David Lazar

[Your Book Title]

[Your book description]

Content

Chapter 2
Who is this 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.

Chapter 3
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 shimmy shimmy slow dancing in the dark that goes into the workaday world of a professional coder.

Chapter 4

A Brief History of Shopify Through a Developer's Eyes

My engineering career started professionally in 1990 at about the same the World Wide Web (WWW) was invented with the introduction of HTTP protocol for the Internet. Before the WWW the Internet was used to exchange email (SMTP), to post messages to Usenet (NNTP) or FTP was used to transfer binary and text files between clients and servers. Archie, Veronica and Gopher were other darlings of the nerd set. Once the WWW helped the Internet concept jump across the early adopter chasm to be embraced by the dot-com 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 the business people sitting on the sidelines with few clues what to do with this new medium, poured money into the foundations of what are today some of the most valuable business properties in the world.

Around 1998 I transitioned my software development focus from C, C++ development for research and development companies to Internet computing. Progress in language development showed you 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.

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 5

The Developers Environment

As a developer with some experience, I have coded for many processors in many languages. From Z-80 Assembly language on the TRS-80, 68000 C on an Amiga 500, C++ on a Sun Workstation to PHP on a homebrew Pentium, the tools used have not varied too much. These days, coding web applications on a laptop in a cafe is not unusual, and having beautiful tools to do so is a most welcome change. Developers are finally able to choose from some very fine tools indeed, and that was almost unheard of 5 or more years ago.

The Text Editor

Vim, Emacs, TextMate, UltraEdit, Sublime Text, Eclipse, Visual Studio, Notepad, … I am in no position to shed any light on this as I cannot even touch type. I get by only in the sense that I type at about the same speed I think. Maybe I think slow and my typing matches? 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 kerfuffle happening out there at the coffee shop. Wish I could do that. As a manager I would use that as a test. If you want a job coding for me, and you cannot type, you cannot be a real coder. Simple. Although, that puts me on the breadline.

My text editor has to have nice colors, easy on the eyes like Solarized. Has to syntax highlight Liquid, Haml and Sass, autosave all my code, and be a pleasure to search with. Sublime Text 2 fits that bill. I have no use for editors that have ftp or SVN built-in like that is something special.

The Web Browser

Obviously every developer needs a good web browser to work from, 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.

Versioning your work

Shopify will version templates as you change them. However, this versioning is not terribly useful to a team and certainly is not something you want to rely on for your only backup. I suggest learning a version control system like git. A basic skill that will payoff in spades if you've never experienced it. You can version everything you work on. Every line of code, every proposal, even your binary work like images and assets that you might not ordinarily think of as something you version.

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 off to them. The mumbled excuses and apologies you get to hear about not seeing your invoice when they clearly did but conveniently forgot is a perk of the self-employed.

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. 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 luxury you've provided them. Writing a manual 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 just 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 utility available that can be installed on any computer that has the Ruby programming language on it. The utility provides a few basic commands but the gist of it is that you can easily download a whole shop so that it resides on your computer's filesystem for editing. Once you do that, you can immediately check the entire codebase into a version control system. I like to use git for this task. You can choose to use any version control system you like. 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 loop is adddedto 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 live. 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.

Localhost Development on a Laptop, Desktop or Other Devices

With text editing, version control, and watching for code changes in place, a developer is almost ready to tackle any kind of 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 primitive days of web application development we had to ensure that we had a web server like Apache installed on our machines, in addition to scripting languages like Ruby, Python and PHP. That is an extra level of hassle we can do without these days. Conveniently the Ruby language comes with several extremely simple but fast web servers that make local development a breeze. For example the thin webserver by Marc-Andre Courneyer. Other options exist like Pow from 37Signals, WebBrick, Mongreal, Nginx, etc. You have to find one you're comfortable with, and stick with it so as to be able to quickly whip up an App to test a concept out or to just be portable and not tied to a single server on the Internet. Being able to develop 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. These compilers allow a developer or designer to setup variables, establish styles to be mixed in with other styles, to extend existing styles and to ensure big and complex style sheets are managed with smaller, moduler files. Less can be compiled using Javascript, and Sass is compiled with Ruby. The advantages are somewhat spectacular in my opinion. You can build a complex Shopify theme's CSS using these tools and gain a lot of very interesting control. For example, to allow a client to change a color of a heading element, or the behaviour of a list, the same variables that you would use in Less or Sass can be programmed in Liquid. This means that a small Less or Sass effort can expand to a much larger more complex CSS file, that in turn becomes part of a Shop, but can be easily customized.

Chapter 6

Common Questions

There are a lot of reasons to love a hosted platform like Shopify since the responsibility for a lot of the small but necessary implementation details of running a website are Shopify's responsibilty. By offering an environment that provides most of the basic features needed to run an e-commerce website, Shopify's platform has created a community of businesses that can share insight and knowledge. It is comforting to know that when an issue arises with respect to how to configure a shop, or take advantage of a built-in preference or how to exploit a feature, the community contributions are there as answers.

One of the earliest and still most common question concerns the customization of products. The gamut of how people want to customize products is immense. Glass baby bottles etched with a monogram, handbags with initials stitched into the leather, silver pendant jewelry with the name of the dog or the newest twins on the block, the presentation of a form to collect this information is seemingly a source of endless discussion.

The evolution of Shopify saw the introduction of cart attributes to support the feature of passing extra data with the order, allowing a shop owner to process the order with the extras. Additionally for simple notes there is also the cart note feature. The cart attributes are a key value pair and an order can have as many of these as needed to fully define an order. Anything can serve as a key and good examples are the identification number of a product, or the variant. The value that can be stored with the key is a string. Therefore, it is quite simple to capture some needed monogram for a product.

```
      attribute['I-am_a_key'] = "welcome to outer space astronaut!"
attribute[12345678] = "David Bowie"
attribute[44556677] = '[{name: "quick brown fox", action: "jumped the lazy dog"}, {name: "lazy hen", action: "laid no eggs t
```

Those are three examples of setting an attribute key to some value. Using Javascript it is pretty simple to experiment and set the cart attributes, and then check that they were set correctly too. An interesting fact about using cart attributes is that since the key can be a variant ID, and the value is a string, you can store JSON representations of customization data. The third example I present shows this. The cart attribute for ID 44556677 that could be some variant in my shop inventory has been assigned a string that is JSON or Javascript Object Notation. 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 Quick Brown Fox and the other Lazy Hen”. Now, when rendering the cart, you could show these names and actions along with the variants themselves. It is possible that the person checks out with this order, and the result displayed in the Shopify admin for the order would not only show the two variants, but in the order note section the shop keeper would see the key and value

```
44556677: '[{name: "quick brown fox", action: "jumped the lazy dog"}, {name: "lazy hen", action: "laid no eggs today"}
```

Now, that is a little ugly it is true, and some shop keepers might balk at having the customizations like that. It is quite possible to deal with this issue with more sophistication. Instead of directly storing the customization data in cart attributes, they could be stored in a cookie, or in a browsers localStorage. That way during the entire shopping session, the site manages the customizations as JSON, but before actually submitting the order, the customization code transfers the JSON to plain english. As an example, if my variant ID represented a Farm Animal, for $24.00, I could rewrite the attribute to be

```
attribute["Farm Animal"] = "Name: Quick Brown Fox, Action: Jumped the Lazy Dog, Name: Lazy Hen, Action: Laid No Eggs"
```

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.

Chapter 7

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 Ham Radio operators of yesteryear, collecting countries, and cities is in and of itself a perk of the job.

Before corporations knew about the Internet, when cocaine and limos still ran the music industry and Newspapers and Magazines made money, there was little to no public e-commerce. Goods and Services, check. Amazon orders with next-day FedEx deliveries to your door? Not a chance. As Nortel sacrificed itself on the alter of greed laying fiber pipe and higher speed networking machines for the goldrush E-Commerce was rising to be the star it is today. Now, everyone wants in, and Shopify is a wonderful platform to begin with.

What some clients do understand is that while Shopify does not try to offer everything to everybody, what it does offer is sufficient for most needs and purposes. There are plenty of clients that will approach Shopify with pre-conceived notions of what they need in an e-commerce platform and when you tell them Shopify does not support their needs directly, 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 examples of very common rants I hear all the time.

- 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?
- It’s not bilingual? What kind of crap is that? I have to setup two shop? Tabernouche!

After six years or so, I am sure Shopify support is pretty tired of hearing the complaints too, but interestingly enough, as that time has passed, the platform has not only grown in capabilities, but it has shown a certain amount of resilience too, as it has absorbed ideas from the public and rolled-out new features. There is a bit of method to the madness.

One of my latest favorite ways to explain to clients why they cannot have the cookies and milk instantly, is short story about the history of one of my favorite programming languages, namely Javascript. It was created by a programmer with lots of experience at developing languages in a mere 10 days. Today, it is on virtually every smartphone in existence, every iPad and all web browsers. It is the ligua franca of the Internet. And for all it’s flaws from day one, it remains much the same today a decade later. You cannot put the genie back in the bottle. Shopify cannot undo the patterns they laid down six years ago either. Software is mankinds most fiendishly difficult construct ever and so we go with what works, for as long as possible before we dare to change it, especially when it is working.

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 multipage checkouts. Are there enough Ph.D dissertations awarded clearly showing that? Discount Codes are another funny trend these days. As Groupon and Living Social have shown, handing people coupons is good for business. Maybe not for Groupon itself or for the vendors that try it and get swamped by large volumes of orders forcing them to take tiny margins followed by fallow times. Do coupon clippers repeat? Gimmicks?

In Shopify, a Product has no price. A Product’s Variant has a price though. If you set a price on a Variant, you cannot change it willy nilly at any point in a checkout. 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 the bundle, it should be cheaper! And yes! I believe! I am on my knees, looking up to the skies and believing! 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 cost $10 each, and the bundle of them costs $30. It cannot be $25 just because. Some clients shake their heads at this.

My ideal client will understand my points here… and want me to ensure their site works like this…

I want to buy something off the Internet. I am prepared to part with my hard-earned Paypal balance or heck, even add to my Visa/Mastercard debt for it. 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? Amazing how Amazon messes this up for me. I have credit at Amazon.com I cannot use at Amazon.ca. I want something from Amazon, but only Amazon.com where I have credit sells it. I put it in my cart, but at checkout they warn me I cannot buy that product. No explanation, just a no. So I am off Amazon and looking at smaller sources. Here is where Shopify comes in. Or the they shall not be named others.

If I find it at a boutique shop, I expect the process to remain pretty simple. I want it. Give it to me. Take my money. Thanks! If I want to game the system, can I find a coupon to save some dollars with? Of course! With Lenovo, it used to be pretty easy to order something and find a coupon for a decent discount. Not sure they like that.

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 8

Shop Customizations

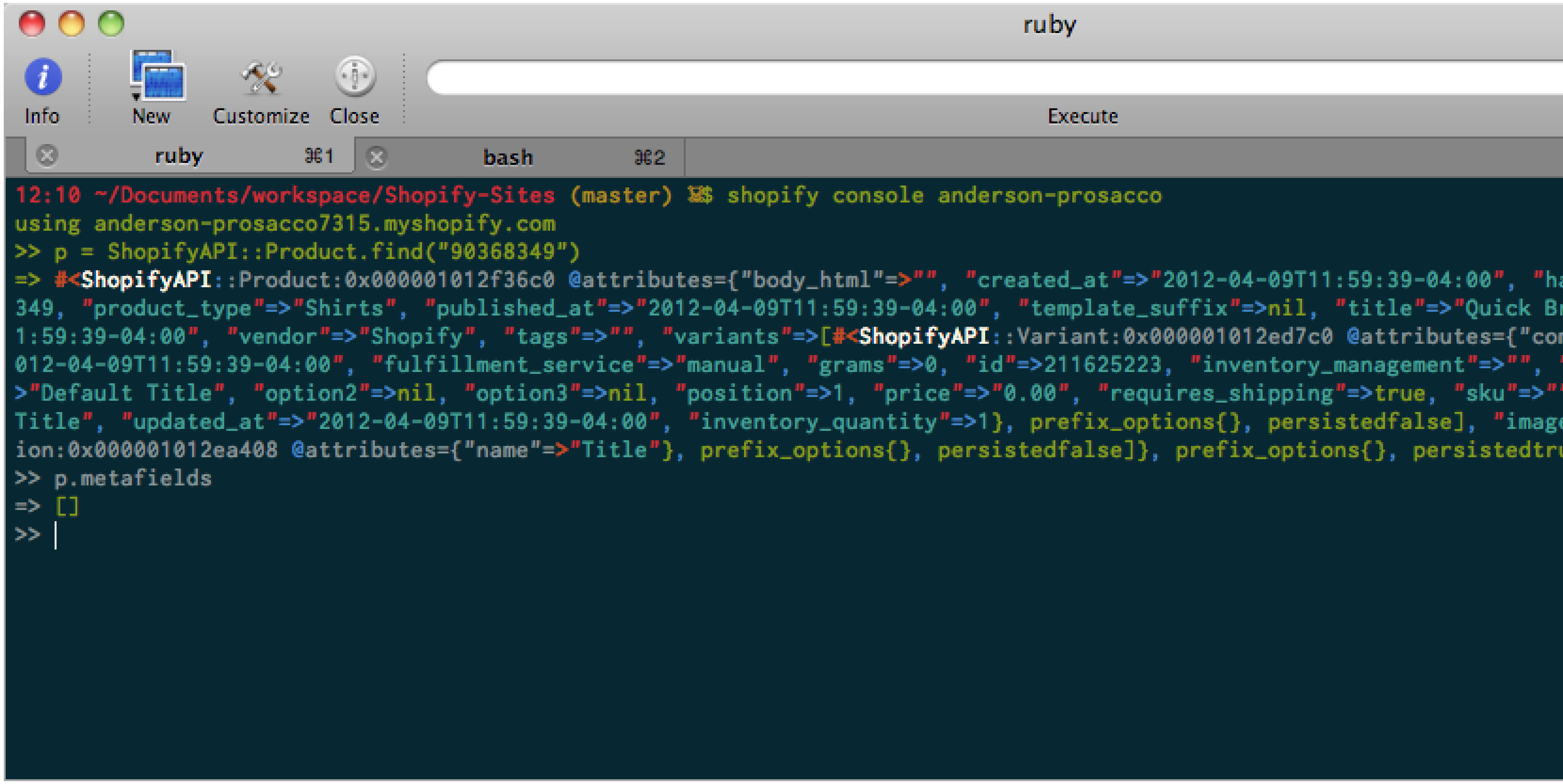
Shops are presented with the usual modern web protocols and languages such as HTML, Javascript and CSS. Shopify's templating language Liquid is crucial in allowing a shop to render it's resources where needed. If you throw in the presence of both a client-side Javascript API and the Shopify API for Apps, there are an incredible variety of ways available to customize a shop on the Shopify platform.

Chapter 9

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 little 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 entire behaviour of the sales process is in order. If the inventory is not organized correctly, too much work can be expended in a fruitless quest for more sales.

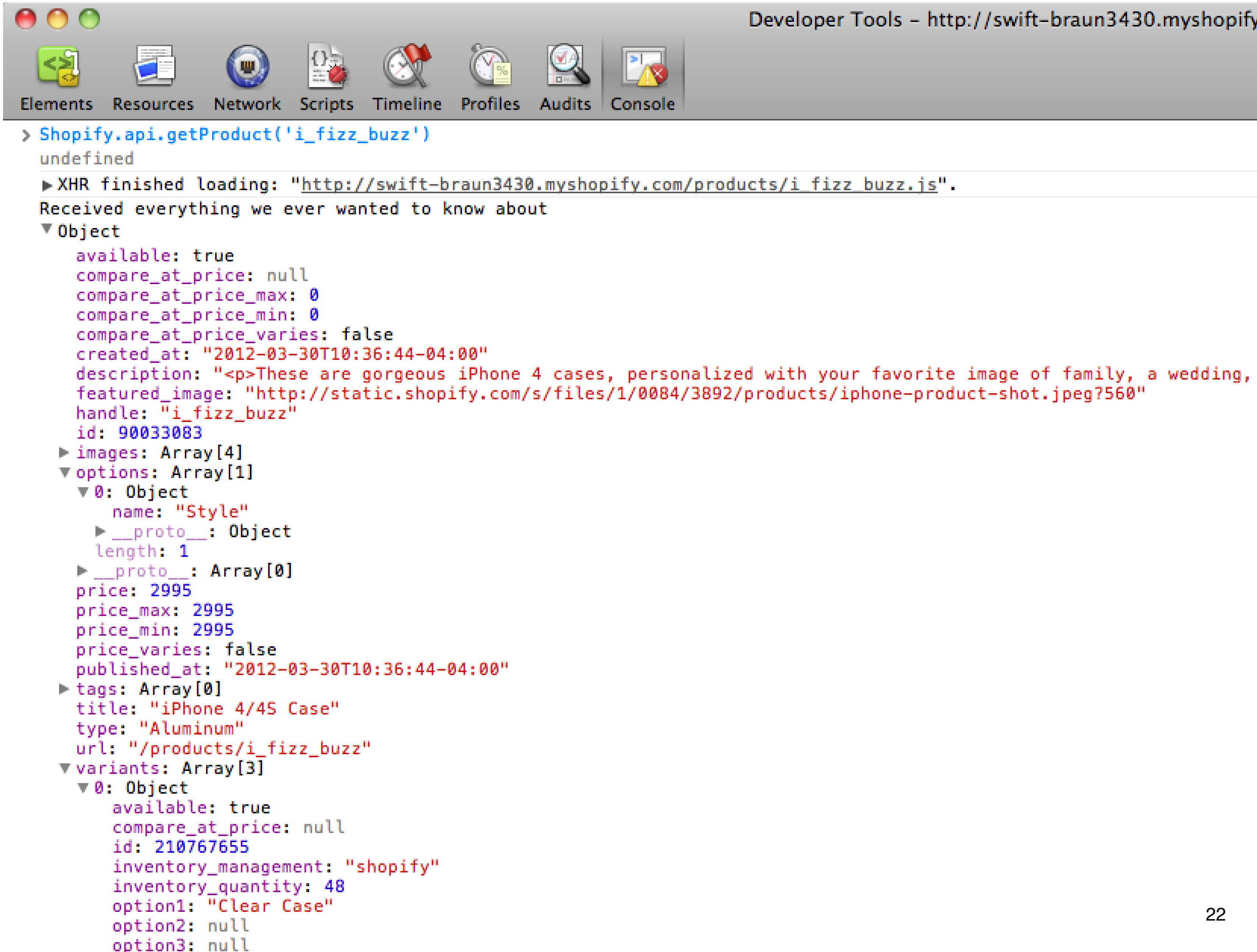
A product can be described and will require at a minimum a title. That title will then be turned into a handle that is a reference to the product in the shop. A quick example is a product with the title Quick Brown Fox. Shopify will set a handle to this product as quick_brown_fox and the so the resource will be found online as the shop's URL appended with /products/quick_brown_fox. Without further action a product could be created and added to inventory that would have a zero dollar price, infinite inventory, and no tags, images, description or other useful attributes. It would exist for the purposes of showing up on the site, and to be accessed by 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:



```
12:10 ~/Documents/workspace/Shopify-Sites (master) ⌘$ shopify console anderson-prosacco
using anderson-prosacco7315.myshopify.com
>> p = ShopifyAPI::Product.find("90368349")
=> #<ShopifyAPI::Product:0x000001012f36c0 @attributes={"body_html"=>"", "created_at"=>"2012-04-09T11:59:39-04:00", "ha
349, "product_type"=>"Shirts", "published_at"=>"2012-04-09T11:59:39-04:00", "template_suffix"=>nil, "title"=>"Quick Br
1:59:39-04:00", "vendor"=>"Shopify", "tags"=>"", "variants"=>[#<ShopifyAPI::Variant:0x000001012ed7c0 @attributes={"com
012-04-09T11:59:39-04:00", "fulfillment_service"=>"manual", "grams"=>0, "id"=>211625223, "inventory_management"=>"", "
>"Default Title", "option2"=>nil, "option3"=>nil, "position"=>1, "price"=>"0.00", "requires_shipping"=>true, "sku"=>"
Title", "updated_at"=>"2012-04-09T11:59:39-04:00", "inventory_quantity"=>1}, prefix_options{}, persistedfalse], "image
ion:0x000001012ea408 @attributes={"name"=>"Title"}, prefix_options{}, persistedfalse]}, prefix_options{}, persistedtru
>> p.metafields
=> []
>> |
```

So, with just a couple of keystrokes, we found a product and we able to see it had 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 color, 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 colors, 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.



```
> Shopify.api.getProduct('i_fizz_buzz')
  undefined
  ▶XHR finished loading: "http://swift-braun3430.myshopify.com/products/i_fizz_buzz.js".
  Received everything we ever wanted to know about
  ▼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,
     featured_image: "http://static.shopify.com/s/files/1/0084/3892/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
```

The quick 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 stores render 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 the Shop uploaded images for the product, I can 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 names, so that that name 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 color, represented by clickable color boxes. A click on any of the color boxes would be recognized as blue, red or perhaps green. We could change the background color of the main image to match. Or we could load new images of the product with "beaver faux fur" colorized 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 colors, 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 10

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 to date, a statement I believe to be true, 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 like become unstable and prone to outages, breakdowns and destroyed reputation. 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. So, by offering an API 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
```

So, 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. Now, once the fulfillment company has accepted the order(s) and sent them off to their final destination, the person would 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 inbox and sanity. The best of them will use HTTP POST to send the order with a tracking number back to the App, allowing the App to 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 80 different vendors, my life will be miserable unless I use the API. So I do. 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. I have a deal with my vendors giving them 80/20 or sometimes 60/40 splits of that profit, and it is assigned per sale too, so that we can make different deals depending on popularity at the Shopify checkout. All of that is an App that can simply be plugged into a Shop. None of that is possible without an App.

The Shopify API is a crucial variable in the equation most enterprises try and work out before going into business. If they know ahead of time 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 option that deserves close attention. Adding an App can add a monthly or one-time fee, but 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 might result in a steady 15% growth in sales. Many companies are looking for a reliable hosted e-commerce platform and a 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 11

How Should I Handle Webhook Requests?

This is a big one. As explained above, if Shopify doesn't receive a response within 10 seconds of sending a webhook, it assumes there's a problem and marks it as failed.

A common cause for this is an app that does some processing when it receives a webhook request before responding. In the normal web-app world this is desired as you need to send data back to the user. When processing webhooks on the other hand, all that's needed is a quick 200 OK response that acknowledges receipt of the data.

Here's some pseudocode demonstrating what I mean:

func handle_webhook(request) process_data(request.data) respond(200) end To make sure that you don't accidentally run over the timeout limit, you need to defer any processing processing until after the response has been sent. In Rails, Delayed Jobs are perfect for this.

Here's how your code should look:

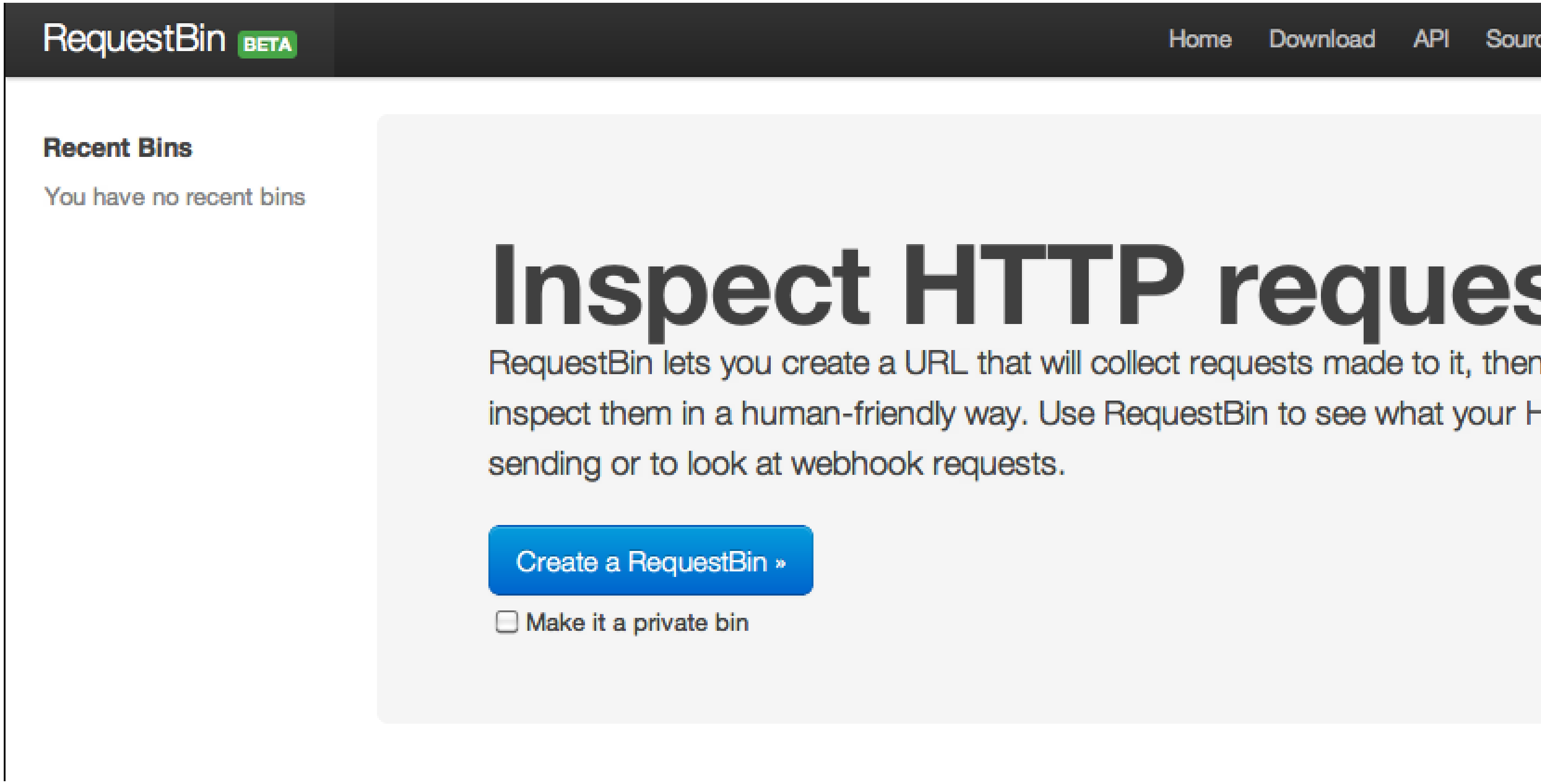
func handle_webhook(request) schedule_processing(request.data) respond(200) end Even if you're only doing a small amount of processing, there are other factors to take into account. On-demand services such as Heroku or PHPFog sometimes need to spin up a new node to handle the request, and this action can take several seconds. Even if your app is only spending five seconds processing data it'll still 'fail' if the underlying server took six seconds to start up.

The interesting world of WebHooks

Shopify does a fine job of introducing Webhooks, and indeed there are some pretty nifty use cases they provide as well. They have a best practices too, which needs to be digested. In my experiences with Webhooks I have run into all sorts of interesting issues, so I will dedicate some time and space to explaining them in my own way. First you should really go through the Shopify explanations available here

Shopify WebHooks Documentation

First off, when you are dealing with Shopify Webhooks you are in the Email & Preferences section of a Shopify Admin site. You can setup a webhook very simply there. Just pick the type of webhook you want to respond and type in a URL pointing at the receiver. For the purposes of those without an App to quickly hook up to a shop, there are some nifty webhook testing sites out there. Let's take one quick example with RequestBin. The first thing I will do is create a WebHook listener at the Request Bin website.


RequestBin BETA

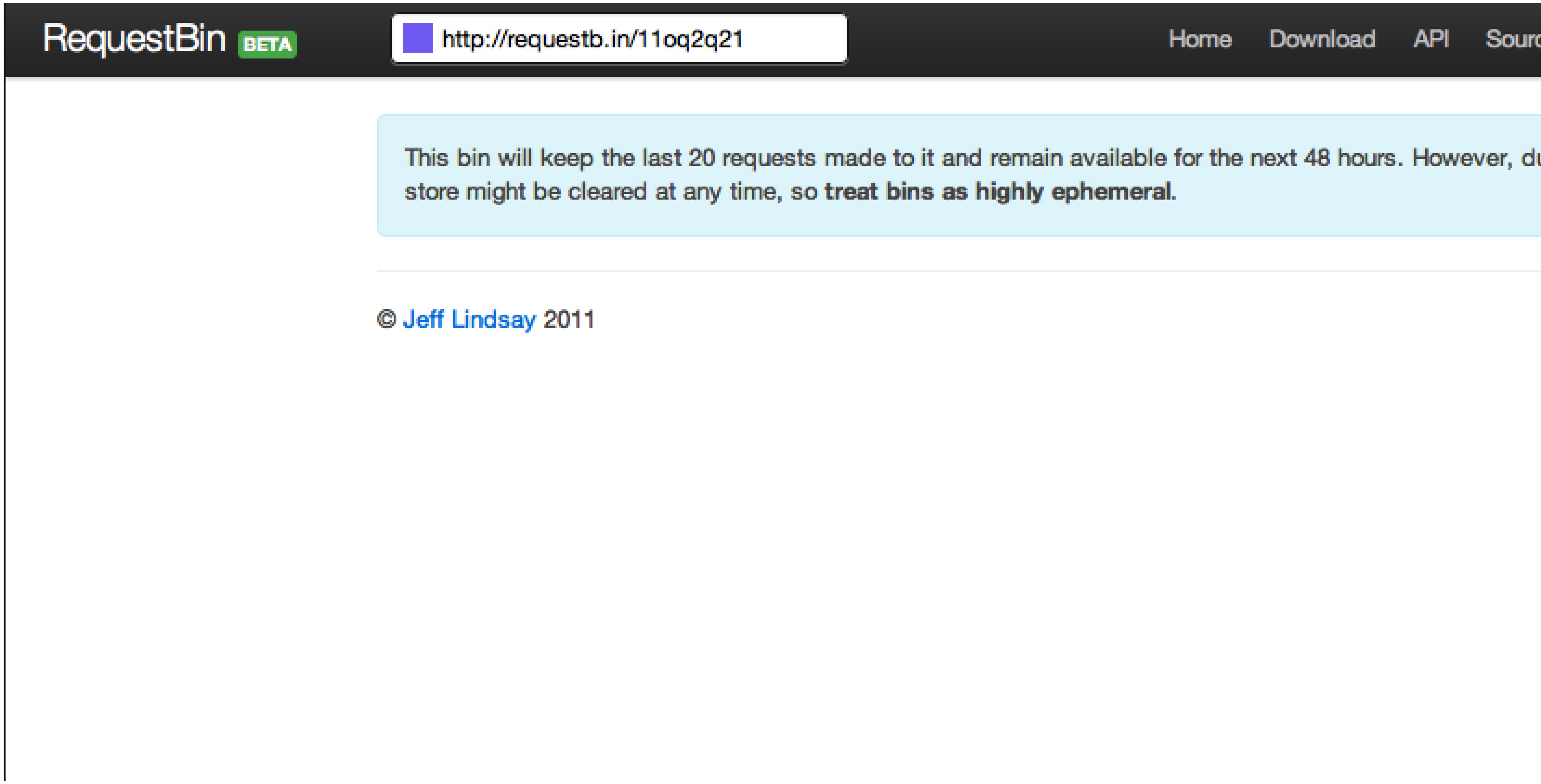
Recent Bins

You have no recent bins

Inspect HTTP reques

RequestBin lets you create a URL that will collect requests made to it, then
inspect them in a human-friendly way. Use RequestBin to see what your H
sending or to look at webhook requests.

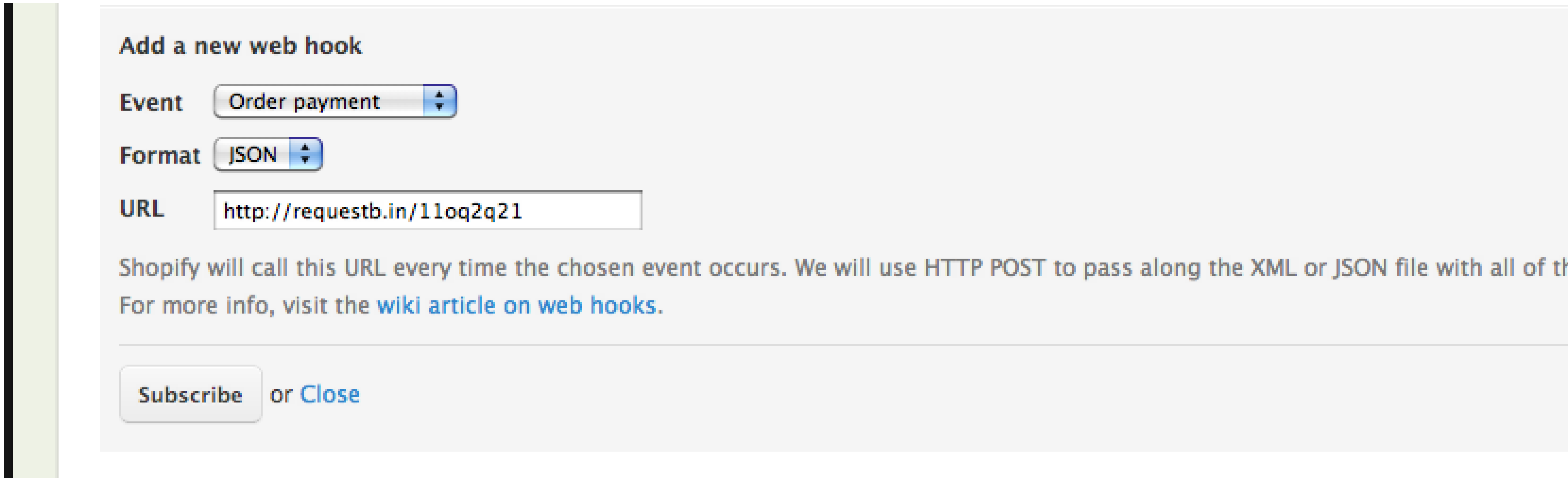
Pressing the Create a RequestBin button creates a new WebHook listener for me. I am assigned a URL that I can use for testing. Note that I can also make this private so that only I can see the results of WebHooks sent to the RequestBin.



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

© Jeff Lindsay 2011

My listener is now the nice simple URL that I can copy into the Shopify Webhook creation form at my Shop’s Email & Preferences administration section. http://www.postbin.org/155tzv2 where the code 155tzv2 was generated just for me. Flipping over to my Shopify Admin, I can create the webhook I want to test, now that I know where to send it.



Add a new web hook

Event Order payment

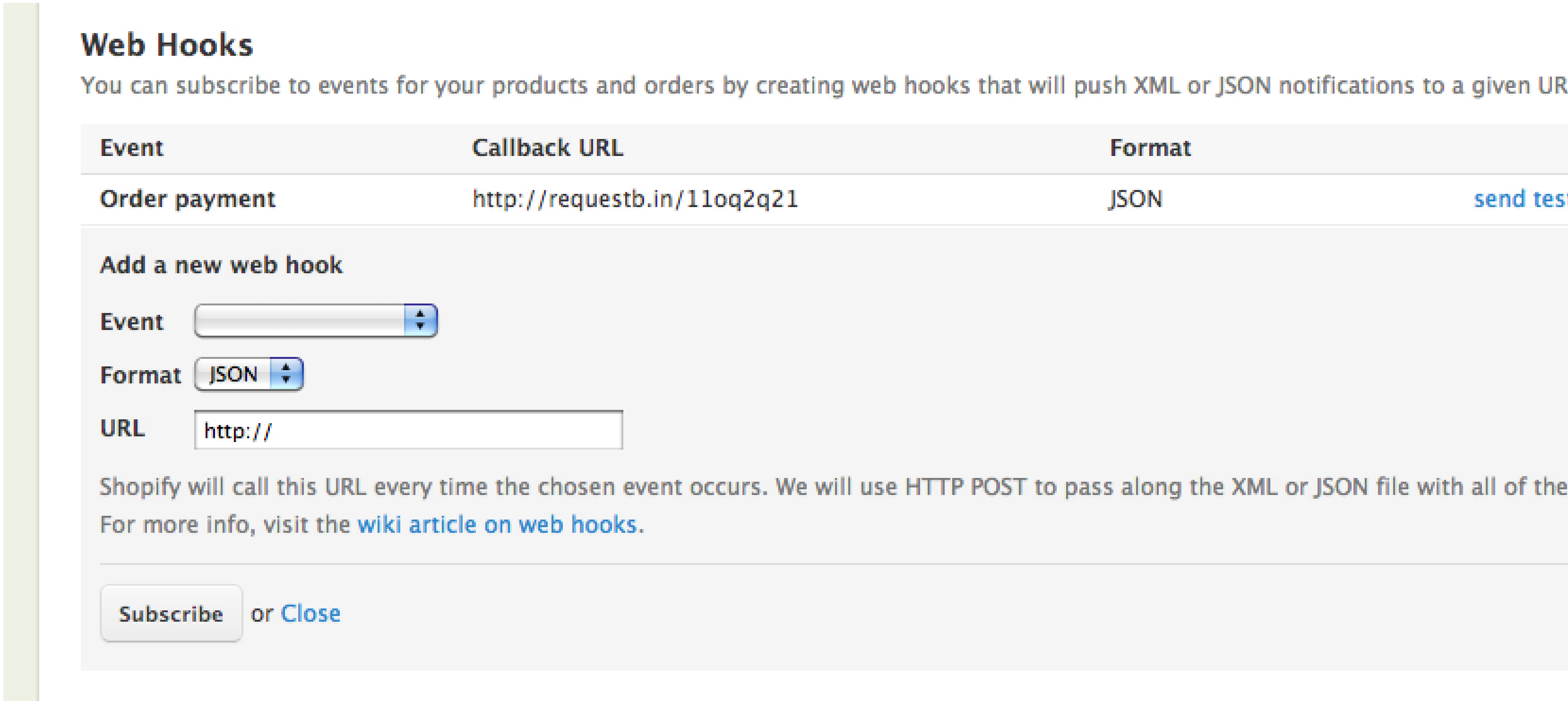
Format JSON

URL http://requestb.in/11oq2q21

Shopify will call this URL every time the chosen event occurs. We will use HTTP POST to pass along the XML or JSON file with all of th

For more info, visit the wiki article on web hooks.

Once created, I can now send a webhook to this little service any time I want by clicking on the send test notification link and standing by for a confirmation that it was indeed sent.



Web Hooks

You can subscribe to events for your products and orders by creating web hooks that will push XML or JSON notifications to a given UR

Event	Callback URL	Format	
Order payment	http://requestb.in/1loq2q21	JSON	send tes

Add a new web hook

Event:

Format: JSON

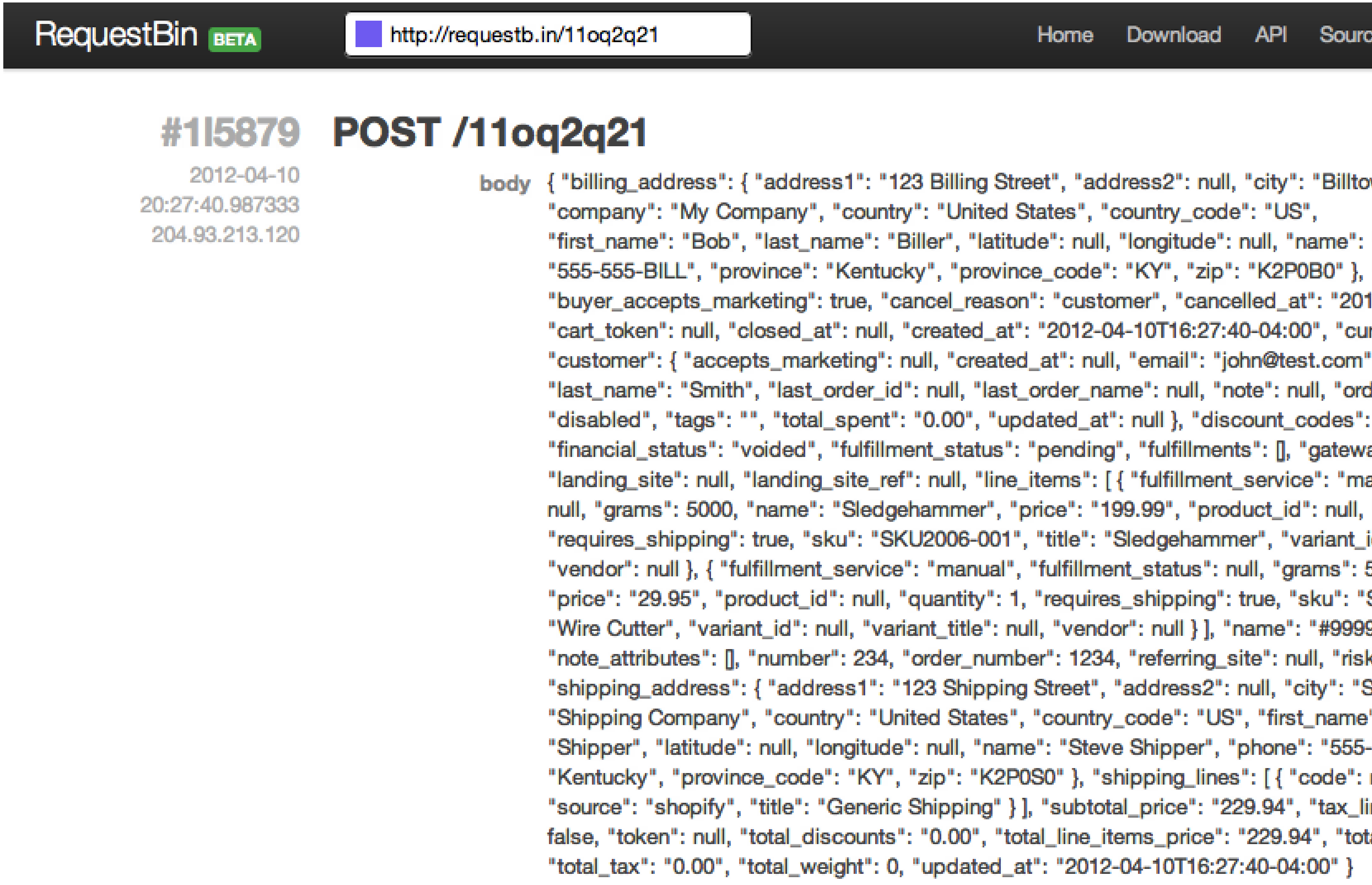
URL: http://

Shopify will call this URL every time the chosen event occurs. We will use HTTP POST to pass along the XML or JSON file with all of the

For more info, visit the wiki article on web hooks.

The ability to delete a WebHook as well as test it in the Shopify Admin interface 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

2012-04-10
20:27:40.987333
204.93.213.120

body
{ "billing_address": { "address1": "123 Billing Street", "address2": null, "city": "Billto
"company": "My Company", "country": "United States", "country_code": "US",
"first_name": "Bob", "last_name": "Biller", "latitude": null, "longitude": null, "name":
"555-555-BILL", "province": "Kentucky", "province_code": "KY", "zip": "K2P0B0" },
"buyer_accepts_marketing": true, "cancel_reason": "customer", "cancelled_at": "201
"cart_token": null, "closed_at": null, "created_at": "2012-04-10T16:27:40-04:00", "cur
"customer": { "accepts_marketing": null, "created_at": null, "email": "john@test.com"
"last_name": "Smith", "last_order_id": null, "last_order_name": null, "note": null, "ord
"disabled", "tags": "", "total_spent": "0.00", "updated_at": null }, "discount_codes":
"financial_status": "voided", "fulfillment_status": "pending", "fulfillments": [], "gatewa
"landing_site": null, "landing_site_ref": null, "line_items": [{ "fulfillment_service": "ma
null, "grams": 5000, "name": "Sledgehammer", "price": "199.99", "product_id": null,
"requires_shipping": true, "sku": "SKU2006-001", "title": "Sledgehammer", "variant_i
"vendor": null }, { "fulfillment_service": "manual", "fulfillment_status": null, "grams": 5
"price": "29.95", "product_id": null, "quantity": 1, "requires_shipping": true, "sku": "S
"Wire Cutter", "variant_id": null, "variant_title": null, "vendor": null }], "name": "#9999
"note_attributes": [], "number": 234, "order_number": 1234, "referring_site": null, "risk
"shipping_address": { "address1": "123 Shipping Street", "address2": null, "city": "S
"Shipping Company", "country": "United States", "country_code": "US", "first_name"
"Shipper", "latitude": null, "longitude": null, "name": "Steve Shipper", "phone": "555-
"Kentucky", "province_code": "KY", "zip": "K2P0S0" }, "shipping_lines": [{ "code":
"source": "shopify", "title": "Generic Shipping" }], "subtotal_price": "229.94", "tax_li
false, "token": null, "total_discounts": "0.00", "total_line_items_price": "229.94", "tot
"total_tax": "0.00", "total_weight": 0, "updated_at": "2012-04-10T16:27:40-04:00" }

Looking closely at the sample order JSON 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 have to agree 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 oreders. 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:

1. Ensure the WebHook order data actually came from Shopify, and that you have the Shop identification to work on.
2. Ensure you do not already have this Order processed as it makes no sense to process a PAID Order two or more times.
3. You know you need to parse out the credit card used, and the shipping charges, and the discount codes used if any.
4. 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's 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 secure exchanges 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 App requesting that the WebHook be created, so Shopify in turn grabs the shared secret associated with the App and makes it available to the WebHook. Before Shopify sends off a real WebHook, it will be able to use the shared secret to compute a Hash of the WebHook payload and embed this in the WebHook 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 of 200 OK. If that response is not received, the same 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 that WebHook to the App. Of course, the ten seconds is not practical, so the time 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 duplicate WebHooks. 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 events, 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 so-called duplicate is issued and now the App might have a problem.

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, if that Orders is detected by the WebHook processing paid orders, it can be ignored as a duplicate. This is not really a terribly 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 more robust way to handle this is for an App to take advantage of a Message Queue (MQ) service. All incoming WebHooks can be directed to a MQ. Once an incoming WebHook is successfully added to the

MQ, 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 really makes no difference as the MQ welcomes any and all WebHooks, duplicates or not.

Having now built our App to direct all incoming WebHooks to an MQ, a process has to be used that pops off any new WebHooks in the MQ for processing. There is no longer any concern over processing speed and the App can now do all the sophisticated processing it needs to do. We can now be certain if we have seen an orders/paid WebHook before 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 to process WebHooks and so JSON is a natural fit for those applications too. 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 one they prefer.

Cart Customization

Without a doubt one of the most useful but also the most difficult aspects of front-end Shopify development is in the use of the cart note and cart attribute features. They are the only way a Shop can collect non-standard information directly from a Shop customer. Any monogrammed handbags, initialed wedding invitations, engraved glass baby bottle etc. will have used the cart note or cart attributes to capture this information as 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 feature of Shopify 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 12

Command Line Shopify API

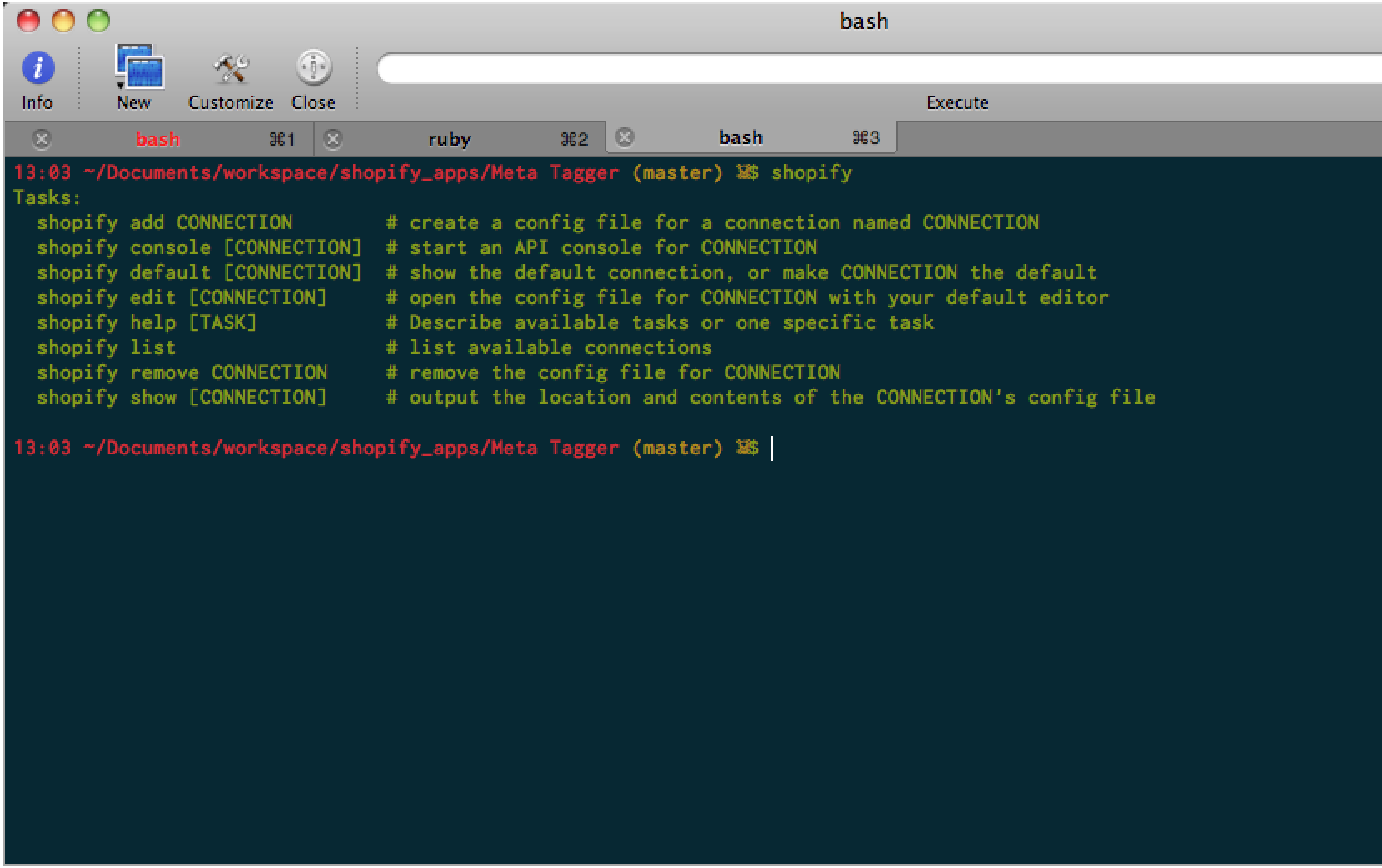
For the pusposes of quickly developing an App being able to fire off requests to a Shop and process the responses is essential. While WebHooks are a source of incoming data from a Shop, they offer only a small subset of interesting possibilities compared to what is available through the API.

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.

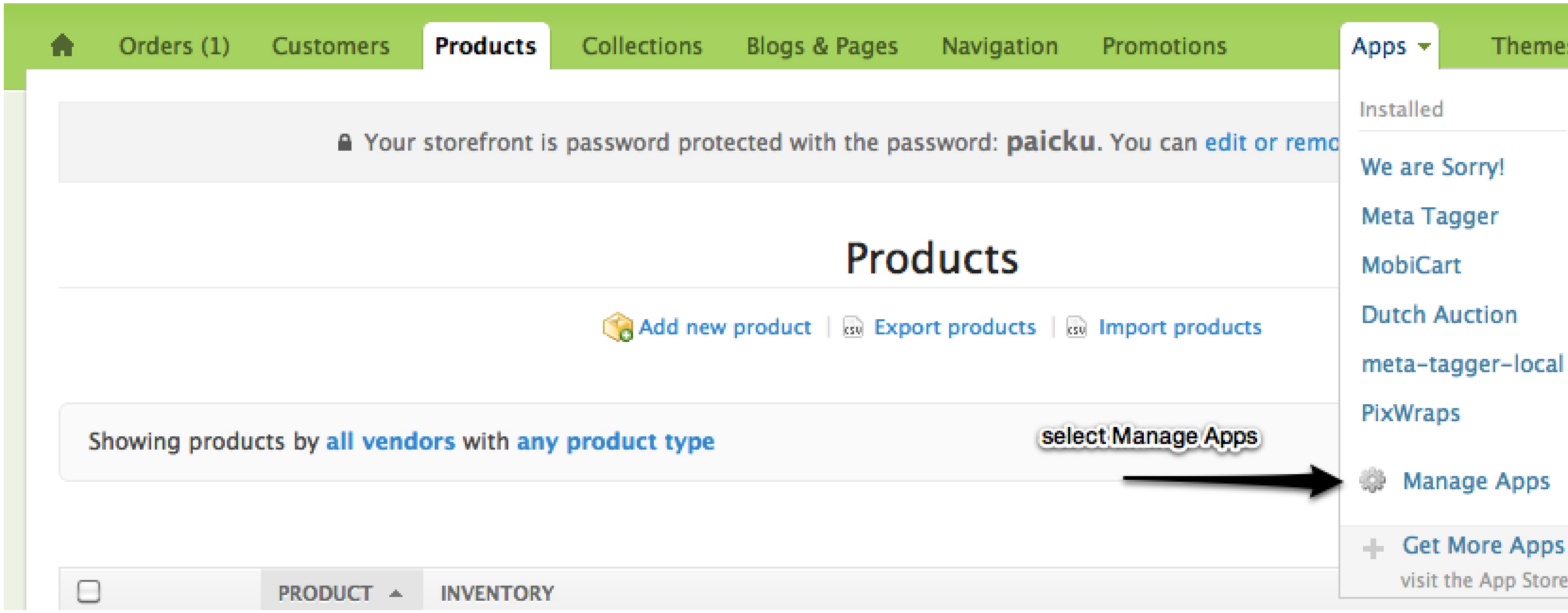
When I use Ruby, and I check my installed gems using the gem list command, I see the shopify api gem in my list and I can test for the Shopify CLI by simply typing in the command shopify.



```
13:03 ~/Documents/workspace/shopify_apps/Meta Tagger (master) ⌘$ shopify
Tasks:
  shopify add CONNECTION          # create a config file for a connection named CONNECTION
  shopify console [CONNECTION]    # start an API console for CONNECTION
  shopify default [CONNECTION]    # show the default connection, or make CONNECTION the default
  shopify edit [CONNECTION]       # open the config file for CONNECTION with your default editor
  shopify help [TASK]             # Describe available tasks or one specific task
  shopify list                    # list available connections
  shopify remove CONNECTION       # remove the config file for CONNECTION
  shopify show [CONNECTION]       # output the location and contents of the CONNECTION's config file

13:03 ~/Documents/workspace/shopify_apps/Meta Tagger (master) ⌘$ |
```

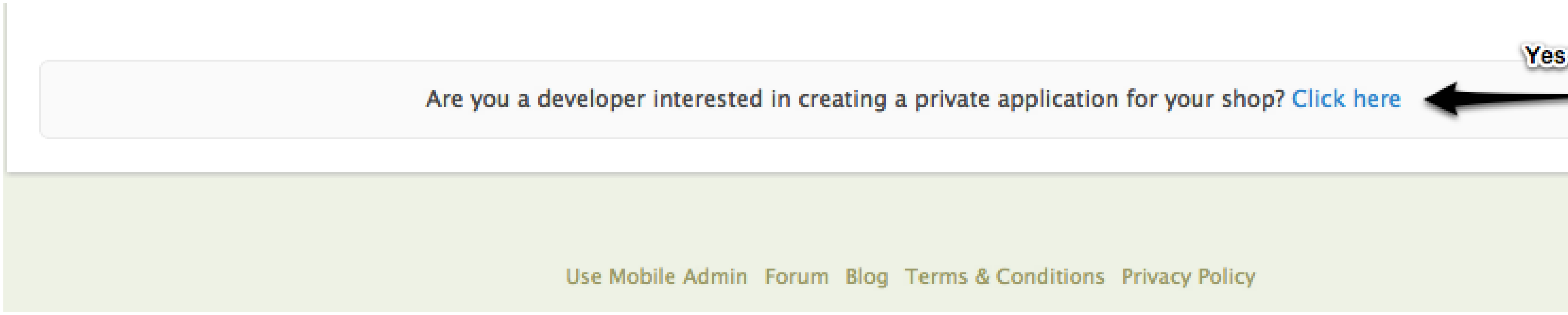
There are just a few options listed and in order to really cook up some interesting examples we will use the configure option that comes with the command. The options required to configure a shopify session are the Shops API key and a password. To get those values, we will use the Shop itself. For some 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.



Your storefront is password protected with the password: **paicku**. You can edit or remo

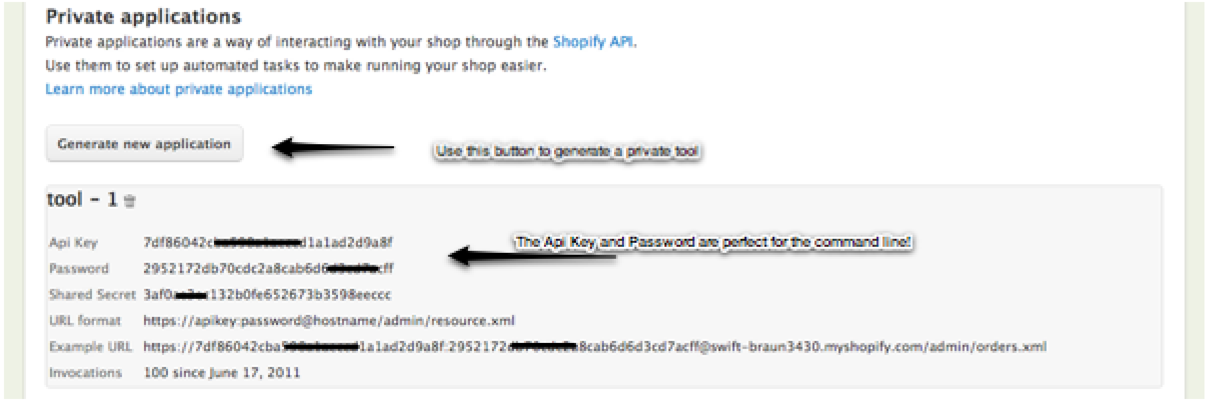
Products

Showing products by all vendors with any product type



Are you a developer interested in creating a private application for your shop? Click here

Use Mobile Admin Forum Blog Terms & Conditions Privacy Policy



Private applications

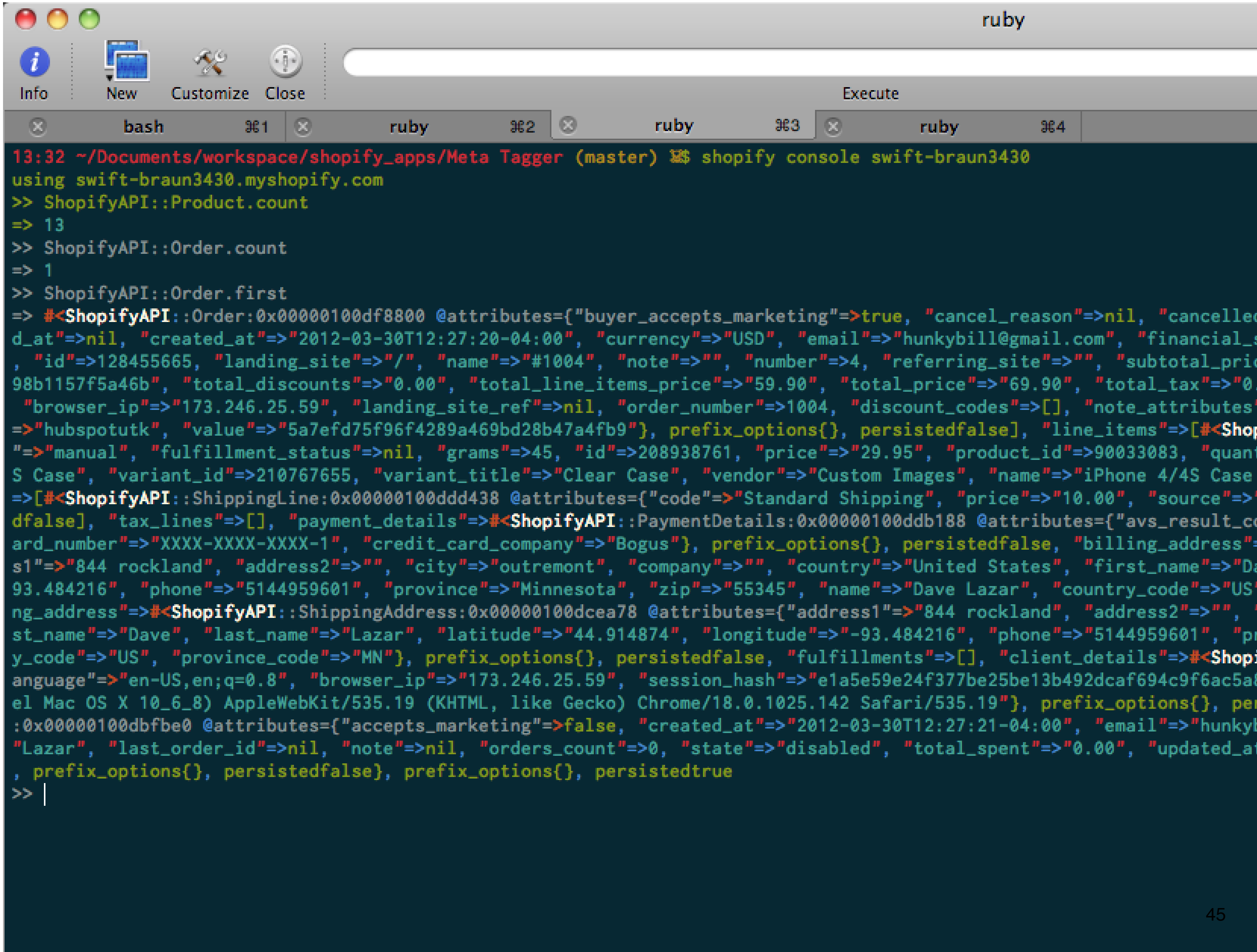
Private applications are a way of interacting with your shop through the Shopify API.
Use them to set up automated tasks to make running your shop easier.
Learn more about private applications

tool - 1

Api Key	7df86042c[redacted]1a1ad2d9a8f
Password	2952172db70cdc2a8cab6d6[redacted]cff
Shared Secret	3af0[redacted]132b0fe652673b3598eeccc
URL format	https://apikey:password@hostname/admin/resource.xml
Example URL	https://7df86042cba[redacted]1a1ad2d9a8f:2952172[redacted]8cab6d6d3cd7acff@swift-braun3430.myshopify.com/admin/orders.xml
Invocations	100 since June 17, 2011

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.



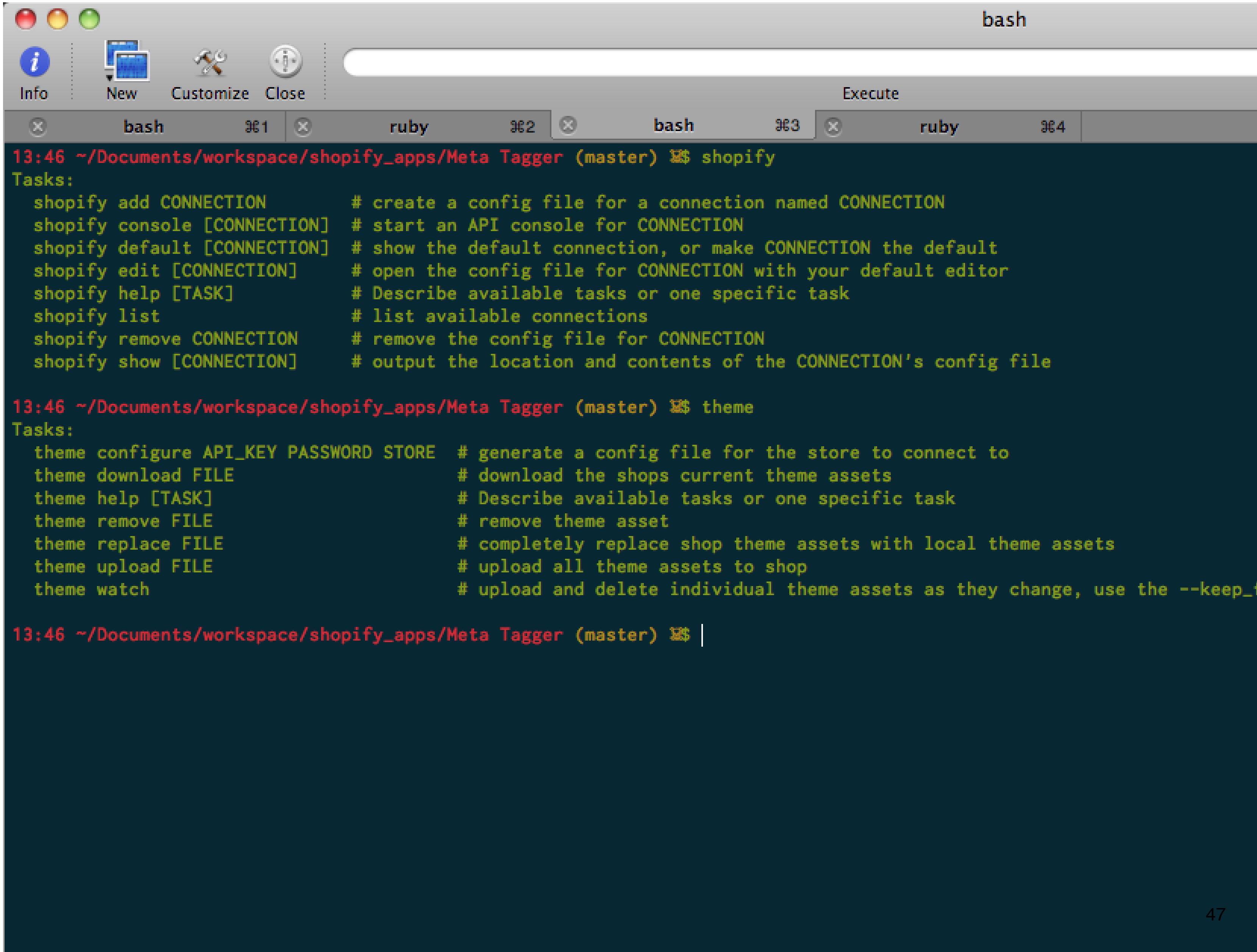
```
13:32 ~/Documents/workspace/shopify_apps/Meta Tagger (master) ⌘$ shopify console swift-braun3430
using swift-braun3430.myshopify.com
>> ShopifyAPI::Product.count
=> 13
>> ShopifyAPI::Order.count
=> 1
>> ShopifyAPI::Order.first
=> #<ShopifyAPI::Order:0x00000100df8800 @attributes={"buyer_accepts_marketing"=>true, "cancel_reason"=>nil, "cancelle
d_at"=>nil, "created_at"=>"2012-03-30T12:27:20-04:00", "currency"=>"USD", "email"=>"hunkybill@gmail.com", "financial_
, "id"=>128455665, "landing_site"=>"/", "name"=>"#1004", "note"=>"", "number"=>4, "referring_site"=>"", "subtotal_pric
98b1157f5a46b", "total_discounts"=>"0.00", "total_line_items_price"=>"59.90", "total_price"=>"69.90", "total_tax"=>"0.
 "browser_ip"=>"173.246.25.59", "landing_site_ref"=>nil, "order_number"=>1004, "discount_codes"=>[], "note_attributes"
=>"hubspotutk", "value"=>"5a7efd75f96f4289a469bd28b47a4fb9"}, prefix_options{}, persistedfalse], "line_items"=>[#<Shop
"=>"manual", "fulfillment_status"=>nil, "grams"=>45, "id"=>208938761, "price"=>"29.95", "product_id"=>90033083, "quan
S Case", "variant_id"=>210767655, "variant_title"=>"Clear Case", "vendor"=>"Custom Images", "name"=>"iPhone 4/4S Case
=>[#<ShopifyAPI::ShippingLine:0x00000100ddd438 @attributes={"code"=>"Standard Shipping", "price"=>"10.00", "source"=>
dfalse], "tax_lines"=>[], "payment_details"=>#<ShopifyAPI::PaymentDetails:0x00000100ddb188 @attributes={"avs_result_c
ard_number"=>"XXXX-XXXX-XXXX-1", "credit_card_company"=>"Bogus"}, prefix_options{}, persistedfalse, "billing_address"=
s1"=>"844 rockland", "address2"=>"", "city"=>"outremont", "company"=>"", "country"=>"United States", "first_name"=>"D
93.484216", "phone"=>"5144959601", "province"=>"Minnesota", "zip"=>"55345", "name"=>"Dave Lazar", "country_code"=>"US
ng_address"=>#<ShopifyAPI::ShippingAddress:0x00000100dcea78 @attributes={"address1"=>"844 rockland", "address2"=>"", 
st_name"=>"Dave", "last_name"=>"Lazar", "latitude"=>"44.914874", "longitude"=>"-93.484216", "phone"=>"5144959601", "p
y_code"=>"US", "province_code"=>"MN"}, prefix_options{}, persistedfalse, "fulfillments"=>[], "client_details"=>#<Shop
anguage"=>"en-US,en;q=0.8", "browser_ip"=>"173.246.25.59", "session_hash"=>"e1a5e59e24f377be25be13b492dcaf694c9f6ac5a
el Mac OS X 10_6_8) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.142 Safari/535.19"}, prefix_options{}, pe
:0x00000100dbfbe0 @attributes={"accepts_marketing"=>false, "created_at"=>"2012-03-30T12:27:21-04:00", "email"=>"hunky
"Lazar", "last_order_id"=>nil, "note"=>nil, "orders_count"=>0, "state"=>"disabled", "total_spent"=>"0.00", "updated_a
, prefix_options{}, persistedfalse}, prefix_options{}, persistedtrue
>> |
```

Now we can really test out potential code quickly without incurring much overhead at all. Any kind of code supported by the API can be tested with the fewest possible keystrokes and minimum effort. For example, we know we can access Metafields for a shop just by providing an ID for the resource but what is the real 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 this little utility, we can add connections to as many Shops as we want, and we can quickly list them all. So when working on a client shop, one of the first things I would do is use the Admin to create a private Tool. With the API key and Password, I can thus use the Shopify console tool to quickly test out any custom queries I may want to build into an App for the client.

The second very useful command-line tool that Shopify provides is not in the shopify_api gem, but is a separate gem called shopify_theme. Once this little gem installed, you can use the command theme at the command-line. Theme is a wonderful developer CLI. The first thing I do with a new client is create a directory to hold all the files of the client's theme. Once I have this directory in place, I can switch into that directory and begin work on the client site.

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



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

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

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

You can see from the listing of available options that the theme commmand is slightly different from the shopify command. Nonetheless it uses the exact same API key and Password to setup a shop. Once we provide those and the configuration is written out as a file, we can being work. I like to first download the client's entire theme into the working directory so that I can examine their Javascript, and Liquid assets.

```
        $ theme download
```

Once the assets are all downloaded, the first thing to do is setup all the files under version control. When I started programming there was no such thing as version control software, and even into the 1990's there was not much to write home about. Microsoft never offered anything of much use as Visual Source Safe was notorious for bugs that lost code, and CVS was just plain bad. SVN was intended to fix all the problems with CVS, and it works, but it has pretty serious limitations too. Open Source has promoted alternatives like Git, Mercurial and Bazaar and these days, it seems like Git is the choice for many people. So, I use Git. I make a local repository and store all the client's code under version control for safety.

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

Now I am ready to code. As a quick example, I could fix a bug in a client shop in their Javascript, and improve the site just like that. Before I do anything, I set the theme command to watch the directory for any changes. As soon as a change is registered to a file, for example I saved a line of code, the theme watcher will transmit the changes to the client site. I can quickly verify my edit worked (or not) using the browser, and continue in the fashion. When I am happy a small change is a good change, I simply repeat my previous command with a new message.

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

Now I am done. Wonderful. I have their code under version control, and I can make all my edits using my favorite code editing tools. If in two weeks time, the client wants more work done, can simply use the theme download tool to grab any new files or changes to files. Git will tell me what those are, and I can investigate.

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.