O'REILLY®

Early Release

RAW & UNEDITED

Front-End Architecture

A MODERN BLUEPRINT FOR SCALABLE AND SUSTAINABLE DESIGN SYSTEMS

Front-end Architecture

Book Subtitle

Micah Godbolt

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Front-End Architecture
by Micah Godbolt

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Meg Foley
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the Early Release Edition
2015-07-30: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491926710* for release details.

978-1-491-92671-0

[FILL IN]

1. prologue
2. intro
3. the 4 pillars
4. code
    a. HTML
    b. CSS
    c. Redhat Code
5. Process
    a. Build Tools
    b. Workflow
    c. Redhat Process
6. Testing
    a. Unit Testing
    b. Visual Regression
    c. Performance Testing
    d. Redhat Testing
7. Documentation
    a. Style Guide
    b. Pattern Library
    c. System/Onboarding
    d. Redhat Documentation
8. Closing

Table of Contents

Origins ..... **vii**

1. The Discipline of Front-end Architect **15**

15

Adopting an Architectural Process 16

2. The First Pillar of Front-end Architecture: Code **23**

A. Appendix Title **81**

Index **83**

Origins

A New Discipline

In the beginning was the web, and the web was good. Well, in this case, the beginning was the early 90's, and good meant a site had found its way onto the Yahoo index page and the little visitor counter at the bottom of the table laden, animated gif infected page was spinning.

But that's all you really needed in those days. As long as you were feeding your fellow web-circle subscribers with click-throughs, all of the webmasters were happy. And those webmasters? Yes, they were masters of their domain...literally! Websites were simple creatures, and the webmaster was their keeper, tending to the tangle of HTML and wondering if these new Cascading Style Sheets were something to be bothered with - most had already written off JavaScript as a passing fad.

But like any medium, the web grew. Javascript stuck around, CSS was useful for more than just setting the page's font-family and font color, and the Webmaster eventually found themselves at a crossroads. Their website traffic continued to grow, and the technologies of the web continued to expend (OH! Transparent gifs!). There were just too many new things to keep track of, and so much work to do, so eventually they were all forced to specialize. On one hand, they really liked animated gifs, the under construction signs, and the ubiquitous marquee tags, but on the other hand Perl was the best language ever to be created and will undoubtedly power every website from here into eternity.

So when it came to hiring a second person to help run the Webmaster's prized precious domain, they needed to decide if they wanted to continue to be a <blink> tag master and hire Perl script kiddie, or the other way around.

Eventually these decisions had all been made, and the modern web team began to assemble like a news team called to a large seashell. At every stage, every crossroad, former webmasters found themselves needing to focus on a smaller chunk of the web process.

Newer, more focused roles also attracted artists, writers, business analysts, engineers and mathematicians to the industry. As these roles developed, and members that gravitated toward them become more and more proficient, the web began to form a new set of designations...or Disciplines.

Those Who Strategize About Content

Very early on in the development of the web there was a particular personality that seemed to believe the words on any given page were at least as important as the design, the code or even the Search Engine Optimization (everyone knew that keyword stuffing was the way to go). Before they came along content had always been a problem to deal with later. "Just throw up some lorem ipsum into the design and move on". The client will eventually fill it in with real, quality, inspired content before the site goes live...really...always.

These lovers of words were ultimately quite vocal that the web was content, and that it deserved our time and attention. It was always a battle, but occasionally they were invited into early planning meetings, and on occasion they were contacted to lend their expertise on proper SEO techniques or to develop a consistent voice and tone for the editorial strategy. They were making progress, and though the fight was difficult and lonely, the results were worth while.

Thus went each of these lone warriors until the fateful day where they happen to come across another logophile, and they realized that they weren't alone in this struggle! This single kindling of a friendship quickly grew into a blaze of new connections. Eventually a community of these word disciples was formed, and they continued to focus their efforts on convincing others to treat content as a valuable asset, and not a liability.

A dozen years passed and the fight for content was far from over. But even as one more designer was asked to "just make something up" for the homepage copy, a new rallying cry could be heard in the distance. December 16th, 2008 was the day that Kristina Halvorson stood up high atop A List Apart Blog and raised a banner for Content Strategy. She asked us all to "take up the torch" and begin "treating content as a critical asset worthy of strategic planning and meaningful investment". Those who practice Content Strategy were to "Learn it. Practice it. Promote it". They were to become Content Strategists. And like that, a Discipline was born.

Ms. Halvorson's article was not the first one to approach the topic of content strategy, but what it did was define the heart, soul, and purpose of Content Strategy, and the Content Strategist. Overnight, a large group of web content creators realized that they had a name, a discipline and a collective calling. These Disciples would usher in an era of blogs, podcasts, and conferences revolving around this simple notion that "content mattered".

A Responsive Web Was Born

Around the same time, a man in a black turtleneck got up on stage and utterly ruined everyone's conception of what an internet connected device was. For the first time in the history of the web we were truly forced to accept that websites were not browsed solely on 1024 x 768 pixel screens in the comfort of our offices and living rooms with the fat pipes of broadband internet. The introduction of the iPhone ushered in a new era of web connected devices with a multitude of screen resolutions, varying capabilities, fluctuating connection speeds and inconsistent input modes. As developers we could no longer make assumptions about our user and the properties of the device they were using to view our websites.

In response to this shakeup we tried a number of different solutions. We tried relying on pinch and zoom, or double tap to zoom, leaving our sites mostly untouched, or we used user agent string detection to redirect any mobile device to a stripped down, mobile friendly m-dot website. Neither solution really solved the problem. Pinch and zoom sites were difficult to navigate in order to finalize purchases, or sign up for services, and increasing mobile traffic meant increasing lost revenue. M-dot sites, which more user friendly for mobile devices, required development teams to maintain two separate websites.

Many m-dot sites languished, failing to be updated as frequently as their larger sibling, or the reduced feature set of the m-dot site forced users to switch to desktop devices to do anything more than get directions or place a phone call. It was obvious that something needed to be done. Though some considered the iPhone a passing fad, it was soon quite obvious that the future of the web lived inside of these small, personal screens.

Three years after the release of the iPhone, the web development community got the mobile solution they had all been waiting for. On May 25th, 2010 Ethan Marcotte penned a lengthy article on A List Apart called simply, "Responsive Web Design". This article did not describe some new discipline, or a banner for embattled developers to gather under, instead "Responsive Web Design" or RWD described a method for creating a new breed of website that would respond to the size of the users device and mold itself to that viewport. He described this process, not as some new or emerging technology, but rather a collection of already existing tools and techniques.

RWD was nothing more than a combination of the following:

- Fluid grids
  - Percentage based widths rather than fixed pixel dimensions
- flexible images
  - 100% width images fill the container they are put inside of, and flex as the viewport changes size
- media queries

— Being able to specify different styles for different viewport sizes, we could now change page layout based on the size of the screen.

All of these techniques had been available in the browser for years before Mr. Marcotte's article, but just like the article on Content Strategy, Responsive Web Design was able to clearly explain how these tools could be put together to achieve a result that everyone was desperately looking for.

I can still remember reading that article for the first time. The imagery used in his examples make my first expose to Responsive Web Design very easy to recall. I remember practically smacking my forehead, because none of it was rocket science, but the resulting effect was magic. Shortly after reading it I built my first site using those techniques...it was a horrible site, but it worked! Since that day I have not been involved in a single project that did not embrace those 3 pillars of responsive web design. In a single blog article my career, our profession, and our industry was transformed almost overnight.

The Seeds of Front-end Architecture

It was with this history in mind that I started to think about the notion of Front-end Architecture in the middle of 2013. As a Drupal Front-end Developer I knew full well the pains and frustration felt by Content Strategists. The front-end styling, or the theme, was always an afterthought. It was a layer of 'pretty' applied to the default markup after the designers and back-end developers had finished their work. The challenges we faced couldn't have been better exemplified than in the order in which people were brought onto a project. As part of an agency I saw projects start, designs debated over, functionality developed...and THEN a Front-end Developer was pulled onto the project to apply whatever design was tossed over the wall to whatever markup was thrown at us.

Having gone through this process several times I knew the pain and frustration I was going to experience as I tried to deconstruct a set of mobile and desktop photoshop documents into a theme that could be applied to the div soup that Drupal spit out. Speaking to rails friends about the challenges of styling a website navigation I eagerly confessed "One does not simply change Drupal navigation markup", and it was true! Once that markup was set, and the developer had moved onto another task, the chance of modifying the combination of divs and lists and links was all but a dream. This inevitably lead to ridiculous CSS hacks to make a design, never meant to fit the default navigation markup, actually work in production.

For many years a Front-end Developers worth was measured in their ability to create these Frankenstein design patterns. "Now if I put a pseudo element on the 3rd nested div, and use a background image from this sprite..." was the epitome of our battle plans, and frankly it sucked. We were doing nothing but patching up holes in a failing

levy, and our only hope was that the site would launch before getting swept away by the piles of technical debt we were leaving behind us.

I knew then that this process for developing websites wouldn't be sustainable as the complexity of our projects continued to grow. So instead of doing things the way we've always done them (because they've always worked in the past), I decided to start imagining how a project would differ if we made Front-end Development " a critical asset worthy of strategic planning and meaningful investment". What if we had a voice in things like CSS frameworks, documentation tools, build process, naming conventions or even the markup itself?! I started to wonder what a large scale project would look like if UX development fed PHP development, instead of the other way around?

Would this create a revolution? Would others pick up the same torch and start to "Learn it. Practice it. Promote it"? But before we could all rally under a single banner, it was important to understand what that banner stood for. What were our demands? How would we accomplish our goals? What would we be called?

What's In A Name

Seeing as my current employer did not have a role doing this type of work, I started to research other similar job titles, hoping to find one that was at least similar. After a bit of digging I found that after Senior Developer the next role was a "Software Architect". Reading the job description made my heart skip a beat! The role of a Software Architect was to come onto a project very early on to discuss the needs of the client with regards to the Drupal platform. What technology stack were we going to use? What were our content types? How was content created, stored and displayed back to the screen? I realized that the role of a Software Architect was to make sure that nothing was ever created by chance, but rather guided by an overarching architecture. It didn't take long for me to convert database and web server to Sass folder structure and build system, and the title of Front-end Architect was born.

Now that I had a job title, I went to modifying the job description thinking what this role would bring to the table, and how it would impact a project given the proper opportunity. These ponderings lead to a quick lightening talk about Front-end Architecture at our annual company retreat, and a submission to CSS Dev Conf that got accepted and forced me to focus my thoughts into a concise 45 minute presentation.

"Raising a Banner for Front-end Architecture", given to a packed room in New Orleans, was a rallying cry for Front-end Developers who had already been on the front lines of this fight. I wanted them to know that they were not alone, that there were others out there to support them and back them up. I also spoke directly to the project managers and sales teams outlining the power of a properly architected front-end and the value that it brought to the team and to the client. The only way we were going to be able to make an impact on projects was to be brought on earlier in the

process, which meant clients paying for those hours, and managers being willing to schedule those resources.

After the talk I heard story after story of developers who finally understood who they were, and the role they played in their organization. Many people found themselves performing the role of Front-end Architect, but had never taken on that title, or felt confident enough to ask for the authority the position should carry. The weeks after CSS Dev Conf saw many people changing their Twitter bio's to read "Front-end Architect". And I, being one of those to make the change, haven't looked back since. No matter the title I currently hold at my current job, I am a Front-end Architect.

CHAPTER 1

The Discipline of Front-end Architect

> Front-end Architecture is a collection of tools and processes that aims to improve the quality of our front-end code while creating a more efficient and sustainable workflow.

When I think about the roll of a Front-end Architect I can't help but make a comparison to a traditional Architect, as they share many characteristics.

An Architect is defined as one who *designs, plans and oversees* the construction of buildings. In many senses, this is just what a Front-end Architect does for a website. The primary difference is that a Front-end Architect's product is not building but a set of tools and processes facilitating the creation of a website. Just like how an Architect spends more time drafting up schematics than pouring concrete, a Front-end Architect's audience is not the end users, but the developers themselves.

Design

The overall look and feel of the website is still squarely in the hands of skilled designers, but as a Designer the Front-end Architect crafts the front-end approach and philosophy. By designing a system all Front-end Developers are going to work within, the Architect sets a clear vision of what the end product, the code, will look like. Think about a building with no clear architecture. The important decisions decisions were all left up to the builders doing the work. One wall built with stone, another with brick, a third with wood, and the fourth omitted because it was trendy.

Once a Front-end Architect sets the vision, the project has a standard in which to test code against. Without a design for the finished product how could we determine wether or not our code, which for all intents and purposes, gets the job done, actually met that standard. A carefully designed system will have checks and balances insuring that all code contributed to that system adds perceivable value, rather than just adding lines of bloat.

Planning

With a clear design in mind, the planning stage involves mapping out the development workflow. What steps will a developer take to write a line of code and see that code through to production. In the most simple case this plan involves FTP'ing into a server, editing a file and hitting save. But in most cases this will involve various combinations of Git (version control), Gulp/Grunt (task runners), Sass (CSS preprocessor), documentation tools, test suites, and server automation.

The goal of the Front-end Architect is to design a well oiled machine that provides quick and painless setup, useful feedback in the form of linting, tests and documentation, and reduces much of the human error that occurs when performing repetitive tasks.

Oversight

Front-end Architecture is never a set it and forget it proposition. No design or plan is ever perfect or complete. The needs of the clients as well as the needs of the developers will change and evolve over time, and a process that was working well in one phase of the project might need to be revisited later to improve efficiency or reduce errors.

The key factor is that a Front-end Architect has the ability and the opportunity to continue to make those adjustments. Modern build tools make it very easy to adjust workflows and distribute those changes out to each team member.

Some people have asked if becoming a Front-end Architect means a role in management, and never writing a line of code again. I can personally attest that not only do I write more code as an Architect, but I get to write in a greater variety of languages, for a larger number of tools, and the code I write has a more significant impact on the team and our ability to succeed. It's not that I write less code, but simply that the audience of my code has changed. A Front-end Developer's audience is the end-user, where as a Front-end Architect's audience is the developers themselves.

Adopting an Architectural Process

Just like other disciplines before it, Front-end Architecture has had to fight a battle of priorities. While we can't imagine why anyone would start building a skyscraper without first consulting an Architect, that is in fact how most large web projects start.

The excuses are numerous: We don't have the budget. We don't have the time. We'll make those decisions after all of the designs and infrastructure is done. Or simply no excuse is required because you get placed onto the project months after the design have been finalized, and development is already in full swing. You now have just a few

months to make a pile of HTML magically look like a stack of PSD's tossed over the wall to you.

This is obviously not the way we want to be developing our projects. As Front-end Architects we believe that there are a number of key decisions that need to be made at the beginning of a project. These decisions are either too difficult to implement later on in development, or the choices shape many other decisions made in the early stages of a project.

Once those decisions are made, continual work must be done to help shape visual design, platform development and infrastructure setup to best meet the needs of the envisioned architecture. By keeping the voice of Front-end Development out of those early decisions, projects run the risk of having to choose between reworking designs/platform/ops and telling the Front-end Developers to make due. I can tell you from experience, betting on the former is always a bad bet.

Therefore, the objective of this book is to express the importance of getting a Front-end Architect involved in a project at a similar times as you would pull in a Content Strategist, Art Director, or Software Architect. The decisions that can be made at this stage of the game are just too important to overlook to put off until later.

What's The Catch

I know this isn't going to be an easy task. The changes I will be suggesting have a tangible cost to them, and for anyone in a position to make these decisions, they will always need to weigh the risk, the cost and the possible benefits. For anyone that hasn't had the experience of working with a Front-end Architect, this can be a difficult risk to take, and an ever more difficult risk to explain to one's supervisors.

The chicken or egg dilemma is that to overcome the objections of spending time and money on a proper Front-end Architecture many stakeholders will require examples of how this has helped projects succeed in the past. This obviously requires one to have worked on a project like this in the past, but how do you get the opportunity to take a risk like this if management is always asking for proof the approach works? I don't know about you, but I have been on a total of 2 different projects over the past 2 years, and at the rate that this industry is changing this means that every new project I start is using a set of tools and process that I have never used in that exact way before. Fortunately for me, and this book, this dilemma didn't stop me from trying to make this vision a reality.

Project Alpha

This past year I was given an opportunity of a lifetime when asked to architect a solution for Red Hat that would let them share their existing website "bands" across multiple company web properties. Having been on the initial build of Redhat.com I knew

the challenge ahead of me, but it also meant that I'd had time to establish the development team's trust, and was given full control over the entire architecture of this project.

I call this an opportunity of a lifetime because it was! Our chicken or egg dilemma had been cracked. My team was given the opportunity to build a design system of significant scale, with sufficient technical support, that would eventually be displayed on a site with an incredible amount of traffic. This would be the project that I'd use to promote Front-end Architecture to my next project...and the project after that.

A Slow, Powerful Start

Our team was incredibly fortunate to be working with Red Hat at that particular time. Having just launched the original site redesign we were in a bit of a feature moratorium. This meant, while we occasionally had a couple bugs to squash, we were otherwise freed up for a number of months to explore ideas of how we'd create and distribute this shared library.

Dispute the sheer amount of legacy code still in production at Redhat.com we were given a blank slate to work with. As a website built on the concept of bands (row after row of unique content), we were fortunate to have a very clean global style state, and the new elements we built would sit under or over existing bands. There were no sidebar styles to override, no HTML tag styles constantly changing around us. We were practically building a brand new website right inside the old one. And just like replacing every board on a ship, one by one, we are hopeful that we could eventually retire the old theme system, and rely solely on our own.

With such freedom in front of us we knew we had an opportunity to create a large wish list....and actually get most of it! That wish list included:

- **Modular Content** - We were big fans of atomic design principals and wanted to reuse small components rather than creating unique band after unique band
- **Exhaustive documentation** - With a large team of front-end developers, back-end developers, designers, marketing managers, ops and various product owners, we had a HUGE audience. We wanted to make sure that whatever we built had documentation that met each one of their needs.
- **Comprehensive testing** - We'd been burned too many times by large chunks of front-end code being merged into master breaking code written months prior. So we were determined to test our design system with the same level of coverage as we had at the application level with behavior testing.
- **Streamlined processes** - We wanted to mirror the git flow system that worked so well at the application level, but we needed to break feature branches into smaller component sized code chunks. We also wanted to automate all of the error prone, time sucking, by hand processes we had been doing in the past like updating style guides, creating icon fonts and deploying new code.

These four areas of focus took us a bit of time to develop, setup and perfect. There is no denying that our first few months were less productive than they might have appeared if we'd continue to build things like we had in the past. But once we got the foundation in place and we started to build out our system we quickly realized the power of what we'd built.

With every new request to build a couple bands, or even a full 12 band page, we found that we had less and less work to do. Instead of approaching each new band as a unique snowflake, we were able to break it down into its smallest parts, determine if a new layout, component, or component functionality was required, and then quickly build up instructions for the developers of how to implement this feature.

Every story ended with exhaustive documentation, a suite of regression tests and code the conformed to the standard set down by our original Architectural decisions. We were able to do all these things with significant velocity because we were working in a static environment (no database, no servers) and we could use Grunt to perform any task from rsyncing code to our remote servers to deploying a new tagged release to production.

I'll dive much deeper into the details of this approach in later chapters, but the take-away message is that even though these tools and processes took some upfront effort, we were literally laughing during planning meetings at how trivial our work had become. Of course, this trivialization of our work was a good thing as it allowed us to pour more time back into our design system. In the end, working inside of this Design System was a joy, and with every story we were excited about how much more powerful it could become.

Armed and Dangerous

Therefore, armed with this experience, I am confident that the software we wrote, the processes we created, the techniques we honed, and the lessons we learned will be a sufficient example of success to convince future projects of the validity of this approach.

With these processes, techniques and lessons poured out into this book, I hope you can walk into your next project with the confidence to fight for your Front-end Architecture. Fight to get onto the project earlier, so that you can help influence important decisions. Fight for quality tools and processes so that you can build smarter, more reusable code. Fight to make your front end design system matter. Take up the banner for Front-end Architecture.

Every building needs a solid foundation, four walls and a roof. These are are non-negotiable. The foundation holds the walls, the walls hold the roof, and the roof keeps you safe and dry. As an Front-end Architect we are under a similar obligation any time we are involved in the creation of a new property. We are tasked with being champions for the essential tools and processes required to make this website a success.

Throughout the rest of this book I am going to be spending most of our time looking at what I see as the Four Pillars of Front-end Architecture. These four groups of topics, technologies and practices are the foundation of a scalable and sustainable design systems. They provide a series of serious discussions that need to be made about the Front-end, and should be had at the earliest point possible in a project. These conversations will help to set expectations in code quality, the time and effort to finish each user story, and the workflow process that will get all of this done in a timely manner.

These pillars in no way prescribe the only way to do something, or even the best way to do something. Each decision needs to be made in the context of the project that you are in. Sometimes the decision will even be to not do anything! Projects smaller scale, or transient nature might not require the same level of foundation as a Fortune 500, customer facing property meant to last several years.

Lastly, do not take the upcoming chapters as a list of topics to master. To be a Front-end Architect is to be in a state of constant learning. This constant state of learning is what defines us. Our inch deep, mile wide understanding of the entire Front-end Development space is what allows us to be champions for new technologies and methodologies. It is our ability to spend an hour with a new framework or grunt plugin, and come away with a reasonable understanding of its strengths and weaknesses, that allow us evaluate if it could be a valuable addition to our workflow. So if you find yourself overwhelmed by the sheer number of technologies and topics in the rest of this book, just remember that none of use are masters at all of these. I personally range from expert in several of them, competent at many of them, and completely new to some of them.

So, enough dancing around these pillars, lets dive into them and discuss the significance of each.

Code

When it comes down to it every website can be broken down into a bunch of text files and media assets. When we start to look at the sheer number of lines of code produced in the making of a website it is incredibly important that we set expectations for the code that is written.

In this pillar we'll be looking at the various types of code we'll be writing and the different considerations we need to make for each when defining those expectations. We also need to determine the role 3rd party code in our system, and how everything ties together.

Testing

In order to create a scalable and sustainable design system we need ensure that any code we wrote yesterday isn't devalued by the code we write today. Our code isn't written in a vacuum, but rather is part of a much larger system. Creating a plan for appropriate test coverage will ensure that yesterday's code continues to provide the value it did on the day you wrote it, and that all new code is written to provide additional benefit to the system.

In this pillar we will take a look at four different methods we have for testing our sites. Sometimes, depending on the team size, these tests will be split between front-end, back-end and ops, but a solid understanding of each of them will valuable when communicating with those other teams.

Process

As we are miles away from a workflow of FTP'ing into a server, editing a file and hitting save, it is important to start thinking about tools and processes that will create an efficient and error-proof workflow. Our build processes are getting more complex, as are the tools that create them. These complexities bring with them risk of over-engineering and unnecessary abstraction, but they also bring incredible gains in productivity, efficiency and consistency.

Just as our workflows have evolved, so has the way we work. We are no longer spending our hours making the CMS markup 'look like' some photoshop comp. As we move to designing in the browser, and creating responsive HTML prototypes, we are in a position where we are writing all of the HTML and the CSS before the feature is even implemented in the CMS. This incredible role reversal needs to be supported by a change in development process.

Documentation

It seems that no one sees the value of spending time on documentation until a key member of the team is about to leave, and then it's "stop everything and document all the things". As a Front-end Architect you will be a champion for documentation that is written at the same time as the process or artifact being developed.

This pillar will look at those various types of documentation your team might need to write, the tools used to make publication easier, and the types of end users that will ingest the content.

CHAPTER 2

The First Pillar of Front-end Architecture: Code

One of the first challenges you'll face as a Front-end Architect will be to tackle the markup written by your developers and produced by your CMS. The web doesn't exist without HTML. Strip away your CSS and JavaScript and you will be left with raw content and controls.

Text, images, links, forms and submit buttons: this is all the web really needs, and it is the foundation of anything you'll ever create on the web. Start off with bad markup, and you'll be writing bad CSS and bad JavaScript to make up for it. Start with great markup and you'll be be able to write more scaleable and maintainable CSS and JavaScript.

Markup of the Web's Past

It wasn't that many years ago that our approach to HTML markup was something quite similar to laying out a page in a brochure, magazine or newspaper. It isn't much of a surprise though, as many of us came from a print background, or were working with designers or product owners with a print background. Before responsive design became the standard (and even some time after) most web properties were treated like a multi page print project. When the work was divvied up you'd be assigned a page and you would start at the top and work your way down the DOM.

In the past our markup typically fell into one of two camps.

Dynamic Markup: 100% Automation - 0% Control

In the world of web publishing it was pretty common for the front-end team to have little to no control over the markup. This was often due to feature development

(which included HTML output) being done weeks or even months before the front-end team came onto the project. It was made worse by the fact that the origins of the markup were obfuscate by complex rendering processes, and did not come from a single template. This made updating the markup extremely difficult for anyone not familiar with the complexities of the CMS back-end, and as the back-end devs had moved onto other tasks, there was rarely time to go back and make any major changes to it.

The effect of this constraint is that CMS's and the back-end developers would err on the side of too much markup and too many classes in order to give the "themer" the ability to target any element on the page, and never be without a wrapper div. In the end, we'd get this:

```
<div id="header" class="clearfix">
  <div id="header-screen" class="clearfix">
    <div id="header-inner" class="container-12 clearfix">
      <div id="nav-header" role="navigation">
        <div class="region region-navigation">
          <div id="block-system-main-menu" class="block block-system block-menu">
            <div class="block-inner">
              <div class="content">
                <ul class="menu">
                  <li class="first leaf">
                    <a href="/start" title="Getting Started with Drupal">Get Started</a>
```

This little snippet, pulled directly from the Drupal.org homepage, shows how your header can have 10 levels of nesting before getting to a single piece of content. The sad thing is that this is a relatively tame example! I can tell you from experience that it can go much deeper.

This 'div soup' might have helped us when our job was to match a static photoshop comp to a page full of markup, but as our needs matured, we longed for more control.

Static Markup: 0% Automation - 100% Control

If we were working on a small project, or simply had a page where we had a huge body field to fill out, controlling the markup was a pretty simple process. While this situation offered great flexibility, it also meant that we were responsible for maintaining all of the code. Changes that would be simple in a CMS template had to be propagated throughout the page by hand. So we'd write markup like this:

```
<header>
  <section>
    <nav>
      <div>
        <ul>
          <li>
            <a href="/start">
```

To keep things simple, 'semantic' markup was preferred, relying on HTML 5 elements and their relative positioning in order to apply styles. Without classes on our markup, and having been burned by primary level styles bleeding down to secondary level anchors, we often ended up with long descendent selector chains like this:

```
header > section > nav > div > ul > li > a {
  color: white;
}
```

This specificity nightmare ensured that every selector we wrote for our hover and active states was at least this long. You don't even want to see how secondary navigation is styles. Let's move past our past and look at more current practices.

Striking a Balance Between Control and Automation

As the Front-end Architect it will be important to evaluate the processes that produce your markup. How much control will you have over the order of the content, the elements used and the CSS classes applied to them. How difficult will it be to change these things in the future? Are templates easily accessible, or will you need to task a back-end developer with the change? Is your markup even based on a templating system? Can you propagate changes throughout your entire system, or is it a manual process? The answers to these questions might dramatically change the way your approach CSS.

Modular Markup: 100% Automation - 100% Control

The utopian state that we are all striving for is a situation where every line of HTML in our site is programmatically created, yet we have full control over the templates and processes used to create that markup. It's very possible that you'll rarely reach this state. Even in the best situations there is user generated content that will have little to no automated markup, and regardless of a CMS's ability to expose HTML templates, sometimes it's just easier to let the CMS determine the markup for things like your forms or navigation. But even if you stand at 90% 90%, a Modular approach to your markup will provide you the flexibility you want with the automation you need.

Modular markup differs from dynamic markup in that we no longer cede power over to the CMS to determine what markup should be used to output any given content. This allows us to visually connect two elements on the page by using the same markup, even though the CMS would have used completely different markup. Modular markup also differs from static markup in that, being programmatically created, we have the ability apply a system of classes to the markup and not rely on the element tags and position to determine their visual appearance. Lets look at that navigation example again with a generic modular approach.

```
<nav class="navigation">
  <ul class="navigation-container">
    <li class="navigation-item">
      <a href="/start" class="navigation-link">
```

At first glance this approach seems quite verbose! And while I will not argue that point, I will argue that it is the perfect level of verbosity. With a class on every element we no longer have to rely on styling tags or using element position to determine visual appearance. Compared to the dynamic markup, this markup is much cleaner, and dare I say more "modular"? This navigation pattern could be used in several places throughout the site. It is not markup that was first printed by the CMS and then styled. It is markup that was created, styled, and then integrated into the website's navigation system.

It All Leads to a Design System

So how do we get started with this modular approach? Well it all starts with changing the way that we create our pages. You see, there is no page. It's a myth. A website page is a relic of our past. What is a page? Is it the content at a certain URL? Well what is the guarantee that the content at a given URL remains the same each time you visit? What happens if you are logged in? What happens if the content on that page is filtered by the time of day, your location, or your browsing activity? The sooner that we realize we are no longer building pages, but Design Systems, the sooner we can start to create some amazing things on the web.

A Design System is the programatic representation of a website's Visual Language. The Visual Language is an artifact created to express how the website visually communicates its message to users. It is a collection of colors, fonts, buttons, image styles, typographical rhythms, and UI patterns.

Just as a spoken language can broken down into nouns, verbs and adjectives, our job, as front-end developers is to deconstruct this visual language into its smallest pieces, so that we can create rules about how to put it back together again. It's by breaking down the visual language that we learn how we can use it to create various sentences, paragraphs, chapters and novels. The goal of this conversion is to create a scaleable and maintainable code base that faithfully reproduces anything that the language is able of expressing.

We'll dive further into creating design systems in future chapters, but it is important that we understand what we are creating, because before we do that, we need to decide how the design system is going to be attached to our markup.

The Many Faces of Modular CSS Methodologies

There are almost as many CSS methodologies today as there are CSS or JavaScript frameworks. But unlike CSS or Javascript frameworks, which are usually all or nothing, and come with a bit of baggage, a CSS methodology is more of a philosophy about the relationship between HTML and CSS than a structured codebase. It seems that almost every day you hear about a new approach using some new namespace, leveraging data attributes, or even moving all of the CSS into JavaScript. The great thing about all of these methodologies is that they all bring something interesting to

the table, and help you to learn something new about how HTML and CSS can be intertwined. There is absolutely nothing wrong with creating your own methodology, or starting with a popular one and then modifying it to your taste.

There is no single perfect approach, and you might find that one project fits best with one, and another project works better with another. So if you are wondering where to start when deciding on your own approach, it is best to take a look at a few of the more prominent methodologies, and see what does, and what does not resonate with the project you are about to tackle.

OOCSS Approach

```
<div class="toggle simple">
  <div class="toggle-control open">
    <h1 class="toggle-title">Title 1</h1>
  </div>
  <div class="toggle-details open"> ... </div>
  ...
</div>
```

The two main principles of OOCSS is to separate structure and skin, and to separate container and content.

Separating structure from skin means to define visual features in a reusable way, so that they can be reused. This simple toggle element is small an reusable in many different situations. It can be displayed using various skins which will alter its physical appearance. The current skin of "simple" might have square corners, but the "complex" skin might have rounded corners and a drop shadow.

Separating container from content means to stop using location as a style qualifier. Create reusable classes like 'toggle-title' that will apply the required text treatment regardless of what element it is used on, and let an H1 look like the default H1 if no class is applied to it.

This approach is very useful when you want to provide your developers with a large set of components that they can mix and match to create their UI's. A great example of this approach is Bootstrap, a system full of small objects adorned with various skins. The goal of Bootstrap is to create a complete system that is capable of creating any UI that a developer might need to put together.

SMACSS Approach

```
<div class="toggle toggle-simple">
  <div class="toggle-control is-active">
    <h2 class="toggle-title">Title 1</h2>
  </div>

  <div class="toggle-details is-active">
    ...
  </div>
```

```
  ...
</dl>
```

SMACSS (Scalable and Modular Architecture for CSS), while it shares many similarities to OOCSS, is an approach differentiated in the way that it breaks the system of styles into five specific categories.

1. Base - How markup would look without classes applied to it
2. Layout - Dividing the pages up into regions
3. Module - The modular, reusable parts of your design
4. State - Describes the way that modules or layouts look under given states or contexts
5. Theme - An optional layer of visual appearance that lets you swap out different themes

In the above example we see a combination of module styles (toggle, toggle-title, toggle-details), submodule (toggle-simple) and state (is-active). There are many similarities between OOCSS and SMACSS in how they create small modular pieces of functionality. They both scope all of their styles to a root level class, and then apply modifications via a skin (OOCSS) or submodule (SMACSS). The only real differences between the two (other than SMACSS opinions about code structure) is the use of skins instead of submodules, and the 'is' prefixing of the state classes.

BEM Approach

```
<div class="toggle toggle--simple">
  <div class="toggle__control toggle__control--active">
    <h2 class="toggle__title">Title 1</h2>
  </div>

  <div class="toggle__details toggle__details--active">
    ...
  </div>
  ...
</dl>
```

BEM, or Block Element Modifier, is the third methodology we are going to look at, and is on the other side of the spectrum from SMACSS. BEM is simply a class naming convention. It makes no opinion about the structure of your CSS, it only suggests that every element is labeled with a class that describes:

- Block - The name of the parent component
- Element - An individual element inside of the block component
- Modifier - Any modifier associated with the block or element

BEM uses a very terse convention for creating these class strings, which can become quite long. Elements are added after a double underscore 'toggle__details' and modifiers are added after a double dash 'toggle__details--active'. This makes it very clear

that ‘details’ is an element and that active is a modifier. The use of a double dash also means that your block name could be ‘news-feed’ without confusing feed for a modifier.

The advantage of this approach over OOCSS or SMACSS is that every class is fully descriptive of what it accomplishes. There are no “open”, or “is-active" classes. While those classes make sense in their context, outside of that context we don’t have a clue what is open, or is-active. While BEM might seems redundant or overly verbose, when we see a class of toggle__details--active we know exactly what it means: the details element, inside of the toggle block, is active.

Picking What is Right For You

In the end, picking a solution that works for you is always the only thing that matters. Don’t pick a convention because it’s popular, or because another team is using it. All three approaches give you extremely similar tools to work with, and will integrate with a Design System in very similar ways.

At Redhat.com we settled on a mix of SMACSS and BEM, which I will get into at the end of this pillar. So don’t be afraid to experiment, combine ideas, or come up with something completely unique! Just be aware of the prior art, be able to express why your approach will solve the challenges your project faces, and have a team willing to commit to a single, unified approach. If you decide to use OOSMABEM, then more power to you! I look forward to reading about it.

The greatest things about this industry is that we can freely sit down with fellow developers and have a cup of coffee. The greatest thing? Really? Yes, really. And here's why. We work in an industry based on open standards, open source software, open information and open learning. The tools and techniques we use may be changing at an incredibly fast pace, but working in such an open industry is the key reason we're able to keep up with them. Now this may surprise you, but there are other industries where you would never, EVER sit down with a fellow practitioner to talk shop, unless they were paying you to do it. In these industries every piece of knowledge, every trick, every preset, every macro, every document, every shortcut is up for sale, and the last thing you'd want to do is sit down with a potential competitor and freely trade that information.

Now compare this to the web. We thrive on sharing knowledge. We publish blogs, record video tutorials, create public code repos, write on Stack Overflow, respond to questions on IRC, distribute Codepens, Gists, Sassmeisters, JSbins, and Pastebins, all so that others can learn the things that we know. Getting a cup of coffee and discussing your views on current web practices and new CSS frameworks is the most basic expression of how we share knowledge and how we learn in this industry.

So yeah! We work in an industry where inviting an associate out for a cup of coffee is not only acceptable, but it is a valuable practice! These cups of coffee can lead to things learned, business connections, new jobs, and even great friends. Suffice it to say, many years of talking shop over cups of coffee, beer, tea, or kombucha, has lead me to believe that this social interface is one of the greatest assets we have in this industry.

The best part about having consistent caffeine or alcohol fueled conversations with other developers is that I always have a pulse on what people are excited about, and in what direction they are headed. This is important to me not because I depend on others to know what to do next, but rather because I am able to validate the things I am learning and the discoveries I, myself am making. Every year I look back at how my approach to CSS has evolved. So wether my newest obsession is preprocessors, build tools, styleguide driven design, or component based design systems, I excited find that others are coming to the exact same discoveries.

The CSS I am writing today looks nothing like what I was writing even 3 years ago. Site build after site build I learn something new and try to apply that to my next project. It's a very organic process of gathering the things I've learned, with the things I've read, and trying to apply an improved approach to the unique problems I face in my next project. Each iteration brings a marked improvement in my technique, and my understanding of a scalable and maintainable design system. Along with each iteration are the excited conversations with various co-workers, industry colleagues, or random conference goer. With each of those conversations I'm consistently

amazed that I was not the only person to have a euphoric epiphany about using data-attributes to handle my component variations.

As a Front-end Architect, you might not need to know every tiny CSS bug in some obscure version of Opera mini, but you do need to understand the major trends CSS, and be able to put together a plan that will set your team up for success. If you haven't been keeping up on the current CSS trends, or your euphoric epiphanies aren't coming as fast as you wish they would, let me catch you up on where we were a few years ago, why it didn't work, and where we are not.

Specificity Wars and the Pains of Inheritance

It wasn't that many years ago that we were still dealing with the 100% dynamic, or 100% static markup that I described in the previous chapter. Regardless of the side of the spectrum we were on, the effect on our CSS was that we almost always started from a global scope and worked our way down, getting more specific with each new level of the cascade. We'd start with general styles on the each elements, like our header and paragraph tags, and then apply specific styles to the elements inside of the various sections of our page.

```
<body>
  <div class="main">
    <h2>I'm a Header</h2>
  </div>
  <div id="sidebar">
    <h2>I'm a Sidebar Header</h2>
  </div>
</body>

<style>
h2 {
  font-size: 24px;
  color: red;
}

#sidebar h2 {
  font-size: 20px;
  background: red;
  color: white;
}
</style>
```

Now every H2 is red, except for the sidebar where the H2 is white with a red background.

This concept was easy to understand and made a ton of sense coming from a print background. Every time you put an H2 in the sidebar, it would be styled the same. That is until we come across a calendar widget in the sidebar that should be using the

original header colors. But that's okay, we could just add another class and overwrite the offending sidebar styles!

```
<body>
  <div id="main">
    <h2>I'm a Header</h2>
  </div>
  <div id="sidebar">
    <h2>I'm a Sidebar Header</h2>
    <div class="calendar">
      <h2>I'm a Calendar Header</h2>
    </div>
  </div>
</body>

<style>
h2 {
  font-size: 24px;
  color: red;
}

#sidebar h2 {
  font-size: 20px;
  background: red;
  color: white;
}

#sidebar .calendar h2 {
  background: none;
  color: red;
}
</style>
```

The issues with this approach are numerous:

1. *Specificity:* Wether you're dealing with ID tags, or just long selectors, overwriting a selector always requires some attention to the level of specificity.
2. *Resetting colors:* To get back to the original H2 color we have to specify it again, as well as overwrite the background.
3. *Location dependance:* Now our calendar styles are dependent on being in the sidebar. Move the calendar to the footer and the header size will change.
4. *Multiple Inheritance:* This single H2 is now getting styles from 3 different sources. This means we can't change the body or sidebar H2's without affecting the calendar
5. *Further Nesting:* The calendar widget might have other H2's inside of individual calendar entries. Those H2's will need an even more specific selector, and will now be dependent on 4 sources for its styles.

A Modern, Modular Approach

The previous chapter on HTML foreshadowed on a few of the modern, modular tenets that the majority of frameworks are employing to deal with the problems of the above approach. While each had pretty differing opinions about the exact markup, many of the opinions each bring about CSS are either explicitly shared, or at least extremely compatible. Let's take a quick look at a few of those key tenets and how they help solve the problems we had above.

OOCSS brings the idea of "*Separating container from content*" where we learn to stop using location as a style qualifier. There is nothing wrong with having a sidebar on your site, and to style that sidebar in whatever way you'd like, but the influence of those sidebar styles stop once you get down to the contents inside of the sidebar. "#sidebar h2" means that every H2 element placed into the sidebar is going to have to either accept, or fight off the styles applied by that selector. "#sidebar .my-heading-class" means that a single heading can opt in to that style while the calendar module's heading, and ever every other heading tag in the sidebar remains untouched.

SMACSS brings us the idea of separating our layout and our components into completely different folders, further creating a divide between the role of the sidebar and the role of the calendar module. Now that we understand the sidebar's role is one of layout, we don't even allow element styles inside of that partial. If you are going to place something into the sidebar and want to have it styled, that element needs to be part of a component, and defined in the component folder.

BEM, while not necessarily a CSS methodology, teaches us the value of having a single source of truth for every class used in your markup. Instead of classes that ebb and flow depending on their context, or proximity to other selectors, each BEM class can be traced back to a single set of CSS properties unique to that selector.

```
<body>
  <div class="main">
    <h2 class="content__title>I'm a Header</h2>
  </div>
  <div class="sidebar">
    <h2 class="content__title--reversed>I'm a Sidebar Header</h2>
    <div class="calendar">
      <h2 class="calendar__title>I'm a Calendar Header</h2>
    </div>
  </div>
</body>

<style>

/* Components Folder */
.content__title {
  font-size: 24px;
  color: red;
}
```

```
.content__title--reversed {
  font-size: 20px;
  background: red;
  color: white;
}

.calendar__title {
  font-size: 20px;
  color: red;
}

/* Layout Folder */
.main {
  float: left;
  ...
}
.sidebar {
  float: right;
  ...
}
</style>
```

The issues with this approach are fixed:

1. *Specificity:* Changing your ID's to classes is a good start in stopping the specificity wars, but even better is flattening every selector's specificity to "1" and stop using the specificity "winner" to determine the applied styles.
2. *Resetting colors:* Even better than lowering specificity is only using a single selector to apply styles to each element. This way your module styles never have to fight with your sidebar or site wide styles.
3. *Location dependance:* With no styles scoped to a single layout, we don't have to worry about what happens to the calendar when we move it from the sidebar to the main section.
4. *Multiple Inheritance:* With each of the 3 headings getting their own unique class, we are free to change any of them without fear of affecting the other. If you want to make changes across multiple selectors look into preprocessor variables, mixins or extends to handle that for you.
5. *Further Nesting:* Even at the calendar level, we still haven't applied a single style to our "H2" elements. We have a clean, blank slate to work from inside of our calendar module. No need to override base, sidebar and calendar styles before writing a new set of heading styles.

Other Principals to Help You Along the Way

Single Responsibility Principle states that every thing you create should be created for a single, focused reason. The styles you apply to a given selector should be created for a single purpose, and it should do that single purpose extremely well.

This doesn't mean that you have a individual classes for "padding-10", "font-size-20" and "color-green". The single purpose we're talking about is not the styles that they apply, but rather where the styles are applied to. Let's look at the following example:

```
<div class="calendar">
  <h2 class="primary-header">This is a Calendar Header</h2>
</div>

<div class="blog">
  <h2 class="primary-header">This is a Blog Header</h2>
</div>

.primary-header {
  color: red;
  font-size: 2em;
}
```

While the above example appears to be quite efficient, it has clearly broken our Single Responsibility Principle. The class of ".primary-header" is being applied to more than one, unrelated element on the page. The "responsibility" of the primary-header is now to style both the calendar header and the blog header. This means that any changes to the blog header is also going to affect the calendar header unless you do the following.

```
<div class="calendar">
  <h2 class="primary-header">This is a Calendar Header</h2>
</div>

<div class="blog">
  <h2 class="primary-header">This is a Blog Header</h2>
</div>

.primary-header {
  color: red;
  font-size: 2em;
}

.blog .primary-header {
  font-size: 2.4em;
}
```

This approach, while effective in the short term, brings us back to several of the problems we had at the beginning of the chapter. This new header style is now location dependent, it has multiple inheritances and has now introduced a game of "winning specificity".

A much more sustainable approach to this problem is to allow each class to have a single, focused responsibility.

```
<div class="calendar">
  <h2 class="calendar-header">This is a Calendar Header</h2>
```

```
</div>

<div class="blog">
  <h2 class="blog-header">This is a Blog Header</h2>
</div>

.calendar-header {
  color: red;
  font-size: 2em;
}

.blog-header {
  color: red;
  font-size: 2.4em;
}
```

While it's true that this approach can cause some duplication (declaring the color red twice), the gains in sustainability greatly outweigh any duplicated code. Not only will this additional code be a trivial increase in page weight (gzip loves repeated content), but there is no guarantee that the blog header will remain red and enforcing the single responsibility principle throughout your project will insure that further changes to the blog header are done with little work, or possible regressions.

Single Source of Truth takes the Single Responsibility Theory to the next level in that not only is a class created for a single purpose, but that the styles applied to that class come from one, single source. In a modular design, the design of any component must be determined by the component itself, and never imposed on it by a parent class. Let's take a look of this in action.

```
<div class="blog">
  <h2 class="blog-header">This is a Blog Header</h2>
  ...
  <div class="calendar">
    <h2 class="calendar-header">This is a Calendar Header</h2>
  </div>
</div>

/* calendar.css */
.calendar-header {
  color: red;
  font-size: 2em;
}

/* blog.css */
.blog-header {
  color: red;
  font-size: 2.4em;
}

.blog .calendar-header {
```

```
  font-size: 1.6em;
}
```

The intention of the above styles are to decrease the size of the calendar header when it is inside of a blog article. From a design standpoint that might make perfect sense, but what you end up with is a calendar component that seemingly changes appearance depending on where it is placed. This conditional styling is what I like to call a "context", and is something I use quite extensively throughout my design systems.

The main problem with the above approach is that the decreased font size originates from the blog component, and not from within the calendar component. In this case there is not a single source of truth, the truth is scattered across multiple components. The challenge of "multiple sources of truth" is that it makes it very difficult to anticipate how a component is going to look placed on the page. To mitigate this problem, I would suggest simply moving the contextual style into the calendar module code.

```
<div class="blog">
  <h2 class="blog-header">This is a Blog Header</h2>
  ...
  <div class="calendar">
    <h2 class="calendar-header">This is a Calendar Header</h2>
  </div>
</div>

/* calendar.css */
.calendar-header {
  color: red;
  font-size: 2em;
}

.blog .calendar-header {
  font-size: 1.6em;
}

/* blog.css */
.blog-header {
  color: red;
  font-size: 2.4em;
}
```

With this approach, we are still able to decrease the size of the calendar header when it is inside of a blog article, but by placing all of the "calendar-header" contextual styles into the calendar file, we can see all of the possible variations in a single location. This makes updates to the calendar module easier, as we know all of the conditions that it might change, and allows us to create proper test coverage for each of the variations.

Component Modifiers

While this approach does improve clarity, it can become a bit difficult when you end up with dozens of contexts for each possible containing component. In a case where the calendar header is smaller inside of dozens of different contexts, it might be time to switch from contextual styles to modifier classes.

Component modifiers, or skins, or sub-components (depending on the methodology you prescribe to) allow you to create multiple variations of a component to be used in various circumstances. They work in a very similar way to contexts, but the qualifying class is part of the component rather than above the component.

```
<div class="blog">
  <h2 class="blog-header">This is a Blog Header</h2>
  ...
  <div class="calendar calendar--nested">
    <h2 class="calendar-header">This is a Calendar Header</h2>
  </div>
</div>

/* calendar.css */
.calendar-header {
  color: red;
  font-size: 2em;
}

.calendar--nested .calendar-header {
  font-size: 1.6em;
}

/* blog.css */
.blog-header {
  color: red;
  font-size: 2.4em;
}
```

In the above example we have created a “calendar--nested” modifier using the traditional BEM syntax. This class by itself does nothing, but when it is applied to the calendar the elements inside of the component can use it as a local context and change their appearance.

With this approach we can use this modified calendar skin anytime we want, and we will get that smaller header (along with other changes if we want). This keeps all of your component variations in a single file, and allows you to use, or not use them, anytime that you need, not making them dependent on some random parent class.

Type something here!
Last fall was the launch of the Redhat.com website. I'd only been on this multi-year project since the spring prior, so while I was pretty familiar with the code architecture at launch, I'd had little opportunity to shape that architecture. The scope of the project, and our looming deadline meant I spent most of my time getting the work done, and little time wondering if that work I was doing truly met the needs of the organization.

In the end, the site launched, and by all measures it was a success. Well, all measures except for one. You see, the site was performant enough. The UI was efficient, and few argued that the site wasn't quite attractive. But I can still recall that fateful afternoon when I was asked a very simple question..."How modular is our design? We'd like to be able to share small parts of our theme with other company websites."

It took me a little while to recover from the chuckle that welled up inside of me. You see, being quite familiar with the markup, javascript and css written for this project, I knew that this design was the antithesis of modular! We did a lot of great work on this theme, but creating a modular design was never even a consideration. If someone was looking to render a single band of content with our styles they would first need to load the following:

- Bootstrap css: 98kb

This site didn't leverage too much of the bootstrap library, but all of the CSS was written with the assumption that bootstrap had already washed all over it.

- Core Site CSS ~500kb

Though each band of content typically had a single file associated with it, the styles from that single file were never the single source of truth for that band. Styles cascaded in from several different locations and were often overridden based on location or page class.

Sure, we could pull out a single band and consolidate all of the required styles into a single file, but doing so would basically mean completely recreating the component Sass partial from scratch, and we'd still have a problem with trying to make the markup modular.

The markup approach that the project used was to style from the band down. We had several types of bands, and most of the styles were scoped to and repeated inside of that band class. Below is an extreme example of a H3 inside of a hero band.

```
.about-contact .hero1 .container>section.features-quarter>section.f-contact h3
```

You can see how not only is this style scoped to a single page (about-contact), but we need to make sure that the features-quarter section is the direct decedent of the the

container class so that we aren't accidentally styling descendants of the wrong section element! This top down styles approach meant that every change we did required longer and longer, more specific selector. But it also meant that none of the contents of a band could be easily rearranged or replaced as the markup order was incredibly strict.

So, when I was asked about how modular our design was, and if we could start sharing out styles out to other departments, there was only one thing that I could say. I said that we'd have to completely rewrite the markup and the CSS for anything we wanted to share, and while we were at it, we should update the markup on our site as well.

I was quite certain that such a drastic departure from our current design approach would laughed right out of the room, so you can understand my surprise when not only did they say "yes", but that we were given several weeks to work out the new system while the newly launched site was in code freeze. So here we had a fully fleshed out design, a very capable development team, and carte blanche to create a modular, scaleable and sustainable design system that could be deployed into a live, highly trafficked, high profile site as we built it. All I could think was…..where do we start!

Breaking the Design Down

As I said above, the original approach we took was to design from the top of the band down. The appearance of the contents and the way that it was laid out was completely determined by the type of band we were using. We had logo walls, a hero band, testimonials bands, blog teaser bands, testimonial bands, you name it. Each of those bands had their own sass partial, and all of the styles were scoped under the band name.

In a way this worked well. You usually knew where to go to update a given band's styles. But the problem was that we had to continually create more and more bands every time we needed to support a new design. We had a well established visual language, but that had never been translated into a design system.

So the first task we undertook was breaking the design down into its smallest possible pieces. We knew that once we had the building blocks of this design system, we could create anything that the visual language needed to communicate. The first step was to look at our designs and break them down into repeatable layout patterns.

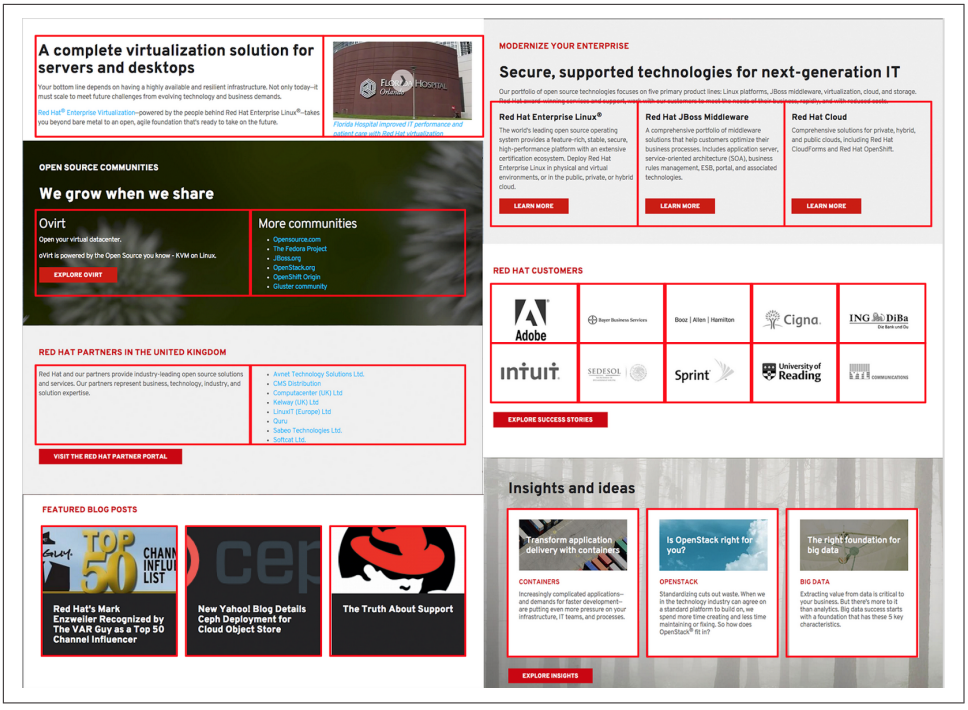


Figure 2-1. The various layouts of our design system

As we took a look at several of our most common band types, we noticed that most of them shared a number of common layout patterns. We had content with a sidebar, equal sized content in columns of 2 or 3, we had galleries of images or icons that spanned 5 per row, and we had black and white cards that held their content inside a bit of padding and a background.

We quickly realized that if created a method for reusing these few simple layout concepts, we could create the layout for every single band in our entire website. We wouldn't need a logo wall band with a specified layout, and a insights band with a different layout. We could simply have a single band that had many layout options. In the same way we could create card layouts that did nothing more than apply padding and a background to the contents inside of them.

In the same way as we broke out all our various layouts, we started taking a look at the contents of each of these layouts and realized we could reproduce a majority of them with just a small handful of small components.

A component, as compared to a layout, describes the visual appearence of a small piece of content. Components are meant to be dumb and flexible. They have zero opinion about their backgrounds, their widths, or their external padding and margins.

The power of this relationship between layouts and components can be found in the fact that you can drop any 3 components into a 3 column layout and it'll look like they were meant to go there, all without a single extra bit of code. Each component will fill the entire width of the column it was placed in, and the first line of the block quote will be level with first line of the blog entry, which will line up with the image in the 3rd column.

Figure 2-2. No component has top margins, so three different components will all line up.

Once we realized that this powerful system was based on a few simple, yet highly important relationships, we set down to codify these relationships in a more official form. We wanted to make sure that as we introduced new layouts and components to the system, each accepted merge request would conform to a series of rules. We called these our Road Runner Rules.

The Road Runner Rules

Shortly prior to us contemplating these new rules, my twitter stream had been bombarded by a story about Chuck Jones and a list of 9 rules that he'd written for the writers and animators of the Road Runner cartoon. This list set forth the rules under which the entire Road Runner Universe was meant to run. Rules like "The road runner cannot harm the coyote, except by going 'beep beep'", and "No dialog, ever, except 'beep-beep'" helped guide the animators and writters of each cartoon to create a consistent and cohesive universe.

It was this consistant and cohesive universe I wanted to create for my own team. I knew that the only way we could avoid writing a swift road-runner kick to the coyote's jaw, followed by a fowl cry of "can't catch me sucker!" would be to distill the rules that govern this relationship into a small paletable list of Road Runner Rules.

Our rules are the following:

1. A **layout** never imposes padding or element styles on its children. It is only concerned with their horizontal or vertical alignment and spacing.
2. **Themes** and other data attributes never force changes in appearance, they are always a context that layouts, components and elements can subscribe to.
3. A **component** always touches all four sides of its parent container. No element will have top or left margins and all last children (right or bottom) will have their margins cleared.
4. The **component** itself never has backgrounds, widths, floats, padding or margins. Component styles only target the elements inside.
5. Every **element** has a single, unique, component-scoped class. All styles are applied directly to that selector modified only by contexts and themes.
6. **Elements** never use top margins. The first element touches the top of its component.
7. **Javascript** is never bound to any element class. Functionality is "opted in" via data attributes.

These rules cover the specific relationship between layout and component, but they also cover other areas of the design system including themes, elements and javascript.

Now just like a good looking styleguide has a styleguide that defines how it should be presented, and a solidly written schema is written to the spec of its parent schema, our Road Runner Rules are written with their own set of governing rules.

Now I know what you're thinking...That's excessively meta and simply an excercise in absurdity to create rules for your rules, but no! When I first wrote my list of Road Runner Rules, trying to describe the system I was trying to protect, I started with almost twice as many rules as we have now. As I neared a good baker's dozen rules I found that with each rule I wrote, I would think of 2 or 3 more.

With the thought making a set of rules old enough to buy me a drink, I realized that I wasn't making rules anymore, but rather I was writing the documentation for our entire system. And the problem with THAT was that I already had written our documentation! What I needed was a small set of immutible rules, not a fully developed set of instructions.

I didn't need to describe how the coyote would order products from an Acme catalog and that they would show up seconds later in his mailbox, but I did need to enforce that no "no outside force [could] harm the coyote", and that it was only his ineptitude or the failure of those Acme products that could harm him. I knew I had to weed out my list, and look for the nuggest of absolute truth beneath them. So revision after revision our team reworded, rewrote and deleted items on my list until we had a

cohesive list of rules that all followed a common pattern. These common patterns are what lead to our Rules of the Road Runner Rules. They are:

1. Only include immutable rules, not general instructions
2. Always boil each rule down to its most simple expression
3. Always state the rule first, then explain "If not that, then what".
4. Every rule should include one of the following: always, never, only, every, don't, or do.

These rules helped us to avoid writing general instructions containing several sentences that never actually got to a point that could be used to just incoming code. Astute readers will also note that all 4 of these rules actually comply to the Rules of the Road Runner Rules.

These rules were derived from, and describe a design system of layouts, components and elements. Let's dive into some of the more interesting decisions we made regarding our approach to HTML and CSS.

A Single Selector for Everything

I've spent too much of my life creating generic, universal classes that could be applied to any element, only to realize how difficult they were to maintain as the project grew. Because the classes were universal, and could be used on just about anything, it was usually easier to create a new class than to update the original one, due to the chance for visual regressions. Therefore, one of the things I like the most about the BEM (Block Element Modifier) approach is the idea that every single element on the page has a single, unique, descriptive class applied to it.

Single Responsibility Principle

In most circles the Single Responsibility Principle, applied to CSS, means that each class has a small, focused responsibility, and that one class will set the box model properties of an element, while another sets the typography, and a third sets the color and background.

But for me, the Single Responsibility Principle, applied to CSS, means that every class I create is created to be used for a single purpose, in a single place. This means that if I make a change to ".rh-standard-band-title", I can be confident that the only effect this will have on our site is to change the appearance of the title of rh-standard-band.

This also means that if we decide to deprecate "rh-standard-band", we can completely remove all of the associated CSS without fear of breaking some other component that "hijacked inheritance", and became reliant on that CSS. It's because of this desire to not "Use and Abuse the Cascade" that I make sure that every class is only used for the single purpose that it is created.

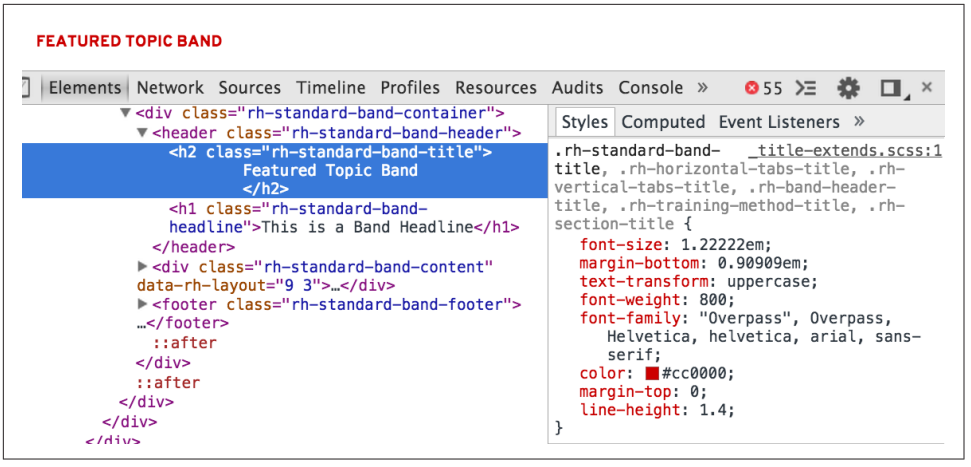


Figure 2-3. Each class is created for a single purpose

Single Source of Truth

Once we have a page full of single classed elements we can be pretty confident that our changes to "rh-standard-band-title" won't affect any other part of the system, but what is to say that our "rh-standard-band-title" can't be affected by something else. This is why it is so important to maintain a Single Source of Truth for every component, and by extension, every element on the page. This means not relying on any H2 styles, or "header H2", or any other selector outside of ".rh-standard-band" to modify this element on the page.

This is not to say that our title can never be altered or modified by an outside force. On the contrary, I'm going to talk about modifiers and contexts next. What this does mean is that anything that these modifiers or contexts do to an element will be defined in the same place as the element's original styles, not in some context partial. So while I have no objections to ".some-context .rh-standard-band-title", these styles will always be defined in the "standard-band" Sass partial, and never anywhere else.

Opt-in Modifiers

As I mentioned above, I have no objections to having modifiers on my components, but in every single instance these modifications need to be opt in. What this means is that if I define a modifier for Component A, that same modifier will have absolute no effect on Component B unless I specify in the Component B partial what that modifier does.

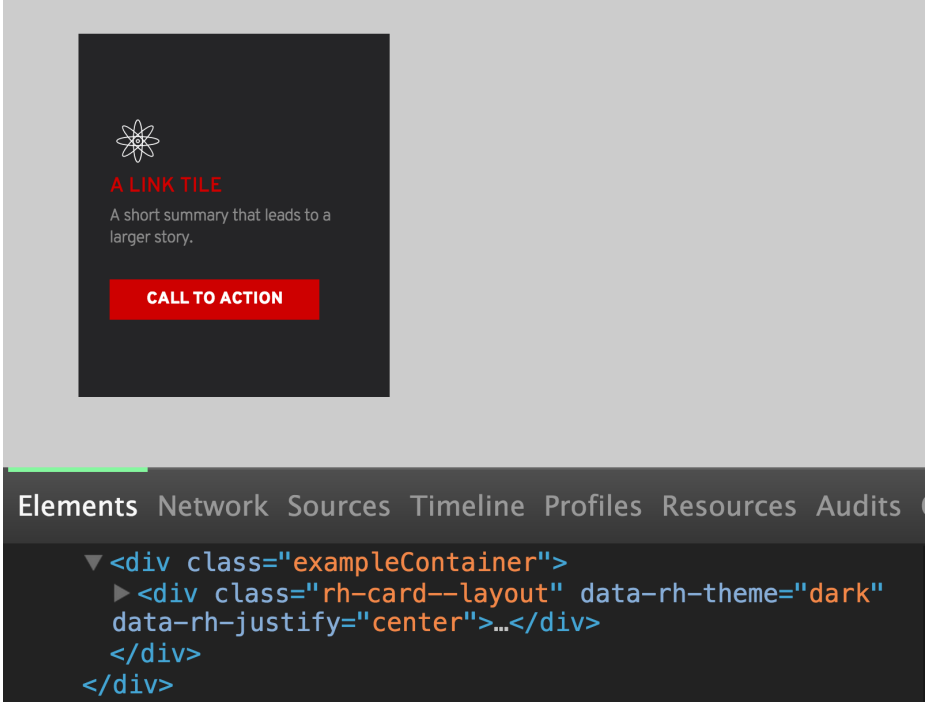
Before we dive into an example, let me explain one architectural choice I made in regards to modifiers and contexts (below). While BEM, SMACSS, and OOCSS all have conventions for modifiers, themes, or skins, they all require adding modifying classes to the block or element. I decided to take a different approach that wouldn't

require any additional classes. I was really determined to, as Ben Frain puts it, "Let the Thing be the Thing". I never wanted anyone to be confused as to what was the "Thing's" class, and what was a modifier, so I decided that all modifiers and contexts would be put inside data attributes, like this.

```
<div class="foo" data-bar="baz">...</div>
```

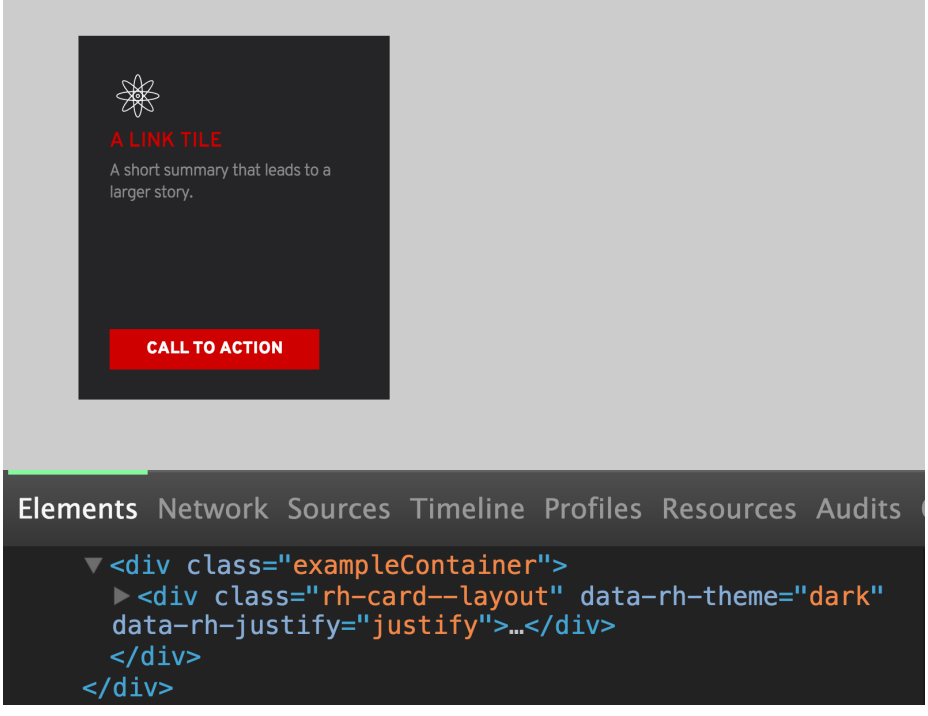
This separation had another benefit other than distinction of purpose and role. Classes are very one dimensional, whereas a data attribute is two dimensional, having the attribute itself and the value passed into it. So while you'll find classes using a namespace to define their parent "align-left, align-right, align-center", a data attribute has an explicit namespace, and can therefore pass any necessary value into it "data-align='left', data-align='right', data-align='center'". Sure, it's a few more characters, but using data attributes makes it really obvious that our component has a property of "data-align" and that it can be set to a variety of values.

Ok, now back to that example.



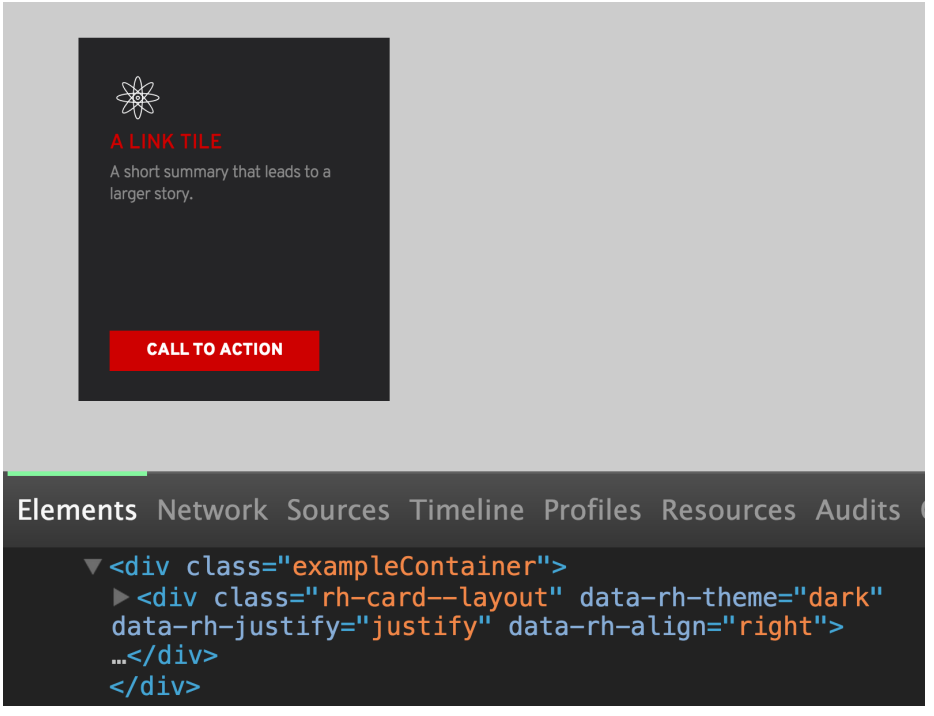
The `rh-card--layout` is layout tool that we use to wrap content inside of padding and background. As you can see above, we've gone with a dark theme, and this has an effect on the card because we've defined what `data-rh-theme="dark"` does to the card in our card--layout Sass partial. In the same way we've defined that `data-rh-`

`justify="center"` will center all of the content in the card using a little flex-box magic. Along with "center" we've also specified other justify values, including simply "justify".



So with a single switch in a data attribute, we can modify the appearance of this card based on values that we set right inside of the same Sass partial that defines the card's padding and other styles.

Now one thing that we have NOT defined in the card Sass partial is alignment. We typically leave left/right/center alignment as an option for individual components using `data-rh-align`, so we've intentionally chosen to not opt-in to that property. This means that no matter what modifiers other components or layouts might create, they will have zero effect if applied to the card.



Opt-in Context

One of the mantras of our design system is that a component should look the same regardless of where you place it. This means that our H2's aren't styled properly because they are in the sidebar, and it also means that if we drop a component into the footer that its styles aren't going to get blown away by some footer specific styles.

But just as we want our components to be resilient and predictable, we also want them to be smart and flexible. This is the reason that we came up with a context system. A context style means that a component can define the way it behaves when it is inside of some other parent element, or when a parent has a specific data attribute. Again, we aren't setting all H2's in our sidebar to be green, but if we had a `.widget-title` that needed to be green when it was inside the sidebar, we could do that! Lets look at at our card example again.

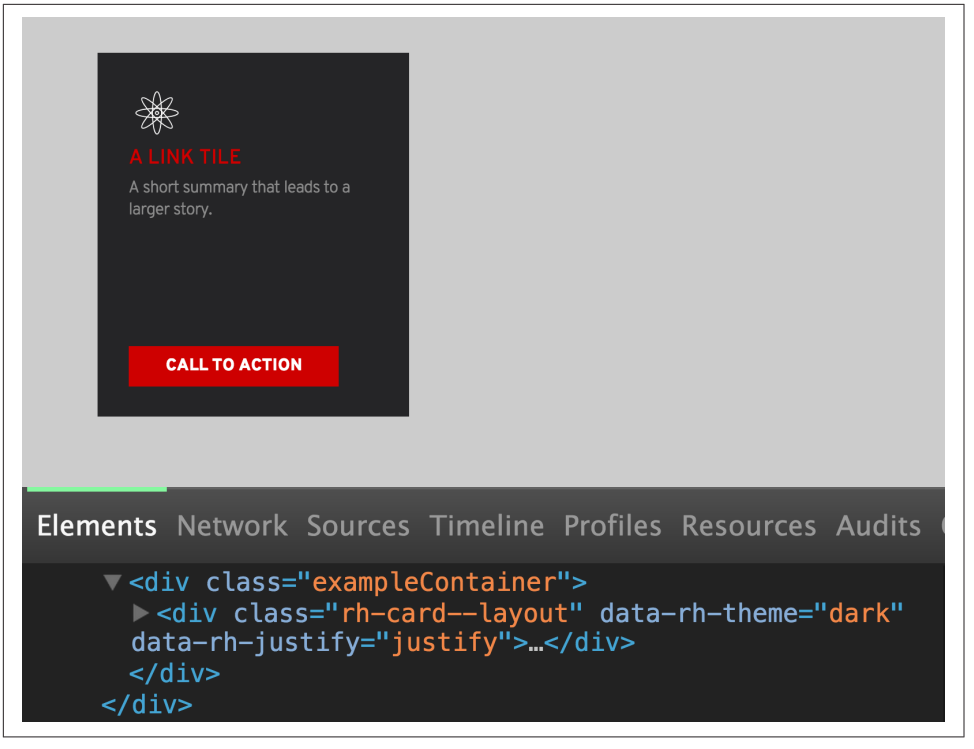


Figure 2-4. This dark theme card has a white icon

The `data-rh-theme` property actually acts as a modifier AND a context. It's a modifier in that it is what changes the card background from black to white, but it is also a context for the elements inside of it.

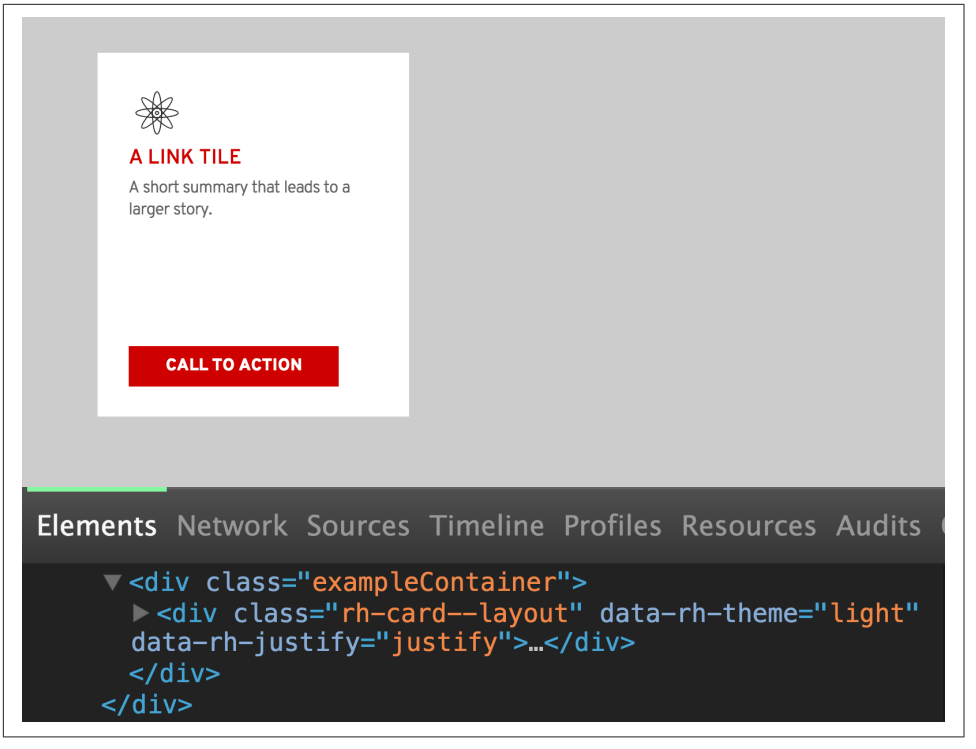


Figure 2-5. This light theme card has a black icon

When we switch from the dark to the light theme you can see that not only does the background change color (the modifier), but the icon changes color as well from white to black. I'm not saying that our context is forcing all of the child text elements to change color (the title and button didn't change at all), because the actual CSS for changing the icon from black to white lives right inside of the "link-tile" Sass partial, and is written specifically for that icon.

```
[data-rh-theme~="dark"]
.rh-link-tile-icon {
   color: white;
}
```

Figure 2-6. The icon subscribes to the dark context to turn white

```
[data-rh-theme~="light"]
.rh-link-tile-icon {
   color: #252527;
}
```

Figure 2-7. The icon subscribes to the light context to turn black

These opt-in contexts again allow us to create variations for any component without affecting the original component. They provide controlled variation, scoped to the component classes, defined solely inside of the component Sass partial.

Sure, I know. This means doing a little bit of repetitive work if we want the same modifier or context to affect multiple components. But I have never regretted this decision as our system grew. Not only could we make it easier to re-use modifiers and contexts through mixins and extends, but having a finite number of component variations helped us avoid hard to find bugs and improved our ability to provide comprehensive visual regression coverage. We could open up any component Sass partial and know without a doubt all of the component's possible variations.

Semantic Grids

With a solid understanding of how we were going to build our components, the next thing we needed to do was figure out how to combine them together into different layouts. In the past, when we were less modular, we wrote styles for each unique band, including layout. When we had a band full of logos, we gave the band a class name and then explicitly applied a layout to that band. The problem with this approach is that nothing, other than the entire band, was reusable. If we had another

band that had a list of logos, but included other content and used a different layout, we had to build an entirely new band for that content.

But now that we've broken down this logo wall to a collection of images, buttons and headers, we need similarly modular layouts that can be applied to this content. What this means is that we need to move layout back into the DOM, something we fought so hard against with the semantic layout movement. But unless we want to assign unique classes to every possible combination of components, and style each one of them individually, we're going to need to have a way to assign layout via markup.

Fortunately there is middle ground between the bootstrap grids of old with their containers, rows, and column containers, and applying unique layouts to the ".logo-wall" band, and every other new combination of content that needed layout.

Our solution was to create a collection of common grid patterns that could be applied to a layout via a data attribute. With the attribute set on the parent element, all of the child elements would fall into whatever grid value was passed in. This allowed us to keep our separation between layouts and components, while still providing a solution for setting our layouts in the markup.

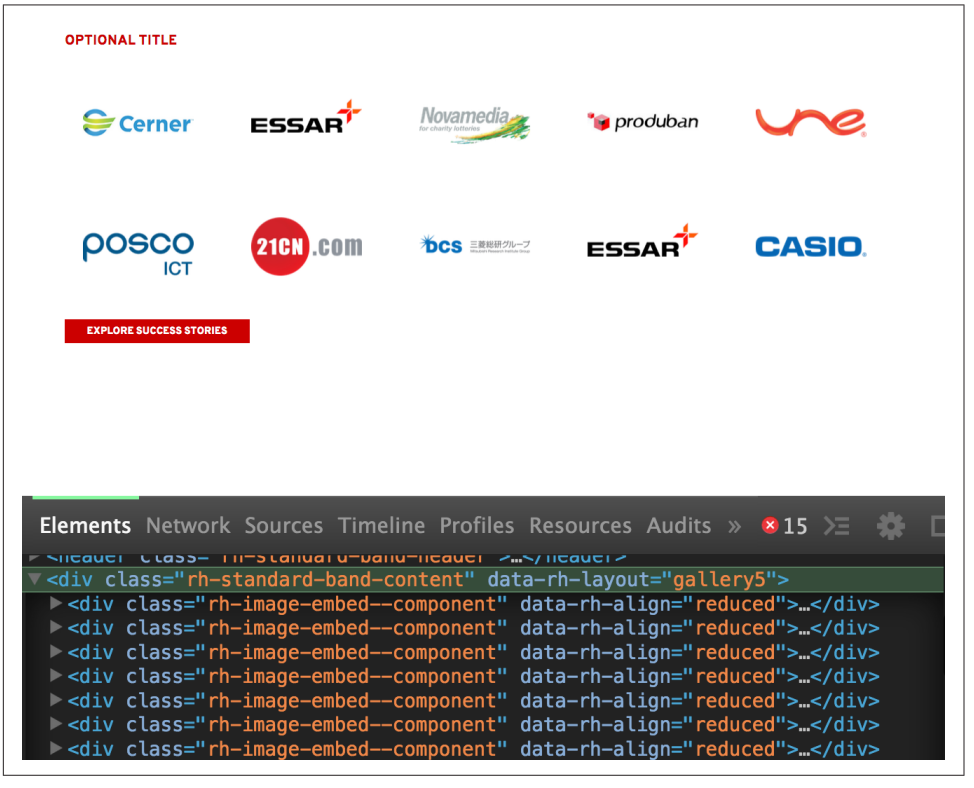


Figure 2-8. A single data attribute allows us to set the layout on a large group of images

In the example above you can see that we are applying a `data-rh-layout` attribute to the band content region (content, along with header and footer are the band's 3 main regions). Once again we see the power of data attributes for organizing the options available for each component or layout. With this single attribute we can pass in dozens of different values, each with their own unique, yet related styles.

In this case, with "gallery5" passed in, the CSS is applying roughly a 20% width to all of its child elements. By placing all of the grid information in the parent element, we are able to drop any component into the content region without ever adding a class, or wrapping it in some row or column container. So when we want to change this logo wall from a 5 column to a 4 column layout, the only thing we need to do is change a single attribute on the parent element.

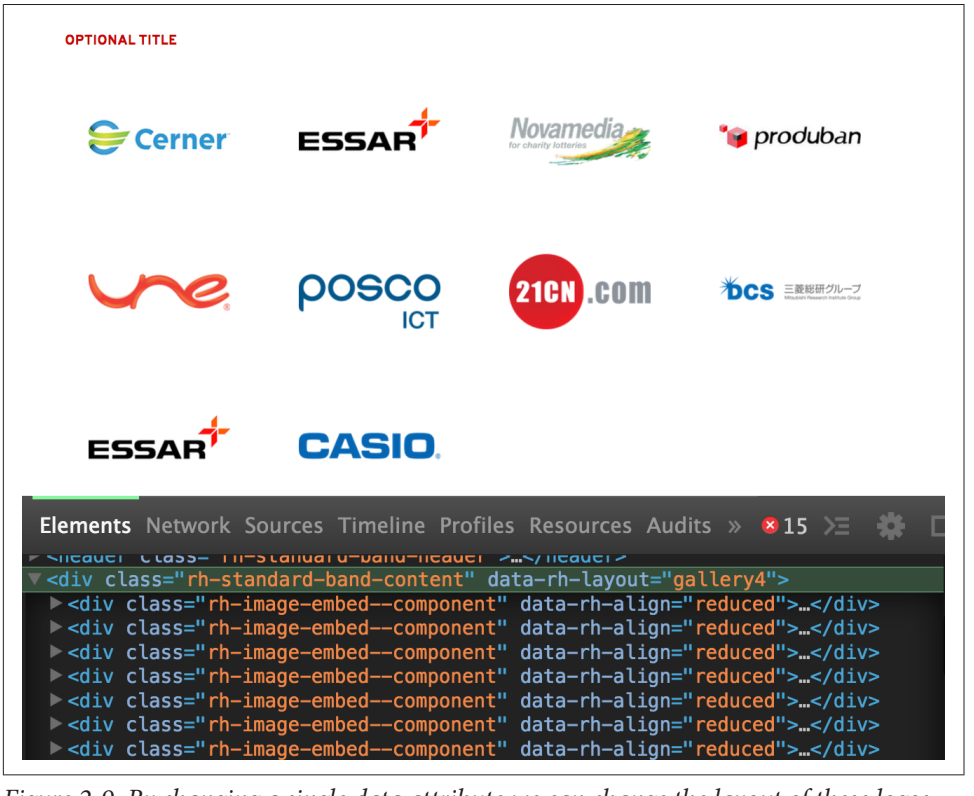


Figure 2-9. By changing a single data attribute we can change the layout of these logos

With a single attribute change we have changed the entire layout of the band. We didn't have to write new CSS (modular) and we didn't have to apply classes to the content (semantic). It's a win win! Now we have an inventory of layouts including various galleries and all of the most common combinations of our 12 column grid (6 6, 3 9, 4 4 4 etc). These layouts cover 99% of the content we need to create, and if we

find that they aren't meeting our needs for a specific use case, we can create a custom grid layout, document it, and now it is available for anyone to apply to their content.

With Great Power Comes…..Well... Great Power

Now that we have a set of modular and customizable containers that can house, and apply layout to our components, we can start building our logo walls, and every other band our site requires. We have a design system, a process for communicating anything that our design language is capable of expressing. This process of communication will be the focus of our next pillar, as we look at the how we turn ideas into finished, deployed code in our website.

As an architect one of our greatest roles is to oversee the development of the website and design system. But anyone who has in a project of significant scope knows that it is impossible for a single person to oversee every aspect of a project. With team members numbering in the dozens, weekly commits in the hundreds and lines of code in the thousands, it is beyond reason to think that a single person can evaluate the impact of every single piece of code entering the system. Yet, this is exactly what we need to be doing.

Newly introduced code can introduce bugs, or regressions, in many different ways. Some regressions will affect the outcome of system computation resulting in incorrect product prices, while others affect the ability of a user to successfully check out using the site shopping cart. Other regressions are visual and while you can still technically complete the task, the experience is made difficult to use because of broken or inconsistent user interface. Lastly, the price may be correct, the shopping cart functioning properly, and the user interface complete, but the performance of the site renders the site virtually unusable on given devices or from certain geographic regions. These four types of regression stem from very different parts of your code base, but they all have a similar impact: loss of sales. Fortunately, each of them have very similar solutions: testing!

While we as Architects cannot spend our time watching every single line of code being committed, we are able to set up various suites of tests to verify our application is working properly.

- Tests are written while the site is being built, or even before actual code it written.
- They are living code, committed with, or next to the system repository.
- All tests must pass before any code is merge into the master branch
- Running the suite on the master branch always returns a passing response

This means that no code is ever merged into the codebase that breaks the systems ability to calculate, perform critical business actions, display the correct UI, or deliver a performant experience. So instead of trying to oversee thousands of

lines of code, an Architect is able to focus on quality, complete test coverage. These tests break down into four different categories that we are going to cover in this section of the book. They are:

Unit Tests

Behavior Tests

Visual Regression Tests

Performance Tests

Unit testing is the process of breaking down your application to the smallest possible functions, and creating a repeatable, automated test to make sure they continue to produce the same output. These tests are the heart and soul of your application. They provide the foundation that all future code is built upon. Without unit tests it is possible that a seldom used function could remain broken for months without noticing. With unit tests, every system function can be verified before a single line of code is even merged into the master branch, let alone pushed to production.

As a Front-end Architect, your primary role is making sure that developers have the tools necessary to make them as efficient as possible. Unit tests are one of those essential tools for building an application of any scale. Wether your application logic is written mostly in a back-end or front-end languages, there are plenty of options to fit your workflow. So wether you are running PHPUnit with PHP, NodeUnit with Node, or QUnit with Javascript, you will find mature, stable platforms to built your tests upon.

Though your technology stack (and the tests associated with them) might be left up to the Software Architect, it is quite probable that a number of your Front-end Developers will be writing code requiring tests. Therefore it is important to become familiar with as many of the suites as possible. Acquiring mastery, or even proficiency, with all of them is not something we typically have the luxury of doing. But a solid understanding of the basic concepts will help you and your team write more testable code, and get up to speed with any framework quickly.

Lets go over some of these basic concepts now, and then we'll have an opportunity to look at the code in action.

The Unit

"Do one thing, and do it well" is the mantra when you are building an application with unit tests. Too often we write functions that try to do too many things. Not only is this very inefficient, as it does not lead to re-use, it also makes these functions very difficult to test.

Consider this simple function: Given a customer's address, the function determines the cost to ship a product to that customer from the nearest distribution center.

Lets break down this function a bit. The first thing that happens is that our function uses the address to find the nearest distribution center. Using that distribution center address the function then determines the distance between the center and the customer's address. Lastly, using that distance, the function calculates the shipping cost to move the package from point A to point B. So even though we have a single function, the function is performing three separate actions.

1. Look up distribution center nearest to given address
2. Calculate a distance between two addresses
3. Evaluate shipping cost for a given distance

Going back to the idea of "doing one thing, and doing it well", it is pretty obvious that our address lookup action would be better accomplished by combining three separate functions. Now we have a function that allows us to find the nearest distribution center to any given address. We also have a function that will calculate the distance between two given addresses. Lastly, we can return the cost of shipping a product any given distance, regardless of where that distance came from.

More Re-use

Now these three functions can be used throughout our entire application, not just for calculating shipping costs. If we have another part of the application that needs to find a distribution center, or to calculate the distance between two addresses, those functions have already been created. Therefore we aren't reproducing small units of functionality over and over again in separate, larger functions.

Better Testing

Instead of testing every possible way our application has of determining shipping, we can instead test each individual, re-usable function. As our application grows, the number of new functions needed to create new functionality decreases. In the end we have a less number of less complex functions performing more advanced functionality than larger, more complex functions could produce.

Test Driven Development

When you first approach unit testing you probably write up some functionality that meets a business goal (like our shipping example), and then work to refactor it into smaller, reusable, testable little bits. Only then do you consider the tests you want to write to make sure these functions never break. Test Driven Development turns that idea upside down by putting the tests first, before any production code is ever written.

But if we write tests for functions we haven't created, won't they all fail? Exactly! Test Driven Development (or TDD) sets out to describe how a system should work when written properly.

For our shipping example TDD would write three tests, one for each of the three functions required to perform this business function. The developer's job is to turn those failing tests into passing tests. First we write a function that properly looks up the location of the nearest distribution center, and we have one passing test. We then move on to the next failing test and knock each of them out until we have three functions that meet the requirements of their tests.

With this done, not only have we built the functionality required for our application to look up shipping cost, but we have complete test coverage for these functions.

Looking at some code

At its core, unit testing is extremely simple. The basic idea is to call the function being testing, passing it preset values and describing what the result should be. Lets look at how we'd do this for our function that calculates the shipping cost for a given distance.

```
function calculateShipping(distance) {
  switch (distance) {
    case (distance < 25):
      shipping = 4;
      break;
    case (distance < 100):
      shipping = 5;
      break;
    case (distance < 1000):
      shipping = 6;
      break;
    case (distance >= 1000):
      shipping = 7;
      break;
    }
    return shipping;
  }

QUnit.test('Calculate Shipping', function(assert) {
  assert.equal(calculateShipping(24), 4, "24 Miles");
  assert.equal(calculateShipping(99"), 5, "99 Miles");
  assert.equal(calculateShipping(999), 6, "999 Miles");
  assert.equal(calculateShipping(1000), 7, "1000 Miles");
};
```

QUnit has several operators for testing assertions, including ok() for testing boolean values or deepEqual() for comparing complex objects. In the case above we are using the equal() function to compare the value returned by calculateShipping() with an expected result. As long as calculateShipping(24) returns a value of 4 (which it will in the above case) our test will pass. The 3rd value "24 Miles" is used to label the pass/fail statement in the test output.

With these tests (and others) in place we have a single suite to run run that will assert whether or not our system is working. If someone were to change the name of the calculateShipping() function, or if the file was removed from the build system, this test would return failures, and we could fix the problem before the new code was pushed to production.

QUnit is capable of doing much more than than the example above. For example it is capable of performing tests on synchronous and asynchronous functions. QUnit can also interact with the web page that the tests are loaded on (remember, this is all just Javascript). So if your tests includes values being returned when a mouse is clicked, or a key is presses, QUnit has you covered there too.

How Much Coverage is Enough

Determining proper test coverage can be a very difficult balancing act. If you aren't running Test Driven Development (where nothing is written without tests), it will be important to determine how much coverage is enough coverage. Test everything, and your development process can get bogged down, don't test enough and you risk regressions slipping through.

Fixing The Gaps

If you are implementing unit tests on an existing project, you most likely won't have the time or budget to write 100% test coverage for current functionality. And that's ok! The beauty of test coverage is that even a single test adds value to a system. So when determining where to start writing tests, look for the biggest wins first. Sometimes the biggest win is writing tests for the most simple parts of your system. Just like paying off your small credit card debt before trying to tackle your larger debt, writing some simple, but still valuable tests, will be a great place to build momentum.

Once you have a working suite providing some basic coverage, start looking at the parts of the system that are either the most critical, or have had recurring trouble in the past. Create stories for your backlog for each of them and make sure to pull them forward as often as possible.

Coverage from the Start

If you are fortunate enough to be starting a new project as a Front-end Architect, your job is not just to get a testing framework set up, but to make sure that the development process itself is prepared for unit testing. Just like writing documentation, or performing code review, writing unit tests take time! You'll need to make sure that any story requiring tests is given the extra time required to write, and verify the necessary test coverage.

At Redhat.com, every user story starts with a set of tasks, and time, to develop and verify the test coverage required for that feature. So if a new feature is estimated to take 8 hours of development time to complete, the assumption is that the entire story

is going to take 12+ hours when we factor in the time for testing. This dramatic increase in time can often be a hard sell, so it is often the Front-end Architect's job to play diplomat and salesperson. Even though this is a 50% increase in the time, we know that this test coverage will save us dozens of hours in the future that would have been spent tracking down bugs.

As I said above, not every feature requires the same amount of test coverage. But the assumption is that every story STARTS with tasks for test coverage. As long as those tasks are only removed when everyone agrees that coverage isn't necessary, we can be confident that any feature requiring coverage is given the time needed to finish that task.

The Proper Type of Coverage

For some functionality the best way to provide test coverage is not Unit Testing at all. If you are writing javascript that creates visual changes to the DOM, you might be better off testing that with Visual Regression Testing. Or if your code is used to facilitate some specific workflow, you could have better luck testing with Behavioral Testing.

In the upcoming chapters we'll be looking at 3 other types of test coverage: Behavioral Testing, Visual Regression Testing and Performance Testing.

Type something here!
Tell me if this is a familiar scene. You've been working on your website contact form for the last few weeks, trying to tweak and nudge the form fields until they look exactly like the photoshop mockup. You've meticulously compared every margin, padding, border and line height. Your lead generation tool is now "pixel perfect", and the product owner agrees...this is the contact form to end all contact forms. With this code securely committed you move on to your next task and stop having recurring nightmares about browser drop down rendering discrepancies.

Weeks later you are surprised to find a familiar sight in your ticket queue...the contact form. Some designer, business analyst, or quality assurance engineer took a ruler to your design and compared it to the latest photoshop comp, and found a list of discrepancies.

But?! Why?! How?!?! Tracking down the culprit is a luxury you do not have time to pursue. So you sit down with a list of tweaks and get to work hoping that this is the

last time you'll have to touch this contact form, but resigned to the fact that you'll probably see it a few more times before the site launches.

The Usual Suspects

My favorite sound in the world is the cry of a decision maker as they scream how "feature X is totally broken!!" Translated into developer terms, this usually means that a few of the fonts are incorrect, or that some vertical rhythm needs fixing. It doesn't matter if the feature had been signed off and agreed upon, there is a difference between what is live and the Photoshop file the decision maker has been pouring over for the past week.

Having had this happen to me over and over again, let me explore a few of the common reasons that this merry-go-round has so much trouble stopping and letting you off.

Unknowing Developers

Any code that you can write without defect can be broken by just a few errant lines written by another developer. Someone else, working on some other form component, didn't realize that the classes they were styling were shared with your contact form. These changes could have happened in the weeks since your code was committed, or they could have been written at the exact same time as you were working, but merged in after your branch.

Small cosmetic changes to unrelated pages are often overlooked by the QA team. With dozens or even hundreds of pages to test, there is no way that they would catch a 2 pixel change to a label's font size.

Inconsistent Designs

Allow me to let you in on a little dirty secret about Photoshop. When one designer changes the font size of a form label in one file, it doesn't magically change in all of the designers PSD files. Wait what?! There isn't a single *sheet* prescribing all of the element *styles* in a *cascading* fashion? Nope!

Worse still, even if all of the designers communicate this font size change, this doesn't cause a ripple in space and time that updates every PSD trapped in an email thread, basecamp conversation, or dropbox folder.

Depending on which designer, business analyst or QA engineer is reviewing the contact form, and which version of whatever PSD they happen to be looking at, there is a 9 in 10 chance on any given day that your form has a defect (and therefore is totally broken). So a new story is created to address these defects, and you can only hope that the changes you are making aren't going to make even more work for you the next time a designer takes a peak at the contact form.

Waffling Decision Makers

According to the law of infinite probability, given enough features, poured over by enough decision makers, there is a 100% chance that someone will find something that they want to change.

Change is inevitable and, given the proper development model, it is completely acceptable. But when change masquerades as defects (or a distinction is never made), developers end up spending a ton of time building features that are nothing more than prototypes.

Now there is nothing wrong with prototyping a feature before the final build, actually that is generally a really good practice! But prototypes need to be quickly iterated designs ending in a final, agreed upon product. Asking a developer to create a single prototype every sprint cycle, and then revising it every other sprint, is not only a great way to hobble a developer's productivity, but it is a horribly inefficient way to prototype.

A Tested Solution

While each of the above scenarios highlights some deeper, organizational issues, they can all be mitigated by a single thing...proper test coverage.

Testing?! Really? But this is front-end development, we don't write tests for HTML and CSS. That's just a programming language thing, right? What's next, test driven Front-end Development? For every line of CSS we write, we need to write a test for it first?

Fortunately the reality of Front-end testing is nothing like the functional testing of Ruby or Javascript. The end result of our work is the visual representation of content on a web page. And because our end result is visual, so is our testing method.

Visual Regression Testing allows us to make visual comparisons between the correct (baseline) versions of our site and versions in development or just about to be deployed (new). The process is nothing more than taking a screenshot of each page and comparing the pixels to find differences (diff).

With these baseline images either committed to the repo, or marked approved in some testing database, we now have a record of the exact signed off, agreed upon, pixels that make up any particular feature (in our case, a contact form). Before any line of code is committed back to the master branch, Visual Regression Testing provides us a method to test every feature in our site, and make sure that nothing unexpected has visibly changed.

We will also be guarded against bug reports that are nothing more than inconsistencies from one PSD to another. With a signed off baseline committed to our codebase, we can re-run our tests and confidently reply that our code is correct. In the same way, we will be able to discern between actual bugs and YACR (yet another change request).

The Many Faces of Visual Regression Testing

Visual Regression Testing comes in many different flavors, using a variety of technologies and workflows. While there are new tools being released into the open source community all the time, they typically include a combination of a small set of features. Here are a few of the categories that most tools fall into.

Page Based Diffing

Wraith is a good example of page based diffing. It has a simple YAML setup file that makes it very easy to compare a large list of pages between two different sources. This approach is best used when you aren't expecting any differences between the two sources, like when you are comparing pages from your live site with the same pages in staging, just about to be deployed.

Component Based Diffing

BackstopJS is a great tool for doing component, or selector based diffing. Instead of comparing images of the entire page, a component based tool allows you to capture individual sections of a web page and compare them. This creates more focused tests and removes the false positives when something at the top of the page pushes everything else down, and everything comes back as changed.

CSS Unit Testing

Quixote is an example of a unique class of diffing tools that look for unit differences, or variations in returned values. Quixote can be used to set TDD style tests where the test describes values that should be expected (title font-size is 1em, sidebar margin is 2.5%), and checks the web pages to see if those assertions are in fact true. This is a great approach for testing troubled areas such as the width of columns in a layout that keeps breaking. Or it can be used to assert that branding protocol has been followed and the logo is the correct size and distance away from other content.

Headless Browser Driven

Gemini is a comparison tool that uses PhantomJS, a headless browser, to load webpages before taking screenshots. PhantomJS is a javascript implementation of a webkit browser. This means that it is incredibly fast, and consistent across various platforms.

Desktop Browser Driven

Gemini is unique in that it also supports running tests using traditional desktop browsers. To do so, Gemini uses a Selenium server to open and manipulate the OS's installed browsers. This isn't as fast as a headless browser, and is dependent on the version of the browser that happens to be installed, but it is closer to real world results and can catch bugs that might have been introduced in just a single browser.

Includes Scripting Libraries

CasperJS is navigation and scripting library that works with headless browsers like PhantomJS. It allows tools to interact with the pages opened in the browser. With CasperJS you can move the mouse over a button, click on the button, wait for a modal dialog, then fill out and submit a form, then take a screenshot of the result. CasperJS even lets you execute javascript on the pages within PhantomJS. You can hide elements, turn off animation, or even replace always changing content with consistent, mock content to avoid failures when the 'newest blog post' gets updated.

GUI Based Comparison and Change Approval

Projects like Diffux store test history, and provide test feedback inside of a web based graphical user interface. Baseline images are stored in a database, and any changes to those baselines must be approved or rejected inside of the app. These types of tools are great when you have non technical stakeholders needing to make the final decision on wether the changes are correct or not.

Command Line Comparison and Change Approval

PhantomCSS is a a component based diffing tool, using PhantomJS and CasperJS, that runs solely in the command line. Test runs are initiated via a terminal command, and the results, passing or failing, are also reported in the terminal. These types of tools work especially well with task runners like Grunt or gulp, and their output is well suited for automation environments like Jenkins or Travis CI.

###Need to wrap this chapter up. Demo moved to redhat case study section.

The purpose of testing is to protect users from a degraded or broken experience, and poor website performance is one of the quickest ways to give your users a degraded and broken experience. Therefore performance testing, while not a test that points out system, behavioral or visual regressions, is an important part of our testing arsenal.

Performance Testing measures key metrics that affect a user's ability to use your website including page weight, number of requests, time to first byte (TTFB), load time, and scrolling performance.

The key to performance testing is to set a proper budget and stick to it. How you set the budget, and how you stick to it will determine how effective the tests will be in your project.

Setting A Performance Budget

Creating a performance budget means setting target values for each key metric and then continually testing those metrics before each code merge, or deployment. If any of the tests fail the new feature will need to be adjusted, or some other feature may need to be removed.

Just like financial budgets, very few people are really excited about the prospect of budgets. To most, a budget means spending less, getting less, having less fun and most importantly....less! Less isn't much fun in a world where we are always being told that we deserve more. As a designer we feel that our creativity is being stifled if we can't toss around hi-res images and full screen video with reckless abandon. As a developer we think that we can't do our job without a CSS framework, a couple Javascript frameworks, and dozens of jQuery plug-ins. Less is no fun!

As a person that has been living within a financial budget for the past 4 years, I certainly understand what it means to not get everything I want. On the other hand, when I do make a large, budgeted purchase, I do so without a single bit of guilt or debt. In the same way, performance budgets allow us to "spend" our budget responsibly, and without regret.

The ultimate affect of this discipline, as Dave Ramsey puts it, is "If you will live like no one else, later you can live like no one else". Just like fiscal discipline and financial budgets, UX discipline and a performance budget will help us to achieve our ultimate goals, which include a performant website and engaged user.

While a financial budget is typically based off one's income, a performance budget has more to do with external factors than internal ones.

Competative Baseline

One method for determining your performance budget is to look at your competition. While saying "at least I'm better than so-and-so" is no excuse for a poorly performing website, it does insure that you have a competitive advantage over your competition.

So start by looking at a few of your key competitors' website homepages and other key landing pages, and then compare load times, page weight and other key metrics with your own website. The goal here isn't to just match their metrics, but rather aim and beating them by 20% or more. So if your competitor's product listing page loads in 3 seconds, make sure that your site loads it's product listing in 2.4 seconds or less. This 20% advantage over your competitor is the difference required for a user to recognize the difference between the two tasks.

This is obviously not a single time process, but something that needs to be monitored regularly. You can be assured that your competitor is looking for ways to improve and optimize their own sites. And if they had been looking at your site to determine their budgets, you've now pushed them to reduce their budgets as well!

Averaged Baseline

Regardless of your competition, it is always important to compare your competitive baselines to industry average and general best practices. There is no reason to settle with mediocre just because your competition is throwing off the curve.

HTTPArchive is a great service that measures and records the average value of various website metrics across almost a half a million websites. As April 2015 here are a few values of note:

- Page Weight: 2061kB
- Total Requests: 99
- Cacheable Resources: 46%

Therefore if you want your website to feel faster than most websites, you might consider setting a goal of having 1648kB website, that is served with 79 requests, of which 44 are cacheable.

So now that we know a few methods for setting our budget, what are the budget items we need to consider when setting our tests up?

Raw Metrics

Page Weight

The most basic test of website performance is to look at the overall weight of the webpage, that is, how many kilobytes of data need to be transferred from the server to the browser in order to fully render the page.

In short, websites are getting fatter. Between April 2014 and April of 2015 the average website grew from 1762 to 2061 kilobytes, a 17% increase year over year. Reaching back to April of 2011 the average page was a skimpy 769kb!

While not the only factor affecting the load time of your website, it certainly plays a large part. Page weight also has another side effect as we remember that more and more people are accessing our sites on mobile devices, and they are paying to download those bytes. The heavier your page, the more you are going to be costing your customers, especially in developing nations. Consider checking out whatdoesmysitecost.com to see what that new carousel is costing your mobile customers in Germany.

When looking to reduce the weight of your pages, there are a few obvious places to start.

- Images make up 61% of an average website's page weight.
  - — Optimize your PNG files, considering reducing quality of some JPG files
  - — Take advantage of the new responsive image <picture> tag and "srcset" attribute to download appropriately sized images
  - — Simply set a budget and don't add image weight without removing another image
- Too many custom fonts will quickly weigh your page down
  - — Set a font weight budget and consider not adding that second or third font
  - — Also consider necessary font weights, as each font weight adds weight to the font file

    - While icon fonts are great, be mindful of the file size as they can grow large quite quickly. Split the font up if one set is used for one section of the website, and another set for others. Also consider using inline SVGs instead as you'll gain many of the benefits of icon fonts while only needing to load the required SVGs
- JS frameworks, jQuery plugins and CSS frameworks often add a great deal of weight for little reward
    - Many sites are moving away from jQuery as vanilla JS is sufficient for their needs, especially if targeting modern browsers
    - jQuery plugins, while they might offer some "wiz-bang" functionality, they can often weigh in pretty heavy. Consider if the same thing could be done with CSS for modern browsers with reasonable fallbacks for older ones.
    - Large JS frameworks like Angular or Ember might accomplish what you need done, but might come with more weight than required to get the job done. If all you need from Angular is the view layer, you might better off looking React or even Mustache.
    - CSS frameworks are often a kitchen sink. They include every little imaginable style you could ever possibly need. While this might be great for prototyping, starting your website with hundreds of kilobytes of CSS and JS is putting yourself in quite a hole as you start to add even more styles to customize your design.
- Minification and compression
    - Javascript can be programmatically minified during your build process, and your servers set up to gzip files before sending them to the browser. These are both vital steps to reducing page weight.

Number of HTTP Requests

The browser is required to perform an HTTP request for every single file needed to fully render a page. Due to the fact that every browser has a limited number of concurrent HTTP requests it can make at a time, a large number of individual files means that the browser has to make numerous round trips to the server to collect all the necessary files. The effect of these rounds trips are compounded on slower networks, so limiting the number of round trips needed to gather the required files will pay off greatly.

You can reduce the number of round trips in a few ways:

- Reduce the number of HTTP requests
    - Instead of serving up dozens of individual CSS and Javascript files, concatenate them into single files
    - Combine individual image files into a single image map or icon font. You'll find many great tools to do this for you automatically (Compass, Grunt/Gulp plugins)

  - — Lazy load assets not required for initial page load. This could be javascript that isn't needed until the user interacts with the page, or images that are far below the initial load window.
- Increase the number of assets retrieved per round trip
  - — Splitting up your assets across different servers (or CDN's) will allow the browser to pull down more assets per round trip, as the limitations on concurrent connections is per server.

Timing Metrics

Time to first byte (TTFB)

Time to first byte is a measurement is the number of milliseconds between the browser requesting the webpage, and the first byte of the webpage being receive by the browser. It is a measurement of the paths between the browser and the server including DNS lookup, initial connection and receiving of data. This value is not the best judge of a website's performance, but it is a valuable number to keep an eye on.

Time to start render

A more useful time measurement is the "time to start render". This measurement is the time in which the user starts to see content on the page. This means that any blocking files have been loaded and the browser is able to start drawing out the DOM. Improving this number can be done by deferring blocking JS/CSS, putting critical CSS inline in page head, replacing image assets with data URIs, and lazy loading any other blocking content to be pulled in after the document has completely rendered.

Time to Document Complete

Once all of the initially requested assets have been loaded, the document is considered to be "complete". Time to Document Complete doesn't include assets pulled in by javascript, so lazy loading of assets won't increase this metric.

Hybrid Metrics

PageSpeed Score

PageSpeed is a website tool and Chrome extension made by Google that analyzes the performance and usability of a website providing a score out of 100, and explaining ways that the user can improve that score. Tests include:

- Presence of render-blocking JavaScript or CSS
- Landing page redirects
- Image optimzation

- File minification
- Server response time
- Server compression
- Browser Caching
- Tap target size
- Viewport properly configured
- Legible font sizes

Speed Index

"The Speed Index is the average time at which visible parts of the page are displayed. It is expressed in milliseconds and dependent on size of the view port."

This hybrid timing metric provides a score that takes into account many of the metrics above, and combines them with a measurement of what the user is actually able to see of your site as it loads. Speed Index is one of the best measurements of actual end-user experience for a website.

Setting Up Performance Tests

Now that we know what types of metrics we can test, and how to set performance budgets, lets take a quick look at a few methods for automating the testing process. Wether you are testing a single website, or dozens of them, no one wants to perform these measurements manually.

Grunt PageSpeed

The first tool we'll look at for automating this workflow is Grunt PageSpeed. As the name implies, this is a Grunt plugin that allows us to run Google's PageSpeed test on our website. So rather than plugging your URL into the test page, or using a Chrome extension, this Grunt task can be ran before every merge request or continuous integration build.

To setup Grunt PageSpeed we start with the standard commands to install and require our plugin

```
$ npm install grunt-pagespeed --save-dev

// Added to Gruntfile.js
grunt.loadNpmTasks('grunt-pagespeed');

// Added to grunt.initConfig inside of Gruntfile.js
 pagespeed: {
  options: {
    nokey: true,
    url: "http://redhat.com"
  },
```

```
    desktop: {
      options: {
        paths: ["/en", "/en/services"],
        locale: "en_US",
        strategy: "desktop",
        threshold: 80
      }
    },
    mobile: {
      options: {
        paths: ["/en", "/en/services"],
        locale: "en_US",
        strategy: "mobile",
        threshold: 80
      }
    }
  }
```

This will allow us to automatically run both desktop and mobile tests on an array of pages inside of our base URL (in this case redhat.com). As long as our score comes back over 80, the tests will pass, if they are below 80 we'll get a failing test, which signifies that changes need to be made to hit our threshold again.

Grunt Perfbudget

Another great Grunt tool is Grunt Perfbudget. This Grunt plugins taps into the WebPageTest API written by Marcel Duran, allowing us to programmatically pull results from webpagetest.org and compare them with our set budgets. If you haven't been to webpagetest.org yet, you'll be in for a treat. It is able to test numerous metrics for your site while simulating different types of connections and locations around the globe. I won't get into everything the site can do, but after 5 minutes of viewing the results for your own site, I'm confident you'll love the plethora of information it provides.

So lets see what this looks like set up in Grunt:

Note: You can currently get a limited use API key at the webpagetest website

```
$ npm install grunt-perfbudget --save-dev
```

```
// Added to Gruntfile.js
grunt.loadNpmTasks('grunt-perfbudget');
```

```
perfbudget: {
  default: {
    options: {
      url: 'http://www.redhat.com/en',
      key: 'SEE_NOTE_ABOVE',
      budget: {
        visualComplete: '4000',
        SpeedIndex: '1500'
      }
```

```
        }
      }
    }
```

This setup allows us to automatically run the redhat.com homepage through the entire webpagetest suite of tests, and check returned values against the budgets I have set. In this case I have set my Visually Complete timing metric to 4000 milliseconds and the Speed Index to 1500. If either of those tests come back above our budget, we get a nice big error message telling us to go revisit the most recent code push and see what we did to break our budget.

So with some proper automated testing in place, and a competitive budget in place, you'll be in a good place to continue developing features and making improvements to your website while being sure that none of the changes you push out ever break your budget.

Visual Regression In Action

PhantomCSS has been my go-to tool for the past few years because it provided component based comparison, with a headless browser, and scripting library, that could be integrated with my current build tools. So let me walk you through the setup of PhantomCSS and how we are currently using it at Redhat.com.

The Testing Tools

PhantomCSS is a powerful combination of 3 different tools:

1. PhantomJS is a headless Webkit browser that allows you to quickly render web pages, and most importantly, take screenshots of them
2. CasperJS is a navigation and scripting tool that allows us to interact with the page rendered by PhantomJS. We are able to move the mouse, perform clicks, enter text into fields and even perform javascript functions directly in the DOM
3. ResembleJS is a comparison engine that can compare two images and determining if there are any pixel differences between them

We also wanted to automate the entire process, so we pulled PhantomCSS into Grunt and set up a few custom grunt commands to test all, or just part of our test suite.

Getting Grunt Set Up

Now before you run off and download the first Grunt PhantomCSS you find on Google, I'll have to warn you that it is awfully stale. Sadly someone grabbed the prime name-space and then just totally disappeared....like not even a tweet for the past 2 years. This has lead to a few people taking it upon themselves to continue on with the codebase, merging in existing pull requests and keeping things current. One of the

better ones is maintained by Anselm Hannemann and can be found here. First you'll need to import Anselm's Grunt PhantomCSS project into your Grunt project. He doesn't have a NPM namespace picked out yet, so we'll just be pulling it in directly from Github:

```
npm i --save-dev git://github.com/anselmh/grunt-phantomcss.git
```

With that installed we need to do the typical Grunt things like loading the task in the Grunfile.js

```
grunt.loadNpmTasks('grunt-phantomcss');
```

Then set a few options for PhantomCSS, also in the Gruntfile.js. Most of these are just default:

```
phantomcss: {
        options: {
          mismatchTolerance: 0.05,
          screenshots: 'baselines',
          results: 'results',
          viewportSize: [1280, 800],
          },
          src: [
             'phantomcss.js'
          ]
      },
```

- **mismatchTolerance:** We can set a threshold for finding visual differences. This helps account for anti aliasing or other minor, non critical differences
- **screenshots:** Choose a folder to place baseline images in
- **results:** After running comparison tests, the results will be placed in this folder
- **viewportSize:** We can always adjust the viewport size with Casper.js
- **src:** Just a path to our test file, relative to our gruntfile

Our Test File

Next, in the phantomcss.js file, this is where Casper.js kicks in. PhantomCSS is going to spin up a PhantomJS web browser, but it is up to Casper.js to navigate to a web page and perform all of the various actions needed. We decided the best place to test our components would be inside of our styleguide. It shared the same CSS with our live site, and it was a consistent target that we could count on not to change from day to date. So we start off by having casper navigate us to that URL.

```
casper.start('http://localhost:9001/cta-link.html')
.then(function() {
  phantomcss.screenshot('.cta-link', 'cta-link');
})
.then(function() {
  this.mouse.move('.cta-button');
```

```
    phantomcss.screenshot('.cta-link', 'cta-link-hover');
});
```

After starting up casper at the correct page, we use Javascript method chaining to string together a list of all the screen shots we need to take. First we target the .cta-link and take a screen shot. We aptly call it "cta-link". That will be its base file name in the baselines folder.

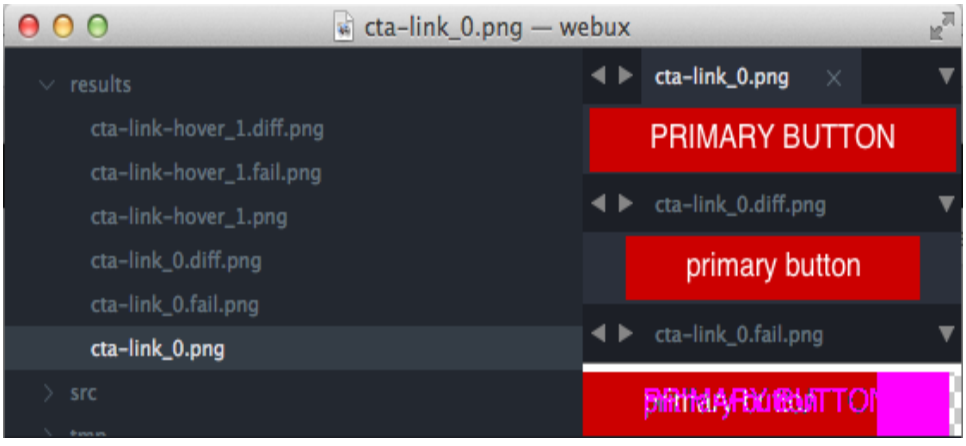
Next we need to test our button to make sure it behaves like we'd expect when we hover over it. We can use Casper.js to actually move the cursor inside of PhantomJS so that when we take our next screenshot, and call it `cta-link-hover`, we get the following:

Making A Comparison

With those baselines in place we are now able to run the test over and over again. If nothing has changed, images created by the subsequent tests will be identical to the baseline images and everything will pass. But if something were to change.....say someone accidentally added the following to their css while they were working on some other feature:

```
.cta-link {
  text-transform: lowercase;
}
```

The next time we ran our comparison tests we'd get the following:

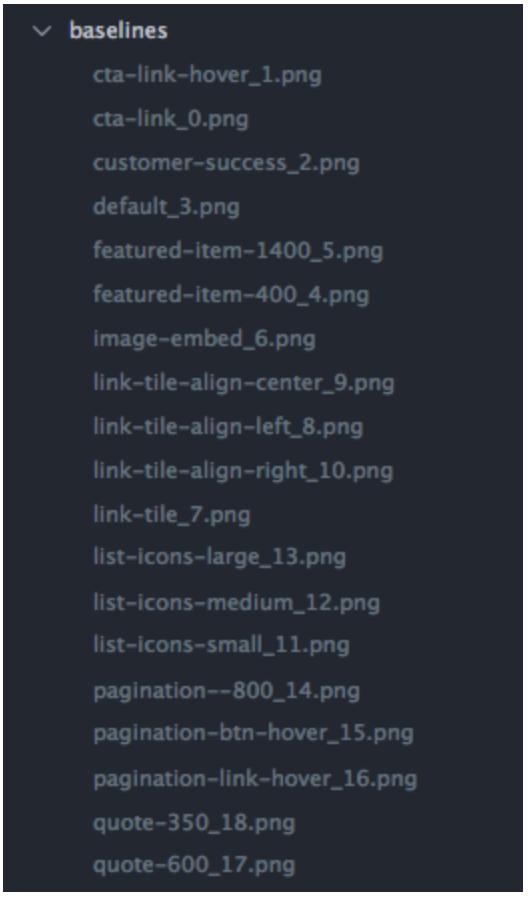


As expected, the change from uppercase to lowercase created a failure. Not only was the text different, but the button ended up being shorter. The 3rd "fail" image shows us in pink which pixels were different between the two images.

Running The Entire Suite

After doing this for each component (or feature) we want to test in our styleguide, we can run `$ grunt phantomcss` and it will do the following:

1. Spin up a PhantomJS browser
2. Use CasperJS to navigate to a page in our styleguide
3. Take a screen shot of a single component on that page
4. Interact with the page: click the mobile nav, hover over links, fill out a form, submit the form etc
5. Take screen shots of every one of those states
6. Compare all of those screenshots with baseline images we captured and committed when the component was created
7. Report if all images are the same (PASS!) or if there is an image that has changed (FAIL!)
8. Repeat this process for every component and layout in our library.



What Do We Do With Failing Tests?

Obviously, if your feature branch is concerned with changing the appearance of a component (adjusting font size or background color etc...) you are going to get failing tests. The point is that you should only be getting failing tests on the component you were working on. If you are trying to update the cta-link and you get failing tests on your cta-link *and* your pagination component, one of two things happened:

1. You changed something you shouldn't have. Maybe your changes were too global, or you fat fingered some keystrokes in the wrong file. Either way, find out what changed on the pagination component, and fix it.

2. On the other hand, you might determine that the changes you made to the cta-link actually should have affected the pagination too. Perhaps they share the same button mixin, and for brand constancy they should be using the same button styles. At this point you'd need to head back to the story owner/designer/person-who-makes-decisions-about-these-things and ask them if they meant for these changes to apply to both components, and act accordingly.

```
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/cta-link-hover_1.fail.png
Visual change found for cta-link-hover_1.png (47.62% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/cta-link_0.fail.png
Visual change found for cta-link_0.png (47.45% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/pagination--800_14.fail.png
Visual change found for pagination--800_14.png (26.92% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/pagination-btn-hover_15.fail.png
Visual change found for pagination-btn-hover_15.png (42.57% mismatch)
```

Moving from Failing to Passing

Regardless of the reasons you might have had some false positives, you will still be left with a 'failing' test because the old baseline is no longer correct. In this case, just delete the old baselines and commit the new ones. If this new look is the brand approved one, then these new baselines needs to be committed with your feature branch code so that once your code is merged in, it doesn't cause failures when others run the test.

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   baselines/cta-link-hover_1.png
        modified:   baselines/cta-link_0.png
        modified:   baselines/pagination--800_14.png
        modified:   baselines/pagination-btn-hover_15.png
```

The magic of this approach is that at any given time, every single component in your entire system has a "gold standard" image that the current state can be compared to. This also means that this test suite can be run at anytime, in any branch, and should always pass without a single failure.

Making it Our Own

I started to work with Anselm's code at the beginning of the Red Hat project and found that it met 90% of our needs, but it was that last 10% that I really needed to

make our workflow come together. So as any self respecting developer does, I forked it and started in on some modifications that make it fit our specific implementations. Let me walk you through some of those changes that I have in my branch labeled alt-runner

```
//New Settings

//Package.json
"grunt-phantomcss": "git://github.com/micahgodbolt/grunt-phantomcss.git#alt-runner",

// Gruntfile.js
phantomcss: {
        options: {
          mismatchTolerance: 0.05,
        },
        webux: {
          options: {
            altRunner: true, // Turn on altRunner
            screenshots: 'baselines',
            results: 'results',
            viewportSize: [1280, 800],
          },
          src: [
             'src/sass/**/*-test.js' // select all files ending in -test.js
          ]
        },
      },
```

Place Baselines in Component Folder

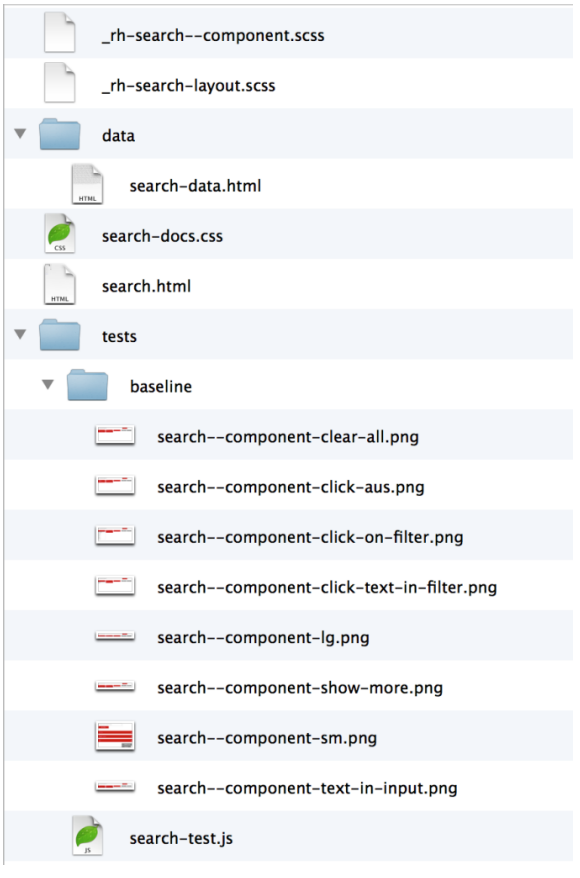
One thing important to us was good encapsulation. We put everything in the component folder....i mean everything!

- Sass files for typographic and layout styles
- HTML partial with the component's canonical markup
- Data file used to generate the component display when fed into our templating engine
- Documentation file that feed into our Hologram style guide explaining component features, options and implementation details
- Test file with tests written for every variation of the component (open/close, light theme/dark theme, field empty/field filled etc...)

Because of this, we also really wanted our baseline images to stay inside of the component folder. This would make it easier to find the baselines for each component, and when a merge request contains new baselines, the images are in the same folder as the code change that made them necessary.

The default behavior of Grunt PhantomCSS is to place all of the baselines, for every test in our system, into the same folder. And then when we ran our regression tests,

all of those images were placed into a single results folder. With dozens of different components in our system, each with up to a dozen different tests, this process just didn't scale. So one of the key changes we made in the alt-runner was to put baseline images into a folder called **baseline** right next to each individual test file.



Run Each Component Test Suite Individually

Secondly, I changed the test behavior to test each component individually, instead of all together. So, instead of running all 100+ tests and telling me if the build passed or failed I now get a pass/fail for every component.

```
Running "phantomcss:webux" (phantomcss) task
No changes found for cta-link.png
No changes found for cta-link-hover.png
>> All 2 tests passed!
No changes found for pagination--800.png
No changes found for pagination-btn-hover.png
No changes found for pagination-link-hover.png
>> All 3 tests passed!
```

OR

```
Running "phantomcss:webux" (phantomcss) task
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/cta-link.fail.png
Visual change found for cta-link.png (47.45% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/cta-link-hover.fail.png
Visual change found for cta-link-hover.png (47.62% mismatch)
>> 2 tests failed.

Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/pagination--800.fail.png
Visual change found for pagination--800.png (26.92% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/pagination-btn-hover.fail.png
Visual change found for pagination-btn-hover.png (42.57% mismatch)
No changes found for pagination-link-hover.png
>> 2 tests failed.
Warning: Task "phantomcss:webux" failed. Use --force to continue.
```

Test Portability

The last change I made is that I wanted my tests to be more portable. Instead of a single test file, we had broken our tests up into dozens of different tests files that Grunt pulled in when it ran the test.

The original implementation required that the first test file start with casper.start('http://mysite.com/page1') and all subsequent files start casper.thenOpen('http://mysite.com/page2') . This become problematic because the order in which Grunt chose to run these files was based on alphabetical order. So as soon I added a test starting with a letter 1 earlier in the alphabet than my current starting test, my test suite broke!

The fix was relatively easy as I just needed to call casper.start as soon as Grunt initiates the task, and then all of the tests can start with `casper.thenOpen` without any problems.

```
casper.thenOpen('http://localhost:9001/component_-_cta.html')
    .then(function () {
        this.viewport(600, 1000);
        phantomcss.screenshot('.rh-cta-link', 'cta-link');
    })
    .then(function () {
        this.mouse.move(".rh-cta-link");
        phantomcss.screenshot('.rh-cta-link', 'cta-link-hover');
    });
```

Conclusion

After getting these tests in place we were able to confidently grow our design system with new component after new component. With every new addition of code we had a suite of tests we could run to make sure that ensured that none of our previous work had been compromised. Our coverage not only included the basic appearance of a component, but every possible variation or interaction. As new features were added new tests could be written. If a bug slipped through, we could fix it and then write a test to make sure it never happened again.

Instead of our design system getting more difficult to manage with every new addition, we found it getting easier and easier. We could now consistently utilize and adapt current components, or create new ones with little fear of breaking others.

APPENDIX A

Appendix Title

This Is an A-Head

An appendix is generally used for extra material that supplements your main book content.