of thumbnails or color swatches is assigned a click listener and that is the key to ensuring the right variant is selected. A click on an image that represents the product with the style "faux beaver fur" can trigger the change event on the hidden HTML select element setting it to the value "faux beaver fur" and the callback function will be called with the correct variant. Fabulous and simple. Clicks on an image can be so much more intuitive than selecting text from a drop down element. Perhaps the product has not only faux beaver fur but also an option called colour represented by clickable colour boxes. A click on any of the colour boxes would be recognized as blue, red or perhaps green. We could change the background colour of the main image to match. Or we could load new images of the product with "beaver faux fur" colourized to match.

Knowing how Shopify inventory works with respect to variants and options is crucial to building out shops with advanced capabilities. To take a shop to a level where shoppers can easily add the product of their choice into the cart and buy it without guesswork or too much reading is essential. Knowing that you can upload as many images of a product as you want does not make it easy to swap the images when options

are selected. One technique is to use the Shopify administration interface to add text to product image alt attributes. Those should be used for improving SEO but they can also be used to make a primitive image swapper work. For example, if the alt tag was set to a color that matched the variant title, you could swap the image currently visible with one that matches color wise when a variant change is detected. A more sophisticated approach would be to examine the name of the image itself. Basic regular expressions can be used to parse a filename into all of it's parts. That might be white_lily_formica_solipstic_grande.jpeg where you can decide that the click was on a *white lily* colored *formica solipstic* variant and that the image is Shopify grande in size. Now, with a tiny bit of code, you could substitute the image uploaded as black_lily_formica_solipstic.jpeg and you could set the size to medium using the source black_lily_formica_solipstic_medium.jpeg.

Sometimes you have to present a list of colours but the sizes should remain as an HTML select element. Don't hide the select element for size changes. Be aware that when showing and hiding elements like the ones Shopify renders in their option_selectors.js file, that they are using a numbering scheme derived from the representation of the product object too. If there are three options, they are listed as option1, option2, option3 and that you might have the actual options in an array where option0 is option1, coming from a click on a select element with the DOM ID product-select-option-0 depending on how you called the select element in your theme (in this case ID was product-select).

In summary, it is possible to present your Products in Shopify with pretty amazing effects triggered by clicks on elements other then HTML select elements, while maintaing control over the current variant, meaning you can update prices and keep the add to cart action enabled or disabled depending on inventory settings.

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 run from modest brick and mortar headquarters, have relatively small employee head counts but count millions and perhaps billions of people as users of their services. Little of their success can be attributed to pounding the pavment door to door, or by flooding the TV with advertising. Service oriented companies leverage the Internet itself for their growth and one of the underlying reasons for their rapid growth, adoption and success is due the concept of the API or Application Programming Interface.

What better way to introduce the public at large to your service than to allow them to build it, populate it and enjoy it using their own labour and tools. YouTube, Facebook and Twitter would not exist without user generated content. How to ensure everyone can contribute to your service without being overly technical or specialized? How to 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. Provide a simple mechanism everyone can take advantage of and the ball starts rolling.

Shopify released their API after some years of processing a steadily growing number of transactions (perhaps some would see it as explosive growth). A period of time during which Shopify surely learned not only to understand 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 codebase. Once the early established shops crossed the chasm from early adopters to profitable e-commerce enterprises, there was a corresponding demand for more and better control of the underlying processes.

Shopify chose to establish their API using the software architecture known as REpresentational State Transfer (REST) accepted by a majority of the Internet community. The API provides support for accessing a

Shopify shop and creating, reading, updating or deleting the resources of the shop. Almost all e-commerce companies offer some sort of API but there is a 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 in advancing the capabilities of many thousands of shops. Whenever the general question is asked "Can Shopify do such and such?", I am often answering the question with a confident "Yes it can, you simply need to use the API."

If we accept that software is one of the most fiendishly diffcult man-made constructs invented and if we accept that even after nearly seventy years of computing research and development we are still stumbling along like drunk sailors, we must learn to accept and set limits while we sort it all out. Shopify has established a working e-commerce system transacting hundreds of millions of dollars per year for many thousands of merchants, but it cannot stray too far from the model that currently works. If all the many requested features were added, the system would likely become unstable and prone to outages, breakdowns and a destroyed reputation for reliability. We know some of the most expensive human mistakes ever can be attributed to computer bugs no one ever knew were there till it was too late. 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 to make running an e-commerce shop on the Shopify platform easier for merchants.

A good example would be the task of fulfilling an order. Shopify has a setup 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 your option? Your supply chain from Manufacturer DrubbleZook to warehouse Zingoblork with shipper USPS 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 handle the fulfillment. An App running in the cloud listening 24/7 for incoming orders. And when it gets an order it robotically follows a set of instructions that ensures the Shopify merchant is going to be 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 cart issuer. The App can take the order from Shopify and format it for Zingoblork and their special needs. It is 2012 as I write this and Zingoblork warehouse 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 IT!

EDI which we won't even bother to describe, but suffice it to say, the Chevy Vega of exchanges

SOAP which makes me want to run away and mow grass for a living

The App accepts all orders thanks to the API and perhaps in combination with 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 person that bought the goods 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 and place those in a holding pen accessible only by a special FTP account. Others will simply send these codes to the shop via email completely destroying the shop keepers email inbox and sanity. The best fulfillment companies will use HTTP POST to send the order with a tracking number back to the App allowing the App to automatically create a fulfillment using the API with the tracking code. Shopify automatically detects the creation of a new fulfillment and sends an appropriate email. This is wonderful since the App and the API together can close the loop automatically.

Without an API it would be impossible to offer this level of customization to a shop. Another interesting use of Shopify is when a shop is opened up to sell products on a consignment basis. If I accept 1000 of your widgets to sell in my shop for $20 each you want to know if you have made $20 in sales or $400 in sales or even $4000 in sales. If I accept 1000 widgets to sell on behalf of 20 or 30 different vendors my life will almost certainly be miserable unless I use the API. I record each and every line item sold as a sale assigned to the product's vendor. Every product has a selling price and a cost price and so we know the difference between those should be the profit to split between me and my vendor(s). I have a deal with my vendors giving them 80/20 or sometimes 60/40 splits of that profit. We really should assign that percentage to each sale as well so that we can make different deals depending on the day of the week, the colour of the sky or the

popularity of the product at the checkout. That is an App that can be plugged into a shop. None of that is possible without an App.

The presence and functionality of the Shopify API is a crucial variable in the decision equation most enterprises work out before going into business with the Shopify platform. If a business knows the fixed costs to run a shop for a year are $1000 and that their theme will cost them $5000 to make the shop attractive, the API is the only other key option that deserves close attention. Adding an App can add monthly or one-time fees, but they can also save hundreds of hours in manual labour. Paying for a custom App to be developed to address a special need can cost a few thousand dollars but can result in measurable growth in sales. Many companies are looking for a reliable hosted e-commerce platform and partner(s) to work with to bring their e-commerce vision to life. Knowing they can use the API to add the little extras is often enough incentive to choose Shopify over competing platforms.

Chapter 8
How to Handle Webhook Requests

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

Here's some pseudocode demonstrating what I mean:

```
def handle_webhook request
  process_data request.data  # Note that the process_data call could take a lot of time
  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
  schedule_processing request.data # this takes no time so the response is quick
  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 since they often put Apps to sleep when they are not busy. This can take several seconds. If your App is only spending five seconds processing data it'll 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 reading and should be thoroughly understood to get the most out of using WebHooks. In my experience with Webhooks I have run into all sorts of interesting issues so I will dedicate some effort to explaining them from an App developer perspective.

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 that are free to use. Let's take one quick example and use RequestBin. The first thing I will 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. The result is a generated URL that can be used for testing. Note that one can also make this test private so that only you can see the results of WebHooks sent to the RequestBin.

This bin will keep the last 20 requests made to it and remain available for the next 48 hours. However, during beta the data store might be cleared at any time, so **treat bins as highly ephemeral.**

© Jeff Lindsay 2011
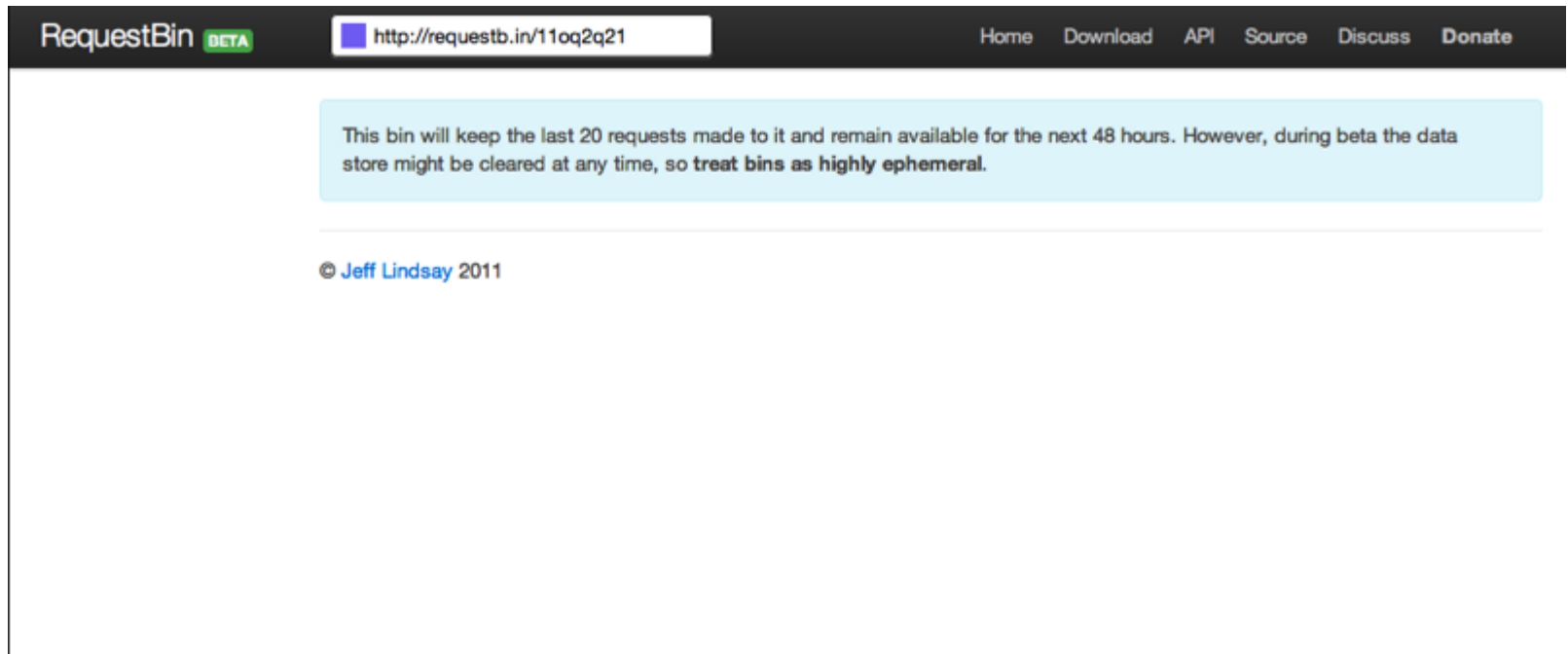
Newly Created RequestBin

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

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

Testable WebHook

The ability to delete a WebHook as well as test it in the shop has on occasion burned me. In my haste to deal with a situation involving WebHooks I have been guilty of accidentally pressing the trashcan icon and removing a WebHook that should *never have been removed*. Ooops! It can take only seconds of carelessness to decouple a shop set for live e-commerce sales from a crucial App running and connected to the shop. Be careful when clicking around these parts!

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

#1l5879   POST /11oq2q21

Headers   Content

body { "billing_address": { "address1": "123 Billing Street", "address2": null, "city": "Billtown", "company": "My Company", "country": "United States", "country_code": "US", "first_name": "Bob", "last_name": "Biller", "latitude": null, "longitude": null, "name": "Bob Biller", "phone": "555-555-BILL", "province": "Kentucky", "province_code": "KY", "zip": "K2P0B0" }, "browser_ip": null, "buyer_accepts_marketing": true, "cancel_reason": "customer", "cancelled_at": "2012-04-10T16:27:40-04:00", "cart_token": null, "closed_at": null, "created_at": "2012-04-10T16:27:40-04:00", "currency": "USD", "customer": { "accepts_marketing": null, "created_at": null, "email": "john@test.com", "first_name": "John", "last_name": "Smith", "last_order_id": null, "last_order_name": null, "note": null, "orders_count": 0, "state": "disabled", "tags": "", "total_spent": "0.00", "updated_at": null }, "discount_codes": [], "email": "jon@doe.ca", "financial_status": "voided", "fulfillment_status": "pending", "fulfillments": [], "gateway": "bogus", "id": 123456, "landing_site": null, "landing_site_ref": null, "line_items": [{ "fulfillment_service": "manual", "fulfillment_status": null, "grams": 5000, "name": "Sledgehammer", "price": "199.99", "product_id": null, "quantity": 1, "requires_shipping": true, "sku": "SKU2006-001", "title": "Sledgehammer", "variant_id": null, "variant_title": null, "vendor": null }, { "fulfillment_service": "manual", "fulfillment_status": null, "grams": 500, "name": "Wire Cutter", "price": "29.95", "product_id": null, "quantity": 1, "requires_shipping": true, "sku": "SKU2006-020", "title": "Wire Cutter", "variant_id": null, "variant_title": null, "vendor": null }], "name": "#9999", "note": null, "note_attributes": [], "number": 234, "order_number": 1234, "referring_site": null, "risk_details": [], "shipping_address": { "address1": "123 Shipping Street", "address2": null, "city": "Shippington", "company": "Shipping Company", "country": "United States", "country_code": "US", "first_name": "Steve", "last_name": "Shipper", "latitude": null, "longitude": null, "name": "Steve Shipper", "phone": "555-555-SHIP", "province": "Kentucky", "province_code": "KY", "zip": "K2P0S0" }, "shipping_lines": [{ "code": null, "price": "10.00", "source": "shopify", "title": "Generic Shipping" }], "subtotal_price": "229.94", "tax_lines": [], "taxes_included": false, "token": null, "total_discounts": "0.00", "total_line_items_price": "229.94", "total_price": "239.94", "total_tax": "0.00", "total_weight": 0, "updated_at": "2012-04-10T16:27:40-04:00" }

application/json
2330 bytes

2012-04-10
20:27:40.987333
204.93.213.120

WebHook Results

Looking closely at the sample order data which is in JSON format we see there is a complete test order to work with. We have closed the loop 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 we note that the sample data from Shopify is great for testing connectivity more than for testing out an App. Close examination of the provided test data shows a lot of the fields are empty or null. What would be nice is to be able send real data to an App without the hassle of actually using the Shop and booking test orders. For example, say you are developing an Application to test out a fancy order fulfillment routine a shop needs.

You know you need to test a couple of specific aspects of an Order, namely:

Ensure the WebHook order data actually came from Shopify, and that you have the shop identification to work on.

Ensure you do not already have this order processed as it makes no sense to process a PAID order two or more times.

You know you need to parse out the credit card used, and the shipping charges, and the discount codes used if any.

There could be product customization data in the cart note or cart attributes that need to be examined.

This small list introduces some issues that may not be obvious to new developers to the Shopify platform. We can address each one and hopefully that will provide some useful insight into how you can structure an App to best deal with WebHooks from Shopify.

WebHook Validation

When you setup an App in the Shopify Partner web application one of the key attributes generated by Shopify for the App is the authentication data. Each App will have an API key to identify it as well as a shared secret. These are unique tokens and they are critical to 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 clearly knows the identity of

the App requesting the WebHook to be created, so Shopify uses the shared secret associated with the App and makes it part of the WebHook itself. Before Shopify sends off a WebHook created by an App it will be 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 setup with the API will have HTTP_X_SHOPIFY_HMAC_SHA256 in the Request's header. Since the App has access to the shared secret, the App can now 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
```

If we were to send the request body as the App received it to this little method, and the value of the HTTP_X_SHOPIFY_HMAC_SHA256 attribute in the request, it can calculate the Hash in the same manner as Shopify did before sending out the request. If the two computed values match, you can be assured the WebHook is valid and came from Shopify. That is why it is important to ensure your shared secret is not widely distrubuted on the Internet. You would lose your ability to judge between valid and invalid requests between Shopify and your App.

Looking out for Duplicate Webhooks

As explained in the WebHook best practices guide, Shopify will send a WebHook out and then wait up to ten seconds for a response status. If that response is not received the WebHook will be resent. This continues until a 200 OK status is received ensuring that even if a network connection is down or some other problem is present Shopify will keep trying to get the WebHook to the App. The initial interval between retries of ten seconds is not practical for a large number of retries so the time interval between

requests is constantly extended until the WebHook is only retried every hour or more. If nothing changes within 48 hours an email is sent to the App owner warning them their WebHook receiver is not working and that the WebHook itself will be deleted from the shop. This can have harsh consequences, mitigated by the fact that the email should be sufficient to alert the App owner to the existence of a problem.

Assuming 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 Center and 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 extending the time it takes for a complete exchange between Shopify and an App. Sometimes the App itself can take a long time to process a WebHook and respond. 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 apriori 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/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 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 as the queue welcomes any and all WebHooks, duplicates or not.

With an App directing all incoming WebHooks to an queue a queue worker process can be used to *pop off* WebHooks in the queue for processing. There is no longer a concern over processing speed and the App can do all the sophisticated processing it needs to do at it's leisure. It is possible to be certain whether an orders/

paid 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 as JSON. Most scripting languages used to build Apps have XML parsers that can make request processing routine. With the advent of NoSQL databases storing JSON documents such as CouchDB and MongoDB many Apps take advantage of this and prefer all incoming requests to be JSON. Additionally one can use Node.js om the server to process WebHooks and so JSON is a natural fit for those applications. Since the logic of searching a request for a specific field is the same for both formats, it is up to the App author to choose the format they prefer.

Cart Customization

Without a doubt one of the most useful but also a more difficult aspect 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 directly from a shop customer. Any monogrammed handbags, initialed wedding invitations, engraved glass baby bottles etc. will have used the cart note or cart attributes to capture this information and pass it 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 we detect cart attributes with JSON, we can parse that JSON and reconstitute the original objects embedded in there. In my opinion it is this pattern of augmenting orders with cart attributes, passing them to Apps by WebHook and then parsing out the special attributes that has made it possible for the Shopify platform to deliver such a wide variety of e-commerce sites while keeping the platform reasonably simple.

Chapter 9
Command Line Shopify
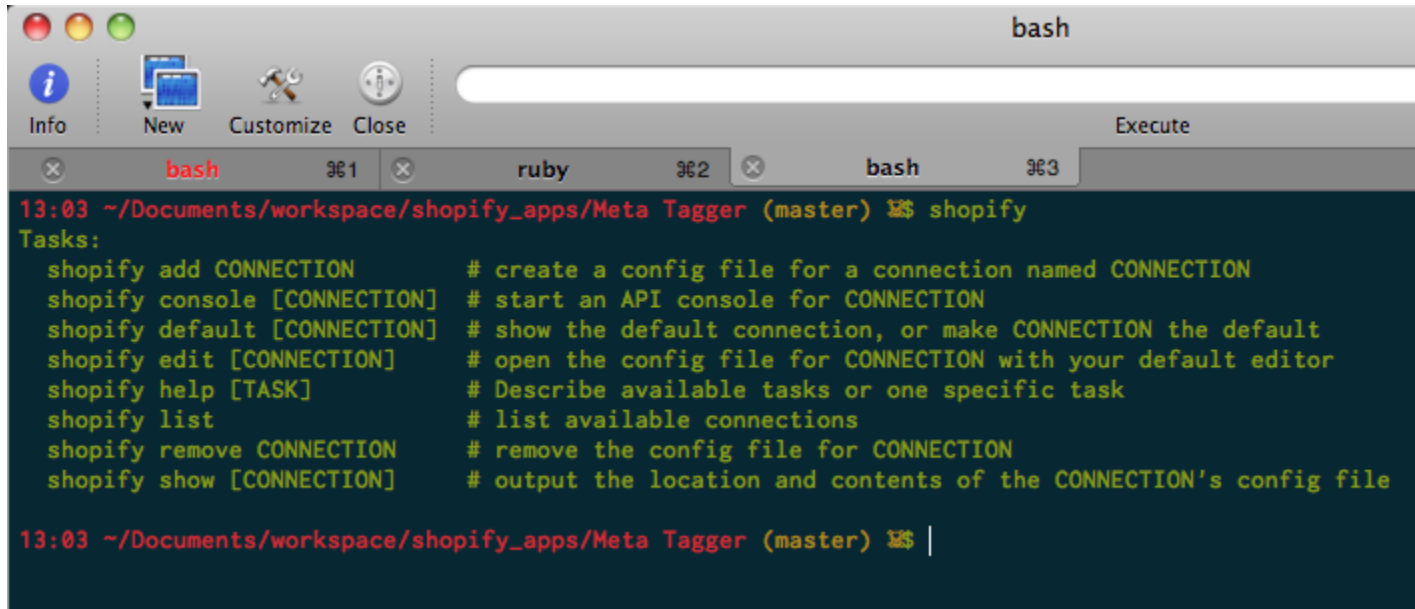
Quick development of Apps requires being able to fire off requests to a shop and to be able to process the responses. While WebHooks are a source of incoming data from a Shop many interesting possibilities are available using a terminal to access the API at the command line.

When firing off a request for a resource from a Shop using one of the formats of JSON or XML, we are using a standard HTTP verb like GET. To quickly send a GET request to a shop with authentication can be an involved setup process using the command line. For example, we could use the venerable curl command to send of a request for a Shop, Product or Collection resource. Doing so by hand can be tedious in my opinion. There are much more sophisticated ways to do this without having to run an App and inspect the responses and run debuggers. While I enjoy single-stepping through live code and inspecting the state of my objects that make up an App, I also appreciate being able to quickly test out a theory or check on code syntax without the burden of that overhead.

Shopify has a super example of how to do just this and it is a small application they bundle with their Ruby gem. Some time ago, 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 lets you quickly write a command-line interface. The current Shopify API gem comes with one of these built-in.

If you have installed the Shopify API gem and you start a terminal session on your computer you will be able to test this out fairly quickly. I use RVM to manage all my versions of Ruby, but there are alternatives including RBENV and the usual nothing special at all I use the default installed with my computer. I am making the bold assumption that most developers are using a computer with Ruby (or any dynamic scripting language for that matter, be it Perl, PHP, Python or others) and that a terminal session is part of their toolkit.

When I use Ruby, and I check my installed gems using the gem list command, I see the shopify api gem in my list and I can test for the Shopify CLI by simply typing in the command shopify.

Available Options for the shopify Command Line Interface

The shopify Command Line Console

There are just a few options listed and in order to really cook up some interesting examples we will use the configure option that comes with the command. The options required to configure a shopify session are the Shops API key and a password. To get those values, we will use the Shop itself. For some developers this will mean using their development shop, and for others, their clients have provided access to their shop so a private App can be created, or they created the Private App for the developer and 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, we use the API key and Password along with the shop's name. Once we have completed the configuration, we can access the shop using a console.

As an example, here is a console session for a development shop, showing how easy it is to query for the shop's count of products, orders, and the details of a single order.

Output of Calling a Shop Using the API

Now we can test out potential code snippets quickly without incurring a lot of overhead. API code can be tested with the fewest possible keystrokes and minimum effort with this handy console. For example we know we can access metafields resources for a shop by providing an ID for the resource. What is the syntax of a call with Active Resource? It can be a challenge to keep perfect API syntax in your head, so we often just try things out to see if they work. With the console I can try a few different call patterns until I stumble on the correct one.

With the shopify utility we can add connections to as many Shops as we want and we can list them all. When working on a client shop one of the first things I do is use the shops admin interface and the App tab

to create a private tool. With the resulting API key and password I can use the shopify console tool to quickly test out any custom queries I may want to build into an App for the client.

The Shopify Theme Command Line Console

The second useful command line tool that Shopify provides is not in the shopify_api gem but is a separate gem called shopify_theme. With this gem installed on your system you can use the command theme to work with shop code directly. Theme is another fine command line tool with a simple workflow. The first thing I do with a new client is create a directory to hold all the files that make up the client's theme.

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

Output of the Theme Command Line Console

You can see from the listing of available options the *theme* commmand is slightly different from the *shopify* command. Nonetheless it uses the exact same API key and password to access a shop's resources. Provide the credentials and the configuration is written out as a file and we can begin work. The first step is to download the client's theme into the working directory so that I can examine their Javascript, and Liquid assets like the templates.

```
$ theme download
```

Once the assets are downloaded I setup the files under version control. I make a local repository with git and store the client's code under version control for safety. You could even go so far as to setup a git remote pointing at github so that your 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 good time to start working. I 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 where the file was downloaded from. I can verify my edit worked (or not) using my web browser and viewing the shop. When I am happy a small change is a good change I commit the change to git and move on to the next task.

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

This is a very productive and safe workflow in my opinion. I have the code under version control and I can make my edits using my favorite code editing tools. If in the future the client wants more work done I can use the theme download tool to grab any new files or changes to files initiated by the client in the meantime. Git will tell me what changes if any are present and I can investigate those. Sometimes it is a simple task to fix something this way. At least you have a fighting chance when you are not the only person touching or editing a client site. Squashing some designers work or other persons files is rarely an acceptable practice!