O'REILLY®

The Little Book of HTML/CSS Frameworks

Jens Oliver Meiert

Foreword by Eric A. Meyer, author of CSS: The Definitive Guide

O'REILLY®

Additional Resources

3 Easy Ways to Learn More and Stay Current

Radar Blog Read more news and analysis about JavaScript, HTML5, CSS3, and other web platform technologies. *radar.oreilly.com*

Web Newsletter Get web development-related news and content delivered weekly to your inbox. *oreilly.com/web-platform/newsletter*

Fluent Conference Immerse yourself in learning at the annual O'Reilly Fluent Conference. Join developers, UX/UI designers, project teams, and a wide range of other people who work with web platform technologies to share experiences and expertise—and to learn what you need to know to stay competitive. *fluentconf.com*

The Little Book of HTML/CSS Frameworks

Jens Oliver Meiert

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

The Little Book of HTML/CSS Frameworks
by Jens Oliver Meiert

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Meg Foley
Production Editor: Kristen Brown
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2015: First Edition

Revision History for the First Edition
2015-02-25: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491920169* for release details.

978-1-491-92016-9

[LSI]

Dedicated to the Google Webmaster Team under Dennis Hwang

Table of Contents

Foreword

In the beginning, there was markup; and lo, it was good. Then came style sheets, which let the author make the markup pretty; and lo, they were also good.

Some of that goodness was rooted in simplicity. HTML was simple, charmingly so, and CSS started out as a small set of presentational suggestions that nevertheless allowed for a great deal of creativity. Even when you account for the fumbling browser implementations of early CSS, it was quite possible to hold in one's head every property, value, and common browser behavior. You could even document them all on a single page for easy reference.

That day is, of course, long since past. CSS has massively expanded in scope, depth, and complexity, and it has penetrated into unexpected places. Trying to keep track of everything CSS has to offer is incredibly difficult, and when you take into account all the varying support profiles and behaviors of desktop browsers, mobile browsers, OS adaptations, embedded firmware, and more, it's practically impossible to comprehend a single snapshot, let alone keep up with ongoing changes.

So it's no wonder there are so many CSS frameworks out there. Frameworks are a great way of taming complexity. Rather than have to remember all the fiddly details of vendor prefixes and syntax, and rather than have to put up with the limitations inherent in the CSS syntax (still no variables!), you can load up a framework and let 'er rip. You can even, depending on the framework, invoke a few simple classes to get precalculated layouts. Frameworks are popular for very good reasons.

On the other hand, in many ways we've traded one form of complexity for another. It's a veritable jungle of frameworks large and small out there, and figuring out how to navigate that jungle requires an expert guide to get you off to a good start. You need that guide not to tell you the specific characteristics of every plant and animal in the underbrush, but to give you hard-won advice on how to approach various situations, what to look for and what to avoid, and thus how to thrive in a constantly shifting environment.

A guide like Jens Meiert.

I've known Jens professionally for many years now, and have been better for it. Jens is someone who always thinks deeply about the Web and the practice of building it, comes to a well-reasoned conclusion, and defends that position with honesty and candor. He cares as much as anyone I've ever known about best practices in web development and will yield to nobody in his defense of that principle.

I know, because when a CSS reset I created became unexpectedly popular, Jens was tenacious in his view that nobody, but nobody, should use a CSS reset. While I didn't entirely agree with his conclusions about resets, I always valued his perspective in that debate, which was (as usual for Jens) an important contribution to the ongoing discussion about best and worst practices in web development. Many of his predictions about how resets would be abused came true. He made a strong case, one that was set on clear foundations and grounded in his passion for web development done right.

Some time later, Jens took that passion to Google and made significant progress in improving the markup Google produced. Anyone who cares about the Web will instantly understand what a huge contribution that was. Now he's applying that same passion to the subject of CSS frameworks.

If you're thinking about using a framework—and there are, as you'll soon read, very good reasons both for and against taking that course—Jens' high-level yet deeply practical advice will help you make the best decision you can. In a like manner, the principles he sets forth here will help you decide if perhaps you should develop your own framework, which is sometimes a much better idea than trying to use someone else's.

To framework or not to framework? Let Jens be your guide. I could not put you in any better hands.

—Eric A. Meyer

Introduction

Many commercial websites these days are based on frameworks, and many personal websites use them, too. Yet what are frameworks, why and when do we need them, and how do we best use or build them?

This little book explores frameworks that govern HTML and CSS (and JavaScript) code. It focuses on HTML and CSS because these are at the heart of every web project. The principles outlined in the book, however, can also be applied to other forms of frameworks.

The goal of the book is to share solid, higher-level ideas around frameworks, trading some specificity for long-term usefulness. We could analyze all the different frameworks that are out right now, but if you wanted to make up your own mind or write a framework yourself, how useful would such review be if you picked this book up again in five years?

While the book attempts to cover all bases, it glosses over some of them, too. Web development has become a large field. Also, as we'll see shortly, framework development pivots around tailoring, and tailoring depends on circumstances. We don't know every project's circumstances, and so we can't generalize everything.

Although written in simple language, the book is geared toward expert web developers, the people who decide about whether and how to use, or whether or not to develop a framework.

It has likewise been written by a web developer. I, Jens, have during my career architected frameworks for OpenKnowledge, GMX, Aperto with their government and business clients, as well as Google. In that time, I've not quite managed to outwit the fast pace of

our industry, but I've found that some principles, methods, and practices contribute to longer-lasting code. That has benefited the frameworks I wrote, and I hope it will benefit you through this book, too.

Acknowledgments

I'd like to thank the following people for their help with this book: Tony Ruscoe for reviewing and advising to the first draft. Asim Janjua, my good friend, for sharing some of his thoughts. Eric Meyer for the generous foreword; if it wasn't for Eric's work, a huge part of the web development world would look grim today, technically speaking. Simon St.Laurent and Meg Foley for guiding the book down the right track at O'Reilly. The O'Reilly staff, particularly Kristen Brown, and the many other friendly, supportive people involved in making this information accessible and enjoyable. Julia Tang for her always loving support. The W3C and WHATWG groups, the Google Webmaster Team, and the people I've worked with over time who made me a better web developer. Thank you.

The Little Book of HTML/CSS Frameworks

Key Concepts

Before we dive into frameworks, let's first go over a few general ideas. We don't have to agree on everything; all we want is to prevent misunderstandings over the course of this book.

First, there are a handful of terms that may be used differently in other contexts:

External (also known as public or open)
: Anything that comes from outside ourselves or our organization and is out of our control. In web development, social site widgets or frameworks are often external.

Internal (or in-house)
: Anything that originates from within our organization and is within our control. In web development, site designs, or site style sheets, are often internal.

Pattern
: A classical design pattern. In web development, the individual elements of a document or app are patterns, but so are document types like a three-column article page.

Cost
: A measure of any negative consequence. Typically expenditures of work, time, or money, but possibly negative changes in, for example, perception, satisfaction, or reputation. In web devel-

opment, for instance, any element added to a page has a cost in terms of reduced page performance.

Tailoring
: The producing and adjusting to precise dimensions and needs. In web development, tailored code is all code that's needed—or going to be needed—by a project, but not more.

Second, some assumptions:

- Code has a cost. For example, there is the cost of development, performance, maintenance, documentation, process, quality, and conversion (though not all of them always apply, and not all of them affect the same groups). Unnecessary code has an unnecessary cost.
- Site owners and developers want to save cost. In particular, they want to save unnecessary cost.
- Tailoring code means removing or, better, not even writing or embedding unnecessary code.
- Good code is code that is of measurably or arguably high quality, where arguably means conforming to common best practices.

High-quality code can be said to be tailored, but it doesn't follow that high-quality code saves cost, at least not as a general rule. Tailored code itself, however, always saves cost. With this first insight, let's begin.

Understanding Frameworks

What Is a Framework?

"Framework" is a broad term, often misunderstood. Conceptually, a framework in the web development sense can be likened to a library: a library not of books but of design patterns, complete with all needed functionality.

For example, the Pure framework knows, with overlap, the following button types:

- Default
- Primary

- Icon
- Active
- Disabled
- Customized

Functionality usually means presentation (styling via CSS) and sometimes also behavior (scripting via JavaScript). The advantage of using a library is that we don't have to code this functionality ourselves, or do so repeatedly. We instead follow the library's instructions for the structural side (markup via HTML).

For example, YAML requires the following HTML code for a horizontal navigation menu:

```
<nav class="ym-hlist">
  <ul>
    <li class="active"><strong>Active</strong></li>
    <li><a href="#">Link</a></li>
    <li><a href="#">Link</a></li>
  </ul>
</nav>
```

The only missing piece or, literally, link, is connecting the library so to have it apply the functionality to the chosen patterns, on basis of the mandated markup.

For example, to use Bootstrap, we must reference something like:

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/
3.3.1/css/bootstrap.min.css">
```

Now that we compared frameworks to fully functional pattern libraries, here's another view. Frameworks can also be seen as just the style sheets and scripts they are, and external frameworks as *shared* style sheets and scripts that get lifted to a higher status. We could indeed pick any style sheet or script or both and declare it a framework!

The implications of this second insight are far-reaching. Although rather trivial, it's one of the keys to understanding frameworks. We'll keep the term "framework" to use common industry language but will at times look at the idea of elevated style sheets and scripts for guidance.

Why Frameworks?

Frameworks promise to save both development and design time. The thinking goes that many of the things site owners and developers want have been done a thousand times, and thus there is no need to reinvent the wheel. Internal frameworks commonly enjoy a more sober regard, so this particularly applies to external frameworks.

If frameworks come with this promise, the question arises whether or not they live up to it. The answer boils down to a cost calculation that is, unfortunately, different for every framework and project. How much development cost was saved? How much was, in turn, spent on training, customization, and upgrades?

Apart from suggesting that we do the math and *think* through every project, the following pages cover frameworks in the necessary detail to empower everyone to form their own theory about the raisons d'être of frameworks.

Types and Uses of Frameworks

While all frameworks provide patterns, we must note general distinctions. For one, there is a difference between internal and external frameworks—the external ones are those that typically get referred to as frameworks. Then, there is a difference between using and developing a framework (note that developers can be users, which makes for some blurriness). And finally, there is a difference between experts and amateurs.

Let's chart this up.

	Expert		*Beginner*	
	Use	*Develop*	*Use*	*Develop*
Internal framework	?	?	?	?
External framework	?	?	?	?

What do you think? Should either type of framework be managed either way, by either group?

Here's what I think. Let's compare.

	Expert		*Beginner*	
	Use	*Develop*	*Use*	*Develop*
Internal framework	Yes	Yes	Yes	Yes
External framework	No	Yes	Yes	No

Please note that developing an internal framework and making it public, as we could even apply to blog themes, is here not considered *developing* an external framework. The decisive factor is the goal during the initial development process. A thorough revision and overhaul of an framework to make it external or internal-only, however, constitutes a development phase, and would be acceptable.

Reflected in the table is the idea that frameworks can be used and developed liberally, with two exceptions. One exception is that *experts shouldn't use external frameworks*; the other is that *beginners shouldn't develop external frameworks*.

The two exceptions stem from a violation of quality standards: while the external framework violates the ideals of the expert (which I will later describe), it is the beginner who would not even know the necessary ideals to create a quality framework.

The internal framework is safe to use or develop in every case because that's the preferred way of developing web documents and apps. Internal beats external every time because external cannot, by definition, know all the needs of the organization and fails many quality standards. Second, internal solutions are the better route for both experts and beginners to stay sharp and to learn, since their mistakes have a smaller impact.

The development of an external framework is safest only with an experienced web developer, who can, following the principles outlined in this book, skillfully build and document it so that it has a better chance to be useful, at a low cost-benefit ratio. For the less experienced developer or the one in a hurry, use of an external framework is thought to be more viable simply because things mat-

ter a lot less for him; he may discern few impacts in quality, and he may not yet have a long-term vision for his project.

Compilation Frameworks

Compilation frameworks are frameworks that include third-party style sheets and scripts. These may be public reset style sheets, but can extend to elaborate UI elements. Skeleton, for example, used to build on Normalize.css, while Blueprint is thought to incorporate Eric Meyer's CSS reset. WrapBootstrap and Flat UI Pro are arguably compilation frameworks because they extend Bootstrap, but we typically find the compilation framework species internally, when institutions build their own frameworks based on existing public ones.

We don't cover compilation frameworks in more detail because they expand on the external frameworks we do cover. But to err on the safe side: composite frameworks mean composite problems, and there's extra work involved in testing and maintaining. Special attention is in order.

Popular Frameworks

There are many dozens of HTML/CSS frameworks that developers have found useful. Here is a representative selection, to give you an impression of what the world of external frameworks feels like:

- 960 Grid System
- 1140 CSS Grid
- Base
- Bijou
- Bootstrap
- Blueprint
- Cascade Framework
- Columnal
- Compass
- CSS Smart Grid
- Fluid Baseline Grid

- Foundation
- Gantry
- Golden Grid System
- Goldilocks
- Gridiculo.us
- Gridless
- Gridlock
- Gumby
- Groundwork
- HTML KickStart
- HTML5 Boilerplate
- IceCream
- Ingrid
- InuitCSS
- IVORY Framework
- KNACSS
- kouto swiss
- Kube
- Layers CSS
- Less Framework
- Metro UI CSS
- Mueller Grid System
- Profound Grid
- Pure
- Responsee
- ResponsiveAeon
- Responsive Grid System
- Salsa
- Semantic Grid System
- Simple Grid
- Skeleton

- Susy
- Titan
- Toast
- Tuktuk
- YAML

(Some readers will remember Choke, too, although that humor may have been rather crude.)

These frameworks all vary in functionality and scope. Some focus on base layouts, while others go all the way into comprehensive print and mobile themes.

Such a list of frameworks is the type of information that goes stale quickly. While some frameworks, most notably YAML (not to be confused with YAML Ain't Markup Language), have been around for many years, other frameworks come and go. It's more useful to obtain said impression from this list, regard it as a snapshot, and, perhaps, make it a starting point to experiment.

Attributes of a Good Framework

Now, what is a "good" framework? What does a framework have that we want to use? What constitutes the framework we may want to build? I've thought about and worked with and discussed this question many times.

In a professional or expert context, "good" usually refers to quality. We can establish this for frameworks as well. A framework should, especially when it's an external one, meet the highest applicable quality standards.

Frameworks tend to be only used after a project reaches a certain size and complexity (a one-pager doesn't need Bootstrap or YAML). They're also done by third parties. As size and complexity makes issues weigh heavier (and since third parties, as we have seen, cannot know a project's full needs), we're in need of some guarantees and safeties.

We can get one such safety if we can curb the bloat that external frameworks in particular bring. We know what helps: tailoring. So a good framework should expressly be tailored.

If we assume a complex project, we're likely not alone working with or on the framework; and if it's an external one, we have no idea whether the developers of that framework speak our language (literally and metaphorically). What helps here is usability. A good framework should be usable.

And then, requirements change just as the times: how do we work with the framework going forward? What if we need to add something, perhaps in a pinch? What helps with that is extensibility. And thus a framework should also be extensible. At least we or the framework should be clear how to extend it.

We're just being professional and reasonable when we demand quality. We gain extra confidence, then, by wanting frameworks that are also tailored, usable, and extensible. Let's look at these three special attributes a little closer and point out the options developers have to get frameworks to that state.

On Quality

It's easy to just say "quality," and, "Yes, I'll have that, too." But what exactly *is* quality? Or, for our purposes, what is quality code? When we think about it—consider lifting our eyes from these pages, and think code quality—we won't struggle to find more than just the ideals of tailored, usable, and extensible. There's also:

- Fast
- Accessible
- Semantic
- Robust
- Scalable
- Documented
- Maintainable
- Valid
- Self-explaining
- Consistent
- "Automagical"
- State of the art
- Simple

- Compact
- Flexible
- Tested
- Fault-tolerant
- Self-correcting
- And more!

This doesn't even include anything emotional we may want to attach to quality, like "pleasant" or "fun." But what we see is that quality has many different faces.

1. A Framework Should Be Tailored

We defined tailoring as "producing and adjusting to precise dimensions and needs." Producing refers to developing a framework—whether internal or external—while adjusting commonly means fitting an external framework. The key here is "precise dimensions and needs." We need to know our needs—otherwise we can neither produce nor adjust something to fit.

One view of tailored code, by the way, is to compare needed code with overall code. That can be hard to measure, because the number of characters or lines in our code doesn't do the trick. But conceptually, tailoring means using as little and yet as effective code as possible, and not more.

What can we do to tailor? The approach depends on the origin of the framework, and that origin makes for a big difference.

An internal framework is relatively simple to tailor: We develop to the needs of our project from the beginning. These needs may be defined by comps (comprehensive layouts) and mocks (mock-ups) or, better, a style guide. Once all needed page types and elements have been specified, they're coded up. If they're all used by the later site or app, the code cannot be anything but tailored (although it can possibly still be optimized and compressed).

An external framework, however, is much more difficult to tailor (by the receiving side, because it's impossible for the originator). In a basic sense, we need to deduct all needed functionality from all offered functionality, and then remove the code that remains. That leads us to the key issues with external frameworks: removing code

may not even be possible, and tailoring then depends on the quality of the framework code and its documentation (e.g., tailoring will require testing, might break the framework, and could make the same work necessary for later updates, if not outright thwarting the ability to move to newer frameworks).

These are big issues that make for good reasons why few people actually go to the length of customizing or tailoring external frameworks (or any external code, for that matter). Yet the outcome—non-tailored and lower-quality code—is not very expert-like, and inferior. And so we see with more clarity why in a professional context, external frameworks shouldn't be preferred. They promise to save cost, only to come with a stiff hidden tax or else bring down the quality of our work.

Now, some frameworks like Bootstrap or Gumby have begun to address these problems by offering sophisticated customization wizards. This is smart, because it significantly alleviates the issues of non-tailored solutions. Framework developers should offer and users use such functionality.

By the way, there's another problem we need to consider: while we're benefiting from either our decision to save cost or to improve quality, our end users benefit mostly from quality. Technically speaking, they are rarely on the list of beneficiaries if we decide to deploy a framework that's bloated but easy to churn out.

To tailor internal frameworks:

- Be clear about needs.
- Build the framework to these needs.

To tailor external frameworks:

- Be clear about needs.
- Customize or modify the framework to these needs (or abstain from the framework).

2. A Framework Should Be Usable

A good framework is not only tailored but also usable. But what is usability for frameworks? It starts with applying the common defini-

tion of usability: ease of use and learnability. And with a universal rule: keep it simple. Simplicity helps everything.

But that's not quite a complete answer, and so we need to differentiate again. The distinction that serves us here is not one between frameworks, but between roles: framework users and framework developers.

For the framework user (who may be a developer himself but is now concerned with working *with* the framework), a usable framework is also easy to understand. That ease of understanding is primarily achieved through clear framework documentation and, where applicable, concise code.

For the framework developer, there's much more emphasis on usable code. Luckily, there are two things we can firmly link with helping code usability: maintainability practices and code conventions (coding guidelines). Adherence to maintainability practices and consistent style are the backbone for usable code.

With slightly smaller boundaries than developer experience, I generally believe there is a subfield of usability: *developer usability*. It could be defined as "the ease of use and learnability of code." Perhaps this field doesn't get much attention because usable code goes under different names, as we just found, but perhaps it would benefit from being treated separately.

To make frameworks more usable for users:

- Keep it simple.
- Follow usability conventions.
- Perform usability tests.
- Provide documentation for framework users.

To make frameworks more usable for developers:

- Keep it simple.
- Aim for self-explanatory code.
- Format code legibly and consistently.
- Follow maintainability best practices.
- Provide documentation for framework developers.

3. A Framework Should Be Extensible

The final attribute to underscore is extensibility. Extensibility for a framework means that it's not just possible, but well-defined and easy to extend it.

Extensibility is necessary for two reasons. First, external frameworks in particular won't offer everything we need, so there needs to be a way to add functionality. Second, especially in large projects, there's a tendency for new patterns to pop up. The problem with these is their uncertainty and uniqueness: they may only be used once or twice and don't warrant a place in the framework core or even near more common extensions. Both their location and handling have to be thought of.

To make up for lacking functionality in a framework, users typically help themselves by pretending they don't use a framework in the first place. That is, they have a style sheet or script that handles everything the framework doesn't cover. That's actually quite OK; the point here is to *be clear* about how such "non-framework functionality" or extensions are handled (and we notice how extensibility is also a user responsibility). If nothing else, extensibility stresses the need for the most basic of all code safeties: a namespace (a framework-specific ID and class name prefix, and the same namespace in JavaScript).

Next, new and rarely used patterns are a challenge that runs in the best families. There tends to always be a need for something new, and there are always document types or elements that are used infrequently. They're one of the biggest contributing factors to code bloat. They are hard to control if they don't get watched and reigned in vigorously. Though I could give a longer dissertation about the matter, an effective counter-practice is to either designate style sheet and script sections for new and experimental code, as well as rare elements—or to even put aside a separate style sheet and script for such purposes. The framework developers should anticipate this and make recommendations, but users should come up with their own guidelines if this piece has not been covered. A documented standard for new code allows better monitoring and better decisions on whether to keep (and relocate) the code, or to remove it.

We've very successfully applied this principle with Google's HTML/CSS framework Go—not to be confused with the programming language, which was conceived two years later. Go came with a

"backpack" library, Go X, which included elements that we used only occasionally. This kept the core very small—4,250 bytes including the Google logo—but offered the use of additional, common-enough elements. Project-specific code made for a third layer that had to be carried by each project style sheet itself.

To make frameworks more extensible:

- Use a framework namespace.
- Define handling of non-framework code.
- Specify where new and rarely used code should be located (also a framework-user responsibility).
- Regularly review new and rarely used code, to either make part of framework or remove (also a framework-user responsibility).

NOTE

Please note that despite all my experience and convictions, I've phrased these rules as strong suggestions. I was tempted to say "must," "must," "must." Whenever we like more dogma in our web development life, we use this verb.

Another thing before we move on: note that no matter the quality of the framework, the *goal* for its use is always on the owners and developers. Frameworks can be likened to cars: a good car should be, say, safe, easy to handle, and economical. And so a good framework should be tailored and usable and extensible. But just as we look at the driver to know the destination for her car, we look at the developer to know the goals for the framework she's using. We can drive a framework against the wall just as we can a car, which is the reason we differentiate between experts and novices. Just to get this out there: a framework doesn't drive itself.

Using Frameworks

Two ways we've been exposed to frameworks are by using and developing them (with some inherent overlap). Our initial definition gives this an interesting spin, as we have seen that we can regard any style sheet or script as a "framework." So anyone who has worked with style sheets and scripts already has a *basic* idea of how to use frameworks.

After all that we've learned, using can't be as complicated as developing, and must mostly depend on the framework. It requires a choice of framework, and then demands two ground rules.

Choosing a Framework

The "pro-quality" choice has been explained as using or developing an *internal* framework, and choosing a framework generally applies to external ones. The choice of an external framework depends on two factors:

1. Which one meets our needs the best?
2. Which one is of the best quality (that is, which one is as tailored/customizable, usable, and extensible as possible)?

These questions underline the importance of knowing our precise needs. It is even important in order to *pick* a framework, as knowing our needs helps determine which framework fits better (tailoring) and comes closer to our extensibility needs (though simple needs don't require extensibility as frequently as comprehensive needs).

The Two Ground Rules of Using a Framework

And of *any* framework at that. These two rules are golden:

1. Follow the documentation

Whether internal or external framework, whether expert or beginner, read and follow the documentation.

This rule is paramount because the second source of quality issues with frameworks and the works created with them (after framework bloat) is user and developer error. Or user and developer misconduct! Some scenarios that illustrate this might be when a pattern is hacked to work, when something has been developed that's actually already part of the framework, when things get overwritten without regard for framework updates, or when something has just been "made working."

When using frameworks, always follow the documentation.

2. Don't overwrite framework code

For reasons that will become clearer in the next section, never overwrite framework code.

Contributing to the expert's dilemma with external frameworks, overwriting framework code can have unforeseen consequences and break things with future updates. Here's an example:

Framework:

```
header {
  /* No layout declarations */
}
```

Overwrite:

```
header {
  position: relative;
  top: 1em;
}
```

Framework update:

```
header {
  left: 0;
  position: absolute;
  top: 0;
}
```

The example, simplified as it is, shows how a seemingly innocent change can have acute consequences. Here, a header is moved by one em. (Note that the example constitutes an overwrite because the framework header is inherently "positioned" and also rests on the initial values for `position` and `top`.) The next framework update, however, switches to absolute positioning. As the overwriting rules come later in the cascade, they prevent the update from working (with the exception of `left: 0;`). In cases like this, overwrites are unpredictable. Overwrites should hence be avoided where possible.

The remedy: For internal frameworks, update the framework, or leave things as they are (as in, no overwriting). For external frameworks, leave things as they are, or create a separate pattern that does the job (like an alternative header, with different markup). Stay away from forking or "patch improvements"; solve issues at the core, or not at all.

NOTE

The more complex the project and the bigger the organization, the harder it can be to display the necessary discipline. Everyone working with a framework needs to follow these two rules, however, to achieve the highest levels of quality and consistency possible.

Overwriting Versus Extending

There is a fine line between overwriting and extending code, especially since overwriting doesn't necessarily mean *changing* code. Here are conceptual examples for both (with a CSS twist but applicable to other languages):

Overwriting:

- A → B: code A changed to, or replaced by, B.
- $A_1 \rightarrow A_2$: code A, like a rule or function, doing 1 changed to doing 2.
- $[A_1 + B_1]$: code A doing 1 extended by code B doing 1, too or differently, in the same file (this is overwriting because the *effect* of the original code changed).
- $[A_1] + [B_1]$: code A doing 1 extended by code B doing 1, too or differently, in a different file (this is also overwriting because the *effect* of the original code changed).
- $(A_1 + B_2$: not a case of overwriting nor extending, because it's exemplifying different code doing different things.)

Extending:

- [A + B]: code A extended by code B, in the same file.
- [A] + [B]: code A extended by code B, in a different file.

There are more cases, especially if we consider additional code snippets (not just "B," but also "C," "D," "E," etc.) affecting code written elsewhere ("A"). And sometimes, overwriting can be intentional or even elegant, so we don't want to rule it out entirely!

The point is, especially for CSS, overwriting can not only have side effects and introduce inconsistencies, but it can make our work extremely complicated. When we look at the case $[A_1] + [B_1]$, for example, we notice that we can face two challenges on debugging: first, what happens exactly (why is A not in effect anymore?), and

second, where does it happen? Extending in the same file, or in another well-defined manner, causes fewer issues.

Developing Frameworks

Developing frameworks is an art form that comes with a lot of responsibility. For external frameworks, it comes with the aura of a daredevil (though naiveté rears a head, too). As we've seen throughout this book, it's by necessity most difficult to build an external framework because we cannot *know* the needs of other projects. And hence, we can hardly avoid shipping something that is incomplete—or that may mutate into bloat.

The following pages describe the basics of writing a framework. The ideas describe the situation of an experienced web developer leading a framework effort in a large organization.

Principles

We've already done our assignment and fleshed out the principles for framework development. A framework should aim for the highest quality standards, and then:

1. A framework should be tailored.
2. A framework should be usable.
3. A framework should be extensible.

These shall serve as every framework's core values (for which we can use the avenues outlined earlier).

Customization, as identified under "1. A Framework Should Be Tailored" on page 10, plays a special role here, for it is the secret weapon—and last line of defense—of the external framework developer. Offering framework customization options is the only way to get closer to tailoring for outside users, users whose projects we will never know.

I decided against including a section about customization because it's not a magic pill for external frameworks, and can stack the whole deck against the framework developer instead of the framework user. This is because the more customization options there are, the more complex the framework gets. Yet that's still only talking frame-

work development. The framework and all its customized subversions, as we'll see shortly, still need to be tested, quality-managed, maintained, and so on.

Prototype

The single most important thing we need to build a successful framework is a prototype. Here we benefit from our recognition that we're really only talking about plain-vanilla style sheets and scripts. Large projects—projects like those for which we now talk frameworks—have always benefited from prototypes.

What do we mean by prototype? In its simplest form, it is a static (internal) website or demo site. It should contain all document types and elements we need in production: the prototype is where we code all the structure (HTML), presentation (CSS), and behavior (JavaScript). And the prototype should include realistic (occasionally intermingled with extreme) sample contents: that's how we test that everything works.

A prototype is an irreplaceable testing ground that we need to obtain the end result we want.

Prototypes follow their own principles, however. They must be, as I attempted to summarize in earlier years (slightly reworded):

- Complete
- Current
- Realistic
- Focused
- Accessible/available
- Managed with discipline
- Maintained
- Communicated/promoted
- Documented

Each of these points is important, but the first three are critical. The prototype has to include everything (all document types and elements), it must be current (any functionality changes must be reflected immediately), and it needs to be realistic (the sample data must

be an as-good-as-possible approximation of how the framework is going to be used outside of the prototype).

Quality Management

In order to be sure that we deliver the quality we're committing to as professionals, we need to verify it. This is done through quality assurance (which aims to prevent issues by focusing on the process), and quality control (which aims to find and fix issues in the end product).

Web development, as a still rather young discipline, knows more quality control than quality assurance. Good examples are validation, accessibility, and performance checks, of which there are plenty. On the quality assurance end, the most prominent example is the enactment of coding guidelines, but some organizations and individuals go so far as to use elaborate infrastructure to continuously test and improve their code. (This is all related to web rather than software development, since in software development, there is a longer history and strong tradition of working with tests.)

For quality assurance, it's useful to:

- Establish coding guidelines
- Define output quality criteria
- Run regular tests (over prototype and live implementations)

For quality control, test:

- Accessibility
- Links (if applicable)
- Performance
- Responsiveness
- Maintainability
- Validation
- Linting
- Formatting

(Incidentally, I run a website hub dedicated to web development testing tools. Check *uitest.com/en/analysis/* for a large selection of them.)

To take a page out of Google's book, it's best to automate such checks. Reviewing tool documentation can give valuable pointers, as a number of tools can be installed locally or come with an API. In addition, there are instruments like Selenium and ChromeDriver that facilitate automated browser testing. As with many of the more complex topics, this book will resort to just showing directions.

Maintenance

We've so far noted how principles, a prototype, and quality management are important in framework development. The last key item to stress is maintenance. Maintenance here primarily means (similar to prototypes) a strong commitment to move forward with a framework. This is important for two reasons.

For one, in the case of external frameworks, maintenance is crucial because publishing a framework is a promise to the user base. That promise is precisely that it's going to be maintained. It's also a promise in *how* it's going to be maintained, in that we do everything in our power not to change any structure, but only the framework style sheets and scripts.

For another, in any framework, a commitment to maintenance acts like another form of safety. The general idea in web development is that the HTML is most important to get right, because it's more expensive—think our cost definition—to change than style sheets and scripts. An explicit commitment to maintenance will keep us from discarding a framework to just "build another," and thus lives up to the vision of CSS-only design iterations and refactorings. (Of course, true structural changes will still always require HTML changes, and with that, eventually, CSS and JavaScript edits.)

A framework, solving widespread and complex development and design issues, comes with an express obligation to maintenance.

Maintaining and Breaking

A word of caution: while we, as framework developers, need to show responsibility and exercise thorough care not to break anything with framework updates, we must also not worry too much about breaking something. On the question what the most important thing was when creating and maintaining frameworks, I responded a few years ago:

> "The more used and hence successful a framework is, the more likely it is that things will break over regular maintenance. […] maintenance is key, yet avoiding things to break is impossible, and the attempt alone unbelievably expensive. And then, while things go wrong, things never go wrong badly. And that leads back to setting expectations: […] people should embrace, not fear, updates to them."

I believe that this advice is still sound. Code can't afford anxiety. If we developers do our homework, occasional breaks can even be a good thing (and maybe the only thing) to educate framework users about ground rules and proper framework usage, and to expose other coding-related issues. As developers we sometimes have to make tough choices.

Updates

The handling of framework updates is delicate enough to deserve a separate section. Updates are generally easier to manage for internal frameworks than for external ones, though updates in a large organization with many developers spread out over many places can be challenging, too.

Here are a few tricks to make framework updates easier:

- Try to avoid HTML changes because of their comparatively high cost. An update should only consist of styling or scripting changes and impart *no* actual work for users (which, to counter the aforementioned definition blurriness, also means developers who work with the framework). The update of framework references can be OK.
- Announce updates in advance.

- Provide a way to test whether the update would have any ill effects. This can happen through something simple like bookmarklets (see "Test Bookmarklets" on page 23), or something more sophisticated like proxying (using a proxy to intercept and change requests to framework files in order to feed updated files for testing).
- Inform users about possible side effects (and use this as an opportunity to educate them about, for example, the problems of overwrites, as explained in "2. Don't overwrite framework code" on page 15).
- Communicate the status of ongoing updates.

What we're assuming here is that we're not just "versioning" frameworks. That's the practice of shipping a framework—let's say, *foo*—and when the first changes come, not updating *foo*, but shipping *foo-2*. And then *foo-3*. And so on. This practice may be an *option* for us, but not a rule. The rule should be to update the framework itself, per the ideas listed here. The reason is that versioning defeats the purpose and advantage of CSS (similarly for JavaScript), which are immediate changes, supported by separation of concerns (HTML for structure, CSS for presentation, and JavaScript for behavior). We'll touch on the vision behind this shortly, but we should strive to do all updates through what we already have. And only for major changes do we look into our toolbox and, always carefully, reconsider versioning.

Test Bookmarklets

CSS bookmarklets are a great low-tech way of allowing users to test framework changes. A short example:

Framework:

```
article {
  margin: auto;
  width: 50%;
}
```

Framework after update:

```
article {
  width: 90%;
}
```

To prepare a test for the update and make sure everything keeps working as intended, take all the changed declarations and all the *removed* declarations, set the removed declarations' properties to their defaults, and on that basis, generate the testing rules:

Test style sheet:

```
article {
  margin: 0;
  width: 90%;
}
```

We're simplifying here by assuming that `<article>` doesn't pick up other margin values from anywhere else. `margin` has to be set because while it's been removed from the new framework, it would still be applied through the old framework, which is in effect on the pages where we want to test the new code. So the test style sheet needs to neutralize all old code—something we could in really tricky cases, as a last resort, also attempt through the `all` property.

Jesse Ruderman's bookmarklet generator is minimal but a fine tool to turn test code into a bookmarklet (by pasting the test CSS and copying bookmarklet code). That bookmarklet can then be provided to any framework user, along with a way to report problems.

Documentation

Though not technically a part of the development process, documentation must be discussed. Anchoring documentation where the development happens has many advantages, from increasing the chances that it's actually done, to being more comprehensive and accurate because it's fresh on the mind.

There are several ways to document, but one of the more effective ones is using a prototype for this purpose too. Sample contents can be turned into documentation describing the page types and elements they're forming, but it's also possible to use hover-style info boxes that share background information and explain the code. (A properly maintained prototype enriched this way may even do most of the lifting of any framework *site*!)

Documentation begins in the code, however, and there, too, we need to exercise discipline. Our coding guidelines should underline this; documentation standards like CSSDOC and JSDOC, as well as tools

that automatically pull such documentation from our code, can be of great help.

The idea behind documentation is to make life easier for ourselves, our colleagues, framework users, and any people interested in the work. Thus, making it part of the development process goes a long way.

Logistics

Our journey, now that we diligently worked through everything relevant to frameworks, is soon over. A bonus aspect concerns logistics. We have covered a few pieces that can be considered logistics:

- Coding guidelines
- Quality-control tools
- Documentation

What we haven't touched are:

- Framework development plans or roadmaps
- Version control systems (like Git, Mercurial, or Subversion)
- Request and bug management systems (like Bugzilla)
- Framework sites (public for external frameworks) with news feeds
- Mailing lists for
  - Developers (framework development team)
  - Users (open to everyone interested)
  - Announcements (low-volume essentials which should go to the developers and users lists, too)
- Trackers for live implementations

A framework site and an announcements list are particularly noteworthy, as they can pay good dividends to framework owners and developers. The site serves as a hub for information and documentation. An announcements list is indispensable to inform customers about new releases and guide framework users.

Support also falls into the logistics category. It does not get more attention here because, for one, we "embed" support at several land-

marks along the way—in quality goals and principles, in documentation and logistics—and for another, support is more of a tangential topic that depends on the complexity and circumstances of the framework and the problems it tries to solve.

NOTE

To repeat, for expert framework development, we need to pay special attention to:

- Principles
- A prototype
- Quality management
- Maintenance
- Documentation
- Logistics

As these are ordered in descending order of importance, our frameworks can probably survive with poor support and gaping docs, but sacrifices in vision, testing, and commitment will break their necks.

Common Problems

Since frameworks are most useful in larger projects, problems involving frameworks tend to be bigger, too. Here are a few of the most common and gravest issues, along with ideas on how to address them.

Lack of Discipline

One of the most severe issues is lack of discipline. For the user, this most commonly means not using frameworks as intended and violating the two ground rules (following documentation and not overwriting framework code). For the developer, this usually means becoming sloppy with quality standards, the prototype, or documentation. The result is the same: sabotage, just from opposite sides of the table.

The solution is not easy to come by. Users of external frameworks are free to do what they want anyway; they may not even notice that an external framework is very difficult to ride in the first place. It's a bit easier internally, where rules can be established, communicated, and enforced. Personally, while I have observed many issues in prac-

tice, I haven't found a cure for this one yet. People are just very creative, and watching how frameworks end up being used is like looking right into the face of human nature (and Murphy's Law).

Lack of a Prototype

Not having a prototype is an equally critical problem, for all the benefits outlined in "Prototype" on page 19. Apart from the fact that framework development is so much harder without a contained environment, maintenance complexity increases by the minute if there is no prototype. A framework without a prototype is essentially freewheeling, out of control. As suggested earlier, a mere collection of static pages—as long as it's complete, current, and realistic—does help.

Lack of Maintenance

If we do not maintain (or stop to maintain), outside of major structural changes or prolonged resource shortages, we miss great opportunities. In the case of external frameworks, it can damage the reputation of those providing the framework. In the case of internal frameworks, it can mean giving up control over the framework-managed docs and apps, and thus slowly being forced into a costly, full-blown relaunch.

Maintenance doesn't mean we should continuously *change* a framework—that may even be hurtful, especially for external frameworks because of the nuisance it creates. Rather, we should regularly monitor, test, and tweak the framework to keep it current. Such care pays off in many ways, be it because it reduces the need for more drastic changes (relaunches, which are pricey) or because everyone's staying in touch and familiar with the framework.

A Vision of Web Development

There is one thing every web developer should aspire to: writing the most minimal, semantically appropriate, valid HTML, and then never changing it. "Never" not in a sense of denial and refusal, since structural changes can always require modifications, but in the sense of a guiding light. The idea of minimal, semantically appropriate, valid markup brings the most out of us as web developers. It leads us not only to supreme markup quality, but pushes us to acquire and exhibit bigger powers in our style sheets and scripts.

The vision is one of highest efficiency, to handle presentational changes only through CSS updates and behavioral ones only through JavaScript updates. Writing HTML, design-agnostic as it should be, has always been underestimated; it's the hardest to write well.

Lack of Accuracy

An assumption we've made thus far is that what our frameworks do is accurate—that is, that they match the underlying needs and designs. That latter part can be a potential source of error if the frameworks we coded or found ourselves using don't match the specs our designer friends sent us (if we're not the designers ourselves). This can lead to all kinds of issues: from not being able to accommodate the original plan (no use for our external framework) to needing structural changes (ouch) to asking the designer folks to rationalize and Photoshop the differences away instead of fixing the framework. We need to watch out for design and style guide divergence.

Lack of Guts

The last big issue is to not have what it takes—even if that's manager support—to pull the plug. Clinging on to something that's not relevant anymore. Something that's not used anymore. That's used wrong. That's a construction ruin. That can't be maintained or extended. Something like that.

Sticking with a broken framework, a failed one, or perhaps a glorious one that has just reached the end of its lifetime can be a challenge. When that happens to us, we need to get over it. As professio-

nals, we have big goals and we want our code to last—but sometimes we fail, and then we need to…suck it up.

Fortunately, there's always more code to write. The next framework —or style sheet, or script—is already waiting for us.

Summary

Frameworks are deceptive. They seem easy. They look like a pretty isolated special topic. And now we've seen how common and complicated they are, like a not-entirely-small meteoroid that passes every single planetary object in our web development solar system. Frameworks are not trivial. If I may distract from the speed with which I typed this down, with brevity as an ~~excuse~~ goal, then any question still open is due to that very fact that they're not.

But I want to recap. Professional web development is about quality. Quality is not easy to define, but one part of it is tailored code. External frameworks without customization options are impossible for users to tailor, and a pain for developers. Internal frameworks are much easier to handle and generally the way to go. Good frameworks aim for the highest quality—to be tailored, usable, and extensible. Framework users should follow the documentation and not overwrite framework code. Framework developers should have principles, a prototype, quality management tools, a maintenance plan, and healthy interest in documentation. And still, things can go wrong.

If they don't, we may be on to the one framework. The one framework for us. Well done.

About the Author

Jens Oliver Meiert believes that people have many talents.

Jens is an established expert web developer who has contributed to technical standards of the Web (W3C), and laid the foundations for large-scale international websites that reach millions of users every day (GMX, Google). He has written a German book about designing with CSS (*Webdesign mit CSS: Designer-Techniken für kreative und moderne Webseiten*, O'Reilly, 2005/2007), developed frameworks and quality management systems for Google (2008–2013), and somehow got nominated "Web Developer of the Year" by *.net magazine* (2011).

Jens is an everyday adventurer, described in another book (*100 Things I Learned as an Everyday Adventurer)*, who has traveled to 6 continents, 70 countries, and more than 400 cities. He has made it his priority to try whatever is new to him, whatever activity or lesson he has the chance to, and completed almost 200 of these in the last 3 years alone. In 2013, Jens quit his job and left his city to go backpack the world, which he's still doing now, 18 months later.

Jens is a social philosopher who has over the last few years studied more than 300 works around the topics of philosophy, psychology, social sciences, public policy, economics, and more. He's an active supporter of and contributor to initiatives that place emphasis on freedom, trust, and rights, and he's focusing his own efforts on breaking through the probably-not-so-accidental apathy we observe in our world today.

Follow Jens's work on his website, *meiert.com*.