

COURSE CONTENTS

- [Introduction](#)
- Lesson 1: Getting Started with Shopify Themes
- Lesson 2: Development Tools and Workflow
- Lesson 3: Principles and Process of Theme Design
- Lesson 4: Designing Layout and Navigation
- Lesson 5: Designing Collection Pages
- Lesson 6: Designing Product Pages
- [Lesson 7: Designing Cart and Checkout Pages](#)
- Lesson 8: Designing Home and Content Pages
- Lesson 9: SEO and Social Integrations
- Lesson 10: Theme Settings
- Lesson 11: Locales and Currencies
- [Lesson 12: Performance](#)
- Lesson 13: The Shopify Theme Store
- Lesson 14: Open Source, Shopify Themes and Community

INTRODUCTION

I started tinkering around with Shopify in 2006 when I was at university.

Back then, the platform was still in the scrappy early stages – the feature set was limited, APIs were undocumented, and the default themes you had to choose from when setting up your store were... well, let's just say there weren't going to win many design awards. Getting stuff to work was often a matter of trial, error and swearing at Liquid code. If you were lucky, Tobi Lütke (then and now Shopify's CEO) would swing by and offer a suggestion in the support forums.

Skipping forward to today, Shopify's documentation and ecosystem has improved a lot, to the point that many designers and developers can make a full-time living swearing at Liquid code.

I'm one of them, and I'm proud to say that the themes and applications I've built on top of the Shopify platform have helped my clients generate millions of dollars in revenue and keep their customers happy.

This course is an attempt to help other people to build successful Shopify themes by sharing what I've learnt.

What this course covers

This course isn't about giving you advanced web development or Photoshop skills. If that's your goal, there are plenty of people out there that can teach that

better than me (I'm at the "knows enough to be dangerous" level in Photoshop, and am constantly looking up the reference for any programming language I'm working with).

The thing is, building successful ecommerce sites – and hence, building successful Shopify themes – is very often about the application of simple ideas, rather than being able to employ the latest web whiz-bangery.

This course is about those simple ideas: the principles and techniques that can be applied when building Shopify themes to make them better. To avoid discussing all of this in a vacuum, the course contains lots of real-world examples and practical techniques.

Knowing and applying the things you learn here will give you the ability to think critically about the design of your themes and deliver more value to clients and customers.

Now, the disclaimer.

As nice as it would be for there to be hard and fast rules around this sort of thing, reality dictates that the decisions we make when building a theme are going to be pretty contextual.

If you think any of the advice in this course is off-base or inapplicable to your situation, use common sense and ignore it when necessary (hard to believe, but I do get stuff wrong). Even better – get in touch and let me know why it doesn't apply in your situation and I can update the material to help others!

Who this course is for

As I mentioned, this course doesn't require advanced design or development skills, but it's going to be handy if you know your way around HTML, CSS and Javascript. Prior experience building Shopify themes isn't required - we're going to be starting from absolute scratch in the first lesson.

If you've already got some Shopify knowledge under your belt (or even if you're a full-blown Shopify Expert), there's still plenty of useful material for you in here as well. Even after years of working with Shopify, I still find little surprises, or come across new and interesting development techniques. (Writing this course has, in fact, hammered home just how much more there is to learn!)

Windows + the command line

In Lesson 2, we will be diving in to the (OS X) command line a little bit. If you're not familiar with the command line or are developing with a Windows machine, don't let that scare you off!

All of the commands, programs and advice covered in this course should be applicable across all platforms - the particular commands may just be a little different.

If you're running in to difficulties (either because the command line instructions aren't clear or you're on Windows), please get in touch with me and I'll step you through your problem. Doing this will not only help you, but also other readers, as I'll add what we learn together

to the course material.

Getting help

If at any stage you feel like you're out of your depth, I'd recommend as a first step to swing by Shopify's excellent walkthroughs on the [documentation pages](#).

I'm always available to answer questions you might have or clarify anything in the course. Contact details are at the end of this introduction.

How this course is structured

Each lesson in this course can stand pretty much on its own, although obviously lessons refer to each other as appropriate. It makes sense to read through the lessons in order, but after that you should be able to come back to each chapter individually as needed.

Each lesson begins with a list of goals and an introduction to the topic at hand, followed by a deeper discussion. Wherever possible, I've included case studies and examples based on my actual experience developing themes for real clients and customers.

Checklists

To help you with the theme development process, the actionable items from each design-focused chapter have been extracted into a single-page PDF checklist. Checklist items are ordered by importance, so reviewing your themes against these checklists should give you an

indication of where your time will be best spent.

These checklists are bundled along with this course in the "Checklists" directory.

Expert Advice

As part of writing this course, I've been lucky enough to interview a number of Shopify experts and get their insights on the various topics covered by the course. Their advice and tips have been included within the relevant lessons.

If you've purchased the **Book + Course** or **Complete** package, you can hear their advice in full with the video interviews bundled with the course. You can access the videos from the HTML files in the "Interviews" directory.

Code Snippets

I've managed to fit most of the code snippets for this course inline with the material, but there are a few examples which are just too long for a PDF format. These are referenced in the text and are available in the "Code Snippets" directory.

Email Review and Screencasts

If you've purchased the **Book + Course** or **Complete** package, you'll be automatically subscribed to the Email Review Course, which is designed to make sure you're getting the most of out the course.

It will periodically deliver screencasts and review material

to your inbox for a short time after you purchase the course. If you'd prefer to receive the review material separately rather than via email, please contact me and let me know.

Questions? Contact me

I've learnt a lot in the process of writing this course, and I really hope you find it valuable. If you ever have any questions, or just want to shoot the breeze, please reach out – I'd love to hear from you!

You can find me on Twitter ([@gavinballard](https://twitter.com/gavinballard)) or send me an email (gavin@gavinballard.com).

LESSON 7

DESIGNING CART AND CHECKOUT PAGES

Goals for this lesson:

- understand the purpose of the cart page and common reasons for cart abandonment;
- learn how to avoid cart abandonment;
- learn design techniques to make pre- and post-checkout experiences as seamless as possible.

It's reported that an [average of 68%](#) of online shopping carts are abandoned before completing the checkout process. While some degree of abandonment is inevitable, stores that put in the effort to optimise the cart and checkout flow can see large improvements in their conversion rates.

Shopify theme designers can aid this optimisation for customers by providing them with timely information, avoiding common user interface mistakes, and reassuring them at every step.

The cart page

The cart page is the last one customers see before they move into the mostly-Shopify-controlled checkout pages.

The aim should be to give users a *complete picture* of their

purchase, so that they aren't met with a surprise when they move to the next step.

It should also make customers feel like they are *in control* of their cart, by allowing them to review, understand and change its contents.

Finally, it should make sure customers *know what to do next* to progress their order into the checkout.

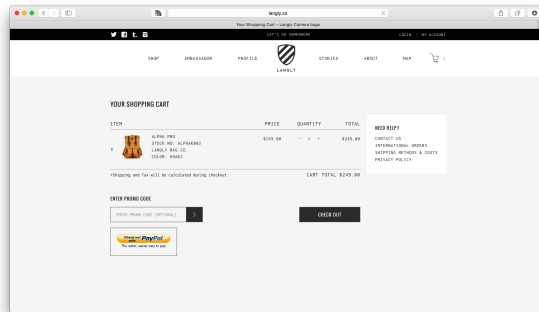
We achieve these design objectives by clearly presenting required information, removing unnecessary page elements, and providing a clear call to action.

What to display on the cart page

Obviously, all *products* currently in the cart should be displayed, along with thumbnail images. Any user-selected options (such as variant options or custom line item properties) should also be displayed.

It's very important that the displayed thumbnail matches the product the user selected, *even if the correct variant information is displayed textually next to the item*. Incorrect thumbnails cause customers to lose confidence in the correctness of their order and may cause them to abandon their cart.

Shopify now makes the linking of product variants with the correct image straightforward, and gives instructions on [how to implement variant images on cart pages](#).



Langly Co provides a simple but accurate overview of cart contents.

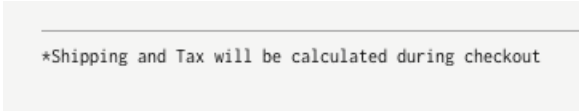
All products should link back to their full product pages, so that the customer can review its details before making a final purchase decision.

Surprises like additional shipping fees or taxes are the [number one cause](#) of cart abandonment, so if possible your cart page should **display shipping fees and taxes**. This can be difficult with Shopify, especially if your shipping and tax rates depend on the location of the customer (as you usually don't have any information about the customer quite yet).

The simplest way to resolve this problem is to prominently include shipping and tax line items on the cart page, but simply add a note indicating they'll be calculated on the next page. If using this strategy, you should also label the element displaying the total price of the order with something along the lines of “before

shipping and taxes”. This approach won’t totally alleviate the surprise of not knowing the final cost of the order but at least it will prepare the customer for additional fees.

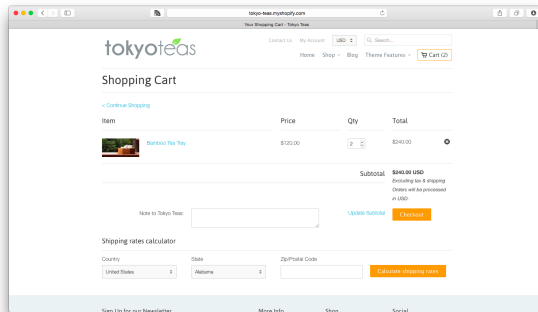
You can see that this is the approach Langly Co took if you take a closer look:

A screenshot of a light gray rectangular box with a thin horizontal line at the top. Inside the box, the text "*Shipping and Tax will be calculated during checkout" is displayed in a small, dark gray, monospaced font.

*Shipping and Tax will be calculated during checkout

Not ideal, but very simple to implement.

A better, but more involved, solution is to ask the customer for their shipping postcode and make use of [Shopify’s Ajax API](#) to fetch estimated shipping costs for the current cart contents and calculate estimated taxes.



The Mobila theme includes a shipping estimator at the bottom of the cart page.

Any **trust signals** used by your site should be reinforced on the cart page. Trust signals include SSL and security badges, payment processor icons, and “as seen in” logos. You may remember from Lesson 4 that I advocated placing these in the footer of the site — in any case, there’s no harm in emphasising these now.

The **call to action** on the cart page is the “Checkout” button, although I’d recommend labelling it something more descriptive like “Proceed to checkout” or even better, “Proceed to secure checkout”. The checkout button should be large, prominent, and not cluttered by other page elements.

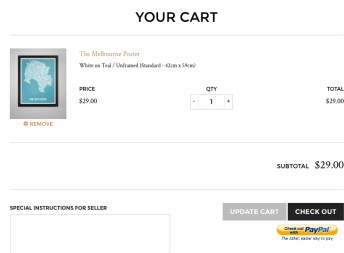
Editing cart contents

If customers can’t change the contents of their cart, you run the risk of frustrating them and losing their business

altogether.

Affordances should be give to customers to change item quantities or remove line items altogether in an efficient manner. Taking the time to add some Javascript interface functionality for this purpose is often a good idea (although don't forget to handle fallback situations).

If you're interested in easily building dynamic cart functionality for Shopify themes, check out my [CartJS](#) library. It's an open source project which aims to make building things like cart editors much, much simpler.



This “Update Cart” button right next to the Checkout button is a potential UX problem.

One default edit pattern that a lot of Shopify themes ship with is a series of quantity inputs for each line item and an “Update” button at the bottom of the screen. Customers can update the quantities of the line items,

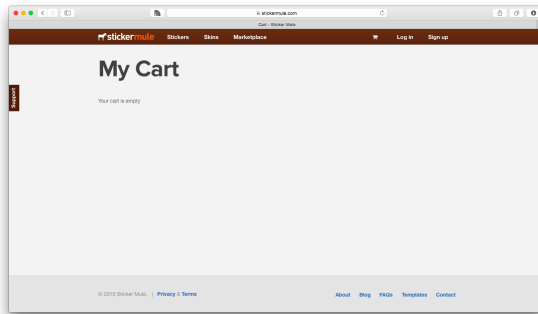
then click “Update” to submit a whole page form.

The process itself is a sensible starting point or fallback if an Ajax update solution isn’t available, but it’s unfortunate that so many themes that implement this pattern (a) label the button “Update” with no clarifying information; and (b) tend to place it directly next to the checkout button, where it competes for attention.

If you must use this pattern in your design, label the action something more descriptive (like “Update item quantities”) and ensure the button is placed well away from the checkout button.

Empty cart page

The empty cart page is an opportunity to encourage customers to explore your store and fill it up, but so many stores leave it as a blank page with a “Your cart is empty.” message.



StickerMule, I love you, but you could do more with your empty cart page.

Adding a list of featured products or a product search box with some suggested terms can help bring customers back in to shopping through the store again.

Upselling on the cart page

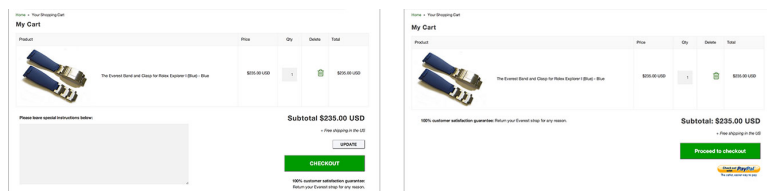
When a customer is on the cart page, the focus should be on moving them on to the checkout. However, many stores use the cart page as an opportunity to upsell or cross-sell their customers in order to generate extra revenue.

I'm generally very cautious about this, and recommend trying to upsell or cross-sell products either on the product page (see Lesson 6) or after the checkout (see below). At the very least, up- or cross-selling on the cart page should be carefully tested to ensure it's not having an overall negative effect on revenue.

Pages that do implement this well on the cart page do so in a way that doesn't take the user away from the cart page and hinder their flow (usually by adding or replacing the product using an Ajax request).

Cart page case study

If you'd like to see an excellent case study on some of the principles discussed in this lesson put to use, check out Ethercycle's post on [raising conversion rates by 133%](#) through making some simple alterations to the cart page.



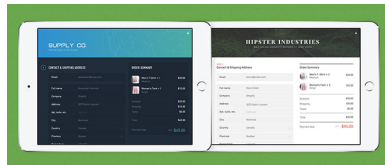
The “before” and “after” shots from the Ethercycle case study look similar but convert very differently.

Checking out

As we noted above, there's a limit to what we can control on Shopify's checkout pages (it's worth mentioning here that this limitation doesn't extend to [Shopify Plus](#) customers, who have access to a `checkout.liquid` template file).

Checkout styling

What we *can* do is customise the styling of these pages to match our theme with a `checkout.scss.liquid` file in our assets directory. Trying to do this styling to get a perfectly seamless style match between the main theme and the checkout is a difficult, fiddly, and thankless task. My recommendation is simply to make the changes needed to create a clear connection in the customer's mind between the main site and the checkout pages.



A couple of Shopify's examples of styling the new responsive checkout.

Account creation

Can you guess what the second most common cause of cart abandonment is after surprise shipping charges? That's right, [requiring customers to create an account](#) in order to check out.

- ☐ Accounts are disabled
Customers will only be able to check out as guests.
- ☒ Accounts are required
Customers will only be able to check out if they have an account created for them by you.
- ☐ Accounts are optional
Customers will have the option to create an account at the end of the checkout process.

This kills conversions.

Requiring account creation is really only appropriate when customers need to be approved before purchasing — wholesale stores, for example. For stores that wish to encourage customers to create accounts, it's better to offer that as an option *after* checkout (see the section on the Thank you page below).

Note that even if account creation isn't *required*, confusing messaging around the pages that give customers the choice between logging in or continuing as guests can have the same effect.

The image shows two versions of the ASOS login/signup form side-by-side. The left version is the older design, featuring a 'Your Account' header with a 'New Customer?' link. It has three main sections: 'Existing Customers' with email and password fields, a 'Forgot your password?' link, and a 'New Customers' section with a long paragraph explaining account benefits and a 'Create your account' link. The right version is the newer design, featuring a 'SIGN IN' header. It has a clean layout with 'Existing Customers' email and password fields, a 'Forgot your password?' link, and a 'New to ASOS?' section with a 'Continue' button. The newer design is significantly simpler and more focused on the user's choice.

ASOS saw a 50% decrease in abandonment when changing from the design on the left to the one on the right. ([Source](#))

Themes need to handle the cases when customer accounts are available, and ensuring the messaging and layout of these screens makes it clear to customers that they can continue with registration.

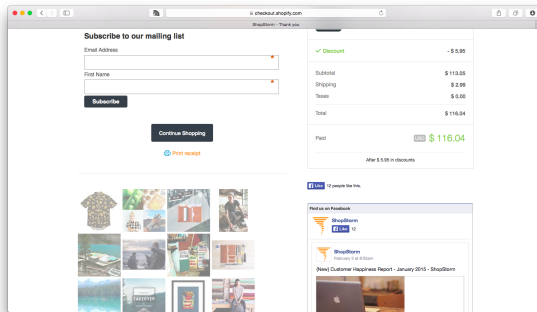
Thank you pages

Even when customers reach the thank you page at the end of the checkout process (hurrah!), your job isn't done.

Now that the high-pressure task of getting the customer through the funnel is over, you can take a bit more time to target them with content that could have been distracting at an earlier stage. Potential uses for the thank you page include:

- Encouraging customers to create an account with the details they just used for checkout;
- Asking customer to join your mailing list or take a survey, now that you have a bit of breathing room to properly explain the benefits;
- Offering discounts on upsales or cross-sales related to the products just purchased;
- Starting customer service off on a good foot by displaying a product usage and care video.

Because control of the thank you page is squarely in the hand of a store owner, these tips are really only applicable to theme developers working with clients — although there's nothing to stop you from suggesting them in your theme documentation!



This screenshot from the amusingly-named Happy Ending app shows the addition of a mailing list signup, Instagram feed and Facebook like widget to the thank you page.

LESSON 12

PERFORMANCE

Goals for this lesson:

- Learn why performance is really important for eCommerce sites and themes;
- Understand why performance is often ignored by theme developers;
- Learn about tools for measuring theme performance;
- Discover techniques to improve the performance of your themes.

Why performance is important

Back in 2009, Eric Schurman at Bing and Jake Brutlag at Google ran a [series of experiments](#) to see how their users responded to increased server delays.

The results were pretty compelling: in Google's case, an added delay of less than *half a second* led to a 0.6% drop in user engagement. Bing went even further and subjected their test users to a full 2 second delay – and saw engagement (and revenue per user) drop by over 4%.

Their conclusion:

*“Speed matters” is not just lip service... delays under half a second **impact business metrics.***

And this performance impact isn't something that's only visible in controlled lab experiments, either:

- Mozilla [shaved 2.2 seconds](#) from their page load time... and downloads increased by 15.4%.
- Barack Obama's campaign [sped their site up by 60%](#)... and donations increased by 14%.

In the context of eCommerce today, I feel that site performance is now actually more important than ever. As more and more businesses move online, consumers have more choices on where to spend their money online, and their patience for less-than-optimal purchasing experiences decreases.

Consumers are expecting a faster web. Businesses succeed with a faster web.

— Steve Souders, [How Fast Are We Going Now?](#)

It can be tempting to dismiss this type of performance concern by pointing to devices that are becoming more and more powerful, and internet connections that are offering more and more bandwidth. I think this is dangerous thinking, for a couple of reasons:

- We're seeing a massive shift to mobile devices

([50.3% of all traffic](#) to Shopify stores is on mobile devices), where processing power isn't as prevalent and 3G connections are spotty;

- There's a great deal of growth in eCommerce in developing markets, where the demand for online sales is high but internet connections are often lagging;
- Even in developed countries, it's common to be shopping online with a poor networking connection (I'm writing this right now in a Swedish cafe with pretty lacklustre WiFi);
- Absolving the developer of responsibility to performance leads to laziness and increasingly bloated sites.

Why performance gets ignored

If you're doing conversion rate optimisation for a client and get a lift of 1%, that's a huge win. You'll be popping champagne and discussing what the business is going to be doing with the tens of thousands of dollars you just made them (hopefully cutting you a bonus cheque, but that's wishful thinking).

So, considering that a poorly performing website can have a negative impact on conversion well in excess of that 1%, you'd image that performance optimisation is high on the list of priorities for theme developers, right?

Nope.

While some sites running on Shopify do take performance seriously, the vast majority of them are grossly

unoptimised – huge images, lots of HTTP requests, poorly structured pages.

When I started work on my [own theme framework](#) for Shopify, I conducted a “performance roundup” of existing themes in the Shopify Theme store. The results were pretty damning – the average theme in the store loaded **2MB** of assets with **58 HTTP requests** on page load.

There's a lot of performance optimization that people just don't bother to do in Shopify.

— Kurt Elster, *Ethercycle*

I've got a theory as to why this is so, and it's pretty simple: clients can't see it, so they won't pay for it.

Unlike all of those pretty images or snazzy Javascript features, a client can't “see” the speed of their site. They might feel something's slightly amiss when a page takes ten seconds to load, but even then they're looking at their site with the mindset of a store owner, not a potential customer.

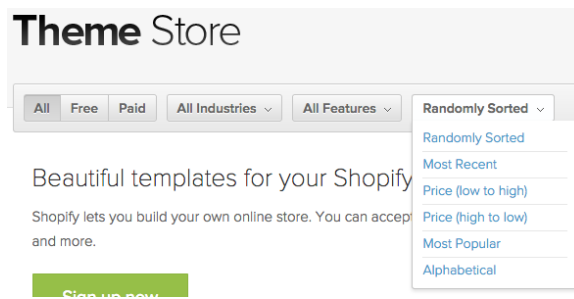
It rarely occurs to them that had the site they were loading not been their own, they would have closed the tab five seconds ago and moved on to the next result in Google.

We designers and developers aren't immune to this problem, either. When we're sitting on a high-speed desktop connection, developing locally and with a

populated browser cache, a lot of the performance issues are going to go unnoticed.

And, just like the client, our minds aren't in the same place when the page does load — we're looking for alignment issues, checking that colours match up, that the new lightbox works as expected.

When these factors combine, it makes it easy as a theme creator to let performance become a low priority.



Notably absent: "Performance".

This is especially true when building a theme for widespread sale (for example, in the Shopify Theme Store). There's no "sort by performance" option in the Theme Store listing – what's going to sell your theme are big glossy images and slick CSS animations.

Performance measuring tools

So, proceeding on the assumption that we want to be one

of the enlightened theme creators that take performance seriously, what's our first move?

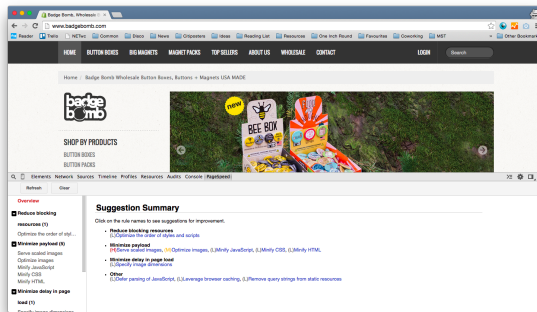
Well, like so many things, the first step in being able to improve performance is to know where you stand currently ("you can't manage what you can't measure" and all that).

There are a number of tools out there to evaluate site performance. I'll cover three of them here, but you don't need to spend too much time worrying about which one is "best". Most performance tools will cover a similar set of metrics — the key thing is to be using *something* before and after your optimisations in order to judge the effect and success of your changes.

PageSpeed Insights

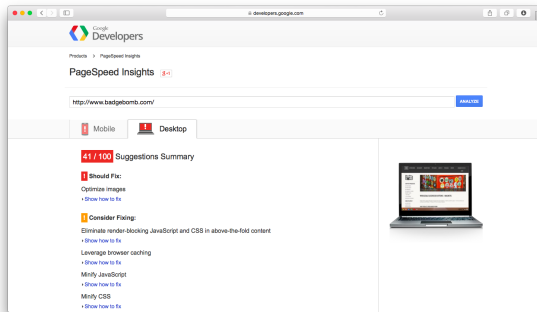
Google's PageSpeed utility is probably the most ubiquitous performance measuring tool, as it's built into the Chrome browser's developer console.

Running the PageSpeed test in Chrome ("Developer Tools" -> "PageSpeed" -> "Analyze") will give you a breakdown of the performance issues with your theme, as well as a prioritised list of what to tackle first:



Chrome's built-in PageSpeed tool.

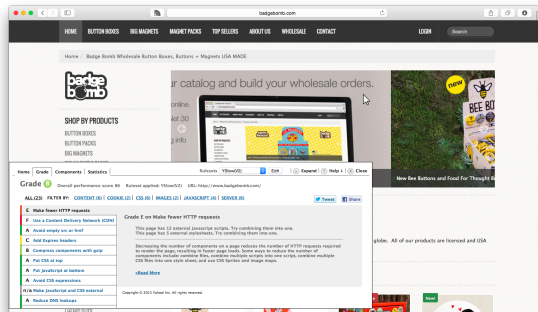
If you don't use Chrome, you can still access all of the same performance data through the online PageSpeed Insights tool. It's simply a matter of heading to <https://developers.google.com/speed/pagespeed/insights/> and entering the URL of your theme:



The online PageSpeed Insights tool. Looks like this theme's got some work to do!

Yahoo YSlow

If you've got a philosophical objection to Google, or just want to try something different, YSlow is a browser plugin available for both [Safari](#) and [Firefox](#) that offers much the same sort of analysis as PageSpeed.

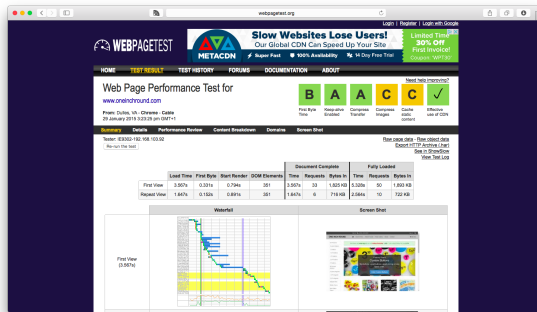


A YSlow report in Safari.

WebPageTest.org

I've only recently come across this tool, but I'm really liking it so far. Unlike the previous solutions, it's not available as a browser plugin — you simply access it at <http://www.webpagetest.org> and plug in the URL of a page to test.

It's a much more advanced tool than PageSpeed or YSlow, letting you specify a huge number of variables when running the test (browser, geographic location, connection type and speed, user agent strings... the list goes on). It also lets you record the test as a video, simulate the failure of particular domains during the page load, and gives you a detailed “waterfall” timeline of your page load.



WebPageTest.org gives you a lot of detail.

WebPageTest.org is also the only one of the tools mentioned here to analyse a site's [SpeedIndex](#), which is a way of measuring how quickly the visible parts of a page are presented to a user (often a much more important metric than the overall page load time).

Which to use?

Unless you're really itching to get into the nitty gritty of performance optimisation, I'd recommend sticking to the simpler reports and analysis of PageSpeed Insights and YSlow for the moment. They'll surface 90% of the performance issues in your theme and identify where you're going to see the biggest wins.

If you're looking for a way of integrating performance testing into your development workflow, check out the [grunt-perfbudget](#) Grunt plugin. It's written by Tim Kadlec, who has a wealth of material on setting a

“performance budget” for your site over on [his blog](#).

Improving theme performance

Being able to get all of this performance information is great, but now the question is: how do we use it? It’s good to have an idea of which things we should be worried about, and which can be safely ignored.

In the case of Shopify themes, any performance optimisations that require server or CDN configuration are out of our hands. Fortunately, Shopify does a pretty good job of optimising these things on their end, so we generally don’t have to worry about them and can focus on the things we can control.

Below, I’ve provided a list of the key metrics I look at when trying to improve my Shopify theme, along with the techniques I use to try to improve those metrics. The remainder of this lesson goes in to each of these optimisation techniques in detail.

Metric 1: Page Weight

How many bytes need to be transferred to the browser to display your site. You should be looking at two numbers here: the number of bytes transferred when the page first loads (before the browser has cached anything) and the number of bytes after caching.

Even if your site is pretty image-heavy, you really should be able to deliver a usable page to a customer with a few hundred kilobytes or less. Using some of the techniques

described in this lesson, you can do this and then pull in larger assets (like high-resolution images) later as needed.

Optimisation techniques focusing on reducing page weight include:

- image optimisation;
- asset minification;
- lazy loading;
- removing content.

Metric 2: Number of HTTP Requests

Every HTTP request your browser makes for an assets requires a round trip from browser to server and back again, along with all of the overhead that implies.

Methods focused improving the efficiency of the requests your page makes are:

- asset concatenation;
- cookieless CDN domains (handled by Shopify);
- CDN hostname distribution (not handled by Shopify, but not much we can do about it);
- more intelligent asset loading;
- removing content.

Metric 3: Time to Load

The term “time to load” is a bit imprecise, as “load” in the context of web performance can actually mean a number of different things. It could mean:

- (a) the time taken for the browser to load your initial HTML and start rendering the page (“time to render”);
- (b) the time taken for the browser to visually display the page to the user (“time to visual completion”);
- (c) the time taken for the entire page, including assets, to load, render and execute (“time to load”).

From the perspective of user experience, (b) is really the most important interpretation, as it’s what makes a site “feel” fast. Unfortunately, it’s also the most difficult to measure with automated tools (although [SpeedIndex](#), a metric developed by Google and offered by WebPageTest.org, does a pretty good job).

The good news is that the optimisation techniques for all types of page load have some overlap, so efforts towards optimising your initial time to render will help improve your other metrics as well.

All of the techniques listed below will have some effect (generally positive!) on your page load times.

Technique 1: Simplify and Remove

You know the old saying, “less is more”? Well, in the case of web performance, that’s definitely true. The “less” of your theme there is — the fewer the images, requests, page elements — the faster your page will load and display.

Sounds sensible and straightforward, right? I agree, but it never seems to quite make it into those “5 ways to speed

up your site and improve your sex life” lists on the internet.

Perhaps the reason for its absence is that it’s not always an easy “quick win”. Unlike some of the techniques we’ll be looking at in a bit, taking the time to look through your hard work for things to *rip out* is difficult, both time-wise and emotionally.

But, just like authoring good writing, authoring performant Shopify themes does require ruthless editing.

The great thing about this technique is that you get some side benefits beyond the performance implications. Just like editing writing makes things much easier for readers, editing your Shopify themes also makes things clearer and easier for your customers, as well as improving your life as a developer by reducing your maintenance burden. It’s a real win-win-win!

If you’re looking for ideas on the sort of things you can do to “edit down” your Shopify theme, here are some ideas:

- **Refactor CSS:** it’s easy to build up technical debt in your CSS files over the course of a site’s development. Refactoring and simplifying your CSS will not only save on stylesheet size, but also make the site more maintainable and consistent for visitors. Check out [this article on refactoring](#) for advice.
- **Kill carousels:** Carousels are not only [horrid for usability](#), they often load lots of heavy high-resolution images. Do your users and your site’s performance a favour by replacing the carousel with

a single (optimised) image.

- ***Replace background images with CSS:*** CSS can do some pretty awesome stuff these days. If you're currently using tiled images for background images, consider replacing them with a CSS background. You'd be surprised at [what you can do](#).

Technique 2: Image Optimisation

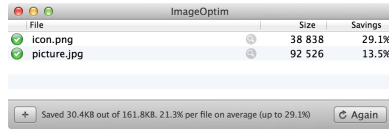
This is the first thing almost every page load optimisation article will suggest. That's not a coincidence – images make up the largest percentage of the weight of most sites, and you can get some big wins with very little effort.

Image Optimisation Basics

Here's the crazy simple way to trim down your theme in less than 50 seconds:

1. Download [ImageOptim](#) and open it;
2. Drag your theme's `assets` folder into the application;
3. Wait to see how much space you just saved.

It really is that simple!



ImageOptim doing it's thing.

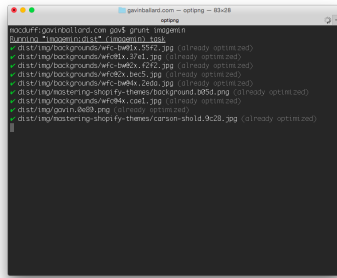
As I was writing this, I ran the assets folder of a theme I've been working on through ImageOptim just to get some screenshots. I've run this theme through ImageOptim before, but I still saved 200kB of page weight from 5 seconds of work. Not bad!

Another tool worth checking out is [JPEGMini](#). Unlike ImageOptim, it only handles JPEGs (who would have guessed) but it does apply some more advanced algorithms and can cut down those files even more.

Automating Image Optimisation

Once you've seen the benefits of image optimisation in action, you're likely going to want to use it all the time. That's when having a Grunt-based workflow (or similar) like the one discussed in the previous lesson "Development Tools and Workflow" is helpful.

Using a Grunt plugin like [grunt-contrib-imagemin](#) (which is built with the same libraries used by the ImageOptim program), we can ask Grunt to automatically optimise all of our images whenever we deploy our theme.



A Grunt task optimising images automatically.

Check out the resources from the tools and workflow lesson, along with the [ImageOptim](#) documentation, for more and to see how to implement this automation into your workflow.

Pushing Boundaries with Lossy Image Optimisation

All of the above optimisation methods are “lossless” – that is, they reduce filesize without any degradation in image quality. That sounds reasonable, but often we can achieve pretty spectacular reductions in filesize with minor reductions in quality – some of which are going to be invisible to the end user.

Let’s take JPEG images first.

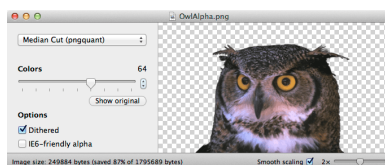
If you’re familiar with Photoshop’s “Save for Web” functionality, you’ll be aware that you have the option of selecting the quality of the final image. By default, this

sits at **60** – but the reality is that you can often drop that quality setting down to as low as **25** without ending up with a dog’s breakfast. This is especially true if you’re dealing with “@2x” images for high-resolution displays.

If you don’t believe me, check out [these scaling examples](#) from Thomas Fuchs’s excellent book *Retinify Me*, which covers the process of creating high-resolution images in lots of helpful detail.

We can also perform lossy optimisation on PNGs.

The same wonderful developers behind ImageOptim have a great utility for this called [ImageAlpha](#). Similar to ImageOptim, you can drop a particular PNG file into this application, select a quality level (for PNGs, determined by the number of colours available), preview the results, and save an optimised version.



Using ImageAlpha is a hoot.

The results can be pretty impressive – here’s a before and after:



*Before (above line, 224KB) and after
(below, 34KB) ImageAlpha (Image
courtesy of <http://pngmini.png>).*

Unlike lossless optimisation, lossy optimisation is not something I'd recommend automating, as you often need to run your eye over the final results to make sure the optimisation hasn't gone too far and everything "looks right".

Screencast on Image Optimisation

If you'd like to see the techniques in this section in action, I recorded a [short screencast](#) on the subject.

Technique 3: Asset Concatenation

For every Javascript, stylesheet or image your page references, a client's browser has to make a HTTP request to fetch and parse that asset. Asset concatenation is the process of combining multiple asset files into a single file

in order to reduce the number of requests the browser has to make, cutting down on this overhead and making for faster load times.

As an example, if the top of your `theme.liquid` contains something like this...

```
<!DOCTYPE html>
<html>
  <head>
    <!-- Include CSS -->
    {{ 'bootstrap.css' | asset_url | stylesheet_tag }}
    {{ 'index.css' | asset_url | stylesheet_tag }}
    {{ 'products.css' | asset_url | stylesheet_tag }}
    {{ 'articles.css' | asset_url | stylesheet_tag }}

    <!-- Include Javascripts -->
    {{ 'jquery.js' | shopify_asset_url | script_tag }}
    {{ 'bootstrap-core.js' | shopify_asset_url | script_tag }}
    {{ 'bootstrap-tooltips.js' | shopify_asset_url | script_tag
  }}
  {{ 'bootstrap-modals.js' | shopify_asset_url | script_tag
  }}
  {{ 'products.js' | shopify_asset_url | script_tag }}
  {{ 'cart.js' | shopify_asset_url | script_tag }}

  ...
```

... then your browser is going to be making lots of requests that it needs to wait on before it continues to render the page. A better strategy is to selectively combine these files to reduce their number, so that you end up with something like this:


```
<!DOCTYPE html>
<html>
  <head>
    <!-- Include CSS -->
    {{ 'main.css' | asset_url | stylesheet_tag }}

    <!-- Include Javascripts -->
    {{ 'jquery.js' | shopify_asset_url | script_tag }}
    {{ 'bootstrap.js' | shopify_asset_url | script_tag }}
    {{ 'main.js' | shopify_asset_url | script_tag }}

    ...
  </head>
  <body>
    ...
  </body>
</html>
```

For stylesheets and Javascript files, the concatenation is very simple — it's just a matter of copying and pasting the text contents of each file one after another. As long as you concatenate the files in the same order that they originally appeared in your HTML, you won't notice any functional difference — but the number of requests the browser has to make is dramatically reduced.

Oh, and incidentally — you should never be including your Javascript `<script>` tags in the `<head>` element, like in this example. Browsers will block page rendering when reading them. Put all of your `<script>` tags at the bottom of your page, just before the closing `</body>` tag.

When to concatenate

As with any technique, you can take this too far, and concatenating *everything* into one file can be counter-productive. Some things you might want to keep separate to your core files are:

Large Javascript libraries

Bundling large libraries like jQuery together with your site-specific Javascript is considered bad practice, as any changes to your site's scripts will force browsers to download everything again.

In fact, a much better strategy for large and common Javascript libraries like jQuery is to make use of a common external CDN (Google's [Hosted Libraries](#) is a good start). Not only does this mean that you don't have to worry about managing the asset within your theme, there's a decent chance that a visitor to your site will already have the library in their browser cache.

Assets using Theme Settings

As we saw in the "Theme Settings" lesson, appending a `.liquid` suffix to asset files (for example, `main.js.liquid`) will cause Shopify to preprocess those assets using Liquid, allowing us to use theme settings within those files.

If your theme takes advantage of this, I'd recommend separating the parts of code that make use of theme settings out into their own asset file as much as possible. Otherwise, you'll run in to a similar problem as that with large Javascript libraries, forcing a large download on your users every time a theme setting is changed.

If you're only using theme settings for a few minor tweaks, you could even just keep a single static asset file and add the dynamic stuff to your theme's `<head>`, for example:

```

<!DOCTYPE html>
<html>
  <head>
    <!-- Include Static CSS -->
    {{ 'main.css' | asset_url | stylesheet_tag }}

    <!-- Dynamic CSS -->
    <style>
      body {
        bgcolor: {{ settings.bgcolor | default: '#ffffff' }};
      }
    </style>

    <!-- Include Javascripts -->
    {{
      'https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js'
      | script_tag }}
    {{ 'bootstrap.js' | shopify_asset_url | script_tag }}
    {{ 'main.js' | shopify_asset_url | script_tag }}

    ...

```

Assets only used on a particular page

When you're bundling your assets with concatenation, the stylesheets and scripts within those files will be available across all of the pages on your site. This is usually a reasonable assumption, and even if a particular script isn't used on a single page, the benefits of having a single asset file usually exceed the downside of having a slightly larger file to download on some pages.

However, if you have large assets that are only used on specific, rarely-used pages, it might be more efficient to keep those assets separate and only load them when needed.

A good example of this would be a Shopify store that offered logged-in customers a page to view their order

history and manage their account.

As most visits to your site won't be from logged in users, and won't be used for account management, it makes sense to avoid bundling the scripts and styles for the account management feature together with everything else.

Automatic concatenation

An obvious downside to concatenation is that it makes management and development of your assets more difficult, by virtue of the fact that all of your scripts and stylesheets are now lumped together in one giant file.

Setting up a way to automatically concatenate your files together means you can get the benefits of concatenation without sacrificing developer convenience. As you might have guessed, I recommend using Grunt to automate this process — see the “Development Tools and Workflow” lesson for more.

Image concatenation

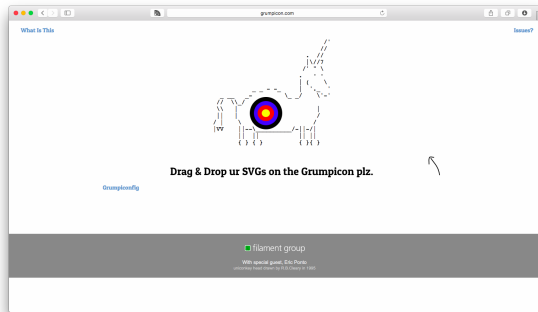
This section has been focused on the concatenation of scripts and stylesheets, but concatenation can be used for images as well! In fact, you've probably seen this before with CSS “sprites”, where sites use a single PNG image to hold all of their icons:



Google's CSS sprite.

The rationale behind this is exactly the same as with stylesheets and scripts: reducing the number of requests that need to be made improves the performance of your site.

The easiest way to do your own image concatenation is through the very excellent [Grumpicon web app](#). It's a drag-and-drop solution that comes with the added benefit of having a unicorn involved.



Any tool with an ASCII Unicorn must be good.

Like all good asset management tasks, the magic behind Grumpicon is available as a Grunt plugin ([grunticon](#)) for adding to your automated development workflow.

Future developments

If you've been following recent developments in web technology, you may have heard of the SPDY or HTTP2 protocols. Without getting too technical (not the least because I'm not across all the details myself), these are next generation technologies which aim to drastically improve the speed of the web.

One of the features of these protocols is “request multiplexing”, which aims to pretty much do the equivalent of asset concatenation at a much lower level. This would mean that developers like us can just use individual assets (easier for us) without forcing the browser to make more requests than Madonna's rider.

However, widespread deployment and support for these protocols is still years away, so for the moment you'll have to stick to the techniques as described. Sorry!

Technique 4: Asset Minification

While stylesheets and scripts aren't usually as bulky as images, they can still have a noticeable impact on the overall weight of your pages.

Once you've followed Technique #1 and simplified and removed as much unnecessary styling and content as possible, you can use asset minification to further reduce the filesize of your stylesheets and scripts.

Minification strips out all of the extraneous information in assets that make it easy for humans to read and write but that aren't needed by your browser (things like newlines and whitespace). In the case of Javascript, minification can also aggressively optimise your code size by doing things like rewriting `var aLongVariableNameUsefulForHumans` to `var a`.

How to minify your assets

If you want to experiment with minification, or just want to perform a one-off minification, there are plenty of online tools that will take a CSS or Javascript file and spit out the minified version (just Google "CSS/Javascript minifier").

However, minification is at its most useful when we can slot it into our workflow once and not have to think about

it again. Once again, the Grunt plugins [grunt-contrib-cssmin](#) and [grunt-contrib-uglify](#) come to our rescue with CSS and Javascript minification respectively.

If you have a different workflow process, or would like to perform minification from the command line, you can check out the comprehensive [YUICompressor](#), which handles both CSS and Javascript. The [uglify](#) tool is also available as a standalone Javascript minifier that can be run from the command line.

Technique 5: Odds and Ends

Properly implementing the techniques above will, in most cases, get you a long way towards improving your theme's performance. If, like me, you're a little bit obsessive about performance, or are in a situation where extracting every last ounce/gram of speed is going to materially affect a store's bottom line, there are still a couple more things you can do to achieve that.

Implement Device-Responsive design

Before “responsive design” came to mean that thing we do dragging the width of our browser back and forth, it had a broader meaning — that websites could respond to a whole range of contexts to deliver the best experience.

A good example of where this idea comes in handy in the performance arena is with “retina” or “high-resolution” images. A common technique for high resolution images over the last few years has been to simply serve all clients a high-res image resized to be half the size. Browsers

then do the work of resizing and displaying appropriately.

The problem with this approach is that it's very wasteful when serving to screens that aren't high-resolution — they're downloading an image that is 4x the physical size for no advantage. "Device-Responsive Design" takes this sort of thing into account by only loading high-resolution images on high-resolution devices, either through the use of media queries or a Javascript library.

For information on how to implement this sort of stuff, go straight to the experts - [Brad Frost](#), [Ethan Marcotte](#) and [Tim Kadlec](#).

Use Lazy Loading

This technique can be really powerful for Shopify stores, as they are often so image-heavy. Lazy loading images involves using a Javascript library to only load images when they should be visible in the user's viewport, improving initial page load times.

It's especially handy when your theme involves very tall pages where lots of visitors won't actually need to load the majority of your content. Matt Mlinac's [lazyload](#) jQuery plugin is a very solid implementation.

Use conditional loading for shims/fallbacks

If you're using "shims" or fallbacks for older browsers (for example, the [RespondJS](#) fallback which allows older browsers like Internet Explorer 8 to use media queries), make sure you're not wastefully downloading them on newer browsers.

You can do this by using conditional comments when loading the script, like this...

```
<!DOCTYPE html>
<html lang="en" prefix="og: http://ogp.me/ns#">
  <head>
    <!-- HTML5 shim and Respond.js support for HTML5 elements
    and media queries -->
    <!--[if lt IE 9]>
      <script src="{ 'js-html5shiv-min.js' | asset_url }">
    </script>
      <script src="{ 'js-respond-min.js' | asset_url }">
    </script>
    <![endif]-->
    ...
```

... or by using Javascript-based conditional loading based on the results of testing with a library like [Modernizr](#).

Master the `async` attribute

As mentioned above, putting `<script>` tags in the `<head>` section of your HTML is a sure-fire way to kill your page loading times, as browsers will block page rendering while they wait for your Javascript to load. Common wisdom says that the best place for including script tags on your page is at the very bottom of the page, just before the `</body>` tag.

This is a great default approach, but with the advent of HTML5, we now have access to the `async` attribute for `<script>` elements, which tells browsers to continue parsing and rendering the page while loading the script in the background.

However, there are some *major caveats* to the use of `async` — namely, that there are *no guarantees* on execution time

or order.

If you're interested in diving in to the nuts and bolts of `async`, [this walkthrough by Jake Archibald](#) is the best I've read.

Check for asset 404s and 301s

Look through the “Network” panel or similar in your browser's developer tools and check that no assets are 404-ing or 301-ing.

Missing assets (404s) are a waste of a request and you should just remove the reference to the asset or replace it with one that works.

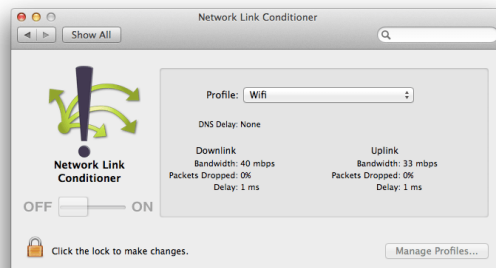
An asset request being redirected with a 301 response code is actually quite common (for example, a request to a `http://` version of an asset may be automatically redirected to the `https://` version). If you see that happening, just link directly to the final URL to avoid the wasted initial request.

Evaluating performance improvements

Once you've made each performance optimisation, it's always a good idea to measure your performance metrics again to see what's changed. Not only will this give you a sense of achievement and motivate you, you'll start to get a good idea of where the easy optimisations are and start to incorporate them into your normal development workflow.

Now, while all of this measuring is handy, there's nothing quite like actually using your site to check whether your customers are getting a good experience! Next time you're on a train, or out with your phone in an area with patchy reception, try clearing your browser cache and loading your site. Bonus points for asking random strangers in coffee shops to do the same and try to buy something from your store.

If you really can't wait to get out of the home or office to test out your performance improvements and you have a Mac, you can make use of OS X's "Network Link Conditioner". Turn your connection settings down to spotty 2G and see if your site holds up.



Condition that link!

Get instructions on how to set up the network link conditioner [here](#).

Final thoughts on performance

Shopify theme developers *have* to wait for your site to load in order to do what they want to do.

Customers don't.