The Guide to ExpressionEngine® Development

A practical method to develop ExpressionEngine projects and version them with Git.

by Ryan Masuga

Foreword by Derek Jones, CEO, EllisLab

The Guide to ExpressionEngine® Development

A practical method to develop ExpressionEngine projects and version them with Git.

by Ryan Masuga

Be a better ExpressionEngine developer right now.

Masuga Design has developed ExpressionEngine sites of all sizes since 2006. We have tens of thousands of hours of experience and now you can benefit immediately by reading *The Guide to ExpressionEngine® Development*.

"With his work building and running devot:ee, dozens of client projects, and ExpressionEngine add-on development, Ryan is an ideal guide for best practices on EE development."

—Ryan Irelan, Mijingo

"Ryan and the team at Masuga Design have been staples in the EE community for as long as I remember. I'm excited about them sharing their knowledge...my copy is going to be dog eared for sure."

—Marcus Neto, Blue Fish Design Studio

"Ryan's contribution to the EE community has been invaluable, whether it's helping someone out on social media or his hard work running devot-ee.com. What would we do without him?"

—Rob Allen, Freelance Developer

First Printing: 2015

ISBN: 978-0-9966336-0-4
eBook ISBN: 978-0-9966336-1-1

Elements used in cover artwork: freepik.com

Masuga Design, LLC
Grand Rapids, Michigan

masugadesign.com
eeguidebook.com

Contents

Thanks

Endless thanks to my family, for whom I spend so much of my time working on ExpressionEngine projects. I'm always grateful for my wife and three joyful kids.

Super big thanks to the hardworking team at Masuga Design. Good stuff!

Thanks to everyone who wrote to me about the non-existent Part 2 of my ExpressionEngine and Git "series." The shame of not following up on that was one of the driving forces to finally write more about EE and Git, and to take the whole thing to the next level. Blog post? Bah! *Let's write a book!*

A shout out to Nathan Barry for inspiring me to sit down and write it out, in order to become an "Authority."

A tip of the hat to Brennan Dunn and Kurt Elster for teaching me the power of positioning and the necessity of "niching down."

Many thanks to every developer who has ever used devot-ee.com, and every client who has ever given us a problem to solve.

A special thank you to Jean Hanks for all her research into the technical stuff it took to put this together, and her help in formatting the different versions.

Thanks to Dren Asselmeier for her help editing and proofreading.

Thanks to Derek Jones for his encouragement and suggestions, and for taking time to write a foreword. Thanks also to EllisLab, for creating and continuing to improve ExpressionEngine.

Finally, thanks to you for taking an interest in what I have to share.

Foreword

As our skills with a tool grow, what we can accomplish increases, as does our reach. ExpressionEngine's renowned flexibility is the result of its modular design, a collection of tools that a craftsman brings together to create great web sites. As the author of this book points out, that flexibility results in a near-infinite number of ways to implement solutions. And as our reach expands, so does the complexity of problems we can solve.

Often critical to success is understanding some truisms. "Don't be clever" for example. "Clever", when it comes to our trade, is a bad word. The more complex the problem, the more critical that our solution is simple. "Keep It Simple Stupid" increases in importance as our skills grow. It protects us from ourselves.

It is then that our implementations are robust, not brittle. Understandable, not confusing. Flexible, not rigid. Ready to solve future problems instead of just the one in front of us today.

There are many books, videos, tutorials, and blogs that can help you learn ExpressionEngine, how to model your content, and build your templates. But there aren't many resources out there that discuss process. Certainly not with the detail and organization the author provides. Even those with many years of ExpressionEngine experience will likely find something new and helpful.

In providing support for our software, we've seen it all, the good and the bad. If everyone adopted the practices in this book, it would be a net gain for ExpressionEngine-powered sites as a whole. I am more than happy to add this book to my recommended reading list for all ExpressionEngine users.

—Derek Jones, CEO, EllisLab

1 Introduction

A funny thing happened on the way to 20,000 words, which was my initial target when sitting down to write a guide about ExpressionEngine. At about 16,000 words, ExpressionEngine 2.9 was released, and it was a release that seemed as stable and exciting as any I'd seen in the past few years.

With 2.9, I thought it was time to take another look at how we were doing our ExpressionEngine builds. We actually hadn't started a *new* ExpressionEngine project in a long time. Most of the client work we had going on was maintenance on older EE sites.

I set up a development sandbox to play around. ExpressionEngine already had some capability for layouts, but we hadn't really used them yet. We had always customized the control panel login for our clients, but never with the view overrides that were now available. I realized we had to seriously change how we were doing things.

Within a couple weeks, we had put together four ExpressionEngine sites with our new structure and process. My experimentation resulted in changing things about our setup and structure, and negated a lot of the now 18,000 words I had already written. As great as I felt our new setup was, I knew it meant a lot of the guide would need to be rewritten.

So, I started rewriting. A few weeks later, we updated our default setup yet again with new refinements, and I realized I would have to rewrite some of what I had already rewritten. This was getting crazy, but I felt we had sites with a higher level of execution and simplicity that were better than anything we'd ever done before. It just makes it hard to pin these ideas down in a "book" if they're always changing.

As I will mention in a section on Directory Structure: "If you're not constantly learning and revising how you do things, you might want to get in the habit of challenging yourself, lest you stagnate and stop growing." You can't accuse me of not eating my own dog food, even if it sets back the release date of something on which I'd been hard at work.

After a big rewrite, and an additional re-rewrite, I ended up well beyond 20,000 words and I think I finally got out everything that I wanted to share, and that it all makes sense. I'm confident this version of the guide reflects my company's current thoughts on setting up sites with ExpressionEngine 2.9+, and versioning them with Git.

Thanks for your interest. I hope that no matter what level ExpressionEngine developer you are, you take something away from this guide that makes you better, even if it's only the thought that "gee, I know more than Masuga does."

—Ryan Masuga, August 2015

Goals of this Guide

I began in web development around 2000. Since then, I've used an employer's bespoke solution, ExpressionEngine and some other systems in between. I've been in and around them for many years. I'm most familiar with ExpressionEngine, so my main goal is to share with you how my company has used it successfully.

For simplicity's sake I'll mostly use the pronouns "I" and "me" throughout (e.g., when referring to "what I know") but at the same time also refer to "we" and "us" as a company at Masuga Design, as we've learned things and refined our processes collectively. I certainly can't take credit for everything.

Share What I (We) Know

I've been in the ExpressionEngine world for quite a while. Many things that seem obvious to me are *not* obvious to some people, so I thought it was time to put down our method of working with ExpressionEngine sites, as it may save a developer like yourself many headaches in the future.

Let's start with something basic: I want to make sure that you know ExpressionEngine is a camel-cased word—it is NOT two words. I still see many so-called "expert" EE freelancers and shops claiming they know "Expression Engine." It is so commonly misspelled that some EE developers even use EE with a space as part of their SEO strategy to capture those searches that contain a space (I mean, c'mon now). ExpressionEngine is one word, camel cased (just like WordPress!). If you're really expert, you'll internalize this fact. It's the details.

We have war stories. About taking over other developers' work. About revisiting our own work years (or even mere months) later. These battles have led us to ask smarter questions and do better development, and have led to some of the methods we use today.

Give You Enough To Go On

I'm not the most advanced Git user in the world. But that's OK, because *I don't need to be* to effectively use ExpressionEngine in a Git workflow. That means you don't have to be an expert, either. You just need enough to go on. My goal is to shed light on our relatively simple Git workflow. We don't use submodules and don't have more than a few branches: master, dev, and topic branches. There are plenty of people out there who know more about Git than I do, and I'm in no way claiming to be an expert.

One danger of ExpressionEngine is that there are 99 ways to do any single thing, and people have found every one of those 99 ways—we've seen it. I want to give you a decent base on which to build your ExpressionEngine sites.

After reading this guide, you should be able to competently set up an ExpressionEngine site in a way that is easy to maintain and version control it with Git.

Achieving this goal makes you a more valuable developer because the work you do for clients will be secure and redundantly backed up, and in the event you're contracted to work with other developers who are using a Git workflow, you will be able to work with them with minimal training.

Help You Help Yourself

Having more skills is never a bad thing. Whatever you learn and can apply to your career will ultimately help you. I think that if you are using ExpressionEngine, learning to structure a site without over-complicating it and then versioning it with Git will make you a much more valuable developer than those who don't have these skills or knowledge.

Although ExpressionEngine is a fraction of the overall CMS market, there are a ton of potential clients out there who have a site on ExpressionEngine or who are looking to have a site built with it. *Even a fraction of the market is a nearly limitless amount of work.* After implementing the ideas in this guide, you will be capable of making and supporting a better product, and those potential clients and all that work can easily be yours.

What This Guide Is Not

We're not talking about templates here!

This guide is intended to instruct on how we set up our site structure and version our sites. It won't go into detail on how to build an ExpressionEngine site. There is no specific site we're setting up. There are no specific template examples, deep meditations on whether or not to use Stash or Structure, or in-depth template tag examples. There are other resources out there for that aspect of ExpressionEngine development, some of which I reference under Resources at the end of the guide.

This guide should convey our experience and suggestions in *structuring a site and versioning it*. The templates are up to you at that point. If you want to nest embeds four deep and drag a client's site to a crawl, that is your choice, but at least you'll be organized and have those templates versioned!

Comparing Apples To Oranges

This guide is also not meant to compare ExpressionEngine to other content management systems. It's exclusively about ExpressionEngine and how we set up and version control our projects with it. If you're looking for information about why you'd use ExpressionEngine over WordPress or vice-versa, Google is your friend. If you're already using ExpressionEngine: let's move on!

Assumptions

> "Is it going to be OS-agnostic—or at least cover Windows and Linux tools & solutions in addition to Mac?"
> *—@sandwich_hlp via Twitter, Feb 12, 2015*[1]

One can't account for everyone or every situation when writing on a subject like this. Different operating systems, various software applications, a plethora of setups, and wildly differing ideologies on how to do anything all come into play, so I need to move forward and make a few assumptions. If I don't happen to mention your favorite Git app, I'm sure you can take the seed of what I'm talking about and apply it to your situation. Besides, this

book is based on how *we* work and what has worked for us, and you can take all of it or none of it!

Your Skills

I'll assume that you're a relatively smart cookie, which is easy to do, because you're reading this guide. You should already have an understanding of how to work with ExpressionEngine sites. Ideally you've already built and launched sites before, but you didn't yet do so with version control. Or maybe you have used version control, but want to confirm you're doing things "correctly." Or you've used version control and want to reassure yourself that you're using it better than we are. All of those scenarios are perfectly fine. What I'm not going to do in this guide is teach ExpressionEngine 101.

You might have opened Terminal before, and you know how to install software. Ideally, when working with an ExpressionEngine site you're also saving your templates as files, which can then be versioned. To that point, if you're working on your files stored in the DB, right in the browser, you might need to seek professional help. How do you effectively and reliably do a search and replace across multiple files when editing them via the control panel? Yes, you can navigate to Tools > Data > Search and Replace and do that there, but how safe do you feel using that when the following warnings are displayed at the bottom of the Search and Replace page:

> BE CAREFUL! THIS ACTION CANNOT BE UNDONE
>
> Depending on the syntax used, this function can produce undesired results. Consult the manual and backup your database.
>
> If you are replacing within templates, synchronize with the database first, or permanent data loss can occur!

I know there are ExpressionEngine development shops that work with the templates as files while in development, but revert to storing the templates in the database on a production site in order to reduce processing overhead. We've never worked on a site where those fractions of a nanosecond were so critical, so I can't speak to that practice. We'll assume your templates are being worked on as files, and deployed as files.

Hardware

We're an all-Mac shop. If you develop with ExpressionEngine on a Windows machine, more power to you. Most of this guide should still apply to you without issue. (We only fire up a Windows machine in this office when we need to do a browser test or two. Fortunately for us, that is becoming more and more rare, but at least testing in Internet Explorer isn't the total nightmare it was a few years ago.) Any server paths you come across in this guide will have a UNIX look to them. If you're developing ExpressionEngine sites on a Windows machine, use your imagination in these cases.

Software

Well, you sort of have to use ExpressionEngine or you're quickly going to be up a creek with this guide. The rest of the applications you use are up to you, but below I explain what we use and why.

ExpressionEngine[2]**.** Version 2.9 was released about halfway through the writing of this guide, and we've done several sites with it already. Although it's the version with which I have the least familiarity, that is the version I'll be referring to when I outline directory structures. By the time you read this, ExpressionEngine 2.10 will have been out a while, and we know that version 3.0 is expected to be released later this year.

MAMP Pro[3]**.** If you're not too technically minded and just want to get down to business putting sites together locally, MAMP Pro makes this a piece of cake. We've been using MAMP Pro v2.x for years, but as of this writing MAMP Pro v3 has been released, and looks to be a major upgrade, with the major bonus of being able to run each local site on a different version of PHP to better keep it in line with what the site may be using on a production server. Either way, this app makes getting local sites up and running totally painless.

Tower[4]**.** This is the premier Git client for Mac. Easy to use, and easy to see your commits and branches. There are other GUI Git apps such as Atlassian's Sourcetree[5] that look pretty solid, but we've used Tower since it was in beta and haven't really found any reason to switch. Any examples in this book will reference Tower, but if you're using a similar app, things should still make sense.

Sublime Text 2[6]. This is a very robust text editor with numerous themes. There are more than one ExpressionEngine code coloring bundles out there for this editor.

I have been using an ExpressionEngine package from Etsur [7], which was a breeze to install with Package Control[8], the package manager for Sublime Text.

There are a couple more out there that might be more complete. One is the ExpressionEngine bundle from Vector Media[9]. Another one from Github user fcgrx appears to be updated more recently[10], and is possibly more complete. (Upon further inspection it appears to be a fork of a fork of the Vector Media version, so it's possible that the Vector Media version is your best bet here.)

Sequel Pro [11]**.** You're going to need local copies of your databases. Don't use a remote DB when developing locally (I talk more about this under Syncing the Database in the Git Chapter). Unless you're just fine with phpMyAdmin, you may want to use an app that provides a friendlier interface for working with your MySQL databases. Sequel Pro is that app. I used to use Navicat, but I've found that Sequel Pro feels more nimble and it's very easy to use. If you're using MAMP Pro version 3 or higher, Sequel Pro is bundled with it, as is something called MySQLWorkbench (which I've never used). Whether you use the MAMP Pro 3.x bundled version or the standalone version of Sequel Pro, you'll find that it's great for helping you manage your MySQL databases.

Terminal. Sometimes you have to get dirty. If you're making a commit that affects more than a few hundred files, the normally wonderful Tower app might crash on you, or hang indefinitely [12] (at least with the initial version). Big initial commits are well served by going into the command-line, but we try to keep our trips into Terminal to a minimum.

History

Let me give you a little background into who I am, what my company does, and what my experience is as an ExpressionEngine developer.

Fig. 1.1 March 2008. Full-blown ExpressionEngine dev by then, in my first office outside the house. Goal of owning a bubble-hockey machine achieved!

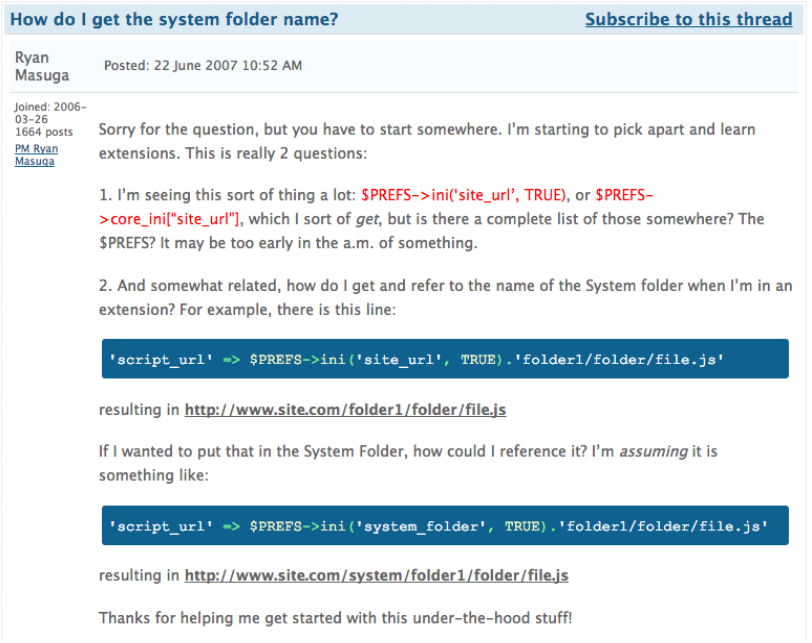
I had been operating Masuga Design as a freelance gig on the side since 2003, but in 2005 I quit my day job and worked out of the spare bedroom, and in 2006 it was finally set up as an LLC and ready to go for full-time development and design **(Fig. 1.1)**. From the beginning, client work has been done almost exclusively in ExpressionEngine.

The story I tell is of the time I landed my first ExpressionEngine site. I think the version at the time was 1.5 something, and I know the control panel was a lovely shade of purple. I was on the phone with the potential client, who was asking me question after question: "Can we do this? Will we be able to do that?" I had the ExpressionEngine Features page open on my screen and was reading it as they asked their questions "Yes, sure…OK, that's possible, yes…" and so on. I got the job.

At the time it was the most money I'd ever earned for a project—a respectable four-figures. This was going to be a huge deal for me. But now it was time to deliver. I'd never built a site on ExpressionEngine. I'd only read the features page. Now it was time to dive in and build the site.

With ExpressionEngine, I was able to execute everything I told the client I could do, so I decided to stick with the CMS. If I could land one good job using it, why not more?

By the next year, I was working on an ExpressionEngine site for a Fortune 1000 company—*for over 20 times the amount of money* I made on my first ExpressionEngine site (I'm still *very* grateful for that project after all these years). That experience taught me that **ExpressionEngine makes it possible for a solo developer to turn out an incredible product for nearly any size client.**



How do I get the system folder name? Subscribe to this thread

Ryan Masuga — Posted: 22 June 2007 10:52 AM

Joined: 2006-03-26
1664 posts
PM Ryan Masuga

Sorry for the question, but you have to start somewhere. I'm starting to pick apart and learn extensions. This is really 2 questions:

1. I'm seeing this sort of thing a lot: $PREFS->ini('site_url', TRUE), or $PREFS->core_ini["site_url"], which I sort of *get*, but is there a complete list of those somewhere? The $PREFS? It may be too early in the a.m. of something.

2. And somewhat related, how do I get and refer to the name of the System folder when I'm in an extension? For example, there is this line:

```
'script_url' => $PREFS->ini('site_url', TRUE).'folder1/folder/file.js'
```

resulting in http://www.site.com/folder1/folder/file.js

If I wanted to put that in the System Folder, how could I reference it? I'm *assuming* it is something like:

```
'script_url' => $PREFS->ini('system_folder', TRUE).'folder1/folder/file.js'
```

resulting in http://www.site.com/system/folder1/folder/file.js

Thanks for helping me get started with this under-the-hood stuff!

Fig. 1.2 The oldest ExpressionEngine forum post of mine I could find, June 2007 (there may have been older but EllisLab "pruned" their forums of 1000's of posts around 2010). And I just found a typo in it…an eight-year old typo. I hate that.

I was frequently in the EllisLab forums helping people as far back as early 2006 **(Fig. 1.2)**, and Masuga Design has been in the EE Pro Network since April 2007. Because we develop add-ons for ExpressionEngine, Masuga Design is in the EllisLab dev preview, which gives us early access to new releases.

I hired my first full-time employee in 2009 and we now have a small team. We have had the good fortune to work on ExpressionEngine projects for companies like Adobe, Google, Toys"R"Us, A&E, Univision, FOX Networks, Alaska.org, Cost Plus World Market, Image Comics, Scripps Networks, and Pep Boys.

Because our skills are broader now with more people, we experiment with other frameworks too, but we still develop nearly all of our client sites with ExpressionEngine. Besides, we've been very successful at developing with ExpressionEngine, from the time I was a solo developer up until we became a small team, so why not stick with what works?

Making devot:ee

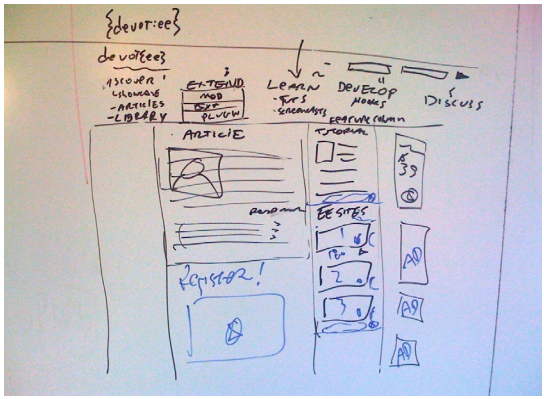


Fig. 1.3 The initial whiteboard sketch for devot-ee.com.

We're also responsible for maintaining devot-ee.com[13], the #1 unofficial resource for ExpressionEngine add-ons **(Fig. 1.3)**. Many of the footnotes throughout this guide link to information found on devot:ee because, if I do say so myself, it's a very important source of information for ExpressionEngine!

By 2008, I had spent an incredible amount of time in the ExpressionEngine forums, and I had also meticulously cataloged almost every ExpressionEngine add-on I could find. (I touched on this in a blog post back in September 2009: *Organizing an ExpressionEngine Add-On Collection*[14].)

After spending all this time in and around EE, I realized something was missing. ExpressionEngine had a great number of add-ons available, but they were scattered everywhere. There were add-ons being released in the ExpressionEngine Forums, some were only to be found on a developer's own site, and a few more advanced add-on developers were starting to keep their add-ons on Github or Bitbucket. This is still the case, but there was no *central* place to look for them, or to compare similar add-ons to see which might better fit your needs. It's almost impossible for me to remember the time when devot:ee didn't exist to serve this purpose.

Also, if you were going to venture into writing your own add-ons, there were practically no resources available at all—it was The School of Hard Knocks. If you wanted to figure out how to do something, one of your best options was to rip open other people's add-ons to see what hooks they were using and how things were done (for better or for worse). That's why we list hooks used for each add-on on devot:ee (at least for Extensions), so that you can find other add-ons utilizing a particular hook you need to use.

If you plan on doing something cool that likely needs to use the `core_template_route`[15] hook, have a look at what other add-ons are using it (currently five) [16] so that you can see how it's been used. The goal there was to help speed up your ExpressionEngine development and reduce your learning curve.

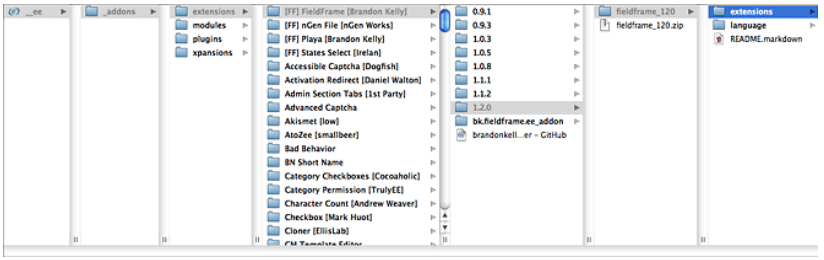


Fig. 1.4 My original ExpressionEngine add-on collection that was the base for devot-ee.com. I turned those directories into channel entries.

The ExpressionEngine community needed a resource like devot:ee, so I took the ExpressionEngine add-ons cataloged on my machine **(Fig. 1.4)** and used that collection as the foundation of devot:ee's add-on library. Because I was confident it would work, I invested about $10,000 of my own money

(practically everything I had in my business account) to pay someone to help design the site, and I built the site on ExpressionEngine (that's obvious, right?). In May 2009 we added the ability for developers to sell their ExpressionEngine add-ons on the site, and we've since sold many millions of dollars worth of add-ons.

So that's a summary of my experience in the ExpressionEngine world, so you know where I'm coming from!

Why Use Version Control?

> "Mistakes were made." —*Absolutely everyone.*

Before we dive into the specifics of version control at all, the first question we should answer is: why bother? Isn't it just going to add a lot of work for you? I'm sure you already have a good workflow editing your clients' templates directly on the live site via FTP. What's the big deal, right?

In 2010, I spoke at the ExpressionEngine Conference held in Leiden, The Netherlands about using Git with ExpressionEngine. You can still see the slides and related links at http://gititon.masugadesign.com/. I raised a number of points that are just as true now as they were when I first delivered the talk. You want to use version control in general because:

- *Not doing so is completely irresponsible and a disservice to your clients*
- Double safety net: commits are made to your local repo before pushing to a remote repo
- Complete history
- Work alongside other developers
- Roll back!
- Work at your pace
- No need for complex `{if member_group}` conditionals or extra test templates

- Make mistakes with impunity. Never again will you install an add-on and worry when it makes your live site white-screen, because you'll never do that on a live site before having done it locally first.

I also wrote an article on devot:ee that (infamously) never had a part two. Part of the reasoning behind writing this entire guide is to right that wrong and share all the things I've learned since. For reference, that article is located here: http://dvt.ee/eegit1. Most of the software in that article isn't as relevant these days, but I'll get into specific software applications later.

One of the main points of my talk was that not using version control is completely irresponsible. Nowadays, a client can use any number of web firms or freelancers to work on their project. Why should they pick you? What competitive advantages do you have? Why would you not want to appear more professional than the next development shop by assuring your client that their site is backed up and worked on in an orderly, professional, systematic way?

By keeping your client sites in version control, their site is redundantly backed up. You have a copy locally, there is a copy at a repository (Github, Bitbucket, Beanstalk, et al.), and you have a copy on the server where the site lives. You might even have another copy on a dev server.

And it's really not that hard to have a basic workflow going with Git and ExpressionEngine. These days you can do it and stay almost entirely out of the command line and while staying in friendly-looking GUI apps like Tower. I'll cover applications later in the guide.

Keep in mind you don't have to use Git. You could use Subversion[17] or Mercurial[18] or some other even geekier system that I'm probably not aware of. I'm writing about Git in this guide because it is what we know and use every day, but I certainly won't fault you for using something else if it suits you. **The main point is: you really should be versioning your work, period.**

2 Practical Development

Keeping things simple and staying practical can be deceptively hard. This chapter doesn't concern itself exclusively with ExpressionEngine. There are a number of points in this section that I think apply to web development in general that I've learned after years of experience…and by making numerous mistakes.

So, before we get into the nitty-gritty of setting up an ExpressionEngine site and versioning stuff, let's talk about more general items that are no less important.

Common Sense

> "Common sense is not so common."
> —*Voltaire, A Pocket Philosophical Dictionary*

These are things that I've found make a lot of sense to pay attention to when working on ExpressionEngine projects. Some seem pretty obvious, but you'd be surprised how lackadaisical people are. I'm not about to say that anyone who develops a site in a way that is not the way we do it is *wrong*. It's just that sometimes I have to wonder what people are thinking.

I say that because very often my company is asked to take over development of an ExpressionEngine site that has already been built and the site owner is either looking to switch developers or have us do a review of the site to see how we could optimize it. Looking at other developers' methods for constructing ExpressionEngine sites is very enlightening, to say the least. Sometimes we've come across some interesting tricks ("Huh, why didn't I think of that?") but many times (and more often than not) our jaw hits the floor marveling at how complicated people make things.

One thing about ExpressionEngine that is its strength as well as its weakness: *there are so many ways to do any one thing*. Rob Sanchez, a long-time ExpressionEngine developer, said this about EE in the (now defunct) devot:ee forums:

> "What makes EE so great is also what makes supporting it so difficult. It's a non-themable, modular CMS, and every site is a bespoke combination of custom templates and add-ons.
>
> *You […] are building a one-of-a-kind Rube Goldberg machine…"*

Keep in mind that I'm in no way saying I'm a perfect EE developer. Let's be clear: *I am not a perfect ExpressionEngine developer.* I've been guilty of every single one of the things I talk about here: not commenting my work, overthinking problems, and relying too much on caching.

So here are some things that I feel make a lot of sense to pay attention to (and again, some of these things apply to web development in general, whether you're using ExpressionEngine or not):

Comment Your Code. Comment Your Freaking Code.

> "If you can't explain it to a six-year-old, you don't understand it yourself."
> —*Albert Einstein*

This sounds almost didactic, but I can't emphasize enough: you really should comment your code. *Comment your freaking code.* I mean this as far as the templates go, but the advice holds true (if not more so) for any custom add-ons you build. I guarantee that at some point you will go back into some dark area of a site you've put together and wonder how that area works, and what the hell you were thinking when you built it. I know I've done it: got too clever by half in building a rickety house of cards and coming back six months later when the client needs a tweak and feeling like a damn fool that I didn't outline the reasoning behind a ridiculously lengthy Switchee and Stash contraption.

On a basic level, we comment every template, and I'll explain how that works. At the top of every template we include a comment block that has a lot of information about the template. By having this block at the top we can see what a template is about at a glance, such as: Is PHP enabled? Why? Does this template use a template route? What URLs is this template responsible for? Is this template embedded? If so, where? Is there any other important info we need to know?

I can't tell you how much having these common comments at the top of each template has saved time over the years, never mind any comments throughout the code. Here's the most basic sample of what you'd see at the top of a template:

```
{!--
Site: winterfell.com
Page: site/index
============================================================ --}
```

At a glance, a developer knows what the template group and template are, and knows there isn't anything else going on.

Sometimes (generally on sites we did not inherit, or sites where we don't need to remain anonymous as the development partner) we include a default "credit" line so everyone knows who's responsible:

```
{!--
Site: winterfell.com
Page: site/index
============================================================
Built by Masuga Design | masugadesign.com
============================================================ --}
```

A template that is embedded elsewhere will get a note:

```
{!--
Site: winterfell.com
Page: includes/_innerloop
============================================================
Built by Masuga Design | masugadesign.com
============================================================
Embedded in one template: reviews/index
============================================================ --}
```

Here's a more elaborate example with a little of everything you might see on a complex template:

```
{!--
Site: winterfell.com
Page: resources/logos
============================================================
Built by Masuga Design | masugadesign.com
============================================================
PHP: output (we need a PHP var in the pagination case)
Uses a route (see config/routes.php)
 - Access to this template is restricted to Super Admins in the CP!
 - Embedded by the main resources template

URLs:
/resources/logos
/resources/logos/[P0]
/resources/logos/[search]&query=[foo]
/resources/logos/[entry_id]

@TODO: why is that extra loop on line 143? - ryan Jul 13 2015
============================================================ --}
```

At a glance, we can see we have enabled PHP (and why we've done so) *and* that there are template access restrictions[19] set on this template in the control panel (which might be something that trips you up when debugging an issue where the client wonders why a certain someone cannot see this section, which otherwise looks like a normal template). This template is also embedded by another template, and we can see exactly what types of URLs it is expected to handle (with variable items represented in square brackets).

It takes practically no time at all to add these comments to the top of a template (and then update them), and it can help prevent headaches later.

We've taken over projects that are absolutely abysmal at giving any indication what is happening. For example, we recently took over a large, highly-customized ExpressionEngine build. This site has tons of custom functionality made possible by numerous custom add-ons. Only a firm that knows what they're doing could have made this site, but...*there are next to no comments, either in the templates or the custom add-ons.* There are tons of custom add-ons, and none of the methods in the add-ons give you any clue as to why they're there or what they do. Never mind the fact that this site has tags for one of the custom add-ons on nearly *every* template. "This add-on has a thousand undocumented parameters and makes an appearance on nearly every template. Pshaw…it's not that big of a deal. Piece of cake to figure out later. Why bother documenting it?" (Cue me passing out).

The end client (who is now *our* client!) needed us to make what, on the surface, appeared to be a five-minute update of one of the sidebar elements. Thirty minutes later, *three* of us were standing here, scratching our heads, quizzically looking at one of the templates like the apes staring at the monolith in *2001: A Space Odyssey*. The sidebar element was a dynamically-named embed *inside* another embed *inside* a Low Variable that was *inside* an MX Jumper (an add-on which was also hacked so that MX Jumper variables could contain other MX Jumper variables)…you get the idea. You still with me? It's every bit as confusing as you think it is.

We had no clue where the sidebar element actually existed. And when we did find it, we couldn't easily modify it because its output was being rendered by a custom function rather than native ExpressionEngine tags. There was no way to tell what parameters or template tags were available for use.

It was as if this site was a completely bespoke solution built on top of ExpressionEngine, rather than a site built *with* ExpressionEngine…and none of it was commented to explain how it worked. I'm willing to bet the original developers would have had just as tough of a time updating the sidebar as we did!

That's sort of an exaggerated (but very real) example. Keep in mind I'm not saying that the original devs did anything *wrong*, because we don't know everything that went into the decision to build it this way (time constraints, client demands, etc.).

Regardless, on any level: commenting your code saves time for developers and money for the client. Yes, writing comments about what you just programmed can be tedious, but it will be worth it ten times over when you have to get back into the code. Forgetting what you were up to only gets worse the longer you stay in the web game. You'll have built more sites, and more developers may have been involved in working on your projects.

Document Core Hacks...and Your Add-on Hacks

If you've worked with ExpressionEngine for any amount of time and built a site of any complexity, you know that core hacks are sometimes inevitable. They happen just often enough for us on highly custom builds that the idea of having to do it doesn't scare us any longer, but we realize that any core hack we add will make the update process that much more error-prone, and we all know the ExpressionEngine update process doesn't need any more potential cause for error (more about updating ExpressionEngine later in this guide).

If you're going to hack core, make sure you document your hacks. This is critical! It's easy to add the hack, make sure it works as you want, and then think, "Oh, I'm sure I'll remember that little tweak later." *Do it immediately.* We document our hacks in two separate files, one for core hacks and one for add-on hacks. They are similarly named (_ee_hacks_core.txt and _ee_hacks_addons.txt), and located in our `/config` directory, which I'll talk more about in Chapter 5.

Documenting add-on hacks is every bit as important as documenting the core hacks. We recently updated an inherited site and it turns out that the popular file-upload module add-on that was installed was hacked. But who knew? We didn't know until the client emailed us asking why the heck the file uploads for particular files weren't working (something we didn't catch in our local testing before deploying the update to the live site). It was only then that I realized the previous developer had added a couple of custom tweaks, and that we would have to re-add those same lines to the new version of the add-on. You can bet I also updated the _ee_hacks_addons.txt file in the same commit, too!

What's that saying? "Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.[20]" I think this is doubly true for undocumented hacks—just replace "Always code" in that quote with "Always document your hacks."

Don't Focus on the Wrong Things

At the 2010 ExpressionEngine Conference in Leiden, Netherlands, there was a workshop day where small teams were given a site to build in a given amount of time (I think it was a few hours). The idea was to see how a particular problem can be solved in various ways with ExpressionEngine. If my memory serves, I think the problem was something along the lines of building a site where users had to reserve a spot at a drive-in movie theater. I was one of a few "judges" wandering among the teams and trying to help out and offer suggestions where I could.

It was interesting to see how some teams chose to spend their limited amount of time. One team had installed an add-on that helps automatically generate htaccess files to remove index.php from the URL. Nothing wrong with that in and of itself, but the obsession with getting it to work correctly meant they hadn't even begun to think about how they were going to solve the larger problems of the site exercise.

I made the rounds of the teams a few times and came back to find them still working on getting the add-on to work correctly—and URLs weren't even a requirement of the project.

Another team spent an inordinate amount of time on the visual design and CSS, rather than the functionality that was required. No team ended up solving the problem at hand very well. Perhaps those teams should have been thinking about how the data should be stored. Are the available theater drive-in spots in a channel? How are reservations done? Should there be a "reservations" channel that relates to theater spots? There wasn't a lot of that sort of planning going on.

I had to wonder if the approach they took to this exercise is the same approach they take to real projects where a client is paying them to help solve their business problems. When you focus on the wrong things, you're going to end up wasting time on things that aren't important to the client, and you're probably going to go over whatever budget you set, which will start to grate on you as the project drags on. I know this from experience.

Try to realize what deserves your real focus and concentrate on that.

Don't Overthink the Problem

> "A child of five could understand this. Send someone to fetch a child of five."
> —*Groucho Marx*

I've been guilty of overthinking the problem in the past, and judging by the projects developed by others that we've looked at over the past few years (and some of them are very well-known), this problem seems widespread. It affects everyone (and again, I'm guilty!). Overthinking the problem happens when it becomes more important to use the latest whiz-bang add-on or to nest something five levels deep in order to make your templates more DRY *just because you can* rather than thinking about the specific needs of the project or the sanity of the content editors who will be taking care of things.

Sometimes, overthinking is a result of not knowing any better. I was recently approached by a new potential client to help fix some issues their site was having. This wasn't a big site—maybe 10-15 pages. When I got into the control panel to look around, I saw that there were equally as many channels. It was readily apparent that the previous developer hadn't read the ExpressionEngine manual. They had created *a new channel* for each "static" page entry, as opposed to making a pages channel and keeping the pages as entries within that single channel. So now we have a site full of 1-entry channels. This is a simple site, overcomplicated. Sure it *worked*, but that is just not how it's done.

Think about it: any line you add to the codebase has to be maintained, either by you or some other poor soul. Any channel you add, any new custom field, any new category group. You always have to ask yourself: in light of the client's needs, am I building this solution and solving this problem in the best and simplest way possible?

There are situations where a solution could be one of many things: a category group, a custom field, a channel, a global variable, or a custom add-on. I've had at least one occasion where the easy way out was a category group, but I wished I'd taken the time to realize what we needed was a channel (maybe this is an example of "underthinking" the problem). Take the time to weigh options carefully so you can future-proof your decisions as best you can.

I've spent hours cooking up a clever solution so that a client could update a section of a site, adding overhead and queries to every template load. You know what happened: they never once updated this section. I didn't think it through. I went ahead and built the clever solution rather than what was needed, and went over budget to do so.

> "Seen a lot of #eecms builds/setups from other devs lately, seems like everyone these days is afraid of just. keeping. it. simple."
> —*@erwinheiser*[21]

I recently spent hours refactoring a site that had too many queries going on. One of the things adding overhead to every page load was the footer. The footer had a ton of links, and the whole thing was dynamic and controlled from a module page in the control panel, and heavy enough that the developer even added cache tags around the output (because even the original dev must have known that this was "heavy"). While going through and optimizing the site, I called the client and asked a simple question:

Me: "Do you ever update the links in the footer, from the area where you can do that in the control panel?"

Client: "Never."

Well, that was easy. I made that area entirely static and haven't thought twice about it since. A while later we realized the logic behind building the main navigation links was so heavy that, if not cached, would add *five seconds* to every page load. Again, talking with the client:

Me: "Hi, have you ever updated the links in the main nav from the control panel? Do those change that often?"

Client: "Never. No."

The main navigation is now static as well. The site is getting zippier (and stays just as easy to maintain, if not easier) by the day.

It's a balance. Regarding updates to pieces of content, would it be easier (or just as easy) for the client to contact you to update it? If they update something so infrequently, they might contact you anyway to remind them how they're supposed to update the content…and then you often end up doing it for them anyway. Some areas are better left alone. *Not all content needs to be managed.*

Regarding how you construct your templates, if it would be easier to make three separate templates for three different sections that have slightly different content and logic that somewhat repeat themselves than it would to make one template that is a 3000-line beast with multitudes of advanced conditionals, by all means…*make three templates.* It just might be possible to be too DRY.

I've just seen a complete overuse of the Stash add-on recently as well. The developers created a complicated Stash list of items from a channel that would only ever show on one template. The list is created and stored at the top, and then is output at the bottom of the template. We looked at it for a few minutes, and scratched our heads wondering why in the heck someone would do this. We then did the next logical thing: pulled the channel entries loop out and removed all Stash tags. Voilà. A much simpler template, and many fewer lines of useless cleverness.

Don't overthink the problem or create solutions where none are needed.

Beware of Caching as a Band-Aid

> "Every time you use caching to hide your shitty performance a kitty cat gets water boarded with Tabasco."
> *—Eric Lamb (@mithra62)*[22]

There are a few great ways to add caching to your ExpressionEngine templates (either with add-ons like CE Cache[23] or Static Page Caching[24], or with solutions like Varnish[25]), but caching (as we've seen it) is generally used in a couple different ways: as a band-aid or as icing on the cake.

Caching as a band-aid is not a smart way to go. When I say "as a band-aid," I mean it is being used to disguise problems. Sure, it can "speed up" your site, but when used as a band-aid, it likely only hides poor coding and less than optimal template setup. "Wow, this template is a *dog*! I know, I'll just cache it. Never mind that I used twenty channel:entries loops when I could have used one because I never did RTFM on how to use ExpressionEngine in the first place!"

Sometimes caching used as a band-aid causes more headaches than it solves, because now you have to deal with cache-breaking and setting up rules for that, or working your cache around any logged in/logged out areas.

Your aim is to build in such a way that your templates perform well on their own. Adding caching (should you decide you need to) can then be used as the icing on the cake. A smaller site with modest traffic might not ever need to go there, whereas a high-traffic site with global reach might definitely benefit by the extra boost.

At the time of this writing, we're working with a newer client who has some pages that are performing *terribly*. We were basically hired to take over their development so we could solve their performance problems. When we signed on, the client was at their wit's end with many of their pages taking 13-17 seconds to load. That is *insanely* bad. At that point, they were wondering if they should even be using ExpressionEngine.

They were talking about using Varnish or other advanced caching methods, but I told them to hang on until I could really look at their templates and see what was going on in there. So I went in and looked…*and I almost didn't make it out alive.*

I saw all sorts of basic mistakes that could easily be fixed, including a couple killers: too many advanced conditionals (this is a pre-EE 2.9 site where that is an issue you still need to be aware of), and far, far, far too many embeds **(Fig 2.1)**.

Here's an example of what we were up against[26] and why caching needed to be added to mask poor coding: the developer needed to feed a dynamic date to a entries in a query module loop[27], which they were doing with PHP, so PHP was enabled for that template on *input*. However, they also needed to use PHP later in the template on *output*. In ExpressionEngine you can't have PHP running on both input and output on the same template[28]. So, they needed to use an embed to get what they were after. The real issue was that they were doing this embed for *every* entry in a loop. And we found that the embedded template had two embeds in it!

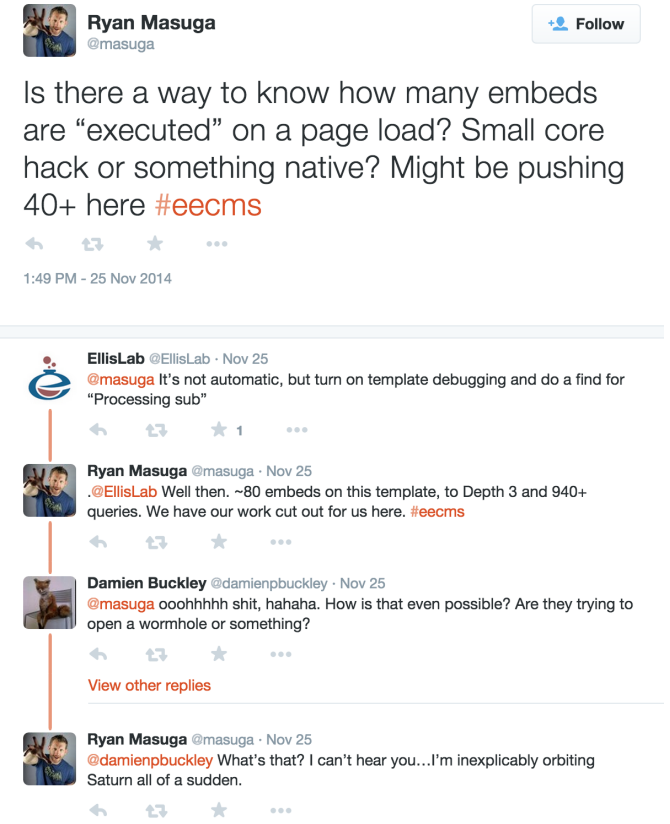


Fig. 2.1 Eighty embeds on a single template load. That ripped a hole in the space-time continuum. I don't think there's ever a case where that is necessary.

So the query module returned 25 results and each result embeds a template that embeds two more templates. This meant that when this template loaded, there were around 80 embeds to a depth of three and over 940 queries, essentially ripping a hole in the space-time continuum. This can alternatively be referred to as "How Not to do ExpressionEngine."

Keep in mind that embeds by themselves are not evil, but nesting embeds in loops without understanding the impact of what that's doing is a problem.

I made the simple move of converting the initial PHP date variable to a global variable (thereby making it so the template no longer needed PHP enabled, which is better for security—see Chapter 3 for more info) and eliminated one of the unnecessary nested embeds which subsequently reduced this template down to 3-4 total embeds. So, one global variable reduced a template's workload by about 76 embeds. After that, we turned the "band-aid" caching off because it really wasn't needed.

The previous developer hadn't even taken care of (or understood) the basics, like sensible template construction. It really is a common rookie mistake to overuse embeds. Everyone does it (I sure did it). My goal here is to get you to realize that this can be dangerous and to stop you from doing that sooner than you would on your own.

I told the client that we would implement caching in some key places *for the short term* (as a band-aid) while we worked on the true underlying problems. The pages are loading OK in most places while we fix the real issues, but woe unto that poor person who hits one of these uncached templates every hour or so and is faced with a 13+ second load time.

We continue to optimize the templates and queries on this project. I'm trying to get the site to a point where it operates very well with no caching at all. Then, if it looks like the pages would *also* benefit from caching, we could look at adding some caching, but this time as icing on the cake.

Dynamic="no" and "Locking Down" Your Templates

We're fans of "locking down" our templates, and leaving nothing to chance. ExpressionEngine makes it easy to create dynamic templates (so long as you RTFM[29]) without having to *do* much (if you have a template group in segment_1 and a template in segment_2 and *something* in segment_3 and your template has a channel entries loop on it, we can make some magic happen!) but we prefer our dynamic "off."

There are two things we like to do on a template that help control how it works: control for any segments that should never be there right at the top of the file, and use clear channel entries loops with dynamic="no."

As for controlling segments that shouldn't be there, you probably know what URLs a given template is expected to handle. For example, a blog or news template might handle these:

```
/news
  /news/P10 (example of a paginated index list)
  /news/entry/dark-wings-dark-words
  /news/category/the-narrow-sea
```

If our news/index template is running the show for all of these URLs (say, with Switchee), we never have more than three segments. So, we don't need someone (or a bot) trying to hit something with four or more segments like:

```
/news/entry/dark-wings-dark-words/thisdumbsiteisstupid/haha-dummy/
```

and getting an OK 200 status code. So first we'll put some rules at the top of the template to keep anyone from putting more than three segments in the URL for this template. There are different ways you can go about it, but I like the redirects to specific pages better than 404's. Here are two examples of how you could prevent the above from happening:

1) **Use a redirect.** At the top of your template, put:

```
{if segment_4 != ''}{redirect="/news/entry/{segment_3}"}{/if}
```

What you put in a case like this may be different depending on what template you're on. In this example, we're taking them back to the "known" news item. You might want to take them back to the main page in your news section instead, so you could use:

```
{if segment_4 != ‘’}{redirect=“news”}{/if}
```

In either case, someone can monkey around with segments four or higher and now absolutely nothing will happen.

2) **Send them to a 404 page:** (this assumes you have one designated[30] under Design > Templates > Global Preferences)

```
{if segment_4 != ‘’}{redirect=“404”}{/if}
```

Here, if someone adds anything to segment_4, then boom—they go right to a 404, with the proper 404 status code.

In addition to preventing against “outer” segments, we like to be more explicit with our channel entries loops by utilizing dynamic=“no”[31] and using the other available parameters. Let’s take a hypothetical channel entries loop for an entry at `/news/entry/dark-wings-dark-words/`.

```
{exp:channel:entries channel=“news” disable="category_fields|member_
data|pagination|trackbacks"}
  <h2>{title}</h2>
  {news_body}
{/exp:channel:entries}
```

Here’s the same loop as we would do it:

```
{exp:channel:entries
  channel=“news”
  disable="category_fields|member_data|pagination|trackbacks
  dynamic="no”
  limit="1”
  url_title=“{segment_3}”
}
<h2>{title}</h2>
  {news_body}
{/exp:channel:entries}
```

Is ours a little longer? Yes, but so what? Setting our entries loops up with stacked parameters serves two purposes: they are easier to read because all of the parameters are there in a clean list, and it’s easier to know how it’s interacting with the URL by not being so “dynamic.” I have a better idea what the second example loop is doing at a glance.

I think this method really shines with entries loops that contain a lot of parameters on complex templates. I've seen some developers go so far as to do their channel entries parameters in a list this way but also *alphabetically*. That's hardcore!

One quick tip that may come in handy if you're developing a template and you find yourself adding or removing parameters over and over for testing: you can add a period in front of the parameter and that will keep it there for reference while disabling it.

Here's an abbreviated example where we were repeatedly experimenting between an entry_id and a custom field:

```
{exp:channel:entries
  channel="promotions"
  .entry_id="{segment_4}"
  search:promotion_id="={segment_4}"
}
```

The entry_id parameter would be ignored here.

Make a 404 Page

> "When life hands you lemons, burrow into the earth. Then give the lemons to the king of the mole men in exchange for his daughter's hand."
> —*@wordlust*[32]

We come across plenty of inherited sites that don't have 404 pages defined. This is simply a lack of attention to detail. It reminds me of a story my uncle Jim told me about how he interviewed people when he worked as a headhunter in New York City. His job was to place executives in high-paying corporate positions. He'd shake the person's hand with a smile and then *look at their shoes.* He wasn't just paying attention to the candidate's general appearance from the waist up, oh no…he was also looking at things other people don't readily see. Quite often the appearance of the shoes told him what he needed to know. Were they new? Were they scuffed up? Were they clean, or shined? Did this candidate take care of their shoes?

One of the first things I do when looking at an ExpressionEngine site (particularly one which we might inherit) is type a bunch of garbage in the first segment to get a 404 page (OK, so I have some weird hobbies).

Set the 404 because, if nothing else, it's the right thing to do. If you're converting a site from another CMS or changing around URL structure, you're probably going to miss out on more than a few 301 redirects, so you're going to want to have a decent, helpful error page in place. All you have to do is make a template that will serve as your 404, and then specify that template in the "404 Page" setting under Design > Template Manager > Global Template Preferences.

Another reason to do a 404 page: in addition to a spot where you can help direct a user to the right place, these are often pages on which you can experiment and have fun. There are tons of listicles and whole websites devoted to clever 404 pages[33]. When your website gives you a lemon…you know, maybe do something interesting.

While I'm discussing 404 pages, let me share a trick I use to help clients know what 404 pages are being hit. This is particularly useful if you have a site with a lot of 301 redirects because it helps you find some you might have missed. In our main layout, we have our Google Analytics code just before the closing `</head>` tag (using the traditional asynchronous tracking snippet[34]). Note the ExpressionEngine conditional for "ga_pageview" in there:

```
<script type="text/javascript">
  var _gaq = _gaq || [];
  _gaq.push(['_setAccount', 'UA-1234567-8']);
  {if layout:ga_pageview != ''}{layout:ga_pageview}{if:else}_gaq.push(['_
trackPageview']);{/if}
  (function() {
    var ga = document.createElement('script'); ga.type = 'text/javascript';
ga.async = true;
    ga.src = ('https:' == document.location.protocol ? 'https://ssl' :
'http://www') + '.google-analytics.com/ga.js';
    var s = document.getElementsByTagName('script')[0]; s.parentNode.insert-
Before(ga, s);
    })();
</script>
```

There is also the Universal Analytics tracking code[35], and in that case our conditional is positioned towards the bottom:

```
<script>
  (function(i,s,o,g,r,a,m){i[‘GoogleAnalyticsObject’]=r;i[r]=i[r]||func-
tion(){
  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createEle-
ment(o),
  m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBe-
fore(a,m)
  })(window,document,’script’,’//www.google-analytics.com/analytics.
js’,’ga’);

  ga(‘create’, ‘UA-12345678-9’, ‘auto’);
  {if layout:ga_pageview != ‘’}{layout:ga_pageview}{if:else}ga(‘send’, ‘pa-
geview’);{/if}
</script>
```

Then, in our 404 template we set the following:

```
{layout:set name=”ga_pageview”}ga(‘send’, ‘pageview’, ‘/404/?url=’ + doc-
ument.location.pathname + document.location.search + ‘&ref=’ + document.
referrer);{/layout:set}
```

That passes some extra information to the pageview, but only on the 404 page. This is useful because whenever a 404 page is hit, you can find it in Google Analytics with info about the page that was trying to be accessed as well as the referrer. It's easy to filter by "404" **(Fig 2.2).**

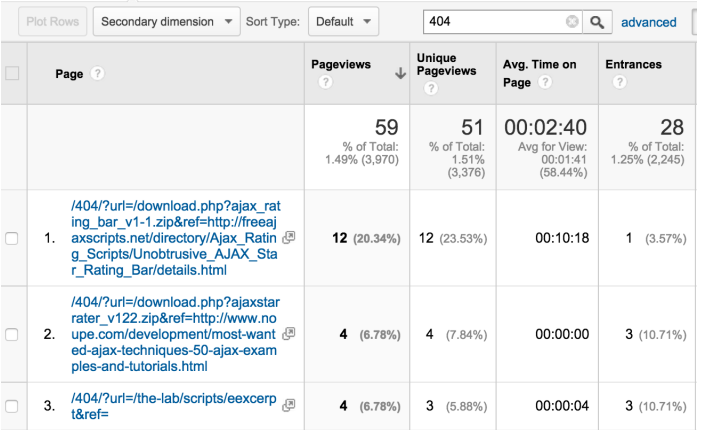


Fig. 2.2 An example of 404 pages in Google Analytics from masugadesign.com. Based on this, we should add a 301 redirect for those pages that lived at "/the-lab/scripts," and probably help people who are trying to hit that old download link, too.

Beware of Inherited Sites

> "Boy, you're gonna carry that weight,
> Carry that weight a long time."
> —*The Beatles*

This one is fresh in my mind after the work we've been doing recently. We are often called by businesses that already have ExpressionEngine sites and for one reason or another they are looking for another EE shop to take over their project. In reality, it's often because the previous developer got in over their head. *We have never inherited a site that didn't have some sort of major issue.* An inherited site is not a gift; it's a challenge. Before you accept one into your life, you're going to want to make sure that you want to work on the site (you love the subject, you care about the cause) and that you'll get along with the client.

The sins of the previous developer(s) can haunt your dreams. Not only that, *they* can make *you* look bad. Often the things you need to fix on the client's site are so pervasive, the holes so large, that you can't get the site to a point where it's workable, let alone move the site forward to solve the client's business goals. You can't go too many months blaming the previous developers for utter incompetence! (And beware of slamming previous devs too hard, because you never know what they were up against as far as timelines, budget constraints, or wild client expectations.)

For example, the site we were working on today needed what sounded like a two-minute update. We should have remembered that we inherited this site *before* saying, "Sure, no problem!" The client had asked us to make an update to text on some buttons. We were to add a field where they could override the default text. We do this sort of thing all the time: add the field, make a conditional to see if the button override text is populated, and if it is, use that text instead of the default.

An *hour* later, two of us had finally pieced together the fact that the default button text was nowhere to be found on the template, but was stored in a Matrix inside a Low Variable, and required the help of a custom plugin with the correct parameters to do a query. Unreal…the previous developer had to write a plugin to get at the text of the button, which appears in *maybe* three

different templates. I asked the client if they ever update the default text of the buttons themselves (and really, in most scenarios, how often *would* a client change the global text default of a button that reads "View Website?") and the answer, predictably, was "no." So this simple text could more-or-less have been hard-coded on the template. Is it any wonder this site had gone over budget and beyond the target date, when even the simplest things had unnecessary levels of complexity? This was a classic case of overthinking the problem, but no matter what the situation was at the time, no matter what the context was, *it's our problem now.*

There are many different philosophies developers have when it comes to putting together and templating ExpressionEngine sites. Inherited sites almost always lean towards being monsters in some way. If you're going to work with a monster, take care to make sure it's your kind of monster. Here's an example of what I mean:

We recently experienced Utter Defeat® on an ExpressionEngine project for the first time. This was a site we inherited and had to jettison after only a couple months. Looking back on the project, I never should have taken it. The site was built to all but completely bypass how ExpressionEngine normally handles URLs, using a witch's brew of add-ons. I knew there was real trouble when I couldn't output a simple, native `{segment_2}` variable on a template no matter what I did. I wasted a few hours reverse-engineering this flophouse of a website, trying to get variables to output in a way I knew should work. Since when can you put a `{segment_2}` on a template and get nothing back when content is clearly in segment_2?

To me, this site was not ExpressionEngine—it was Frankenstein's monster. I realized we couldn't waste another minute on it and found another developer who builds EE sites that way, and we handed it over to them. I don't regret it for a second, but I do feel bad for the client, who was nice, paid on time, but had an EE site that did everything it could to *not* be EE.

Be extremely careful in which site projects you elect to inherit. Make sure you're clear why the client isn't working with the previous developer any longer. Ask the potential client to pay you for a couple hours of your time for you to look over the site and write up a "recommended actions" paper (I've never been denied this request). In addition to helping determine if getting

money out of the client is like getting blood from a stone, you want this time so you can study the codebase, understand the templates, the channel and fields setup, categories, the host, the server—everything. If you do this and you don't get the project, your deliverable is the paper, so the client can still take it and show it to whatever developer they ultimately go with. If you do take on the project, you'll likely have encountered some trouble areas, so you'll have an action plan to start with. I've never met a client who didn't like a plan of action.

Remember, when you inherit a project, all the old baggage becomes your baggage, and you're going to carry that weight a long time.

Add Field Instructions

This takes little effort and has a huge ROI. Field instructions not only help the client, they help you. You *will* forget why a field exists, or where it shows on the front-end. You will forget if you need to enter values manually or if it's auto-populated somehow. You will forget what the image dimensions are supposed to be. Adding instructions when you make a field only takes a moment, and you should do it as you go so it isn't such a mountain-out-of-a-molehill later.

ExpressionEngine makes it easy to add detailed instructions. You can use HTML in them to emphasize or color-code text, and you can add links as well (**Fig. 2.3**). We sometimes link directly to Photoshop files or other reference files such as screencasts to make sure content editors are never lost.

Adding this level of detail certainly makes you look more professional, and helps make the control panel less intimidating, especially on channels that have a lot of custom fields.

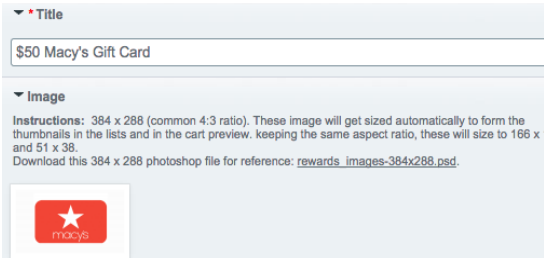


Fig. 2.3 Here's an example of linking to a PSD file so the client can create uniform

images later.

Approaches to ExpressionEngine Sites

There are numerous approaches to building a site in ExpressionEngine. There isn't a right or wrong way to do it, but there are certainly ways that are too clever by half, and end up making more work and giving developers more headaches than they save. As I mentioned earlier, the "danger" of ExpressionEngine is that there are 99 ways to do any single thing, and people have found every one of those 99 ways to do stuff. In the hands of someone not familiar with the system, this can lead to some pretty precarious houses of cards.

> "Is it a coincidence that every #eecms site we take over from problem devs either has Mountee or Structure or both installed?"
> *—John Baxter (@johnwbaxter)* [36]

Some people have never built an ExpressionEngine site without the third-party Structure[37] module, yet others have never used it on a single site, for example. Is either way "right?" Not really. So much depends on the goals of the site, the client, and the content it contains. There are sites that lend themselves to having tons of pages in a highly structured "tree," but there are other sites with many thousands of entries and a sensible enough URL setup that there is no need to have those pages managed by Structure.

Some people use the (newer) native Grid and Relationship fields, while others prefer to stick with Matrix[38] and Playa[39], the tried-and-true, battle-worn third-party versions that came out many years before the native versions of those fieldtypes. Is either side "correct?" Not really. The third-party versions have been around a *long* time and have a ton of features and a very helpful, responsive development team over at Pixel & Tonic. The native fieldtypes are newer (so maybe not as thoroughly tested by you yet), but they're part of the core—and there's something to be said for installing fewer add-ons when it comes time to upgrade an ExpressionEngine site.

In addition to add-ons used or template set-up, there are also different approaches when it comes to directory structure (ExpressionEngine is flexible this way) and multi-environment setups, which is something of importance when it comes to building a site with version control. You'll typically have at least a local development environment and a production site, if not a staging site or other developers working with their own local sandboxes. There is no "right" way to structure an ExpressionEngine site, but there are ways that are *less stupid*, and if anything, one of the goals of this guide is to help you be less stu…smarter.

One well-known example of a different approach to handling multiple environments is FocusLab's Master Config[40]. This is not something we've ever used, so I can't speak to it. At first glance, it looks more complex to set up than what we do (which is why we've never tried it), but it appears to serve much the same purpose. Perhaps after getting a site set up with our method you will find that FocusLab's version is better suited for you. If you do use this, you should read all the PHP to understand what it's doing. If you don't understand everything it's doing, you might consider not using it.

If I ever get around to trying FocusLab's config and I determine it's better than what we're currently doing, then we would certainly switch. I'm not so foolish to assume that our way is the "best" way—there is *always* something more to learn, or areas in which we can improve.

Never assume you know all there is to know, and never be complacent with what you do know. Use what works for the site at hand, and don't be afraid to use a different approach now and then if it's in the best interest of the client, the project, and future development. ExpressionEngine sites can be built to serve many different purposes, from small brochure-type sites to massive corporate monstrosities. One might lend itself to using the Structure module and the other not so much. You don't have to cookie-cut all your sites.

Default ExpressionEngine Installs

Default ExpressionEngine installs are not something we've found very useful in the past, but that's not to say that they wouldn't work for your workflow.

In the past we found that having a default install didn't help us much because we never did a high volume of sites. The sites we typically work on are larger, more customized, and typically have long development lifespans. Some take up to a year to complete. These sites are usually so different from one another that having a base install doesn't make much sense for us. There might be 2-3 add-ons that we do use on absolutely every site, but the time savings of having those in a ready-to-clone install versus just installing fresh copies is negligible.

We recently started taking on smaller projects, meaning we could work on more projects at a time, so we revisited the idea of a default install. Our approach is to have a repo with the most current version of ExpressionEngine, including any patches or bugfixes that have been released.

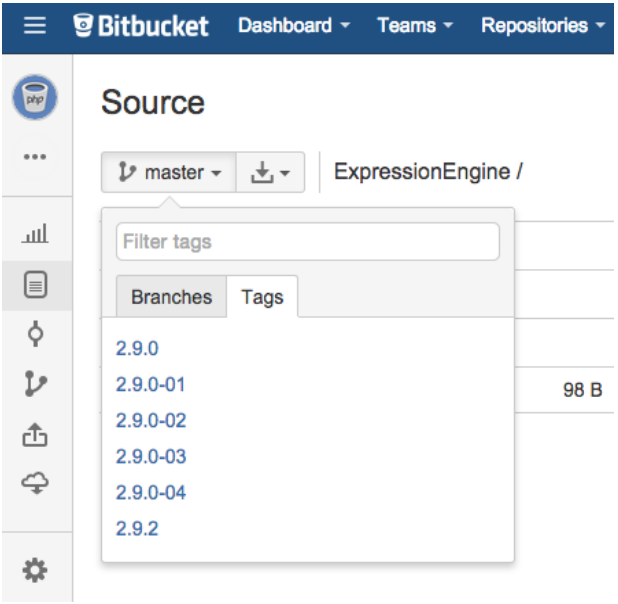


Fig. 2.4 Our default ExpressionEngine install repo.

We tag every release, so we could essentially go "back in time" and download any version of ExpressionEngine that we've tagged. In the figure **(Fig. 2.4)**, you can see we started doing this only recently–at version 2.9.0. Each of the tags after that and before the 2.9.2 indicates a patch was applied or something was tweaked. So, If I needed to go back and download a "fresh"

version of ExpressionEngine 2.9.0, I'd want to make sure to download from tag 2.9.0-04, which would ensure that I downloaded a version of 2.9.0 that includes a patch for Bug #20174[41].

If you're familiar with the ExpressionEngine changelog, you can also see that we skipped ExpressionEngine 2.9.1. The reason for this is that 2.9.2 was released only a day after 2.9.1, and was basically 2.9.1 with a couple big bug fixes. So, there was no need to tag a 2.9.1 release here. By the time I finish writing this guide, there is almost certainly going to be another point release of ExpressionEngine, at which time we'll upgrade this default install. Any new sites we work on get cloned directly from this repository.

As it stands right now, we don't keep any third-party add-ons in our default installation. It's strictly for core files. We might consider doing that in the future, but don't see a significant advantage to doing so. It's just as easy (if not easier) to go download a fresh add-on file from devot:ee when we need it as it is to keep each copy of "must install" add-ons updated in the repository.

There are a couple things to think about with default installs. First, if you are going to keep a default installation, *do not keep one in a public repository.* I've seen developers who otherwise have their heads screwed on straight keep a full install of EE, loaded with commercial add-ons, sitting in a *public* repo at Github ready for anyone to clone. Please, *do not do that*. (To test this I just did an advanced search on Github and found a complete ExpressionEngine install in less than 2 minutes. In fact, there are many of them.) Don't be that person!

The second issue with default installs is that it can potentially enable the "honest" mistake of forgetting to pay for your licenses. You start a new project by cloning your default site with the default add-ons in there, work on the site for a few months, maybe buying a few other add-ons along the way. The project finally launches, and you move on to the next site build. Forget something? Like, buying your ExpressionEngine license and all those commercial add-ons? You're not alone. This happens more often that you'd think. How embarrassing would it be to have multiple clients using the same ExpressionEngine license? Can you imagine them saying to you "we're not even using properly licensed software?"

We are careful to take stock of whether or not the ExpressionEngine license was purchased and by whom. We also inventory every commercial license in use and make sure to buy them all *before* a site launch.

Add-ons

I need to start this section with a big caveat: I started devot:ee in 2009 as a catalog for EE add-ons, and it then became a store, so I certainly have an interest in selling EE add-ons. There are a lot of great add-ons out there, and some I think should be in the core of ExpressionEngine. There are others that replicate core functionality and do it better than core (look at a few of the third-party WYSIWYG options[42], for example, which pretty handily make the built-in Rich Text Editor appear lacking).

ExpressionEngine is also *known* for add-ons. In a conversation I had a few years ago with a former CEO, it was mentioned[43] that the software is seen as more of a platform that can be extended, rather than a one-size-fits-all bloated piece of software. I don't know if that's EllisLab's official stance, but I do think that is still a fair take. ExpressionEngine isn't terribly bloated, and it's easy to extend it with add-ons to do whatever niche things your project requires.

Even in light of all of the above, my advice is: *use fewer add-ons.* Really. There are some things EE does just fine out of the box, without needing an add-on. But when you *do* need an add-on, ExpressionEngine's ecosystem is rich and vast enough to find what you need. (Of course, the first place to look is devot:ee). But many times, we'll go into a site and see so many add-ons installed that do things you can do out of the box with ready-to-go out-of-the-box ExpressionEngine tags. Maybe some people find it easier to search devot:ee than to read the ExpressionEngine documentation.

One problem with using too many add-ons on a site is that upgrades become a hassle. Don't believe me? Try updating a site that's a few years old to the latest version of ExpressionEngine. Even with devot:ee, it's hard to know what add-ons are going to work and which are going to completely stop your upgrade in its tracks.

Another issue is that compatibility and interoperability between add-ons can cause unexpected errors and conflicts. Developers have different levels of skill, ability, or time to test their add-ons against each version of ExpressionEngine. Those developers also may or may not care about ensuring that their niche extension plays well with another developer's niche module. I imagine this is the case in the ecosystem surrounding any content management system.

Purchasing Add-ons

I mentioned this under the section on default installs, but it bears repeating. Make sure to purchase the commercial add-ons and the ExpressionEngine license for each project. I think it is a common trap to forget to make the purchases. If you've developed many EE sites, and you tend to use certain add-ons over and over when you start a new project, you might download a copy of ExpressionEngine, then download add-ons you've already purchased and start building. There is nothing necessarily wrong with this, but make sure to make a note to yourself to buy the EE license and add-on licenses! A good time to do this might be towards the end of a project when you know exactly which add-ons will be used when your build goes to production.

Purchasing your add-ons also avoids potentially embarrassing situations. We've taken over a project where certain add-ons were in use and we needed help with a conflict or a bug. We go to contact the add-on developer for support and the developer, rightly so, asks "What is the license number?" If there was no record of the add-on purchase, then you've got a situation on your hands. Not to mention that this situation, even though we had nothing to do with it, made us look sort of bad. "We inherited this site, I'm telling you!"

We've also heard of a case where a developer had made their "own" add-on on by buying commercial add-ons, copying the code of those add-ons into their own plugin/module/extension combo (or whatever monstrosity they built), and then returning the add-ons. "Hey! That person is using their *own* add-on, so there's nothing fishy going on here at all, so move along." Right. I wanted to strangle this person on behalf of every developer who has spent time trying to help people like that, only to get burned by that insane logic.

Support those add-on developers who do such a masterful job of extending the ExpressionEngine CMS, and support the company that works hard to make ExpressionEngine better every day. That's good karma. (Besides, you're billing the cost of the add-ons through to your client, anyway, aren't you?)

Site Planning

> "The menu is not the meal."
> —*Alan Watts*

We generally do at least two documents at the beginning of every project: the URL Map and a Channel Map. I try not to get too carried away with this stage. I paint with a wide brush and do just enough so that we can get into actually building the site. Our ultimate goal is to build the site—not to plan or talk about building the site; the menu is not the meal!

URL Map

Depending on the complexity of the site, this can be a basic text document or a spreadsheet. Most times, even for pretty large sites, I end up going with a text document. There are sitemap services out there such as Slickplan[44] that help you create very detailed sitemaps, but I've rarely found the need to go to that level of detail, maybe because we've never needed to collaborate on sitemaps. Sometimes extra apps and services are only beautiful distractions. We've sometimes used spreadsheets, but I've found that very often those can waste nearly as much time as they are supposed to be saving you.

Here's an example of what a basic URL map might look like:

```
/

/about-us
  /about-us/locations
  /about-us/team
  /about-us/team/[P0]
  /about-us/team/bio/[url_title]

/contact
/contact/[error|success]

/blog
```

```
  /blog/[P0]
  /blog/[category]/[P0]
  /blog/[url_title]
  /blog/archive/[0000]/[00]/[P0]

/search
/search/results/[hash]
```

Some are much longer than this. Yes, it is super basic, but I can quickly gather whether a page is going to be its own template group, a single template, a static page, a page utilizing Switchee, and so forth. For example, for the "about-us" section, I'd probably go with an about-us template group, with index, locations, and team templates, and the team template will probably use Switchee to figure out what to display based on segment_2 (blank, pagination, or the keyword "bio"). These URL Maps are what you'll find at the top of our templates, too, as mentioned earlier in this chapter.

The items in square brackets are variables that represent typical things you'd see in a URL segment, such as an entry_id or url_title.

If the client has an existing site, we make sure to get a Google site index to see what pages are indexed so we know what the most critical 301 redirects are that we'll have to set up, in the event we can't keep the same URL structure.

Channel Map

In addition to the URL Map, we might also do a Channel Map, which is an outline of the fields and how the content will be stored or related between channels. This is generally better served in a spreadsheet, but sometimes I'm working so quickly the channel info never makes it beyond a sketch on a piece of paper before I go in and start making channel fields.

At its most basic, we're just trying to define the fields and what the fieldtypes should be before we sink a lot of time into it, and potentially have to backtrack. Using the example of the "team" from the URL Map, we might sketch this out, including the field name, the short name and the fieldtype:

```
Title       title
Position    team_position     Text Input
Photo       team_photo        Assets (File upload dir 3 - team)
Bio         team_bio          Textarea (Wygwam)
```

```
Department  team_department  dropdown (static options)
```

Some channels are quite complex and really benefit from you taking the time to think about how they will hold the data. In our experience, the channel map really doesn't need to be more complex than this.

Final Thoughts on Practical Development

This chapter can be pretty easily summed up in a few bullet points:

- **Comment your code.** This simple act can avoid so many headaches later.
- **Document your hacks.** Again, a simple step you can take at the time you make the hack that will save you an incredible amount of pain later, especially when it comes to updating your site.
- **Beware of default installs.** Don't keep these in public repositories and if it's just as much trouble to remove "default" add-ons you *aren't* going to use as it is to add ones that you *will* use, why bother? Also, if your work isn't based on a large quantity of sites built in X days, why bother?
- **Approaches.** You don't have to build your sites in cookie-cutter fashion. Different types of sites might lend themselves to a different line of attack. Go ahead and use a different approach now and then if it's in the best interest of the client, the project, and future development.
- **Purchase your ExpressionEngine license, and your add-ons.** Help support the ExpressionEngine ecosystem and keep your karma points high.
- **For ExpressionEngine sites, plan out those URLs** and really think about how you're going to store the content, but don't get analysis paralysis. You could spend a long time in this part of a project—don't overthink it.

3 Security

ExpressionEngine has a fine reputation as being a very secure CMS. As mentioned prominently on EllisLab's site, ExpressionEngine is:

> "Exceptionally Secure. And we're not talking about ExpressionEngine's self esteem. ExpressionEngine has never had a major security vulnerability, in eleven years and counting. Security in ExpressionEngine is of tremendous importance to our team.[45]"

You can read more on general security guidelines in their docs[46]. There are some things outside the scope of that document that you'll want to do to make sure your setup is secure. Fortunately, these things take minimal effort.

There is already an existing ebooklet on the subject titled *Securing ExpressionEngine 2*[47] by Mark Huot, the "godfather" of ExpressionEngine add-on development. This is a quick read that goes into more detail, and might be worth exploring in addition to the tips supplied here.

You will find that some ExpressionEngine shops don't do some of the following basic things. I've gone to EE sites for large companies put together by reputable developers and have been able to easily see a list of members or find functional registration forms on sites that don't look like they need any sort of member registration.

I've even found EE sites that didn't bother to rename the system directory. This is just as easy to do with WordPress sites. If you know a site uses WP, just try wp-admin.php at the end of the URL to see if you can get to the control panel. If you or I can find these things, you know spammers can find them!

The first six security tips are specifically for the system folder:

System Folder Related Tips

Rename the System Directory

This takes about five seconds to do. Ideally use something difficult to guess, or something that makes sense to you or your client. Once you've done that, just open index.php in the root of your site and admin.php and update the system directory in each:

```
$system_path = './westeros';
```

Use a "Masked Control Panel"

"Masked Control Panel" is sort of the older term for the practice of using admin.php. I still always call it Masked CP because I've been around since the ExpressionEngine 1 days and old habits die hard. Basically you're always telling your content managers to access their control panel via admin.php rather than navigating directly to the control panel at `/system/index.php`, which, by itself, is merely security through obscurity[48].

In any event, you should rename this file, too. If you renamed it to "throneroom.php", you'll then want to update the `cp_url` value in your config file to this value:

```
$config['cp_url'] = "http://example.com/throneroom.php";
```

We generally take this a step further by renaming "admin.php" to "index.php" and putting it in a directory with the custom phrase you selected. Accessing the CP this way feels a little cleaner to me. You would access it like: `http://site.com/throneroom/` rather than `http://site.com/throneroom.php`. You will have to change the system path slightly, because now you have to hop up one extra level (note the second period):

```
$system_path = '../westeros';
```

Your config URL will have to change as well:

```
$config['cp_url'] = "http://example.com/throneroom/index.php";
```

That config variable will actually be much more dynamic than shown in the simple example here. I'll go into much more detail about configuration variables in the next chapter.

Put the System Directory Above Webroot

Going a step further beyond renaming your "admin.php" file, we should take your system directory and move the whole thing out of your public folder, above webroot. The level of effort here depends in part on your hosting setup and whether or not you're deploying your site with version control or just using SFTP. Ideally you have a situation where you can change webroot for your site, so that you can deploy your site to a single directory and have your system directory sitting within that directory, right alongside your public directory. Your web directory would look something like this:

```
public_html
  |_system
  |_public
```

Control Panel at a Subdomain

I don't know how common this is because I don't see it mentioned very often, but you can put your system's control panel at a subdomain, so you would access it like so:

```
http://system.example.com/
```

Sometimes, you might come across a client that insists things be set up this way. In some ways, I feel that is a bit more secure only because it seems less common and Evil People looking for your control panel are probably more likely to look for it by using a trigger word in the first segment rather than a subdomain.

Using a subdomain allows you to do other relatively easy security-minded things such as restricting access to that subdomain by IP address in the htaccess file.

If you're going to use a subdomain for your control panel, I think it's easier to follow the previous step first and put your system directory above webroot. Then you create an A record at your host to point the "system" subdomain (or whatever term you plan to use) to this directory, just like your "www" points to your public directory. It's now as if you have two sites living next to each other, and one is your admin.

This comes with a caveat: make sure that if you're using any third-party add-ons, they will work with this setup. We tried the subdomain method on devot-ee.com and ran into some issues a while ago (meaning circa-2012). I believe it had something to do with an AJAX call a certain add-on was trying to make that was failing. Outside of a third-party add-on (potentially) not playing nice with this method, it should work fine.

Htpasswd Authentication on the Control Panel

One simple thing you can do to help put an extra barrier on your system directory is to add htpasswd authentication. Let's say you want to do this and are using the following method:

- You put your system folder above webroot
- You renamed "admin.php" to "index.php" and put it in a directory in your public folder called "control", so your content manager will access the control panel at example.com/control.

The next step is to add an htaccess file into the "control" directory, so that your structure looks like:

```
public_html
  |_system
  |_public
  |_control
  |_.htaccess
  |_index.php
```

In the htaccess file, add rules for authentication:

```
AuthName "Client Control Panel"
AuthType Basic
AuthUserFile /path/to/.htpasswd
AuthGroupFile /dev/null
Require valid-user
```

Then make sure your .htpasswd file is in the correct location (wherever your AuthUserFile path points) and has credentials in it. The htpasswd file does not need to be in your public folder, and shouldn't be in your public folder. You can generate htpasswds at various sites online. Just google for "htpasswd generator." Alternatively, you can easily make a password in the command line by typing `htpasswd -nb username password`:

```
~: htpasswd -nb jonsnow youknownothing
```

```
jonsnow:$apr1$DWpFuhQN$Q/cjY5BZbBZCIjIooP1G81
```

If you do that on a server where you don't want the password to be stored in command history, you can use `htpasswd -c filenametocreate username`.

```
~: htpasswd -c secret.txt tyrion
New password:
Re-type new password:
Adding password for user tyrion
```

That produces a file with the username and password in it.

When I add this extra htaccess/htpasswd file combo, I keep them out of version control. We don't really need this extra protection when running sites locally, and on a development site we may want to use a different password for the control panel authentication, or we might want to skip CP auth because we're probably going to put htpasswd authentication on the entire development subdomain.

Now when you try to access the control panel, your browser will first present you with a form for htpasswd authentication before you get to the login screen for your ExpressionEngine installation.

I just did this on a site build on ExpressionEngine 2.9.2 and hosted at Arcustech, and I didn't need to do anything else. You *might* have to do one other trick for this to work correctly, and this may depend on your server environment. We had to use this extra trick when protecting devot:ee's control panel:

If you're getting a 404 error trying to access your control panel now that it is protected with htpasswd, you'll want to add one line to your main htaccess file (in your public folder, the htaccess file where you most likely have rules to remove index.php from your URLs):

```
RewriteCond $1 !^(401.shtml)
```

This tip was found in a 2010 forum post at EllisLab (which I can longer find). I spent an age hunting that one down so you don't have to!

Force SSL on the System Directory

If your site is using SSL, you can force access to the system directory to be over SSL. If your control panel is *not* a subdomain you can force it to use SSL by adding something like the following to your htaccess file[49]:

```
RewriteEngine On
RewriteCond %{HTTPS} off
RewriteCond $1 ^(system|some_other_template_group|checkout) [NC]
RewriteRule (.*) https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

If `mod_spdy` is in use[50], or something else that prevents `{HTTPS}` from being available, that version won't work. If you use `mod_spdy`, use this:

```
# Force HTTPS
RewriteCond %{SERVER_PORT} ^80$
RewriteRule (.*) https://%{HTTP_HOST}%{REQUEST_URI} [R=301,L]
```

That takes any regular HTTP request on port 80 and rewrites it to HTTPS.

If your control panel is at a subdomain, you could use an htaccess rule to force that subdomain to be served over SSL. If that's the case, remember that you will need to have a wildcard SSL certificate that will allow you to use SSL on your primary domain as well as subdomains. And if you have a wildcard certificate for this sort of setup, why not put your whole site on SSL while you're at it? SSL all the things!

You could likely mash many of the previous tips together to have an incredibly secure control panel: Put it above webroot, allow access at a subdomain that is restricted by IP address, put htpasswd authentication on it, and force it to be served over SSL.

General Security Tips

Change the Default Member Trigger; Disable Registration

It used to be that ExpressionEngine shipped with membership registration enabled by default, so it fell to the developer to remember to turn this off for sites that didn't require membership. Newer versions of ExpressionEngine now come with this setting *disabled* by default, which is definitely in keeping with EllisLab's own Security Guidelines[51] that suggest using an "approach

of Least Privilege[52]" where you begin a site allowing access to nobody, and grant access only to those that qualify, whether it's a Super Admin or a regular member.

You can find the Member Registration preference under General Configuration when navigating to Members > Preferences in the control panel. If your site doesn't require membership registration, double check to make sure it is disabled, either via the setting in the control panel or by setting the `allow_member_registration` config variable[53]:

```
$config['allow_member_registration'] = 'n';
```

If you are allowing registration, you might be using any number of third party add-ons to gain more control over the member templates[54], which (as of this writing) are still a weak part of ExpressionEngine's core, as they don't use normal templates. The third-party offerings allow you to do member functions but use normal templates in your template groups and usually reserve a channel to store your member data, meaning that your member fields have access to all the available fieldtypes.

If you are using one of the third-party member add-ons, you might want to just "kill" EE's default member section altogether, and an easy way to do this is to change the default member trigger. You can do this via the control panel, or in your config file. I always suggest making these sorts of settings available in the config file.

Anytime there is a config variable available for a site setting, use it—you'll be versioning a "BASE" version of your config file anyway, and it's easy to see many settings all in one place when they're centralized like this. You can also easily change these values depending on your environment. If you put your `profile_trigger` word into your config, you can modify it in a couple different ways.

The first is to just use any random, hard to guess string:

```
$config['profile_trigger'] = 'asdflhasdfmnTyr10naasdgasfgl87asdg';
```

That could ultimately be guessed (however unlikely that would be in reality) because it never changes. Another method I've seen is to randomize this value.

```
$config['profile_trigger'] = rand(0,time());
```

Get Anything You Can Out of Your Public Folder

In addition to the system directory, there are numerous other files you can put above webroot and your site will work just fine. You can put file uploads above webroot (that is to say, the ones that will be served by ExpressionEngine, not those that need to be accessed directly like images). For example, devot-ee.com's downloadable add-on files are stored above webroot. You can also put your templates, all your third-party add-ons, snippets (if saving them as files) and more above webroot. I'll get into this in more detail in the next chapter where I discuss directory structure.

An important point to note: if you are saving templates as files, you'll certainly want to get them out of the public folder. ExpressionEngine ships with an .htaccess file in the *default* template location to prevent access, but we see many ExpressionEngine sites that don't have this file (and all it contains is one line: `deny from all`), perhaps because a template folder was manually made in another location.

Consider that it might be easy for someone to guess where you keep your templates if you have a run-of-the-mill directory structure. I might guess at the following, for example:

```
http://site.com/assets/templates/default_site/site.group/index.html
```

If we were to navigate directly there we would likely see a mess of stuff in the browser. (We've seen this happen in reality on more than one site.) If you view source, you will see the entire, raw EE template code. Channel names, other template names, variables, any ExpressionEngine comments you thought were hidden from public view, queries, PHP code…everything. Sure, it might be hard for someone to guess your template location in reality, but *why even take the chance*?

If any sites you're working on right now have templates in the public directory, try this experiment yourself, right now. I'll wait.

I'll admit—I didn't wait around. I thought about it for a second, then went and double-checked an older (2+ years) site we built where we didn't have our current system in place, and sure enough, the templates are in the public folder. I went to where I knew the index.html file was and voilà—there is a nasty looking page. Then I viewed source and *double voilà*—there is the entire contents of the template, as naked as the day it was born, there for all the world to see. I know what I'll be doing tomorrow.

Watch Out For {exp:query}

Sometimes `{exp:channel:entries}` doesn't cut it or you're doing something that can't be done with native tags, so you decide to use `{exp:query}`. When you do, beware of using template tags in your query. If those tags aren't validated and escaped, you could fall prey to a SQL injection attempt. As an example[55], consider the following:

```
<h2>Member Info</h2>
{exp:query sql="SELECT * FROM exp_members WHERE member_id={segment_3}"}
  Screen Name: {screen_name}<br/>
  Registered Email Address: {email}<br/>
  Username: {username}<br/>
  <a href="{site_url}account/edit/{segment_3}">Edit</a>
{/exp:query}
```

That query seems simple enough (and it is!) but it is an example of the danger of the Query module. If someone were to visit this URL:

```
http://www.example.com/account/member_info/0 or member_id > 1
```

Look at `{segment_3}` there: "`0 or member_id > 1`." They are essentially running this query:

```
SELECT * FROM exp_members WHERE member_id=0 or member_id > 1
```

This request would show all of the member information in the system, which is obviously terrible. This is where it would make sense to make a simple plugin to handle this output. Read the source article for a primer on how to construct a simple plugin to mitigate the security issues posed by using user input in SQL, including `{exp:query}`.

Don't Enable PHP in Your Templates Unless You Really, Really Have To

Trust me, you can build really big sites in ExpressionEngine without using "actual" PHP in the templates. ExpressionEngine doesn't even allow PHP to run in its templates by default—you have to enable that ability[56], per template. One thing I've noticed that has been a common theme in sites we inherit is that there is usually a decent amount of PHP in the templates—sometimes so much so that it looks like a complete PHP application was built on top of ExpressionEngine. Some sites have PHP enabled on nearly every template. This indicates to me that the developer really had no idea how to use ExpressionEngine.

Why is this bullet point in the Security section? If you think you have to use PHP in a template, just keep the following in mind, which is straight from the ExpressionEngine docs:

> *IMPORTANT:* Enabling PHP in a template will enable anyone with editing rights for that template to become a de-facto Super Admin since they can execute any PHP they want in that template, including PHP that can reveal information about your system, PHP that can delete data from your database, etc. Exercise extreme caution before enabling this option if you permit others to edit your templates.

You shouldn't often have to resort to using PHP, but even when you have to do so, you can still keep it out of your templates by putting your PHP into a plugin instead. Developing plugins is outside the scope of this guide, but just remember that on typical "decent" ExpressionEngine builds, you shouldn't often come across a bunch of PHP in templates. In addition to being a security concern and slower than if the PHP were in a plugin, it just feels sloppy.

Debug = "0"

We like to make sure that debug is set to "0" on a production site, otherwise potentially too much information can be revealed to someone when an error is triggered. We don't even like to set this to "1"—which is only supposed to show errors to Super Admins. I suppose there isn't anything wrong with doing that, but my preference is to turn that on temporarily to trigger an error and have debug set to "0" the rest of the time. Typically we'll have

debug set to "1" in our local environments, though (which we try to keep as similar as possible to the production server) so that we can catch errors when logged in as Super Admins and fix them before issues show up on the live site.

This setting is found in two places: your index.php file in the webroot of your site, and as a config variable that can be set: `$config['debug'] = '1'`[57]. The critical one that overrides the other is in the index.php file, and this definitely should be a "0" on a live site.

One reason you don't want to show these errors is that full paths can be displayed (known as a Full Path Disclosure (FPD) Vulnerability[58]). Here's an example that we encountered recently:

```
<h4>A PHP Error was encountered</h4>
<p>Severity: Warning</p>
/home/useracct/public_html/syst3mf0ld3r/expressionengine/third_party/cool_
addon/libraries/cool_addon_library.php:536)</p>
```

In that example, at the very least, a nefarious ne'er-do-well would know that the system folder is in webroot and what the user account name is. They could then spend the rest of their day trying to have fun at your expense.

Be Aware of ACT IDs

A lot of ExpressionEngine functionality can be triggered by Action IDs in the URL, such as `?ACT=123`, where the query string is simply pointing to a number that relates to a row in the exp_actions table in your database.

Depending on what add-ons you have installed, your actions table will have all sorts of actions in it that trigger add-ons to perform things like running cron jobs, or generating PDFs. Because there is often no other key or string needed in the URL, once someone finds a site using ExpressionEngine, they can just run through all the actions from ACT=1 to ACT=10000 to see what they can see.

If all the add-ons you have installed are coded well, someone doing this shouldn't see anything. Some (poorly coded) add-ons might throw an error that reveals too much information when navigated to directly, out of the context in which it might normally be triggered.

It happens that devot-ee.com was hacked this way in October 2012, when someone discovered that by hitting a certain ACT ID with a couple other get variables attached they could see the database.php file. This was a real eye-opener, making me realize that even add-ons by reputable add-on developers can have flaws in them, and that *any modifications you make to your default system (including installing add-ons) can introduce insecurities*.

From an email conversation with Derek Jones, CEO of EllisLab, about the devot:ee hack:

> "…yes, it's true that you can randomly try action request IDs and trigger actions within ExpressionEngine, but the receiving methods should be written with proper validation, authorization, or whatever other routines are necessary to prevent unwanted execution or ill-effects from over-calling. For instance, hitting the member registration action without being logged in or submitting any form data displays an error that you're missing fields."

The way some add-ons are coded, you can't always assume that the developer has written it with "proper validation, authorization, or whatever other routines are necessary to prevent unwanted execution or ill-effects."

I'd like it if each action had a random hash (or something similar) associated with it (or was a harder to guess hash itself) to prevent someone from just breezing through the action numbers. It would feel a little more secure to me, but admittedly, documented problems with people monkeying around with ACT IDs are few and far between.

Limit Super Admin Accounts

Very likely your client and any of the content editors who will regularly be working on the site *do not* need to be Super Admins. We usually make a user group called "Site Admin" that is much like a Super Admin but with a number of things turned off and limited ability to break stuff. A user account like this is almost always more than sufficient for your clients.

We also take this one step further by customizing the control panel and navigation for this new user group with some fantastic third-party add-ons, which I'll go into detail on in a later chapter. **Your site security is only as strong as your weakest Super Admin account credentials.**

Conclusion

We've now seen a number of ways to protect your system, and given you some tips for settings to check, and things to look out for and generally be aware of.

Don't put anything in your public folder that doesn't need to be there, and remember that your entire installation is only as secure as your weakest Super Admin username/password combination. Once a malicious person knows you're using ExpressionEngine and they can find your control panel, you'd better hope your Super Admins don't have account credentials like admin / password123 or you might be kissing this client—and potentially your reputation—goodbye.

4 Setting Up ExpressionEngine

This chapter covers everything up to the point where we get our site into version control. First the installation, then moving things around to be more secure and better fit our view of how a project is set up. As mentioned back in Chapter 1, we're not going to get into anything fancy as far as templates and front-end strategy. There are other EE training sources out there that handle that aspect of development, and I mention those in the Resources chapter.

Install ExpressionEngine

I'm pretty vanilla when it comes to my local setup. I just want to get in and get working. I haven't had the need for a Vagrant virtual development environment or anything remotely complicated. In setting up a site for the purposes of this guide, we'll keep it simple. (Again, I'm making an assumption that you're using a reasonably recent version of a Mac. If that's not what you use, you'll have to use your imagination.)

For getting a sample site together so we can get into versioning it, we'll go "super native" and not bother with Stash, Switchee, Resource Router, or any of the other items we'd normally install. We just need the most basic setup: a locally installed ExpressionEngine build with a "Hello World!" template.

Installing ExpressionEngine is pretty straightforward. We're more or less copying the official docs for this section. I cover *every* step that I go through. Take what you need from it, and leave the rest!

I create a directory in Sites named according to the domain I'm going to use **(Fig. 4.1)**. For this guide, I'll use "eeguide.com." This is *not* where I install ExpressionEngine, though. This is a directory to hold anything associated with the site such as reference files, symlinks to DropBox directories,

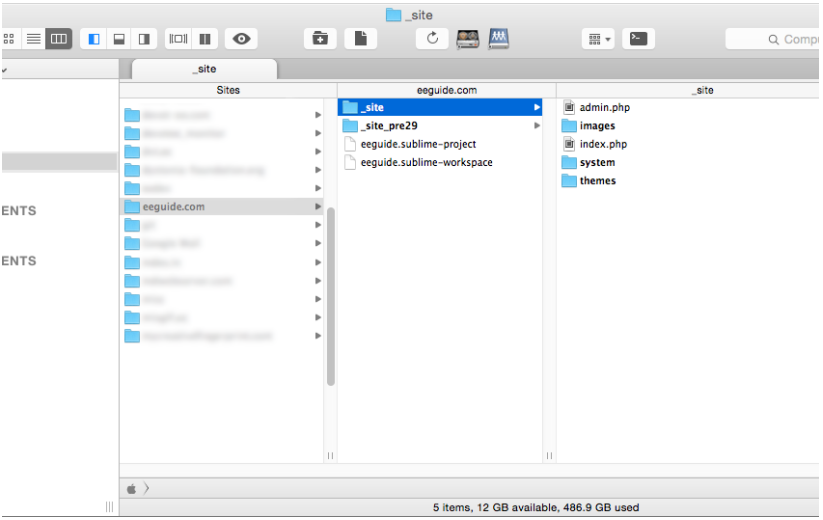


Fig. 4.1 Basic setup in the Sites directory.

Sublime Text project files, or any other miscellany I want to keep organized. Within the main eeguide.com directory, I'll make a "_site" folder. I prefix with the underscore to keep it at the top. This is where ExpressionEngine lives. In fact, to make the "_site" directory, I download a fresh copy of ExpressionEngine from the EllisLab site (after paying for it, of course), unzip it, and rename it "_site."

The top-level directory in Sites usually has the name of the actual domain (with the '.com' or the '.org' and so on), even though we use ".dev" on our sites while in development. So, for example, masugadesign.com has a local top-level directory titled "masugadesign.com" and the local dev site for it is at masugadesign.dev. Same goes for all our sites.

Now we'll move a couple things around a bit before setting up a virtualhost in MAMP. We'll move them now so we don't have to set up the virtualhost twice. Cast your mind back to the Security chapter when we were talking about putting the system directory above web root, as this is what we'll do at this point.

Within your "_site" directory, create a directory called "public," and move everything into it *except* your system folder **(Fig. 4.2)**. This includes admin.php, the images directory, index.php, and your themes directory.

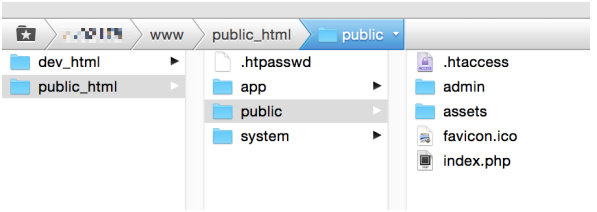


Fig. 4.2 Moving directories into place for better security and organization.

A few notes on this last step: our goal here is to set up the site in a way where most of the core files that drive the site are above webroot, and it is still easy to deploy when we get to the point where we're pushing files out to a production server.

Having your publicly-accessible directory *within* the main public_html directory means we can push our master branch to public_html, and any other branches can live outside (and alongside) the public_html directory. It's a nice way to compartmentalize things so as not to litter your web server with seemingly random files and directories, especially if and when you have a live site, dev site, and any other sites running on the same box.

If this site is going to be deployed to a server where you don't have control over changing webroot, you might want to skip this step. Some hosts don't "officially" allow changing webroot of a site, but sometimes you can get around this by being creative with symlinks if you have the ability to SSH into a server.

For example, on a recent site for which we inherited development, we wanted to improve their ExpressionEngine setup by moving the system directory above webroot, but the client was on a host that didn't officially allow this. The default web directory for sites on that server *had* to be "public_html." So we created another directory called "site_production" and put a "public" directory in that folder. Then we converted "public_html" into a symlink (Symbolic Link[59]) directed to "site_production/public" **(Fig. 4.3)**.

This way the server is still set up to work correctly, but when we deploy our site (using DeployHQ, which I talk about later), we deploy it directly to the "site_production" directory.

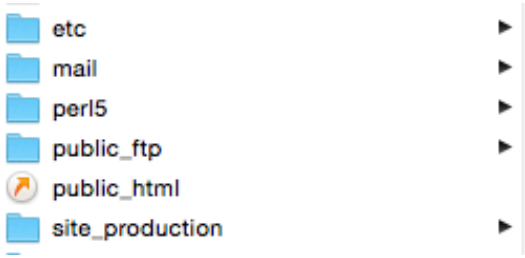


Fig. 4.3 This is how the symlink looks (orange arrow) in Transmit (Mac FTP app).

A symlink is constructed like this:

```
$ ln -s {/path/to/file-name} {link-name}
```

So creating our link involved logging into the server via SSH in Terminal and running a command similar to:

```
$ ln -s /site_production/public public_html
```

A terminal prompt will show you the symlink with an arrow like this:

```
lrwxrwxrwx 1 foo foo 23 2014-09-10 02:04 public_html -> site_production/
public/
```

If we set up any other subdomains on an account like this, we'll do something similar (e.g., dev.clientsite.com would resolve to the "dev" directory, but we would make that a symlink directed to "site_dev/public"). Your mileage may vary depending on the server setup and what is allowed, but we've done this symlink method on more than one site at different hosts with no issues. Again, none of this is necessary if you contact your host and they're able to change webroot for you!

Setting Up a Virtualhost in MAMP Pro

Now we need to make sure we can access the site in a browser. I'll set up a virtualhost in MAMP Pro so that "eeguide.dev" takes me to _site/public.

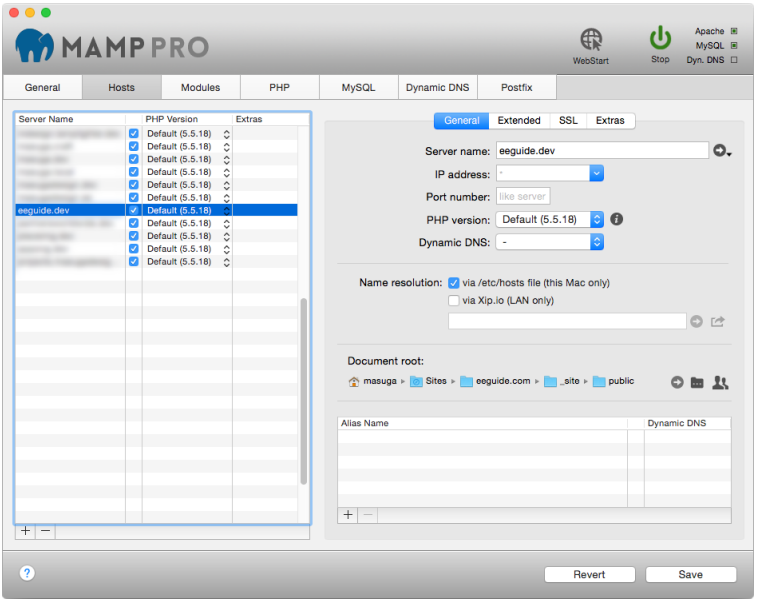


Fig. 4.4 Setting up a virtualhost for local development.

In MAMP Pro (referring to version 3.+ here), go to your "Hosts" tab and click the plus sign at the lower left. Enter "eeguide.dev" in the Server Name field **(Fig. 4.4)**. Then, under Document Root, select the "public" folder you just made. Mine happens to be at `/Users/masuga/Sites/eeguide.com/_site/public`. Click "Save" and then click restart the Apache and MySQL servers. Enter your password at the prompt and wait for those processes to restart.

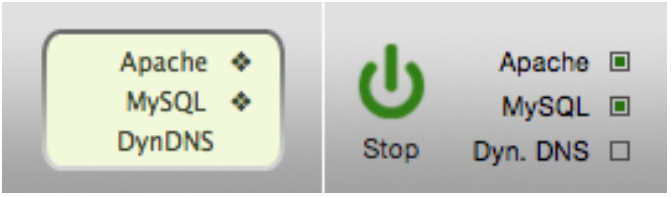


Fig. 4.5 Active MAMP Pro Apache and MySQL, v2 on the left. v3 on the right.

Once they have their little squares filled in **(Fig. 4.5)**, you're back in business. Open a web browser and visit http://eeguide.dev and you should see this message: "Your system folder path does not appear to be set correctly. Please open the following file and correct this: index.php." Perfect! That's exactly what we'd expect to see at this point **(Fig. 4.6)**.

Fig. 4.6 The error message we want to see. We're getting there.

Now we fix that by opening the index.php in your public folder and changing

```
$system_path = './system';
```

to

```
$system_path = '../system';
```

(Note the extra dot)

Now the error message we get in the browser is "Your config file does not appear to be formatted correctly." Great! We're on the right track.

Now hit http://eeguide.dev/admin.php and see that you get the same "Your system folder path does not appear to be set correctly. Please open the following file and correct this: admin.php" error we got before, except note that it's referring to "admin.php" this time. Open the admin.php file and add the extra dot to the beginning of the `$system_path` variable. Now reload the page and voilà: we see the installer.

Start by clicking the "Click Here to Begin!" button, and hope that the next screen shows you the "All pre-installation tests have been completed successfully and no errors were detected" message (**Fig. 4.7**).

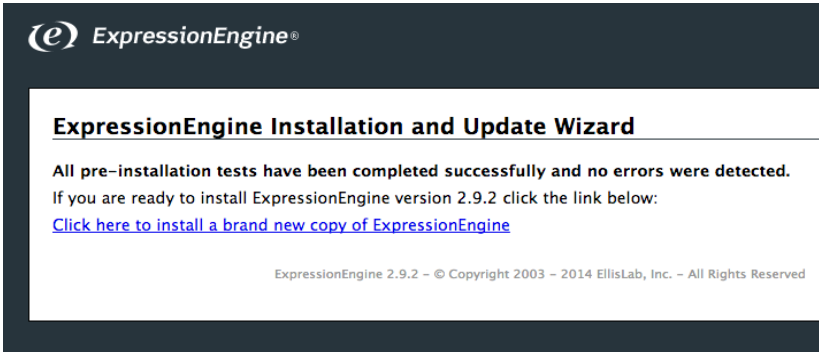


Fig. 4.7 Pre-installation checks out, so we can get on with the install.

At this point you will need to have created a database for ExpressionEngine to continue with the installation, which we'll cover next.

Create a Database

Of course there are various ways to do this, but we're trying to keep things simple. I'll cover two easy ways to create a database.

Fig. 4.8 The MySQL tab on a recent version of MAMP Pro. There are three options for administering a MySQL database.

If you have MAMP Pro installed, open MAMP Pro and click on the "MySQL" tab **(Fig. 4.8)**. There you'll see buttons under "Administer MySQL with." If you want to use phpMyAdmin, click on "phpMyAdmin" which will open in your browser. Switch over to the phpMyAdmin control panel and click on the "Databases" tab. Enter a name for your database in the provided

text field, and click “Create” to make the new database. I created a database called “eeguide” **(Fig. 4.9).**

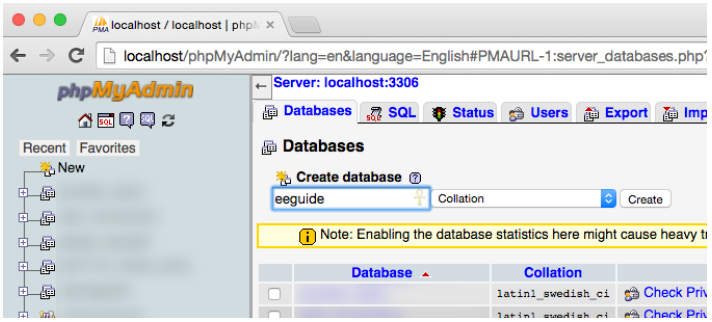


Fig. 4.9 Creating a database with phpMyAdmin.

Another way to create the database would be to use Sequel Pro. I mentioned this fine bit of software way back in Chapter 1. MAMP Pro version 3 and above includes a copy of Sequel Pro, but it’s also available as a standalone application. If you’re using the MAMP Pro version, you access it from the same MySQL tab as phpMyAdmin.

In Sequel Pro, double-click into localhost, and then select “Add Database…” from the “Choose Database…” dropdown. Enter the name of your database and click “Add” **(Fig. 4.10).**

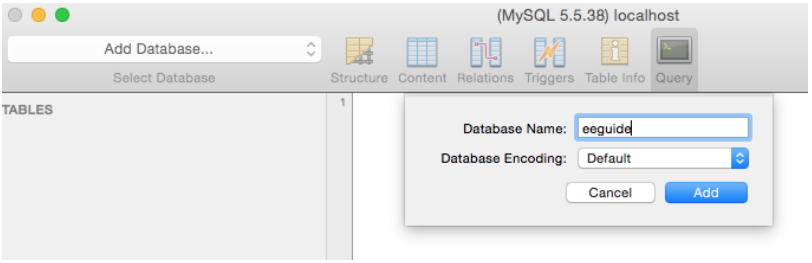


Fig. 4.10 Creating a database with Sequel Pro.

Lastly, ensure is that you have a user who can access this database. I use a generic user to access my local databases. You can create a user and password in either phpMyAdmin or Sequel Pro. If you give the user permission to access all databases on localhost, you can use the same user credentials on all your local ExpressionEngine sandbox installs.

File Permissions

Set file permissions on the files you have in your "_site" directory. This generally means setting a couple files and a few directories to be writable[60]. You can optionally tweak and run this ExpressionEngine permissions utility script[61] if you're familiar with the command line.

Follow through with the rest of the installation as outlined in the ExpressionEngine docs. The Settings page has sections for Software Registration, Server Settings, Database Settings, Admin Account, Site Theme, Optional Modules, and Localization Settings.

I would suggest ensuring the Site Theme setting is set to "None" rather than Agile Records because we will remove all the Agile Records files included in the default download in order to keep our install lean.

I usually uncheck the Emoticon module from being installed as well because I'm not big on having emoticons enabled for our clients. (In fact, I've long disliked emoticons and smileys being in any of my ExpressionEngine installs. So much so that I wrote one of my first extensions to hide the smiley links in the control panel[62] for ExpressionEngine version 1.x, way back in 2007 or so. But I digress!)

Once you've got your settings established, click "Install ExpressionEngine!" You should have ExpressionEngine installed now, so do as the screen says and remove the `/system/installer` directory **(Fig. 4.11)**. We'll never have the installer directory committed to our Git repository at any time. We add it temporarily, do our thing, and then remove it.

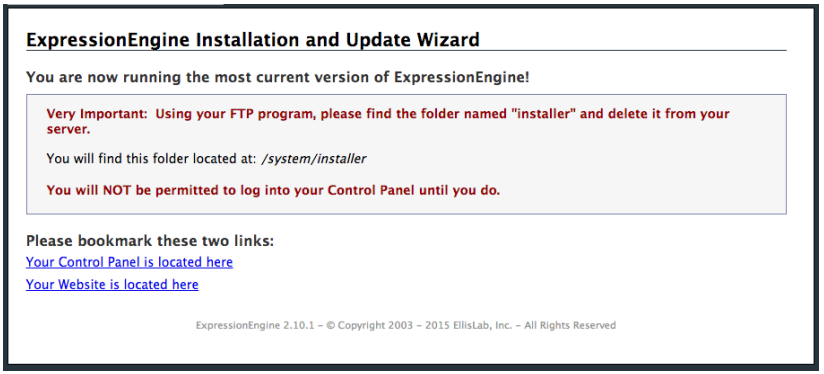


Fig. 4.11 The screen when ExpressionEngine has been installed (or updated).

You should now be able to access your site at http://eeguide.dev (which at this point will be blank), and your control panel at http://eeguide.dev/admin.php.

We now have a super basic setup that works. We're getting close to the point where we can make our initial commit. Our next steps will be to move directories around based on what we covered in the security chapter, clean up files we'll never use, and then create a basic template before diving into version control. But before all that, I'll go into great detail about our directory structure, because it's important to understand why we set things up the way we do and where everything is.

Directory Structure

We love a clean and simple root directory. As you saw when you first installed ExpressionEngine, the directory appeared like the left example in **Fig. 4.12**.

Now our objective will be to reconfigure that setup to meet our goals of a clean root directory *and* better security. After molding a site into our setup, our directories will end up looking like the right example in **Fig. 4.12**.

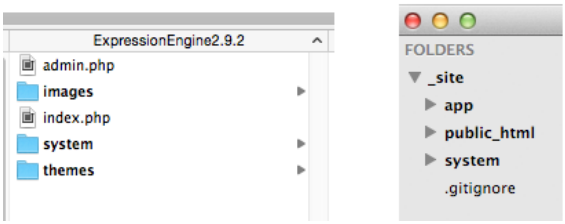


Fig. 4.12 EE directory as installed (left), and after we move things around (right).

Some of our sites might have a few more directories or files hanging out (such as "node_modules" or "Gruntfile.js" which are files that are a part of our workflow that I don't need to get into here), but that's pretty much it. The webroot of the site in this setup is the /public directory.

This setup keeps our system folder out of webroot, and allows us to put as much as we can in our app folder (templates, add-ons, and more) which also resides above webroot. *ExpressionEngine makes it easy to set up our sites this way with minimal effort.*

Also, because our system folder is not in webroot, we don't *have* to change it from the default of "system"—we'll rename the default "admin.php" that lives in the `/public` directory instead—but more about that later.

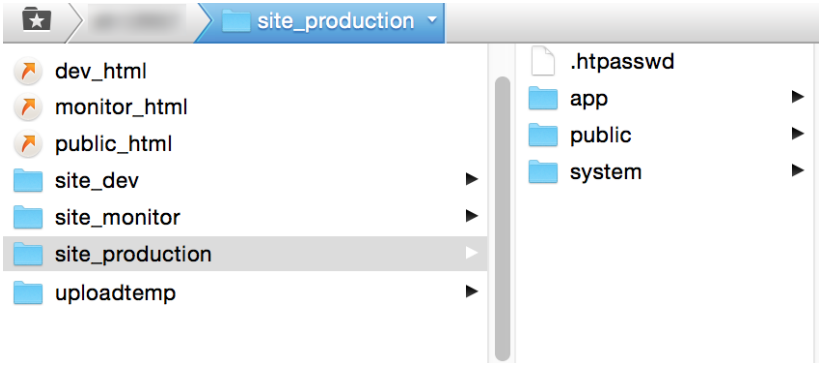


Fig. 4.13 Site root for devot-ee.com. This would be even cleaner if we didn't have to use symlinks.

This is devot-ee.com's site root **(Fig 4.13)**. Ahhh. So clean, so fresh. Almost makes you feel like you could update the site in a matter of minutes, right? Well, let's not get crazy. At least it looks clean, though. But don't make me open devot:ee's `/app/add-ons` directory if you know what's good for you. It might age you ten years to know how many add-ons we make use of on devot:ee. Note that we're using the previously mentioned symlink method here, where the public_html directory is symlinked directly to site_production/public.

I think it's important to go over the main directories in detail before moving our files around and getting our site into version control. I'll go through each of them (and their subdirectories) now.

The App Directory

This is the basic structure of the `/app` directory on every site we put together as of this writing:

```
/app
  |_/add-ons
  |_/config
  |_/src
  |_/templates
```

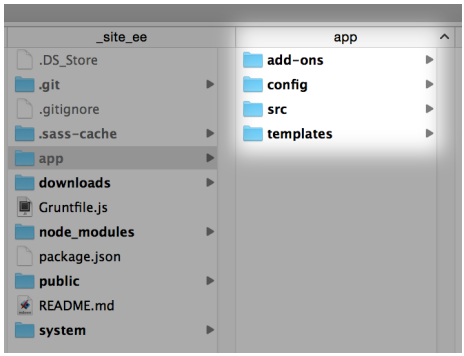


Fig. 4.14 Our app directory contents.

This is likely where you'll spend most of your development time, with the public directory coming in second. The way we currently develop sites, this directory consists of four sub-directories: add-ons, config, src, and templates **(Fig. 4.14)**.

If you use snippets and save them as files, you'd save them in this directory too. We phased out our snippets-as-files approach, with a `/snippets` directory inside `/app`, based on a new add-o-n that was recently released (called "In[63]" from Aaron Waldon, of CausingEffect.com—more about this add-on later in the chapter on add-ons). I touched on Snippets here because it was a consistent part of our workflow until recently.

That's the danger in writing a guide like this: technology and approach can (and should) change. If you're not constantly learning and revising how you do things, you might want to get in the habit of challenging yourself, lest you stagnate and stop growing! The sites we're building now are definitely better than the sites we were building even less than a year ago.

Let's go over the directories contained in the `/app` directory.

The /config Directory

I'm going to cover the config directory first. This directory is a workhorse because so much stems from the env_config.php file that lives here.

We have a default set of files that go in the `/config` directory for each project.

- _ee_hacks_addons.txt
- _ee_hacks_core.txt
- _README.txt
- BASE-env_config.php
- BASE-env_database.php
- BASE-htaccess.txt
- routes.php
- upload-prefs.php

All of the above files will be version controlled. When we start a project and clone the repo to your local machine, your first step would be to duplicate the three BASE files, and rename each file to env_config.php, env_database.php, and .htaccess (the env_ files will stay in this directory, but the .htaccess file will get moved to the public or public_html directory). Those files will *not* go into version control as they are specific to the environment (your machine, your co-worker's machine, the live site, a dev site, and so on).

We use DeployHQ[64] to deploy our sites over SFTP (there are other services that do this too, like Beanstalk[65]). We can set certain files to be ignored in the deployment, so we make sure not to deploy any of the "BASE-*" files to the live site, or either of the _ee_hacks files or the README.

I don't like having a production site littered with unused files. There isn't any reason for those files to reside on the production server because their only real function is in the repository as reference for the developers. So, as deployed, our /config directory looks like:

```
/config
  |_env_config.php
  |_env_database.php
  |_routes.php
  |_upload-prefs.php
```

One of the reasons we keep these files in their own config directory is so we don't have to go into the system folder any more than is absolutely necessary while developing. Most of our work takes place in the `/app` and `/public` directories.

Now let's cover all of the default files in the `/config` directory in turn.

_ee_hacks_addons.txt and _ee_hacks_core.txt

It is *vitally important* to document any hacks you've made to either the core files or the add-ons you've installed. Let me repeat myself, because if there is one point you should take away from this guide, this one ranks right up there: **it is vitally important to document any hacks you've made to either the core files or the add-ons you've installed**, and preferably in files like the ones I'm proposing here. We find it a little more common to hack core than to hack add-ons, so the examples below will be in reference to the _ee_hacks_core file, but the contents of the _ee_hacks_addons file would look much the same.

Don't litter your core files with inconsistently formatted comments about what you've hacked (e.g., line 5263 of some deeply buried core file reads: "hack: made this thing do a thing that it's not supposed to do" while another core file somewhere reads "corehack: removed 10 lines of stuff"). Be consistent with your comments, and then document them in one place. Why make another developer (or yourself 6 months or a year down the line) have to search all files in a project for the word "hack"?

> **Fun Tidbit:** While writing this section, I searched a fresh install of ExpressionEngine 2.9 for the word "hack." Sublime Text searched 1414 files and found 50 matches in 24 files. The results do make for some interesting reading. In fact, some of the comments in the core files are quite entertaining. How about:
>
> "Match the entirety of the conditional, dude. Bad Rick!" or...
>
> "Deprecate "weblog" tags, but allow them to work until 2.1, then remove this." (which, curiously, I found in the 2.9 codebase)

Here is how we currently document our hacks. First of all, hacks in the files should have comments near or around them that look consistent, to make them easier to locate. Ours look like the following:

```
/* =SOF MASUGA hack ========== */

[modified code here]
```

```
/* =EOF MASUGA hack ========== */
```

We put the word "Masuga" in there which makes them very easy to find when searching all the files in a project. Nothing like being associated with being a hack.

At the top of the core hack file, we list all the files referenced throughout the file, because on more complex builds, the hack file can get quite long. It's nice to have an index at the top so you know how many total files are involved.

Index of all hacked core files changed below:

```
/system/expressionengine/controllers/cp/members.php
/system/expressionengine/libraries/Template.php
/system/expressionengine/modules/channel/mod.channel.php
… (and so on)
```

Below that basic list are the more detailed hacks. Each one consists of the following info:

- **Hack** - Short description of what we did
- **When** - What EE version we applied the hack on and the date
- **Reason** - Why did we need to do this? Better be good!
- **Database** - Were any changes needed to the database? (not common)
- **File(s)** - What files were changed, and what code was altered

Here is a complete (real) documented hack. You can see that having all this information at your fingertips would be very useful when updating a site and wondering whether you need to re-apply a hack or not.

```
========================================================
Hack:     Commented out checks for duplicate screen_name when registering on
          the front end or through the control panel.
When:     EE 2.6.1 - Thursday October 24, 2013
Reason:   We need ability to have duplicate screen names.
Database: No change

File(s):  /system/expressionengine/controllers/cp/members.php
          /system/expressionengine/controllers/cp/tools_utilities.php
          /system/expressionengine/libraries/Validate.php

/system/expressionengine/controllers/cp/members.php
```

```
(~ 2148)
remove "[new]" from the validation rules (this covers the back-end registra-
tions)
'rules' => 'trim|valid_screen_name[new]'

/system/expressionengine/controllers/cp/tools_utilities.php
(~ 786-798)
Comment out the whole case 'screen_name' block

/system/expressionengine/libraries/Validate.php
(~ 231)
Comment out if block under "Is screen name taken?"
============================================================
```

Each hack is at least this detailed. When we go to upgrade our ExpressionEngine install we review the changelogs to determine if the hacks need to be reapplied. If we were upgrading that 2.6.1 site to EE 2.10, we'd look to see if the ability to add duplicate screen names is available in the core product. If not, reapply!

_README.txt

We recently had a contractor help us out on a project. It can be a chore to get anyone up to speed with how you do things, but his very words after cloning the site to his local machine and getting set up were "Dude, that was a breeze. Your setup is slick. I like it." I think that was in no small part due to including a README file. The README explains the renaming of the BASE files as mentioned above. Here is what is in the file:

```
Instructions for the ExpressionEngine 2 config files.
============================================================
Each "BASE-*" file is versioned and included with every new installation.
All the files created by copying and renaming the BASE files are NOT ver-
sioned.

tl;dr steps:

  1. Duplicate and rename BASE-env_config.php => env_config.php
  2. Duplicate and rename BASE-env_database.php => env_database.php
  3. Duplicate and rename BASE-htaccess.txt => .htaccess (and put in web
root)

------------------------------------------------------------
env_config.php
------------------------------------------------------------
Duplicate and rename BASE-env_config.php => env_config.php

The default EE config file is located at /system/expressionengine/config/
```

```
config.php, and is versioned. Add this line at the bottom:

<?php include(APPPATH."../../app/config/env_config.php");
Our custom config file is then located at /app/config/env_config.php

This file should contain all config data, starting from BASE config file,
and customized for the site location. This file is NOT versioned.

BASE config file located at /app/config/BASE-config.php

This file is versioned and should contain all base config data

------------------------------------------------------------
env_database.php
------------------------------------------------------------
Duplicate and rename BASE-env_database.php => env_database.php


EE database file located at /system/expressionengine/config/database.php
This file is versioned. Add this line at the bottom:

<?php include(APPPATH."../../app/config/env_database.php");

Our custom database file is then located at /app/config/env_database.php

This file should contain all database data, starting from BASE database
file, and customized for the site location. This file is NOT versioned.

BASE database file located at /app/config/BASE-database.php

This file is versioned and should contain all base database data.

------------------------------------------------------------
.htaccess
------------------------------------------------------------
Duplicate and rename BASE-htaccess.txt => .htaccess (and put in web root)

Active .htaccess file located in web root (/public)

This file should contain all htaccess data, starting from the BASE htaccess
file, and customized for the site location. This file is NOT versioned.

BASE .htaccess file located at /assets/config/BASE-htaccess.txt

This file should contain all base htaccess data. This file is versioned.
Rename as '.htaccess' and place in the root of the install.

------------------------------------------------------------
routes.php and upload_prefs.php
------------------------------------------------------------
These files are versioned and included by the env_config.php file.

------------------------------------------------------------
_ee_hacks_addons.txt and _ee_hacks_core.txt
------------------------------------------------------------
```

```
Update with any hack information that will make it easier to upgrade the
site in the future.
```

With a file that detailed, you *should* be able to get up and running pretty quickly.

env_config.php

This is a complex beast, and easily the most detailed of the files in the config directory. *It's the most critical file to get right when you're getting a multi-environment site set up.* There are so many ways you can control and manipulate ExpressionEngine with configuration variables. We're huge fans of config variables (not only for the core system, but for add-ons, too), and believe there are never enough of them. I would much rather control environment-specific behavior and settings with a file that is easily read and doesn't require decoding base-64 serialized data from the database to figure out.

When you first install ExpressionEngine, the default config.php is located at `/system/expressionengine/config/config.php`. The only thing we are going to do to that file is add one line at the very end to include our env_config file, as I'll outline shortly when talking about the system directory. Once we've added that line, we can add valid configuration variables to our own config, or override values that are in the default file.

If you want to see all of the available config variables and their values for the version of ExpressionEngine that you have installed, make a template and enable PHP on it, then put this on it:

```
<pre>
<?php print_r(ee()->config->config); ?>
</pre>
```

Visit that page in a browser and you will see a large formatted array, like this:

```
Array
(
  [image_resize_protocol] => gd2
  [image_library_path] =>
  [thumbnail_prefix] => thumb
  [word_separator] => dash
  [use_category_name] => y
  [reserved_category_word] => category
```

```
  [auto_convert_high_ascii] => n
  [new_posts_clear_caches] => y
  [auto_assign_cat_parents] => y
  [enable_template_routes] => n
  [strict_urls] => y
  ...
  [site_short_name] => default_site
  [output_charset] => utf-8
  [base_url] => http://eeguide.dev/
  [secure_forms] => y
)
```

Many of the default values are perfectly fine and don't need to be overridden. If you're partial to config variables like we are, it can be a good idea to check this array out every time you install a new version of ExpressionEngine because some config variables are deprecated and new ones are added that may be useful enough to find their way into your default config file. You can also view the "System Configuration Overrides" in the ExpressionEngine docs[66].

The following is what we initially put in our env_config file in all its glory (located in full in a Gist on Github[67]). Depending on the site and its unique needs, much more can be added to this starting point:

```
<?php
/*
Site: site.com
File: /app/config/env_config.php
=========================================================
Duplicate this file and rename as config.php
See: /app/config/_README.txt
=========================================================
Unversioned; included by /system/expressionengine/config/config.php
include(APPPATH."../../app/config/env_config.php");
=========================================================
"Hidden" overrides:
https://ellislab.com/expressionengine/user-guide/general/system_configura-
tion_overrides.html
========================================================= */
  $domain = '';
  $system_folder = 'admin';
  $config['new_version_check'] = 'n';
  $config['license_contact'] = ''; // an@emailaddress.com
  $config['webmaster_email'] = ''; // an@emailaddress.com
  $config['webmaster_name'] = ''; // Name of company or individual
  $config['doc_url'] = 'http://ellislab.com/expressionengine/user-guide/';
  $config['use_newrelic'] = 'n';

/*
```

```
|--------------------------------------------------------------
| Debugging and Performance
|--------------------------------------------------------------
| Set debug to '2' if we're in dev mode, otherwise just '1'
| 0: no PHP/SQL errors shown
| 1: Errors shown to Super Admins
| 2: Errors shown to everyone
|
| Profiler: Only shows for Super Admins when logged in (but not in the CP)
*/
  $config['is_system_on'] = 'y';
  $config['allow_extensions'] = 'y';
  $config['debug'] = '1';
  $config['show_profiler'] = (strpos($_SERVER['SCRIPT_NAME'], $system_fold-
er) !== false) ? 'n' : 'y';
  $config['template_debugging'] = 'n';

/*
|--------------------------------------------------------------
| URLs
|--------------------------------------------------------------
*/
  $system_app_path = realpath(BASEPATH . '../../../app');
  $protocol = (isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] == 'on') ?
'https://' : 'http://';
  $base_url = $protocol . $domain;
  $base_path = $_SERVER['DOCUMENT_ROOT'];
  $config['site_url'] = $base_url;
  $config['site_index'] = '';
  $config['cp_url'] = $base_url . '/' . $system_folder . '/index.php';
  $config['cp_theme'] = 'eclipse';

// Cookies
  $config['cookie_prefix'] = '';
  $config['cookie_httponly'] = 'y';
  $config['cookie_domain'] = '.' . $domain;

// Templates + Themes
  $config['save_tmpl_files'] = 'y';
  $config['tmpl_file_basepath'] = $system_app_path . '/templates';
  $themes_folder = '/assets/themes/'; // Must end with trailing slash
  $config['theme_folder_path'] = $base_path . $themes_folder;
  $config['theme_folder_url'] = $base_url . $themes_folder;

// Localization Settings
  $config['default_site_timezone'] = 'America/Detroit';
  $config['allow_member_localization'] = 'n';

// Sessions
  $config['website_session_type'] = 'c';
  $config['cp_session_type'] = 'c';
  $config['cp_session_ttl'] = 259200; // Seconds. This requires small core
hack. See hack file.
```

```
/*
|--------------------------------------------------------------
| Tracking & Performance
|--------------------------------------------------------------
|
*/
  $config['disable_all_tracking'] = 'y';
  $config['enable_hit_tracking'] = 'n';
  $config['enable_throttling'] = 'n';
  $config['log_referrers'] = 'n';
  $config['gzip_output'] = 'y';

/*
|--------------------------------------------------------------
| File Upload Preferences
|--------------------------------------------------------------
| The included upload_prefs.php file is versioned.
|
*/
  $uploads_folder = '/assets/uploads';
  $upload_path = $base_path . $uploads_folder;
  $upload_url = $base_url . $uploads_folder;
  $config['upload_preferences'] = include 'upload-prefs.php';

/*
|--------------------------------------------------------------
| Image Paths
|--------------------------------------------------------------
|
*/
  $images_folder = '/assets/images';
  $images_path = $base_path . $images_folder;
  $images_url = $base_url . $images_folder;
  $config['emoticon_path'] = $images_url . '/smileys/';
  $config['emoticon_url'] = $images_url . '/smileys/';
  $config['captcha_path'] = $images_path . '/captchas/';
  $config['captcha_url'] = $images_url . '/captchas/';
  $config['avatar_path'] = $images_path . '/avatars/';
  $config['avatar_url'] = $images_url . '/avatars/';
  $config['photo_path'] = $images_path . '/member_photos/';
  $config['photo_url'] = $images_url . '/member_photos/';
  $config['sig_img_path'] = $images_path . '/signature_attachments/';
  $config['sig_img_url'] = $images_url . '/signature_attachments/';
  $config['prv_msg_upload_path'] = $images_path . '/pm_attachments/';

/*
|--------------------------------------------------------------
| Third Party Add-ons
|--------------------------------------------------------------
| The included routes.php file is versioned. (requires Resource Router).
| Add other 3rd-party settings here as needed.
|
*/
  $config['enable_template_routes'] = 'n'; // Disable default template
```

```
routes
  $config['resource_router'] = include 'routes.php';
  $config['third_party_path'] = $system_app_path . '/add-ons';


  $config['minimee'] = array(
    'cache_path' => FCPATH.'assets/c',
    'cache_url' => '//'.$domain.'/assets/c',
    'base_url' => '//'.$domain,
    'disable' => 'no' // Disable Minimee entirely
  );

/*
|--------------------------------------------------------------
| Global Variables
|--------------------------------------------------------------
| If making subdomains or "sub-sites" with their own directory and index
file
| in web root, you can put a global_vars array there that will apply only to
| that site. The $master_globals get added here and apply to every site.
|
*/
  global $assign_to_config;
  if( ! isset($assign_to_config['global_vars']))
  {
    $assign_to_config['global_vars'] = array();
  }

  $master_globals = array(
    //'gbl_disable_fonts' => 'n',
    //'gbl_domain' => $domain
  );
  $assign_to_config['global_vars'] = array_merge($assign_to_config['glob-
al_vars'], $master_globals);

/*
|--------------------------------------------------------------
| Misc Settings
|--------------------------------------------------------------
|
*/
  $config['charset'] = 'UTF-8';
  $config['uri_protocol'] = 'AUTO'; // AUTO|PATH_INFO|QUERY_STRING|REQUEST_
URI|ORIG_PATH_INFO
  $config['deft_lang'] = 'english';
  $config['subclass_prefix'] = 'EE_';
  $config['cache_path'] = '';
  $config['encryption_key'] = ''; // 32 chars, upper/lower letters and num-
bers
  $config['rewrite_short_tags'] = TRUE;
  $config['enable_emoticons'] = 'n';
  $config['allow_member_registration'] = 'n';
  $config['profile_trigger'] = rand(0,time()); // Unneeded. Randomize member
profile trigger word.
  $config['use_category_name'] = 'y';
```

```
  $config['reserved_category_word'] = 'category';
  $config['auto_assign_cat_parents'] = 'y';
  $config['autosave_interval_seconds'] = '0'; // Disabling entry autosave


/*
|--------------------------------------------------------------
| Error Logging
|--------------------------------------------------------------
| If you have enabled error logging, you can set an error threshold to
| determine what gets logged. Threshold options are:
|
| 0 = Disables logging, Error logging TURNED OFF
| 1 = Error Messages (including PHP errors)
| 2 = Debug Messages
| 3 = Informational Messages
| 4 = All Messages
|
| For a live site you'll usually only enable Errors (1) to be logged other-
wise
| your log files will fill up very fast.
|
*/
  $config['log_threshold'] = 0;
  $config['log_path'] = '';
  $config['log_date_format'] = 'Y-m-d H:i:s';

/* End of file env_config.php */
```

We group variables in a way we think makes sense, although more than a few could go in more than one spot. Now let's go through it, section by section so it's clear what's going on.

Opening Comments and Variables

```
<?php
/*
Site: site.com
File: /app/config/env_config.php
============================================================
Duplicate this file and rename as config.php
See: /app/config/_README.txt
============================================================
Unversioned; included by /system/expressionengine/config/config.php
include(APPPATH."../../app/config/env_config.php");
============================================================
"Hidden" overrides:
https://ellislab.com/expressionengine/user-guide/general/system_configura-
tion_overrides.html
============================================================ */
  $domain = '';
```

```
$system_folder = 'admin';
$config['new_version_check'] = 'n';
$config['license_contact'] = ''; // an@emailaddress.com
$config['webmaster_email'] = ''; // an@emailaddress.com
$config['webmaster_name'] = ''; // Name of company or individual
$config['doc_url'] = 'http://ellislab.com/expressionengine/user-guide/';
$config['use_newrelic'] = 'n';
```

You may notice that there are a couple variables missing. Those are `app_version` and `license_number`:

```
$config['app_version'] = '292';
$config['license_number'] = '1234-5678-9012-3456';
```

Those will never change or need to be environment-specific, so those remain in the default config file in the system directory. There is no need to override them. This also makes updating smoother. We used to not keep *any* variables in the default config, but when we went to update ExpressionEngine, if at least the version number wasn't in the "real" config, updates would sometimes hang and never finish. I'll go through upgrading ExpressionEngine in a later chapter.

There are a couple things to do with this block. First, we change the name of the site to the domain of the site we're working on. Sometimes we get rid of the rest of the comments at the top because they're not needed in the env_config file—they still exist at the top of the BASE-env_config file from which we duplicated this file.

Then we update the domain with your local domain, and if you elected to use something other than "admin" for your system folder, you'll switch that as well. Update the `license_contact`, `webmaster_name`, and `webmaster_email`, and we're all set here.

We explicitly set `use_newrelic`[68] to "n" because by default this is *on* and we don't want any extra Real User Monitoring JavaScript added to our pages unless we're certain we're going to use New Relic[69] on this site. It technically won't be added unless the server has the new_relic PHP extension installed, but I don't see anything wrong with being explicit.

We like to set a variable for anything we will reference more than once. This makes it easier to update the config file later if we need to. We like to explicitly set the domain instead of relying on `$_SERVER['HTTP_HOST']`. This is definitely more secure because as I understand it, "In PHP every `$_SERVER` variable starting with HTTP_ can be influenced by the user.[70]"

We set the `$system_folder` to the value of our "masked" control panel directory, so you only need to change that in one spot should you ever need to.

Debugging and Performance

```
/*
|--------------------------
| Debugging and Performance
|--------------------------
| Set debug to '2' if we're in dev mode, otherwise just '1'
| 0: no PHP/SQL errors shown
| 1: Errors shown to Super Admins
| 2: Errors shown to everyone
|
| Profiler: Only shows for Super Admins when logged in (but not in the CP)
*/
  $config['is_system_on'] = 'y';
  $config['allow_extensions'] = 'y';
  $config['debug'] = '1';
  $config['show_profiler'] = (strpos($_SERVER['SCRIPT_NAME'], $system_fold-
er) !== false) ? 'n' : 'y';
  $config['template_debugging'] = 'n';
```

This block includes all the settings related to showing you, the developer, queries and template info when working on the site. The `show_profiler` is set to not show up when you're in the system directory, which we previously set as a variable at the top of the file.

URLs

```
/*
|--------------------------
| URLs
|--------------------------
*/
  $system_app_path = realpath(BASEPATH . '../../../app');
  $protocol = (isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] == 'on') ?
'https://' : 'http://';
  $base_url = $protocol . $domain;
```

```
$base_path = $_SERVER['DOCUMENT_ROOT'];
$config['site_url'] = $base_url;
$config['site_index'] = '';
$config['cp_url'] = $base_url . '/' . $system_folder . '/index.php';
$config['cp_theme'] = 'eclipse';
```

We set `system_app_path` once here so we always know where our "app" directory is located, and can use that in other settings down the line.

The `$protocol` variable handles flipping between SSL or not, based on the request. Then we set `$base_url` by piecing together the `$protocol` and `$domain`, and `$base_path` is simply our document root. These variables will be used frequently. Our `site_index` is blank, because we will not want index.php in our URLs. The `cp_url` is, like the `$base_url`, pieced together from previously set variables. The `cp_theme` is set to be our own theme (not the default!) which we include in our default ExpressionEngine repo from which we start all our projects. This makes it easier to customize the CP and upgrade ExpressionEngine because we're not altering the default CP theme that comes in the core.

Cookies

```
// Cookies
  $config['cookie_prefix'] = '';
  $config['cookie_httponly'] = 'y';
  $config['cookie_domain'] = '.' . $domain;
```

If you want to help hide the fact your site is using ExpressionEngine (for security reasons), you can change the `cookie_prefix` to something other than the default "exp_." The cookie domain is automatically set based on the `$domain` variable we set previously.

Templates + Themes

```
// Templates + Themes
  $config['save_tmpl_files'] = 'y';
  $config['tmpl_file_basepath'] = $system_app_path . '/templates';
  $themes_folder = '/assets/themes/'; // Must end with trailing slash
  $config['theme_folder_path'] = $base_path . $themes_folder;
  $config['theme_folder_url'] = $base_url . $themes_folder;
```

We always save templates as files so we can version control them. With `$config['save_tmpl_files']` we tell the system to save them as files, and with `$config['tmpl_files_basepath']` we tell ExpressionEngine where to find those files, again utilizing the `$system_app_path` variable we set earlier.

For the themes, we set a `$themes_folder` variable and then use that to construct the `$config['theme_folder_path']` and `$config['theme_folder_url']` config variables with our previously set `$base_path` and `$base_url` variables. You can see reference to an "assets" directory in the `$themes_folder` variable—I'll cover that more when we move on to the public directory.

Localization Settings

```
// Localization Settings
  $config['default_site_timezone'] = 'America/Detroit';
  $config['allow_member_localization'] = 'n';
```

Localization doesn't seem to be as big a deal in recent versions of ExpressionEngine, but I've been working with EE long enough to remember it being quite confusing. Some of the confusion stemmed from members somehow having set a different time zone than what the site was set to. So we disable that ability here and set the default to be our timezone. If that needs to be changed based on where the client is located, it's easy enough to do.

Sessions

```
// Sessions
  $config['website_session_type'] = 'c';
  $config['cp_session_type'] = 'c';
  $config['cp_session_ttl'] = 259200; // Seconds. This requires small core
hack. See hack file.
```

This one is a little odd in that the `$config['cp_session_ttl']` is a *hack*, right out of the gate. The first two are straightforward. We want session type for the front-end and the control panel to use cookies. One reason for setting the `$config['cp_session_type']` this way is found right in the docs:

> "The Auto log-in on future visits? option appears on the CP login screen when the cookies only method is used, allowing users to remain logged-in to the CP for up to 2 weeks since their last visit."

We like anything that helps our content editors stay logged in for darn near as long as they please. One of the biggest complaints we hear from our clients is how they "get logged out all the time."

That complaint is a reason we have the `cp_session_ttl` hack in place—`cp_session_ttl` was removed/replaced as of version 2.8.

Now there is a config var named `expire_session_on_browser_close` (that as of this writing we have not currently fully tested as a complete replacement for `cp_session_ttl`). It used to be you could set `cp_session_ttl` to whatever you wanted, and we loved the ability to override the default value if we so chose. Note that this line has a comment that directs you to the hack file (again, all of this is from our default EE install which already has all of these things in place when we start new projects). From the _ee_hacks_core.txt file:

```
Hack:     Increase CP Session TTL with a config variables

When:     EE 2.9.2 - October 21, 2014 (reapplied)
          EE 2.9.0 - August 21, 2014 (added)

Reason:   For some reason this is not configurable, and we think it is too
          short a duration.

Database: No change

File(s): /system/expressionengine/libraries/Session.php
(Line 1422)

  /* =SOF MASUGA hack ========== */
  //return (REQ == 'CP') ? $this->cpan_session_len : $this->user_session_
len;
  return ee()->config->item('cp_session_ttl');
  /* =EOF MASUGA hack ========== */
```

We straight-up tell ExpressionEngine to look for whatever we set in the config file here, because we think the default is simply too short. You can definitely skip this config variable and hack and you'll probably be fine!

Tracking & Performance

```
/*
|--------------------------
| Tracking & Performance
|--------------------------
```

```
|
*/
  $config['disable_all_tracking'] = 'y';
  $config['enable_hit_tracking'] = 'n';
  $config['log_referrers'] = 'n';
  $config['gzip_output'] = 'y';
```

I think the key one here is `disable_all_tracking`. As far as I'm aware, we have never had a need for any of the default tracking ExpressionEngine does, so I see no reason to have any of it on. I don't see any reason to log referrers as most of the sites we use have Google Analytics, which does a great job of this anyway, so having this disabled saves a database query on each page load.

We tell ExpressionEngine to compress the front-end pages with `gzip_output` set to "y," but as the docs state:

> "It's a good idea to enable this setting in most production environments…This setting only controls whether ExpressionEngine itself serves up compressed front-end pages. If the web server is configured to serve compressed pages, this setting will have no effect.[71]"

File Upload Preferences

```
/*
|-------------------------
| File Upload Preferences
|-------------------------
| The included upload_prefs.php file is versioned.
|
*/
  $uploads_folder = '/assets/uploads';
  $upload_path = $base_path . $uploads_folder;
  $upload_url = $base_url . $uploads_folder;
  $config['upload_preferences'] = include 'upload-prefs.php';
```

Even though this environment-specific config file isn't versioned, the upload directory paths are versioned because they're set up in a multi-environment way, so they will always be the same for everyone. We do this by keeping them in a separate, versioned file, and including them in the env_config. When you think about it, a lot of the config variables would be the same for everyone, but we needed to make sure everyone was getting the upload directories, especially, when new ones were added to a site later on. Sure, we'd add them to the BASE-env_config file, but we don't always proactively

go back into that file to see what we might be missing on older, more mature projects.

Here is an example of what is in upload-prefs.php by default when it's cloned from our default install:

```
<?php
/*
Site: site.com
File: /app/upload-prefs.php
========================================================
The array keys must match the ID from exp_upload_prefs.
======================================================== */
return array(
/* 1 => array(
    'name' => 'Foo',
    'server_path' => $upload_path.'/foo/',
    'url' => $upload_url.'/foo/'
  ),
2 => array(
    'name' => 'Bar',
    'server_path' => $upload_path.'/bar/',
    'url' => $upload_url.'/bar/'
  ) */
);
```

The contents of the array are commented with easy examples of what will go in there, should you start adding any file upload locations. The `$upload_path` and `$upload_url` variables were constructed from the `$uploads_folder` and `$base_path` and `$base_url` variables, just before the line where we include this file.

Image Paths

```
/*
|--------------------------
| Image Paths
|--------------------------
|
*/
  $images_folder = '/assets/images';
  $images_path = $base_path . $images_folder;
  $images_url = $base_url . $images_folder;
  $config['emoticon_path'] = $images_url . '/smileys/';
  $config['emoticon_url'] = $images_url . '/smileys/';
  $config['captcha_path'] = $images_path . '/captchas/';
  $config['captcha_url'] = $images_url . '/captchas/';
  $config['avatar_path'] = $images_path . '/avatars/';
```

```
  $config['avatar_url'] = $images_url . '/avatars/';
  $config['photo_path'] = $images_path . '/member_photos/';
  $config['photo_url'] = $images_url . '/member_photos/';
  $config['sig_img_path'] = $images_path . '/signature_attachments/';
  $config['sig_img_url'] = $images_url . '/signature_attachments/';
  $config['prv_msg_upload_path'] = $images_path . '/pm_attachments/';
```

This is a block that covers the various standard ExpressionEngine image upload directories. We set three variables at the top of the set, and then go on down the line. Pretty straightforward stuff here!

Third-Party Add-ons

```
/*
|--------------------
| Third Party Add-ons
|--------------------
| The included routes.php file is versioned. (requires Resource Router).
| Add other 3rd-party settings here as needed.
|
*/
  $config['enable_template_routes'] = 'n'; // Disable default template
routes
  $config['resource_router'] = include 'routes.php';
  $config['third_party_path'] = $system_app_path . '/add-ons';
  $config['minimee'] = array(
    'cache_path' => FCPATH.'assets/c',
    'cache_url'  => '//'.$domain.'/assets/c',
    'base_url'   => '//'.$domain,
    'disable'    => 'no' // Disable Minimee entirely
  );
```

Any tweaks to third-party add-ons will go in this area. On some sites, this area of the env_config can get pretty loaded, but at least all the third-party tweaks are easy to find because they're in one section.

The Minimee stuff is here because we *always* use that add-on for reasons I'll touch on shortly when I discuss the cache directories in the public directory. You'll also commonly find variables here for CE Image or any custom add-on config variables we create.

We put the template routing config items that involve Resource Router here because it's a third-party add-on. This is one of those config items that could have easily gone into another area such as URLs. Even though ExpressionEngine has native routing now, we still prefer this third-party

solution for a couple reasons: 1) it is ridiculously flexible in the event we have to do anything crazy, and 2) all the information for the routes is stored in a file rather than the database (have I mentioned I'm a fan of storing configuration settings in a file rather than the DB?). This is a case similar to upload-prefs, where we would have certainly added the routes to the BASE-env_config file, but we don't want our developers to possibly miss any new routes that might come along as development progresses on a site, so we keep the routes in a separate, versioned file to ensure everyone has all the latest routes, all the time.

Here's an example of what you might find in routes.php:

```
<?php
/*
Site: westeros.org
File: /app/routes.php
============================
Requires Resource Router (http://dvt.ee/resrtr)
============================ */
return array(
  'get-help' => 'site/get-help',
  'action'   => 'site/action',
  'sitemap'  => 'site/sitemap',
  'contact'  => 'site/contact'
);
```

So far, we've never had to use advanced routing that this add-on offers. We use it a lot for pages like the above that live in the "site" directory, but for which we don't want to have "site" in the URL. Alternatively these templates could have their own template groups, but we don't always go so far as to make template groups for single page areas unless we think it might get complex or involve sub-sections or require embeds. I talk more about Resource Router later on when discussing "must-have add-ons."

Global Variables

```
/*
|---------------------
| Global Variables
|---------------------
| If making subdomains or "sub-sites" with their own directory and index
file
| in web root, you can put a global_vars array there that will apply only to
| that site. The $master_globals get added here and apply to every site.
|
```

```
*/
  global $assign_to_config;
  if( ! isset($assign_to_config['global_vars']))
  {
    $assign_to_config['global_vars'] = array();
  }
  $master_globals = array(
    //'gbl_disable_fonts' => 'n',
    //'gbl_domain' => $domain
  );
  $assign_to_config['global_vars'] = array_merge($assign_to_config['glob-
al_vars'], $master_globals);
```

The global variables are normally found in the "CUSTOM CONFIG VALUES" section of the index.php file that sits in the root of your site. There is a line there (approximately line 93 in a default ExpressionEngine 2.9.2 install) that is commented out by default:

```
// $assign_to_config['global_vars'] = array(); // This array must be asso-
ciative
```

We used to keep these in their own file and add an include to the core index.php file to get at the global variables, but we've since found that it's much easier to add the global variables directly to our config, which bypasses the need to add even a single line to the index.php file.

These user-defined global variables[72] are very useful. You can store all sorts of useful, reusable things in them, and access them on any template by wrapping them in curly braces. They can be used as "switches" to enable or disable or show or hide things on templates based on environment. The fun to be had is endless! Some examples, taken from a few different sites we've done (I usually line up the arrows for legibility):

```
$assign_to_config['global_vars'] = array(
  'gv_background_image'   => 'https://d127hlef0e87m.cloudfront.net/pic.jpg',
  'gv_eod_today'          => date('Y-m-d').' 11:59 PM',
  'gv_release_start_date' => date('w') == 0 ? strtotime('today midnight') :
strtotime('previous sunday midnight'),
  'gv_ads_fpo'            => 'y' // y/n: Show dummy placeholder ads for lo-
cal dev to speed up page load times
  "gv_network_cat_id"     => 'not 0'
);
```

Using Regular and Master Global Variables

From our example above, note the `$master_globals` array that is set after the normal `$assign_to_config['global-vars']` array. We don't always need to use that, but there are some cases where it comes in handy. This is a little more advanced, and may not be applicable in most cases. While we're talking global variables, I'll give you an example of how it might be used.

Sometimes you have a single site that has a number of subdomains (or alternatively, "subdirectories"), such as a multilingual setup where the subdomains indicate what language one is using: de.site.com, jp.site.com, and so on. If you were doing the same thing with subdirectories, those would be site.com/de/, site.com/jp/ and so on. This is known as the "old style method" of running multiple EE sites.

We've not done this specifically with *subdomains*, but we have set up numerous sites like this with *subdirectories*. For simplicity I'll continue to refer to them as subdomains. There will be a directory for each of these in your public folder that contains an index.php file and (quite likely) an .htaccess file.

If you are using global variables, you'll have them set in the index.php file for each of these subdomains, and those global variables will apply to that subdomain. If you need global variables that apply to *all* of your subdomains *and* your main domain, you'll want to utilize the `$master_globals`.

Say, for example, you have a site that deals with television networks rather than a multilingual site, and each network gets its own subdomain/subdirectory. The `site_url` for each of these sub-sites is actually different. In my fictional example, our three "sites" will be at these URLs:

```
http://rad.tv
http://rad.tv/hbo
http://rad.tv/showtime
```

So, to begin, we have our `global_vars` and `master_globals` in the main env_config file:

```
global $assign_to_config;
  if( ! isset($assign_to_config['global_vars']))
  {
```

```
    $assign_to_config['global_vars'] = array(
      "network_name"    => '',
      "network_abbr"    => '',
      "network_cat_id"  => 'not 0',
      "network_slug"    => '',
      "network_twitter" => ''
    );
  }



  $master_globals = array(
    'gbl_disable_fonts' => 'n',
    'gbl_load_ads'      => 'n',
    'minimee_debug'     => 'n',
    'minimee_disable'   => 'n'
  );

  $assign_to_config['global_vars'] = array_merge($assign_to_config['glob-
al_vars'], $master_globals);
```

So, our defaults are left blank (except for the category, which is set to "not 0" which is one way to tell a channel:entries loop to load entries regardless of category), and we have some "master" variables that we will want to be the same across all sites.

Then in our public folder we'd create a directory for each sub-site. So we create an 'hbo' directory and a 'showtime' directory.

```
public
  |_/assets
  |_/control
  |_/hbo
  |_index.php
  |_/showtime
```

Then you duplicate the index.php file from the public folder into each of these new directories. Update the `$system_path` with another "`../`" so it hops out one more level:

```
$system_path = '../../system';
```

Then we assign site-specific global variables in these index files, starting with the site_url, and then for all of the same `global_variables` we set in env_config. So, for HBO, we'd have:

```
$assign_to_config['site_url'] = 'http://rad.tv/hbo';
$assign_to_config['global_vars'] = array(
```

```
   "network_name"    => "Home Box Office"
  ,"network_abbr"    => "HBO"
  ,"network_cat_id"  => "2"
  ,"network_slug"    => "hbo"
  ,"network_twitter" => "hbo"
);
```

We do almost exactly the same thing for Showtime, in `/public/showtime/index.php`:

```
$assign_to_config['site_url'] = 'http://rad.tv/showtime';
$assign_to_config['global_vars'] = array(
   "network_name"    => "Showtime"
  ,"network_abbr"    => "SHO"
  ,"network_cat_id"  => "3"
  ,"network_slug"    => "showtime"
  ,"network_twitter" => "sho_network"
);
```

Note that even though we are setting the `site_url` to include what looks like a `{segment_1}`, that is actually *not* treated as a segment—it's part of the `site_url`. Consider then that all the following URLs are the same and "about" is segment_1 in all cases:

```
http://rad.tv/about
http://rad.tv/hbo/about
http://rad.tv/showtime/about
```

If you were to output global variables on the about template that all these sites share, they would output the values from the "site" you're on, but the `master_globals` will be the same across all "sites."

This can be pretty powerful. We've used this method on a number of television network-related sites to keep network content separated and separately branded and it works well. Global variables are your friends!

Misc Settings

```
/*
|--------------------------------------------------------------------------
| Misc Settings
|--------------------------------------------------------------------------
|
*/
  $config['charset'] = 'UTF-8';
  $config['uri_protocol'] = 'AUTO'; // AUTO|PATH_INFO|QUERY_STRING|REQUEST_
```

```
URI|ORIG_PATH_INFO
  $config['deft_lang'] = 'english';
  $config['subclass_prefix'] = 'EE_';
  $config['cache_path'] = '';
  $config['encryption_key'] = ''; // 32 chars, upper/lower letters and num-
bers
  $config['rewrite_short_tags'] = TRUE;
  $config['enable_emoticons'] = 'n';
  $config['allow_member_registration'] = 'n';
  $config['profile_trigger'] = rand(0,time()); // Unneeded. Randomize member
profile trigger word.
  $config['use_category_name'] = 'y';
  $config['reserved_category_word'] = 'category';
  $config['auto_assign_cat_parents'] = 'y';
  $config['autosave_interval_seconds'] = '0'; // Disabling entry autosave
```

This is a block that contains items we don't frequently mess with. Sometimes, depending on the server environment, we might have to mess with the `uri_protocol`, but "AUTO" works in most cases. We generally set an encryption key, particularly if we're using an add-on that requires encoding and decoding encrypted data like our own Link Vault[73].

By default we make sure membership registration is off and only enable that if the site is going to require it. Even then, we typically use a third-party solution to handle the templates that have the registrations forms, and rarely show a default profile page, so we always set the `profile_trigger` to be a random number.

Error Logging

```
/*
|--------------------------------------------------------------------
| Error Logging
|--------------------------------------------------------------------
| If you have enabled error logging, you can set an error threshold to
| determine what gets logged. Threshold options are:
|
| 0 = Disables logging, Error logging TURNED OFF
| 1 = Error Messages (including PHP errors)
| 2 = Debug Messages
| 3 = Informational Messages
| 4 = All Messages
|
| For a live site you'll usually only enable Errors (1) to be logged other-
wise
| your log files will fill up very fast.
|
*/
```

```
$config['log_threshold'] = 0;
$config['log_path'] = '';
$config['log_date_format'] = 'Y-m-d H:i:s';
```

We haven't had a need to use this block often, but sometimes it is useful while a site is in development. To use this, set your `log_threshold`, and set the path for the log file. For example, to send the log file to my app folder, I'd enter something like this:

```
$config['log_threshold'] = 3;
$config['log_path'] = '/Users/masuga/Sites/eeguide.com/_site_ee/app/';
```

Loading a page will create a file there titled like "log-2015-03-29.php" and it will be loaded with the type and level of logging info you requested. Here's a simplified sampling of log file output with debug, info, and error messages:

```
<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');
?>
DEBUG - 2015-03-29 23:01:03 --> Config Class Initialized
DEBUG - 2015-03-29 23:01:03 --> Hooks Class Initialized
ERROR - 2015-03-29 23:01:03 --> Severity: 8192 --> mysql_connect(): The
mysql extension is deprecated and will be removed in the future: use mysqli
or PDO instead /Users/masuga/Sites/eeguide.com/.../mysql_driver.php 73
DEBUG - 2015-03-29 23:01:03 --> Helper loaded: low_variables_helper
INFO - 2015-03-29 23:01:03 --> Minimee: Passed flightcheck.
INFO - 2015-03-29 23:01:03 --> Minimee: Cache file found: `de4cf-
54f8a350e932b5ba08c9e117d3f3c7cc156.1423546172.css`
DEBUG - 2015-03-29 23:01:03 --> Model Class Initialized
DEBUG - 2015-03-29 23:01:03 --> Final output sent to browser
DEBUG - 2015-03-29 23:01:03 --> Total execution time: 0.1679
```

As you can see, that sort of data might come in handy while developing—and we certainly wouldn't want this type of logging running all the time on a production site!

env_database.php

I have seen a lot of fancy environment switch-cases and such in default database.php files, so I'm sorry to inform you that ours has almost no fireworks whatsoever. Our env_database file in this directory contains what was in the database file originally in `/system/expressionengine/config/database.php`. Compared to the config file, the database file is a piece of cake.

We simply create a new variable group for each environment we need to access. Our default BASE-env_database file comes with one group, The env_database file on a live site only ever has one group. I've seen live sites with switch cases around their database connection info, which is checking to see if the current domain is foo.com, dev.foo.com, foo.testserver.com, doofuses-MacBook.local, and so on. Only one of those would ever be true on your live site, so why in the world would we need a switch case or conditionals?

Here is a very typical (local) env_database.php file:

```
<?php
/*
Site: eeguide.com
File: /app/config/env_database.php
============================
Unversioned; included by /system/expressionengine/config/database.php
include(APPPATH."../../app/config/env_database.php");
============================ */
$active_group = 'office';
$active_record = TRUE;
$db['office']['hostname'] = '123.123.123.123';
$db['office']['username'] = 'officeusername';
$db['office']['password'] = 'th30ff!cePa$$w0rd';
$db['office']['database'] = 'eeguide_db';
$db['office']['dbdriver'] = 'mysql';
$db['office']['pconnect'] = FALSE;
$db['office']['dbprefix'] = 'exp_';
$db['office']['swap_pre'] = 'exp_';
$db['office']['db_debug'] = TRUE;
$db['office']['cache_on'] = FALSE;
$db['office']['autoinit'] = FALSE;
$db['office']['char_set'] = 'utf8';
$db['office']['dbcollat'] = 'utf8_general_ci';
$db['office']['cachedir'] = APPPATH.'cache/db_cache/';

$db['local']['hostname'] = 'localhost';
$db['local']['username'] = 'mylocaluser';
$db['local']['password'] = 'MYl0c@LPa$$w0rd';
$db['local']['database'] = 'eeguide_db';
$db['local']['dbdriver'] = 'mysql';
$db['local']['pconnect'] = FALSE;
$db['local']['dbprefix'] = 'exp_';
$db['local']['swap_pre'] = 'exp_';
$db['local']['db_debug'] = TRUE;
$db['local']['cache_on'] = FALSE;
$db['local']['autoinit'] = FALSE;
$db['local']['char_set'] = 'utf8';
$db['local']['dbcollat'] = 'utf8_general_ci';
```

```
$db['local']['cachedir'] = APPPATH.'cache/db_cache/';
/* End of file env_database.php */
```

We might have a group or two in the env_database file, especially for early development, where I might take the database home over the weekend to do a bunch of things before bringing it to back to the office and loading it back on our centralized office machine. All I have to do is change the `$active_group` variable to flip between databases. One other thing we do is change the default 'cachedir' path that ExpressionEngine supplies us when the site is installed, which initially looks pretty lengthy, like:

```
$db['expressionengine']['cachedir'] = '/Users/masuga/Sites/eeguide.com/_
site/system/expressionengine/cache/db_cache/';
```

and simplify that to

```
$db['local']['cachedir'] = APPPATH.'cache/db_cache/';
```

which is more portable. Later, when I talk about moving things into place, we'll see how to tell ExpressionEngine to reference this file.

The BASE-htaccess file

We include this file here in the config directory with the other BASE files. It has *some* common things we'll use in an .htaccess file, but these can vary so widely between servers and environments, we really only keep ExpressionEngine-specific stuff in it by default, such as the ever-popular "remove index.php from the URL." Here's the meat of the file:

```
# site.com .htaccess file

# secure .htaccess file
<Files .htaccess>
  order allow,deny
  deny from all
</Files>

# EE 404 page for missing pages
ErrorDocument 404 /index.php?/site/404

# Simple 404 for missing files
<FilesMatch "(\.jpe?g|gif|png|bmp)$">
  ErrorDocument 404 "File Not Found"
</FilesMatch>

RewriteEngine On
```

```
RewriteBase /

# remove the www
RewriteCond %{HTTP_HOST} ^www\.(.+)$ [NC]
RewriteRule ^(.*)$ http://%1/$1 [R=301,L]

# Remove trailing slash
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_URI} ^(.*)/$
RewriteRule ^(.*)/$ /$1 [R=301,L]

# Removes index.php from ExpressionEngine URLs, but not the control panel
RewriteCond %{THE_REQUEST} ^GET.*index\.php [NC]
RewriteCond %{REQUEST_URI} !/admin/.* [NC]
RewriteRule (.*?)index\.php/*(.*) /$1$2 [R=301,NE,L]

# Directs all EE web requests through the site index file
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ /index.php/$1 [L]
```

This is our *typical* setup. Some sites force www, for example. Some force https. Some sites have tons of 301 redirects or numerous expires headers. This is a basic enough setup for a starting point. Just duplicate the BASE-htaccess.txt file to your public directory and rename it to .htaccess. Then change "site.com" to the name of the site you're working on, and update the system directory name in the block whose lead comment says "but not the control panel."

The /add-ons Directory

By default, all of your third-party add-ons would live in the "third-party" directory, which is located inside the system folder. Ever since EllisLab updated ExpressionEngine to allow us to move this directory wherever we want (which I believe was EE 2.4.0), I thought it made sense to bring it out of the system folder and name it what it actually contains: add-ons.

Not only do we bring it out of the system folder, we bring it out of webroot as well which means all the add-on files are not directly accessible through the public folder, so this setup is a little more secure. Plus, it's fewer directories to have to drill down through if we need to update add-on files. I like the approach of keeping as many non-core files outside of the core directories as possible.

When you first install ExpressionEngine, the third-party directory is at `/system/expressionengine/third_party`. All we need to do is take that directory and drag it out of that location to the 'app' directory that we previously set up, then rename it "add-ons."

Many add-ons come with associated themes. We'll want to move these themes to a new location as well, but because they contain CSS and JS, they still need to be in the public folder. We will end up taking the 'themes' directory that is in the base directory of our default install and move it to `/public/assets/themes`, as I will outline when I discuss the public directory.

Now we only have to tell ExpressionEngine where to find the now renamed 'add-ons' directory and the moved 'themes' directory, and we do that with the available `third_party_path`, `theme_folder_path`' and `theme_folder_url` config variables[74] that we just covered when talking about the env_config.php file.

The /snippets Directory

There are some bits of content that make sense to be "snippets." From the ExpressionEngine docs:

> Snippets are small bits of reusable template or tag parts. You could create a Snippet for any number of purposes, anywhere that you need to reuse a small portion of a template, including partial or complete tags, other variables, etc. Snippets add flexibility and reusability, while making it simple to make site-wide changes by editing the Snippet's source instead of having to modify many templates.

We don't keep a snippets directory any longer because we don't use "pure" snippets (we use a mix of Low Variables[75] and create includes and insert repeated things with Causing Effect's "In[76]" add-on).

The problem with Snippets, for those of us who use version control, is that they aren't saved as files, so we can't version them. We used to solve this problem by installing an add-on called Snippet Sync[77] (for which we have a developer license, because we used to install this on every site).

If we were to use snippets again, a /snippets directory inside the /app directory would be where we keep them.

The snippet sync method wasn't without hiccups, as it seemed like any time we made a change to a snippet file, we needed to load at least one (any) single page in the control panel to kick the snippet sync into gear and update the database with our change. There were two variables that we added to our config file to ensure the system could find them:

```
// Snippet Sync
  $config['snippet_file_basepath'] = $system_app_path.'/snippets';
  $config['snippets_sync_prefix'] = 'snip_';
```

Again, we largely phased out this aspect of our development, but I documented it here because it worked for us for the past few years (and is still active on many sites we've built) and I know many ExpressionEngine developers use Snippets.

The /src Directory

For the purposes of this guide, I won't need to go into detail on the /src (source) directory. I can tell you that we use it for our Grunt[78] workflow (which I also don't go into detail about in this guide). We typically store source SCSS files, JS files, and images that get minified, concatenated, and compiled and end up in our `/public/assets` directory. None of the files in this directory ever make it to the live site because I set our deployment tool to ignore deploying any files in this directory.

Suffice it to say, you can absolutely keep other things in the app directory that won't need to be in webroot.

The /templates Directory

Our whole approach assumes that you're saving your templates as files. I know of some people who build EE sites by coding their templates directly in the text areas provided in the control panel. I can't fathom doing this for a number of reasons. How would you easily and safely do a template-wide search and replace? Open up 20 browser tabs? (See my note way back in the Introduction about this, because if you don't watch out, "permanent data loss can occur!") There is no way a site of any complexity could be efficiently maintained by working on your templates in the control panel and storing them in the database. I don't get it. If you aren't working on your templates as files, I feel you have some bigger problems than this guide can solve.

However, my hope is that reading this guide might persuade you to change how you're currently developing.

If anyone reading this can refute my points above with evidence of a complex ExpressionEngine site that was coded entirely in the control panel and whose templates are all stored in the database, I'd love to hear about it!

Fortunately, ExpressionEngine allows us to save our templates as files. From the docs[79]:

> ExpressionEngine supports saving Template Groups and Templates as regular folders and files on your server, so that you can use your preferred text editor (e.g. Dreamweaver, Coda, BBEdit, etc.) to edit Templates and then FTP the changes to the server.

I think they should append that last sentence to read "…, or allow you to version control them."

You can set templates to be saved as files in one of two ways:

1. In the control panel, under Design > Templates > Global Preferences, set "Save Templates as Files" to "Yes." You will also have to specify the server path to the templates, and make sure that directory has the proper write permissions.

2. Set variables in your config file (our preferred method). There are two config variables:

```
$config['save_tmpl_files'] = 'y';
$config['tmpl_file_basepath'] = $system_app_path.'/templates/';
```

Those lines may look familiar to you from the env_config.php section. The `$system_app_path` is dynamic and helps us with our multi-environment setup. Review the section on env_config.php if you need a refresher.

The contents of the `/templates` directory are straightforward: you have the `/default_site` directory at the next level down, and then you have your template groups within that. As of ExpressionEngine 2.8, with template layouts, our most basic templates directory would look like:

```
site
  |_app
  |_templates
    |_default_site
      |_common.group
      |_layouts.group
      |_site.group
  |_public
  |_system
```

We'll create a very basic front end template in just a bit when I cover moving directories into place. After that, we'll dive into version control.

The /public Directory

This is a pretty straightforward directory containing those files that need to be accessible via the web. Depending on your web server, this directory might be named "www" or "public_html." For most of the sites we work on, we either request that the host change the webroot of the site, or we shell into the server and make the change ourselves and restart Apache, or use the symlink method I mentioned before.

In this public directory, we put a couple sub-directories: admin (or whatever you renamed your admin.php to) and an 'assets' directory which will hold another set of directories including our themes, Javascript, CSS, fonts, etc. At its most basic, the public directory will look like this:

```
public
  |_/admin (or whatever you name your access to the control panel)
  |_/assets
    |_/c
    |_/css
    |_/images
    |_/js
    |_/themes
  index.php
```

The admin directory has one file in it—the "admin.php" file we discussed before, renamed to "index.php" and the system path in it modified to hop out a couple levels:

```
$system_path = '../../system';
```

The assets directory contains a number of subdirectories that hold things like your CSS, JS, images folder, and themes. Here is an example list from a

recent site we did, that is pretty typical of our sites these days:

```
/build
/c
/images
/img_cache
/js
/themes
/uploads
/webfonts
```

A rundown of what those are, really quickly:

- **build** - We use a Grunt workflow where source CSS and JS files and images from the `/app/src` directory get compiled here.
- **c** - This is a cache directory specifically for concatenated and minified JS and CSS by the Minimee[80] add-on. The reason we use Minimee for this instead of relying on Grunt to do the concatenating and minifying of files is because we don't want to keep the minified versions in version control. It's enough to have the source files and the build files versioned.
- **images** - This is the images directory that is present in a default ExpressionEngine install.
- **img_cache** - This cache folder is for CE Image[81] cached files, which is an add-on we install more often than not. The default location for CE Image cached files is in `/images/made`, but I like to have them in their own location outside of the default images directory, and under a slightly more descriptive name. We set config variables for the path to this folder under the Third Party Add-ons section of env_config.php,
- **js** - Holds some source JS files (such as a local fallback copy of jQuery). Not always used, because most JS lives in `/app/src` and ends up in `/public/build`.
- **themes** - The ExpressionEngine themes folder. We set a config variable so the system knows where this directory is.
- **uploads** - We usually put any file upload directories here, except for those that need to live above webroot (such as the add-on files on devot-ee.com).
- **webfonts** - SymbolSet fonts, logo sets and such.

Your mileage may vary here. You may not even want to keep these directories in a folder called "assets"—that's up to you and how you like to do things.

The /system Directory

This is the most straightforward directory. We won't end up doing much with the system directory at all. We like to leave it alone to help facilitate easier upgrading. As of ExpressionEngine 2.9, it consists of three directories and two files:

```
/codeigniter
/EllisLab
/expressionengine
index.html
index.php
```

The way we set things up, there will be almost no need to ever go in your system folder. There are just two files inside `expressionengine/config` that we modify, which are config.php and database.php. Each of these files ends up having only one line added to them, referencing the "env_" version of the files we have stored out in our `/app/config` directory. We add a single line to the end of each of the core files, as I'll explain next when we move things into place. We can leave the variables in the default files if we wish, because any variables we have in our environment-specific files will override them anyway.

Basic Template Setup

This part of the guide is the most basic outline of what we keep in our `/app` templates directory. I mentioned earlier in "What This Guide Is Not" that I won't go into much detail about how to actually put site templates together. We won't be building an elaborate or detailed dev site. We won't be going into the relative merits of using a `{stash:embed}` over a regular `{embed}` vs. a snippet vs. an `{in:clude}` vs. a native layout. This guide is intended to cover everything up to that point: a practical and secure directory structure, and how to version that setup.

However, we'll put just enough together on the front end so we have something we can look at and tweak when we get to the version control chapter, which is coming up next.

At the most basic level, we have the following template groups:

```
common.group
layouts.group
site.group
```

The common.group will hold things like headers and footers, or other chunks that we might reuse in various places across the site. With the more recent addition of layouts to ExpressionEngine, we find we're not keeping headers and footers in the common.group at all any longer, because those "framing" elements will go into a layout.

We always keep our main files in site.group, which includes the site index and most or all one-off pages. For example, we might keep contact.html in site.group, or about.html. Some developers prefer to call that group "home.group" or "main.group" but we've never found a compelling reason to not use "site.group" as our default template group.

Usually we have "Enable Strict URLs" set to "Yes" (which is officially recommended), and means that the first URL segment must only be a valid template group, or your site will display a 404 page. As the docs state:

> This setting determines whether or not ExpressionEngine allows Templates from your default Template Group to be directly accessed in the first URL segment.[82]

So for our one-off pages in the site.group directory, we would normally have to access them at URLs like:

```
eeguide.dev/site/contact
eeguide.dev/site/about
```

This isn't ideal. We don't want 'site' in the URL in this case, so we use Resource Router[83] to make it drop-dead simple to remove the '/site' in the URL for any template that resides in this directory. This allows us to have Strict URLs enabled, but to also remove 'site.' More on Resource Router later in the section on "Must Have Add-ons."

If any section looks like it will get more complex or require extra templates, we put it into its own template group. For example, a blog might be better served by having its own template group to collect the related template files together.

```
blog
  |_ _entry.html
  |_ _sidebar.html
  |_ index.html
```

The templates that get included into other templates and aren't supposed to be directly accessed are prefixed with an underscore rather than a period. As of 2.9.0, the underscore is the default, and that is just fine with us. Before 2.9, the default indicator was a period, but hidden files on Macs are preceded by periods, so if you didn't want to have to worry if you could see your hidden templates, ExpressionEngine allowed you to change the indicator[84]. If for some reason you want to switch the indicator back to a period, you can do so with that config variable:

```
$config['hidden_template_indicator'] = '.';
```

In the next section, while we're moving things into place, we'll create a quick front-end template so our site displays more than a blank page. Once we have that working, we'll get into version control.

Moving Things Into Place

We're going to take our basic setup and move things around based on what I covered under Directory Structure, and we'll clean up some files too. With the default install we have, all of this is already done for us when we clone a new site from that repo, but we'll go through the motions here with the default ExpressionEngine codebase we downloaded and installed.

At this point we already have our system directory sitting alongside (and outside of) our webroot. We moved that already so we wouldn't have to bother changing the virtualhost's webroot more than once.

Rename admin.php

Inside the public directory, I'll create a subdirectory called "control" (you can name it whatever you want), and move the admin.php file into it and rename it "index.php." Now if we try to access our control panel at http://eeguide.dev/control/ you'll see we're out of luck with an error message that reads: "Your system folder path does not appear to be set correctly. Please open the following file and correct this: index.php." So we need to open the file in this directory and add an extra two-dots-and-a-slash to the `$system_path` variable (because this file is now one directory deeper than it was a minute ago):

```
$system_path = '../../system';
```

Now reloading http://eeguide.dev/control/ brings us to the control panel as we'd expect.

Make an /app Directory

Let's create the "app" directory that will sit right next to our public and system directories **(Fig. 4.15)**. Here is where we'll put our add-ons, config files, and templates. Remember, we want to keep as many things out of webroot as possible so that we can be as secure as possible.

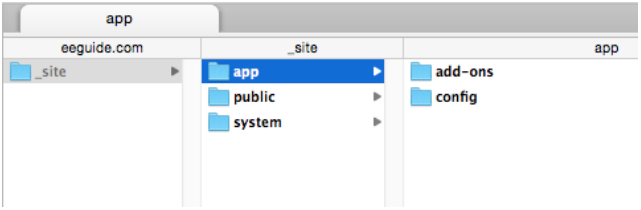


Fig. 4.15 The /app directory holds much of what we don't need in the public folder.

Inside your '_site" directory, create `/app`, and within that, create the `/config` and `/add-ons` directories. Before we get our templates in here, we'll take care of our environment-specific config and database files.

Make env_config.php and env_database.php for our Environment

One of our goals when building an ExpressionEngine site is to stay out of the core files and system folder as much as possible. This makes upgrading ExpressionEngine easier. As of 2.9.2, I think upgrading EE is still something that is harder than it needs to be, so anything we can do to make the process easier is a bonus.

We touched on the `/config` directory not too long ago. Create two files in the config directory: env_config.php and env_database.php. We prefix these with "env_" to signify they are specific to the environment (and these won't be committed to version control) and to distinguish them from the core config and database files.

For the config file, I *could* have you copy the entire contents of our pre-modified version that we already talked about, but instead we'll add a few variables and overrides at a time as we go through each point in the rest of this chapter. I think it will give you a slightly clearer idea of how those variables directly affect settings. We're not adding anything I didn't already go over above, we're just not adding *everything* at this point.

Copy the current contents of your config.php and database.php files into these two new files. Then, we'll just add one line to our default config and database files, which is easy to remember to add back in should it get accidentally blown away in an upgrade. At the bottom of the config.php file in your system folder, add:

```
include(APPPATH."../../app/config/env_config.php");
```

and at the bottom of database.php, add:

```
include(APPPATH."../../app/config/env_database.php");
```

Because we're just including our file at the end of this one, we can dispense with the:

```
if ( ! defined('BASEPATH')) exit('No direct script access allowed');
```

when we copy the default info into our environment-specific files, because it will still exist at the top of the original core files.

Now the system will be looking for the config and database variables in our environment-specific files rather than the defaults. You can reload the front end and the control panel and see that everything works as before.

We like to do it this way rather then having a number of conditionals or switch cases in our configs, because those feel like unnecessary extra overhead. Why have a config file on your production server that is looking through various different cases when only one will ever be true?

Now we'll paste in a block of variables that will be immediately useful for the rest of this chapter. Put these lines at the top of the config, above all the other variables that are there:

```
$domain          = '';
$system_folder   = '';
$system_app_path = realpath(BASEPATH . '../../../app');
$protocol        = (isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] == 'on') ?
'https://' : 'http://';

$base_url        = $protocol . $domain;
$base_path       = $_SERVER['DOCUMENT_ROOT'];

$config['site_url']   = $base_url;
$config['site_index'] = '';
$config['cp_url']     = $base_url . '/' . $system_folder . '/index.php';
```

We saw all those before, when I went over the `/app` directory. Now just fill in the domain ('eeguide.dev') and my system folder ('control'). Remove the `cp_url` variable that was already in the file because we're now using our more flexible version.

Create a Template; Move Images and Themes

If you view http://eeguide.dev at this point, it should be blank. Let's quickly create a template that we can see so we can carry on. Log into your Control Panel at http://eeguide.dev/control/. There are three default areas with links. Under "Create" click "Template" and then "Create a Template Group" **(Fig. 4.16).**

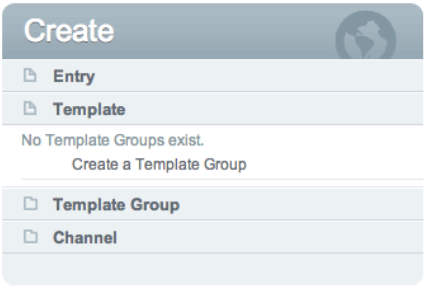


Fig. 4.16 Get started by creating a template group.

Name the template group whatever you want (I like to stick with "site") but the key here is to make sure you check the box for "Make the index template in this group your site's home page?" so that you've designated a template that the system knows is your homepage **(Fig. 4.17)**.

After creating that group, you'll be taken to the Template Management area. Here's where you could quickly set your templates to be saved as files. However, as you know we prefer to do this by setting a variable in our config file. It bears repeating here: **the config file is the most important part of our entire ExpressionEngine install, controlling most of what is going on, and overriding settings in the control panel.** We've already gone into a lot of detail about the config file when I covered the `/app` directory, so now let's see how those variables override the values in the control panel.

Click on Global Template Preferences. Note that "Save Templates as Files" is set to "No" and that "Server path to site's templates" is populated with only a forward slash.

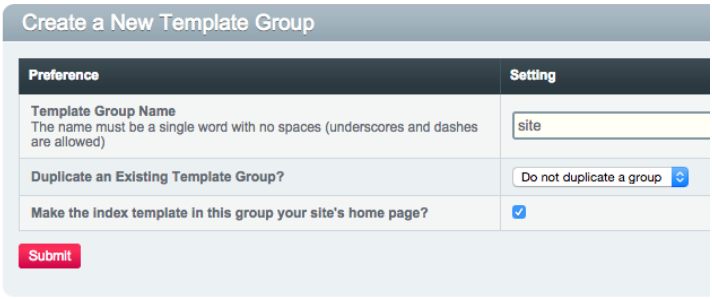


Fig. 4.17 Creating a default template group by designating this group's index template as the site home page.

We're going to update these values by using config variables that I explained earlier in Templates and Themes when I covered the env_config.php file. Paste this block into your env_config.php file, right below where we put the other items:

```
$config['save_tmpl_files']    = 'y';
$config['tmpl_file_basepath'] = $system_app_path . '/templates';
$themes_folder                = '/assets/themes/'; // Must end w/slash
$config['theme_folder_path']  = $base_path . $themes_folder;
$config['theme_folder_url']   = $base_url . $themes_folder;
$config['third_party_path']   = $system_app_path . '/add-ons';
```

Note that while we're in here we're updating the `third_party_path` as well, because we will no longer be using the default third_party directory for our add-ons. We'll instead use the aptly-named 'add-ons' directory we just created.

Now reloading our control panel page shows us that our theme is broken **(Fig. 4.18)**. No problem. Notice the `$themes_folder` path has "assets" in it. That's a folder I discussed earlier that we like to use to keep things organized…and we haven't made it yet. We'll eventually put CSS and JS directories in assets as well.

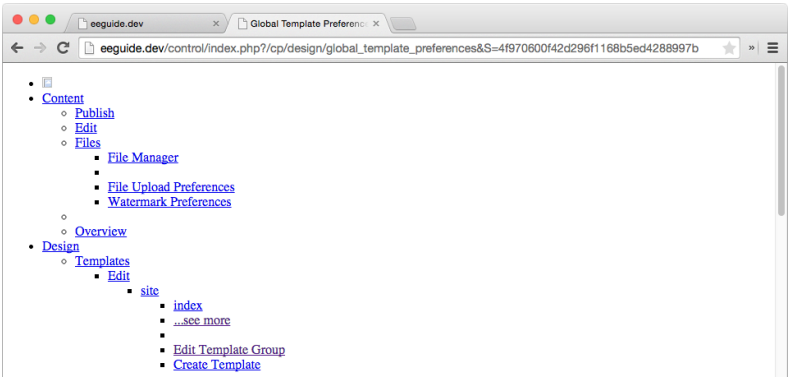


Fig. 4.18 Our control panel theme is broken at this point because we haven't yet made the /assets directory we specified in our config variable.

Go out to your public folder and create an `/assets` directory, and drag the `/themes` directory in there. While we're at it, we'll drag `/images` in there as well**(Fig. 4.19).**

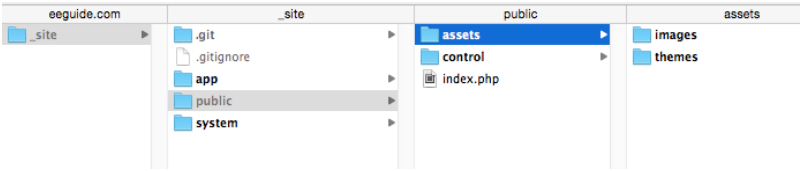


Fig. 4.19 Starting to build out our /assets directory.

Now that our template and theme variables are in our config file and we've cleaned up our public folder, let's finish creating our template(s). Go back to your Global Template Preferences and refresh the page. Note that the control panel styles are now back in place and "Save Templates as Files" is now set to "Yes" and that "Server path to site's templates" is populated with a path **(Fig. 4.20)**. In my case, that path is:

```
/Users/masuga/Sites/eeguide.com/_site/app/templates/
```

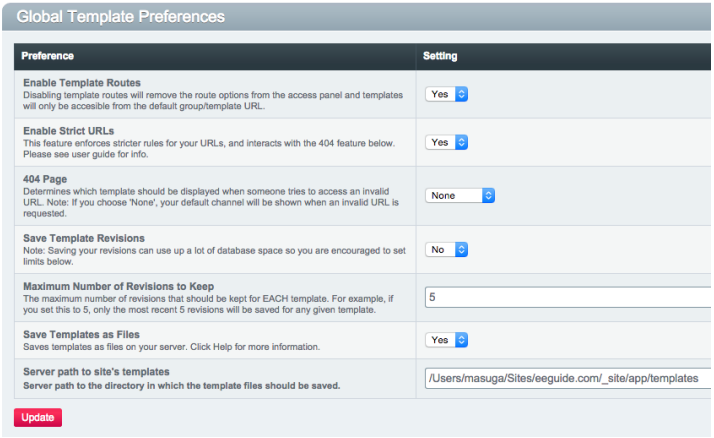


Fig. 4.20 Global Template Preferences. Our control panel themes are working again.

That isn't all we'll be doing to the config file, but this will be enough to show you how the variables there affect the settings in the control panel.
If you update the variables directly in env_config and reload this page, you'll see that they change. If you try to edit the fields here in the control panel and save, you'll see that they revert to the values you tried to change because the config file is the definitive value. If you're not aware that the value is already being set via a config file, this can be maddening (I know, I've been there)!

This is the reason I think it would be such a nice touch, on those fields that are utilizing available config overrides, to visually indicate the values are being pulled from a config file so a developer doesn't spend any extra time scratching their head wondering why the field won't update. I believe it would only enhance the user experience that much more. Maybe in an upcoming version of ExpressionEngine! Suffice it to say that config file overrides work, and we use them to set site (and add-on) values whenever we can.

At this point I'll go ahead and manually create the `/templates` directory in the `/app` directory and set the permissions on it to "777" so the system can update it.

When the folder is created, head back into the CP and navigate to Design > Templates > Template Manager and click on the index template while you're in the site template group. Type anything into the template. I typed "Hi There!" Check the box for "Save Template as File" and click "Update."

If you go back out to view the "templates" directory you created, you'll notice that it now contains subdirectories:

```
/app
  |_/templates
    |_ /default_site
      |_ /site.group
        |_ index.html
```

Your ExpressionEngine install can now communicate with this templates directory so that you can create templates directly in this directory, and EE will know that they are there **(Fig. 4.21)**.

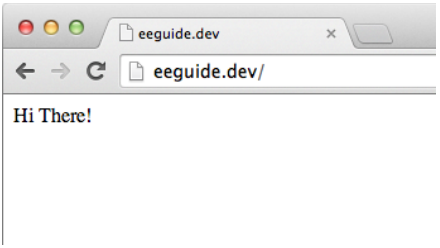


Fig. 4.21 ExpressionEngine now knows where your /templates directory is.

To test that our `/app/templates` directory is being referenced correctly, we can update our index file at `/app/templates/default_site/site.group/index.html`. Type anything, save the file, and reload it in your browser and you should see the change reflected there **(Fig. 4.22)**.

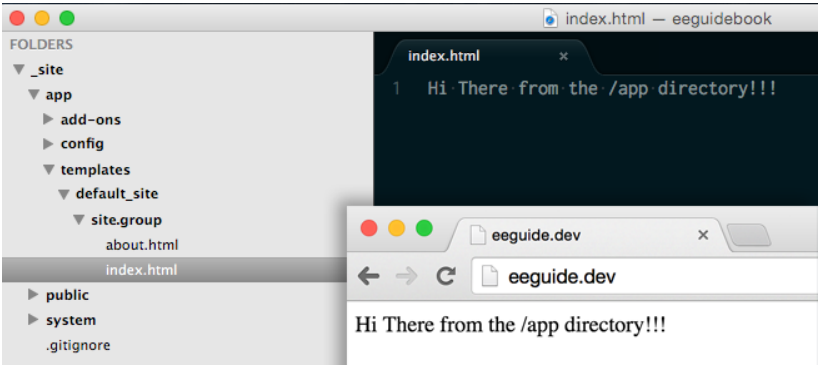


Fig. 4.22 If you visit your site in your browser, it should be displaying whatever it was that you typed.

We won't have to go through this rigmarole with subsequent templates. For example, if you were to create an "about.html" template in your "site.group" directory, you could immediately access it at http://eeguide.dev/index.php/site/about **(Fig. 4.23)**.

Fig. 4.23 ExpressionEngine will now recognize new templates created directly as files.

Note the "index.php" is still required in the URL at this point, because we haven't removed it with an .htaccess rule yet. There's no need to mess with that right now because we're still keeping things simple!

Delete Unused Themes

ExpressionEngine comes with a few theme directories we remove because we will never use them, so there's no need to have them sitting in our repository.

First we remove the two "agile_records" theme directories:

```
/public/assets/themes/profile_themes/agile_records
```

```
/public/assets/themes/site_themes/agile_records
```

Then we delete the two Wiki theme directories:

```
/public/assets/themes/wiki_themes/azure
/public/assets/themes/wiki_themes/default
```

Of course, any time we upgrade ExpressionEngine, with Git we can see that these directories try to make their way back into the repo because they come with core, but we just nuke them again before committing our upgraded files and we're on our merry way.

Our base setup is in decent shape now. This all might feel tedious to you, moving all these things around, setting up config variables, but once you've done it one time, it's a breeze to get sites started. It's also a lot easier if you have a default ExpressionEngine install to start from every time.

Now we have enough to go on to get this site into version control. First I'll cover Git in general, and then we'll get into the specifics of versioning an ExpressionEngine site.

5 Bringing In Version Control

This guide has a goal of getting you to use Git with ExpressionEngine. By now we have a basic site structure set up, so we want to get it into a repo to start tracking of our modifications.

Let's get into why you'd want to use it, what it is, basic commands, tips and tricks, and software. Then let's get the ExpressionEngine site we set up into version control!

Why Use Git?

I've used Subversion in the past, and there are other version control systems out there such as Mercurial, but Git really is one of the easiest to use, and it has had widespread adoption, especially in ExpressionEngine circles. I wouldn't have taken the time to learn it and use it if it wasn't going to be relatively easy to deal with and ultimately benefit me—and this was years ago as a freelancer. Now that we're a small team working on numerous projects simultaneously, it makes even more sense. Earlier I outlined the general points that explain why using version control is a great idea, and here are some of points specific to Git:

- **Git is relatively easy to learn and use.** Some claim it's harder to learn because there are more commands than other version control systems, but for everyday use, the number of commands you need is totally manageable. Version control isn't rocket science.
- **Work at your pace—even offline.** You can commit work, view history, and make branches all without having to be connected. You have a full repository with you all the time. If you have a part of a project that is taking a long time to finish, you could commit to it for weeks before anyone else ever even knows you're working on

it. As a perfect example, right now as I type this I need to make a couple small changes to masugadesign.com, but the repo where our site's code is hosted (Bitbucket) is having (unusual) connectivity issues due to a database upgrade. I can still commit my changes locally. When Bitbucket's service is back to normal, I'll "push" my work out to the main repo at that time.

- **Git is cleaner than Subversion (SVN).** Subversion litters all directories with .svn folders, but Git only uses a single .git directory at the top level of your repo.
- **Many ExpressionEngine developers use Git**. By using Git you're diving into the version control system that many EE devs already use. Many keep their add-on code in repositories at GitHub or Bitbucket or Beanstalk.
- **Branching and merging are easy.** It's easy to make a new "branch" to embark on site updates, and this helps make your development safer, because you're not working on your master branch. You're basically siloing your work into topic branches and only introducing your changes to your main codebase if and when you decide to merge it in.

There are other sources that do a great job explaining the advantages of Git and how to use it. For example, see Git Tower's "Learn Version Control with Git" article[85].

Consider this basic scenario: you inherit an ExpressionEngine site from another developer (or string of developers) and your new client is counting on you to fix some bugs that they just haven't been able to solve. "Well, our first developer had feature X working. It was the second developer who added this other functionality. I think that was when the functionality we need stopped working. Or maybe it was the other way around."

That's fun stuff, and really happened to us when we recently took over development of a site. Nothing was versioned, so there was no history. The client was having a problem when it looked like MX Cloner, an add-on for cloning entries, appeared to no longer work, and they wanted that fixed (understandably). I did a little research and found that maybe, just maybe, MX Cloner and CartThrob might not play well together. But then at some point, someone threw Zenbu into the mix (which I happened to know would

stop MX Cloner from working unless an additional MX Cloner/Zenbu extension is installed—and that wasn't installed).

Seemed like a relatively easy problem to fix, but when exactly did things go wrong? Who knows if MX Cloner was or wasn't working with the version of CartThrob installed on the site? When was CartThrob updated? When was Zenbu initially installed? Was it ever updated? There is no commit history for this site, so…*we'll never know.* One of our first steps was to get that site under version control so we could document the changes we were going to make.

Another example of why Git makes development better: you don't have failed experiments and other junk lying around. On another site we inherited, we were (again) tasked with optimizing it. It was obvious that the previous developer did not use version control. Entire template groups were duplicated: the "search.group" that contained 13 templates was duplicated as "seach_new.group" with what appeared to be the same 13 templates in it, including 4 new templates. Which group is in use? Are both? Can we trust that all of the templates in either of these groups are even being used? What about the curiously titled "wtf.html" (actual template!)? Was that an experiment or some sort of Easter-egg?

You too will find that non version-controlled sites can potentially contain tons of junk like this. This is not an issue on version controlled sites because experiments would take place on topic branches, and be nowhere near the production codebase, unless they had been merged in and the code in them was being used.

I'll give you one more example of how useful it can be to have your work committed to a repo. Over a year ago, a client had us hide a section of a site that wasn't being actively used. Hiding that area affected 5 templates, which I did on a single commit, while putting a message in the commit to the effect of "Hid Section X at client's request." Just today, over a year later, the client said we need that section there, ASAP! Now, I could have gone through all the templates and tried to figure out why that section wasn't showing (hey, it had been a year and I'd barely touched that site—I had no idea why that area wasn't in there.) The easy thing to do, though, was search the repo for the term "Section X." That led me to find the commit I made over a year ago.

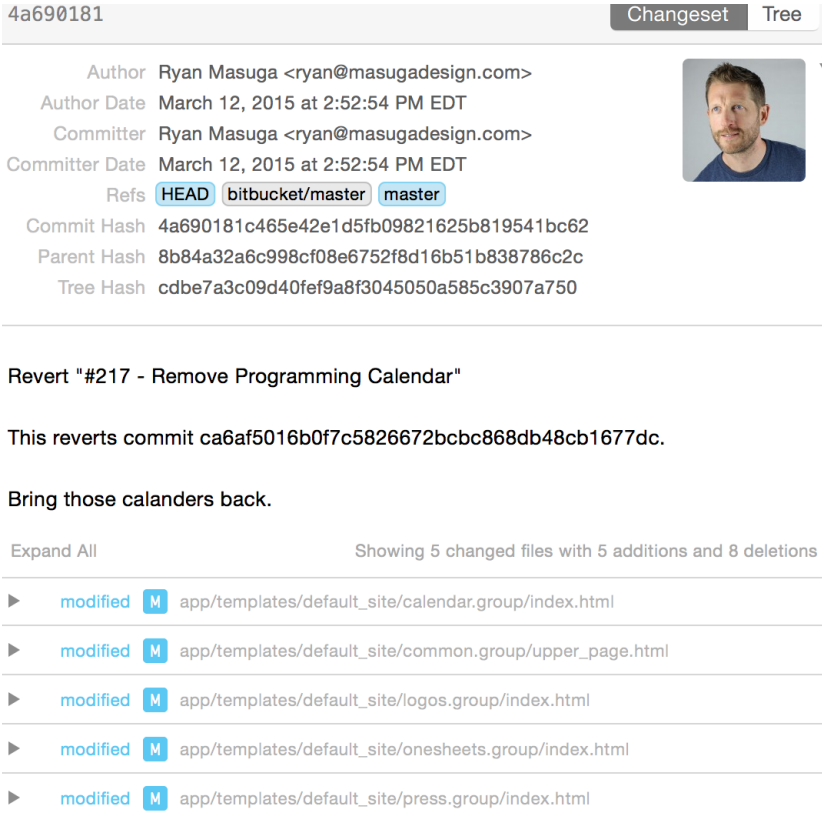


Fig. 5.1 Reverting changes to five files that were made over a year ago.

I reviewed the changes in it and saw that I had commented out or removed a few lines and set up a redirect—all of which spanned five templates. Instead of trying to carefully undo the work there (what if it had been a change that affected 80 templates?) I right clicked on the year-old commit in Tower and selected "Revert 'xxxxxxxx'…" which reverted the changes I made in that commit[86]. Then I committed the revert with a note saying the client wanted to show that area now **(Fig. 5.1)**. I was done in a matter of a couple minutes, and I have a full record and history of what went on. How handy is that?

You don't have to get into real "power usage" to see benefits from using Git on your projects. I'll state that there are any number of people out there that can use Git more effectively than I can, but that only helps strengthen my point that it's pretty easy to use in your development workflow. If I can

understand it well enough, then you can too. These days the GUI apps for Git are so nice that you almost never have to get into Git via the command line (Terminal). That being the case, this guide won't delve into those depths. I'll keep it relatively simple so that you get the overarching idea.

What is Git?

At its core, Git[87] is a distributed version control system that is designed to be efficient and fast.

> Every Git clone is a full- fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Branching and merging are fast and easy to do.[88]

In language your client can understand: at the very least, the codebase of their site is redundantly backed-up. From the Git site:

> This means that even if you're using a centralized workflow, every user essentially has a full backup of the main server. Each of these copies could be pushed up to replace the main server in the event of a crash or corruption. In effect, there is no single point of failure with Git unless there is only a single copy of the repository.[89]

That sounds to me like a big selling point in and of itself. It's also easy to see who changed what and when—even if it's only you making the changes.

Atlassian has a well designed and informative Git Tutorials[90] section with tutorials and clear information about different Git workflows (ours most closely resembles the Feature Branch Workflow[91], and I talk about that more in depth later on).

Install Git

On a Mac, installing Git is straightforward. Git is actually included on OS X, but it's typically an older version. You can download the latest release for your operating system from the Git Downloads[92] page.

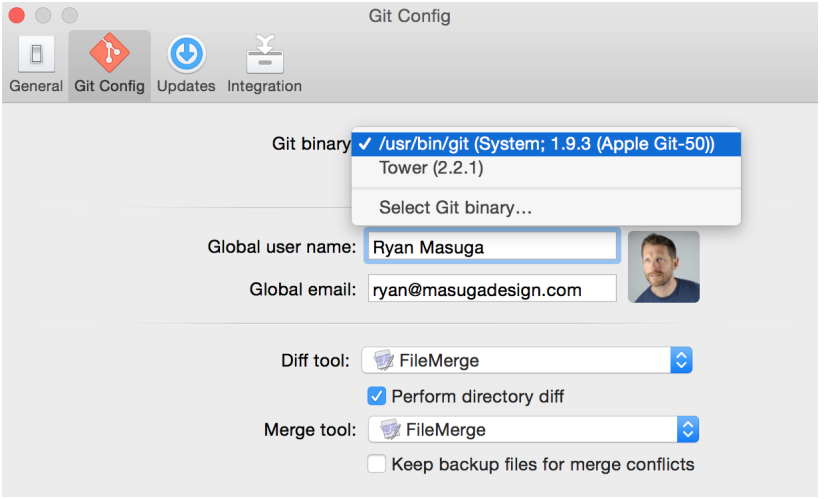


Fig. 5.2 Git is available via Tower app.

I believe you might not even need to install Git that way any longer if you use Tower because Tower includes Git. I haven't used Tower's Git before—I happened to notice it was available in the Git binary dropdown in the settings (**Fig. 5.2**).

It's easy for Windows users to find and download Git, too (though I can't speak to installing it on Windows). Once you have it installed, we're ready to get our ExpressionEngine site into version control.

Basic Git Workflow

There are formal Git workflows like Gitflow[93], but our relatively small team uses a pretty basic master-development-topic branch setup (as mentioned previously, it's close to a Feature Branch Workflow). That is to say we do most of our work on dev or topic branches, and that work is then merged or

cherry-picked into the master branch. We try our best to not work on master directly (which is the branch that represents our "production" site).

Git workflow in 2 minutes or less:

- **git clone / git init**—start off a project by either cloning an existing ExpressionEngine project or set up a new EE site and initialize a repo
- **git commit**—make changes to your repository
- **git push**—push those changes our to a "remote," or central repository
- **git pull**—pull changes others have made from a central repository
- **git branch**—make new branches and commit changes there
- **git merge**—merge changes from branches into development or master branches

Then it's just more committing, pushing, pulling, rinsing and repeating on project after project for client after client until you find yourself happily retired, sipping a margarita from the deck of your beach house while reading the…OK, I'm getting ahead of myself.

Basic Git Commands

Here are a few basic commands we use 90% of the time, with a basic "layman's" description of what they do[94]. An app like Tower makes these operations trivial:

- **commit**[95]—store some work you just did, with a message about it
- **push**[96]—push your recent work (commits) out to a central location, where others can then download and update it
- **pull**[97]—grab changes from a remote repository into your current branch
- **branch**[98]—start new work on a "copy" of your files, as taken from a certain point of another branch
- **merge**[99]—join development histories together (mash the work you did on a branch back into another branch, like master)

- **cherry-pick**[100]—apply changes made in existing commits; sort of like merge, but more granular

We might do the occasional stash[101] and a couple others, but these six are the main players.

Software for Git

We're a Mac-based shop, so we can't speak for other operating systems. As mentioned way back in Chapter 1, we've used Tower App[102] almost since it came out. I think it's great and worth your money—and I don't even have an affiliate link.

Tower might be the most full-featured GUI app for Git, but Atlassian offers Sourcetree[103] which also looks very polished. We haven't used Sourcetree so I can't speak to it.

If you're more comfortable in an app than you are in the command-line, either of these apps will do for you. Tower is easy to use, particularly for the common everyday Git commands that you'll use.

There are other GUI clients for Git, and you can find a list of them for various OS's on Git's GUI Clients page[104].

Versioning an ExpressionEngine Site

Now that you have an idea of what Git is good for, and we've whipped together a quick local ExpressionEngine site, it's time to add version control to the mix.

We're not going to get to wild with advanced Git usage. I'm giving you enough to understand how we work and what we do at Masuga Design. I've never needed to use submodules and as of this writing I've never used "git bisect." We, of course, could stand to learn more just like anyone else.

The hope is that this gives you enough to go on when using Git version control with your ExpressionEngine sites.

Make a Local Repository

Let's start by going into Tower to make our repository.

Fig. 5.3 Creating a new local repository in Tower app.

Click the plus sign at the lower left of Tower's Repositories view and select "Create New Local Repository…" **(Fig. 5.3)**. Navigate to the directory that holds your site files (in my example here, `/Users/masuga/Sites/eeguide.com/_site`) and click "Create Repository." You'll now have a generically-named "_site" repo in Tower, so rename it by right-clicking on it in the left-hand list and selecting "Rename…" Then double-click it to go into the repo. Your setup should look very close to **Fig. 5.4**.

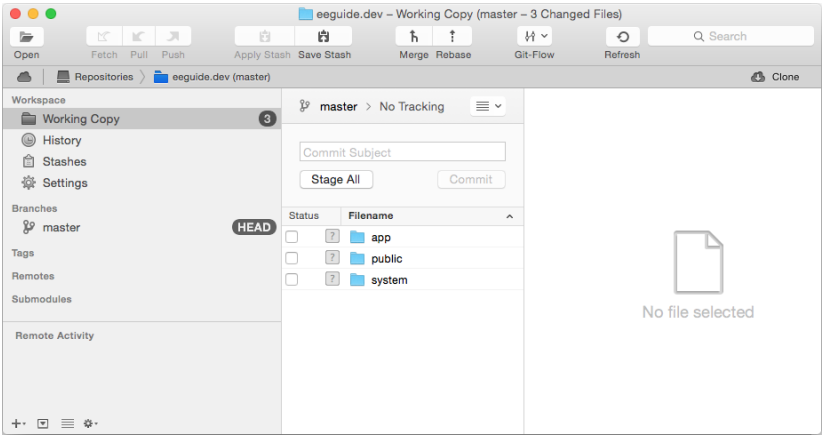


Fig. 5.4 Our initial setup should look much like this after creating our repository.

Tower sees our three directories, and has the "what the heck is this, it's not in version control" gray question mark icon next to each. Before we commit these, we want to make sure to keep some files out of Git, so we need to set up some .gitignore rules first. If you go back out to this directory in the file system, you'll see a hidden ".git" directory **(Fig. 5.5)**.

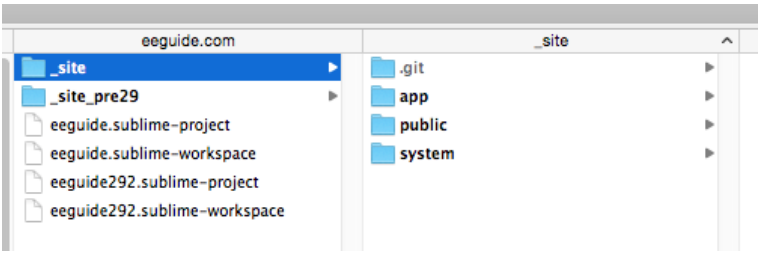


Fig. 5.5 There is now a hidden ".git" directory in our site.

My screenshots are taken using Path Finder[105], which I use in place of the Finder, and I have hidden files and folders displayed by default (their names appear slightly grayed out). If you don't use Path Finder, there are small scripts[106] you can download to toggle invisible files on and off in the Mac Finder so you will have an easier time seeing hidden files like .gitignore, .htaccess, and your .git directory. Showing invisible files by default also makes it easier to see your ExpressionEngine Hidden Templates[107] if you elect to prefix your hidden templates with a period rather than an underscore (the underscore is what we use to avoid this problem altogether).

You could alternatively make the local repo in the command line with "git init."

```
cd /path/to/my/repo
git init
```

It's pretty much that easy. Now we need to ensure that we tell Git what we want to ignore, using a .gitignore file.

Create a .gitignore File

A .gitignore[108] file holds info about every file or directory you want Git to…ignore. This might include environment-specific files, files within cache directories, and files within directories that hold (or will hold) user-uploaded content. The .gitignore file will be housed in the root of our project.

I found an interesting online tool at gitignore.io[109] that generates starter gitignore files for various systems, and it does have one for ExpressionEngine. What it creates is very basic and may not be worth the trouble, but it's an interesting idea.

Our site is pretty basic at this point, so our .gitignore file won't be extensive. We'll start out with a few real basics. The contents of our file will be:

```
# Site: EE Guide

public/.htaccess

system/expressionengine/cache/*
!system/expressionengine/cache/index.html

app/config/env_config.php
app/config/env_database.php
```

A bit of explanation:

.htaccess: The contents of an .htaccess file can be specific to the environment, so we don't want to version it. For example, our production server might have tons of 301 redirects that we don't need locally or on a development server. I've seen setups where the .htaccess file is tracked in Git, but I've never seen the advantage.

System Cache Directory: We don't want to version the ExpressionEngine cache folder, either. However, we still want this directory to get created when someone else clones the repo, so we need to have something in it (this is how Git works—it ignores empty directories). Make sure there is an index.html file in the cache directory that says "Directory Access not allowed" (or similar).

Our first line says ignore the cache directory, but note that the very next line has an exclamation point at the beginning. Those two lines together ensure that Git will ignore all the items in the directory *except* the index.html, so when the site is cloned, at least the directory structure will be preserved because it will bring in the index.html file, but none of the undesired files will come with it.

The process is the same for all the default image directories. For those, you would add rules like the following (your paths may vary):

```
/assets/images/captchas/*
!/assets/images/captchas/index.html

/assets/images/member_photos/*
!/assets/images/member_photos/index.html
```

There are other .gitignore strategies for ExpressionEngine, and you can read about some of them on the ExpressionEngine Stack Exchange[110].

The .gitignore file is pretty important. We want to make sure we're ignoring the right things early on, because having large or unnecessary files in your repo and removing them later keeps them in the history, making your repository larger than it needs to be.

We took over two sites recently, and neither of them had a .gitignore set up well. One of those sites is a very small site, and I was happy to see it was at least in version control when we first looked at the project, but there was only one line in it that ignored the ".esproj" file (the developer must use the Espresso[111] app). Unfortunately, that meant the cache directories and any user-uploaded content directories are included in the repo. This can lead to issues later on, such as we are seeing with the *other* site we took over…

The second project is a very large site with tons of content. For some reason, the original developers did the exact same thing: allowed the contents of the user upload directories to be in the repo, as well as storing SQL dumps of the database right along with the site files (and this database is *huge*). Now, a side note: why the living hell would you keep a .sql file in your repo so that it's deployed to the production site, which might mean you also have to add lines to your .htaccess file to specifically disallow access to it? I'm a huge fan

of not having anything sit on your production site that doesn't need to be there, or potentially create any extra security risks.

Anyway, all of that stuff now lives in the repo, and even if we remove them from being tracked by the repo (but physically keep the files there, which you do by using `git rm --cached`[112]), the repo will be artificially bloated because those things are in the history. The repo I'm talking about is 1.8GB in size! **(Fig. 5.6)** It's currently stored at Bitbucket which has a 2GB hard limit on Git repos. We see warnings all the time about the size of this site's repository. At some point we're going to have to "start over" by making a new repo with a .gitignore file that ignores all the directories it should have probably ignored in the first place in order to get this repo back to a reasonable size.

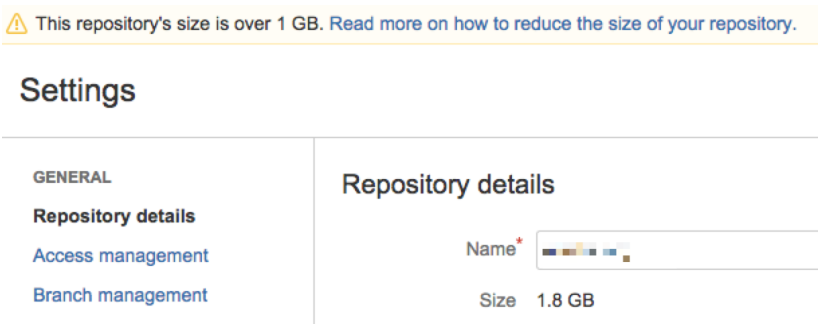


Fig. 5.6 This repo is a 1.8 GB monster. I'm sure there are repos that are legitimately this large, but this one certainly didn't need to be. Probably time to start over, history be damned.

Git is not a *backup* solution—it's a versioning solution. For ExpressionEngine, it's great for your template files, core files, and that sort of thing, but keep the content directories out of there, and in my opinion, keep your .sql backups out of there, too!

Initial Commit

It's time to get this site into Git so we can start tracking our updates. We've moved stuff around into a logical setup, removed numerous files that we won't need, and we've added our extras, like the .gitignore file, and created the repo.

If we're at the same point, your Tower window should look like **Fig. 5.7**, with three directories and one .gitignore file ready to go.

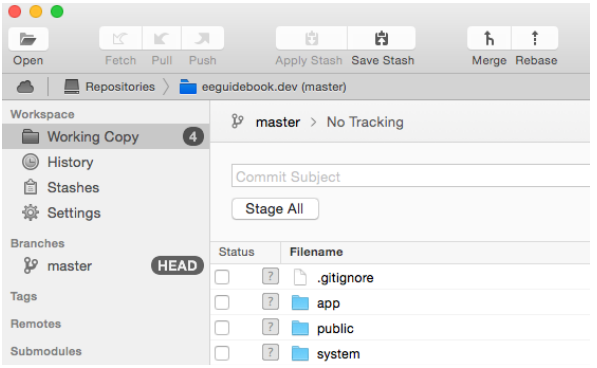


Fig. 5.7 Our .gitignore file added to our original three directories.

Because this is our initial commit and we're adding all of ExpressionEngine's core files, it will be bigger than usual. Clicking "Stage All" shows me that there are 1,707 files to commit—and remember, that's after removing all those theme directories we're not going to use, such as the Agile Records theme.

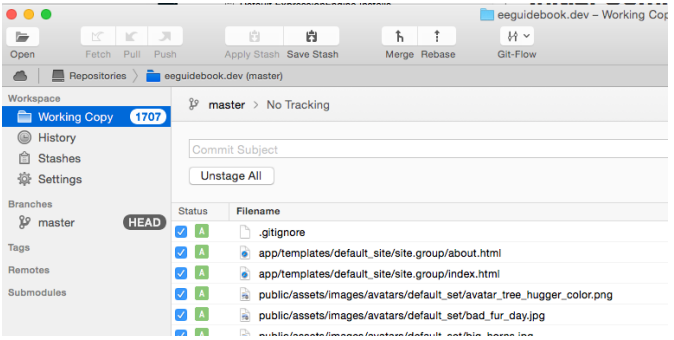


Fig. 5.8 Our ExpressionEngine site is ready for the initial commit.

You'll now see a mile-long list of files with a green "A" next to them, meaning you're adding those files to the repo. It's a great idea, especially on larger commits, to skim through and see if there is anything in the commit that should be unchecked. Sometimes I will have forgotten to add an item or a

directory to the .gitignore file. You can uncheck those items at this point, and then after you've made your initial commit, you can update your .gitignore file again, commit your .gitignore update, and the files you didn't want to track will disappear from the staging list view. Scrolling through our current list, I see that everything checks out and that I want to add all the files I've checked.

Click into the "Commit Subject" field, and type something meaningful like "Initial Commit" or "All systems go" if you're feeling frisky and click "Commit."

Everything disappeared! Not really. You don't have any changed files in your working copy any longer—you're "clean." Click on "History" on the left hand side and you'll see your single commit there, with further details about all the things that changed in the column on the far right **(Fig 5.9)**.

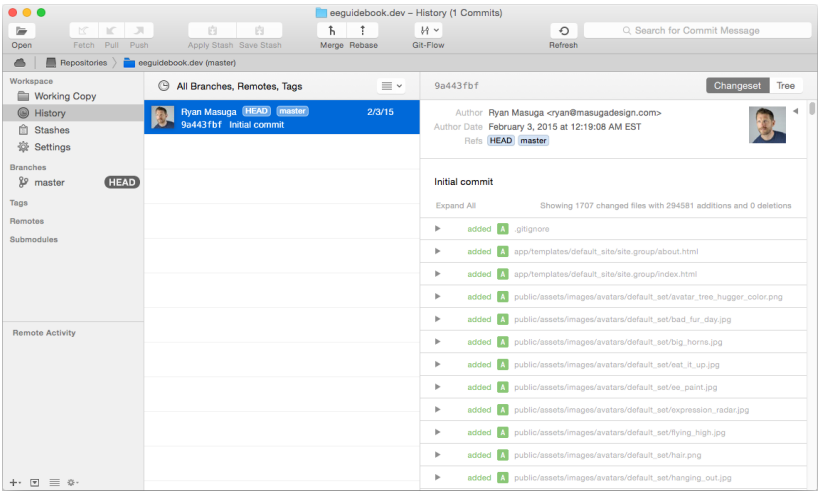


Fig. 5.9 We've made our commit, and now have a history.

A lot of the simple Git workflow is like this: you make changes to files in your working tree, you commit those changes, and then you ultimately push those changed files somewhere. Right now all the changes and updates you're making are in your local working copy only—no one else knows of them.

The next step will be to establish a remote repository, which is sort of a centralized repo, where we will push our changes to, and from where we will pull changes others have made. If you're working solo, you'll do a lot more pushing to your remote repos than pulling from them.

Setting Up a Remote Repo

There are a number of repository hosts, but we'll use Bitbucket in this example, if only because Masuga Design keeps all of our private Git repositories there. The experience would be similar at Github or Codebase or Beanstalk, or any other code-hosting service.

Adding a Remote

One of the cool things about Git is that you can do all your work locally, commit it, undo it, and so on before you ever push it "out there" to the central repository. The first step to getting your work out there is adding a remote to where you can push it. Wherever you add the remote, you're going to do a couple common steps: tell your local working copy where it is, and make the initial push to it.

Our first step is to create the repo at our host of choice. In Bitbucket, just create a new repository, which is a one-step process **(Fig. 5.10)**.

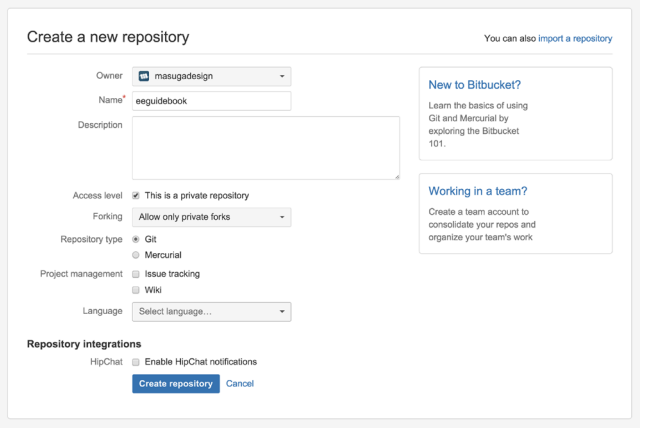


Fig. 5.10 First create a repo at your host of choice so we can set up our remote.

Once it exists, there are a couple ways to easily hook up your local working copy to your remote: via the command line or in Tower.

Add a Remote via Command Line

I frequently do this, and probably because it's more of an old habit than anything. Earlier versions of Tower used to take a while when doing anything with thousands of files, but I don't think the app has that issue any longer.

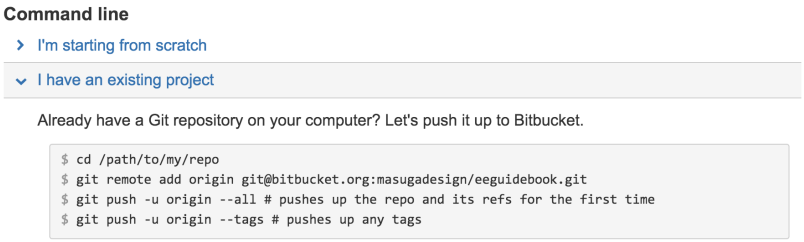


Fig. 5.11 Bitbucket provides clear command line instructions.

When you've created the repo at Bitbucket, the next screen tells you what we need to do to hook our existing project to it **(Fig. 5.11)**.

```
cd /path/to/my/repo
git remote add origin git@bitbucket.org:masugadesign/eeguidebook.git
git push -u origin --all # pushes up the repo and its refs for the first
time
git push -u origin --tags # pushes up any tags
```

This is easy to do and blazing fast. Open Terminal, `cd` (change directory) to where the local repo is sitting:

```
~: cd Sites/eeguide.com/_site/
```

Then we push our work to the remote, which by default is named "origin." This is as good a name as any, although you could change it if you wished. It's simply a convention to use "origin" for the primary centralized repository, but there's nothing special about the name. A repo can even have multiple remotes, which we've had to do on occasion, but we don't need to go into that.

We don't need to worry about pushing tags, but I included that line here because by default that is included in the instructions Bitbucket displays after creating a new repository.

Add a Remote in Tower

It's a breeze to add your remote in Tower, too. In the left sidebar, right click on the word "Remotes" to get the context menu and then select "Add Remote Repository…" **(Fig 5.12)**.

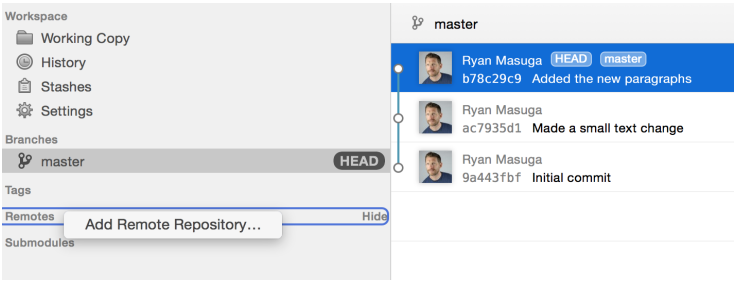


Fig. 5.12 Adding a remote in Tower from the left sidebar.

In the next step, enter the name of your remote and then the repository URL **(Fig 5.13)**. Filling in those two fields reveals a couple more fields where you confirm your Bitbucket account and your SSH key (which I've previously added to my Bitbucket account).

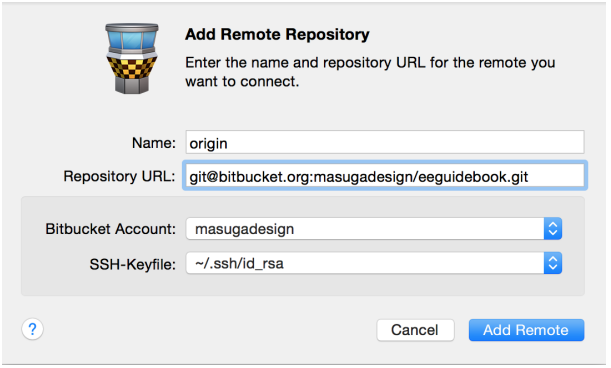


Fig. 5.13 Finalize adding a remote in Tower.

Clicking "Add Remote" will make the remote pop into the left sidebar.

Either way will hook our local working copy up to the empty, centralized repository. We'll be able to push our updates to the repo, and pull updates from it as well. But first, let's make a few changes.

Making Changes

Now we'll make a simple update and commit that so you can see how it will appear in your working tree. Then we'll commit our change as the second commit.

First, I'll open up the site's index page at `/app/templates/default_site/site.group/index.html` and add a line of text **(Fig. 5.14)**.

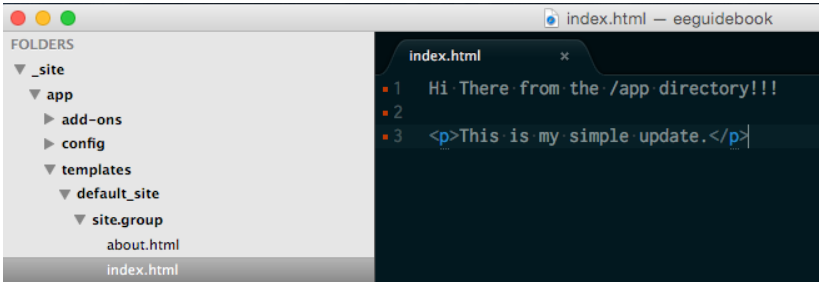


Fig. 5.14 Making a simple change to the homepage template.

If I save that file and go back into Tower, I'll have a numeric indicator "1" next to Working Copy in the left sidebar because now there is one file altered that the repo is going to need to know what to do with.

Clicking on "Working Copy" in the left sidebar shows me that the file I worked on has been Modified (indicated by the blue "M" next to the filename in the center column) which also implies that this file isn't new, but is already in our repository, being tracked by Git **(Fig. 5.15)**.

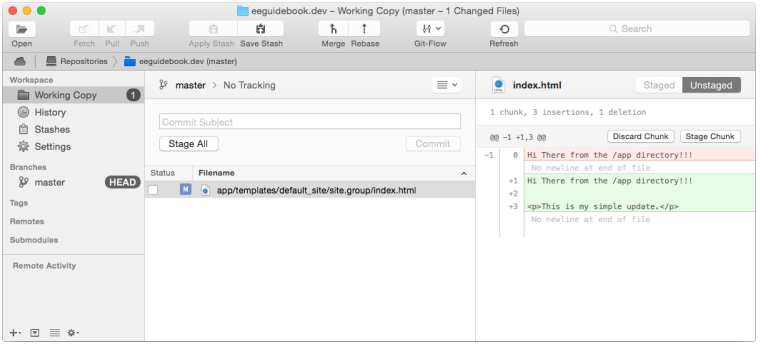


Fig. 5.15 We see our modified file, and the lines that have been changed.

In the column on the far right, I see specifically what lines have been changed. The line highlighted in red shows what has been changed or removed, and the lines highlighted in green show the change.

To commit this change, I check the Status box to "stage" my change, type a commit subject, and click submit. Going back into the "History" tab, I can see we now have two commits on our project **(Fig. 5.16)**.

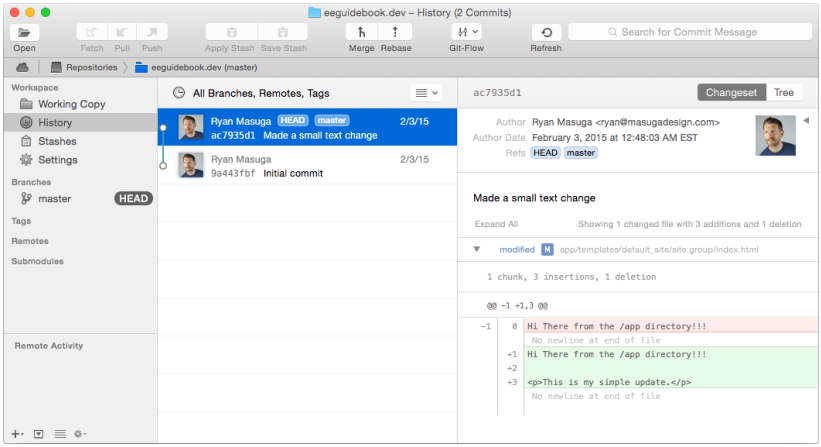


Fig. 5.16 We now have two commits in our history.

Stage Changes in Pieces

One thing Tower makes really easy is the ability to stage parts of a single file in different commits—down to the "chunk" or even the line. This comes in handy if you're doing a lot of work on file, and you want to commit that work in smaller groups. Or, for example, let's say you made a site-wide change to all your templates and want to do that as a commit, but one of those affected templates already had other work done on it that you're not ready to commit. You would be able to separate the work out on that example template and commit it separately. Let's try that now.

I'll add a few more lines to our index page **(Fig. 5.17)**.

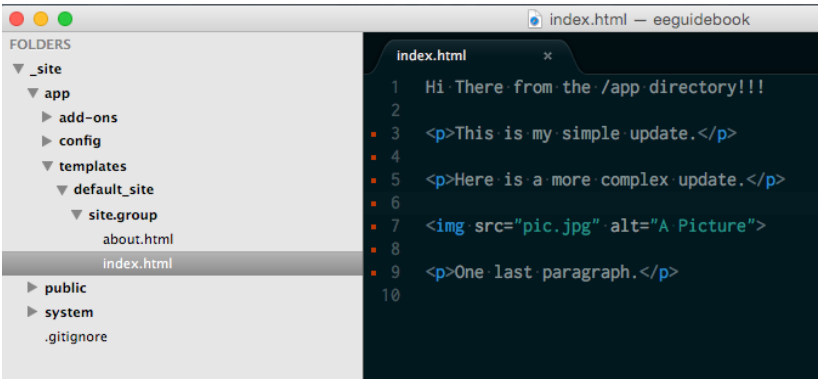


Fig. 5.17 Adding a few more changes to our index file.

In this (very simple) example, I want to commit the text changes I made, but not the image, because I'm still working on that.

In Tower, you have the ability to "Stage Chunk" or to select individual lines to be staged. In this case, I don't want to stage the whole chunk, just everything but the image line. When I'm next to any of the green highlighted lines, I can hover over the line numbers to the right and click them. The

"Stage Chunk" button turns into a "Stage Lines" button. Here I select the 6 lines I'm good with, and click "Stage Lines" **(Fig. 5.18).**

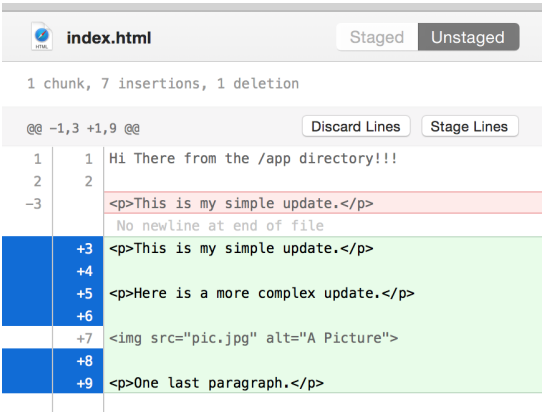


Fig. 5.18 Selecting particular lines to stage and commit.

You'll notice that the blue icon for a modified file is now two icons and there is a minus sign rather than a check. That indicates that part of the files has been staged, but that there are other modifications to it **(Fig. 5.19)**.

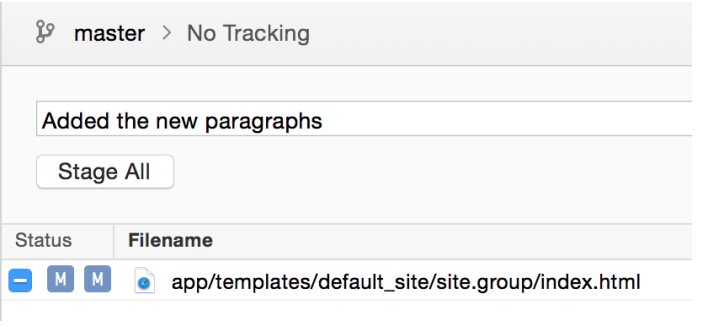


Fig. 5.19 Tower indicating that part of the file has been modified and staged.

I add my commit message, and commit the work.

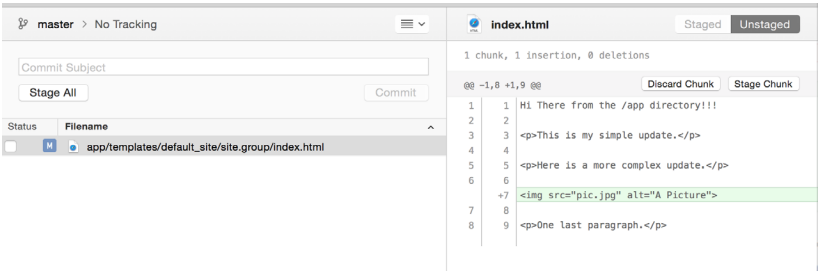


Fig. 5.20 The image line is all that is left to commit.

Now I'm left with a single blue "M" indicator and you can see that the image line is the only one not committed, and it's still highlighted in green **(Fig 5.20)**. On second thought, I don't want an image there. I can right-click on the filename in the middle column and select "Discard Local Changes…" and our Working Copy is back to a nice, clean state **(Fig. 5.21)**.

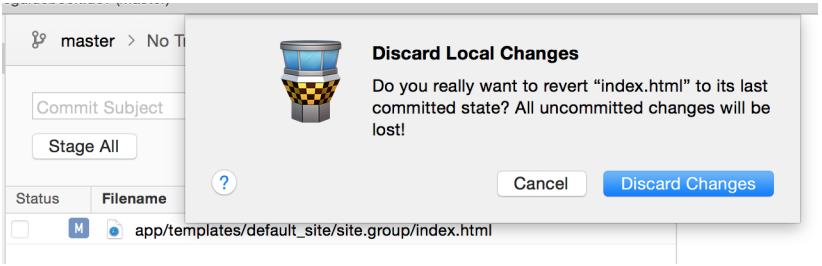


Fig. 5.21 I discard the changes that I don't want to commit.

Now our History shows three commits on our master branch (which is also our only branch right now). Again, any and all commits we've made so far live only in the local copy of our repository, on our machine. No one else knows of these changes.

Now that we've made a couple tweaks, let's get this work pushed out to the remote repository.

Pushing and Pulling From the Remote Repository

We now have an ExpressionEngine site running locally, we've made an initial commit, we've set up a remote repository, and we've made a couple changes locally already. The next step is to push our files (publish our branch) to the remote repo.

If you're doing this via the command line, you `cd` to the directory where your local working copy is and then use the `push` command like so:

```
cd /path/to/my/repo
git push -u origin --all
```

If you're doing this in Tower, click the "Push" button in the top toolbar and you'll be presented with a "Publish Branch" dialog box. Everything is already filled in the way we want it, so click "Publish Branch" and our files will be hurled across the interwebs to land safely in our remote repo **(Fig. 5.22)**.

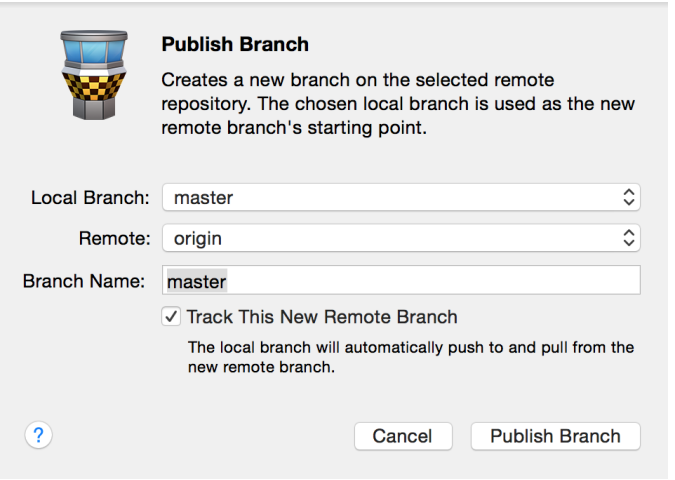


Fig. 5.22 Push to our remote repository in Tower.

Once that's done, notice where we are. If we compare Tower to Bitbucket to Terminal, you'll see everything looks the same in all three **(Fig. 5-23-5.25)**.

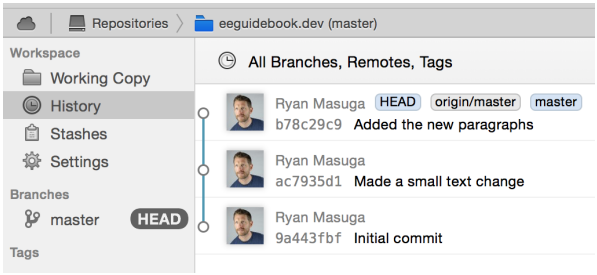


Fig. 5.23 Our three commits as shown in Tower.

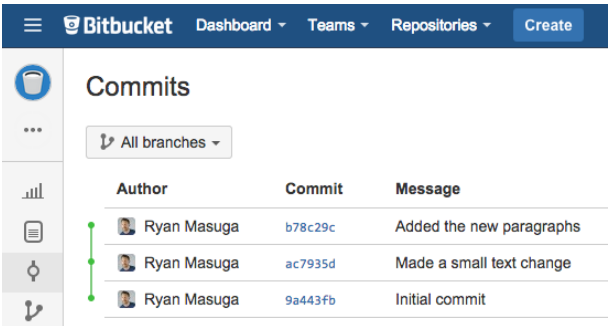


Fig. 5.24 Our three commits as shown in the remote repo at Bitbucket.

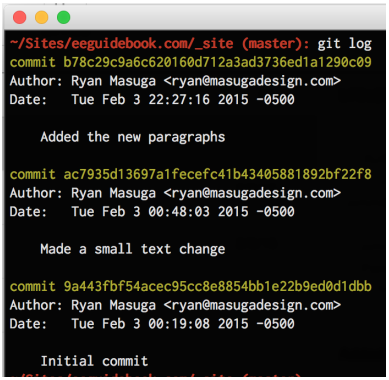


Fig. 5.25 Our three commits as they appear in the command line.

In Tower, notice the small tags in the commit: HEAD, origin/master, and master. Our current working HEAD is the latest commit, and our master branch (local) and origin/master (remote) are at the same point.

Now that we've published our branch, it is essentially "backed up" off site, and if we wanted to share our work with anyone else, they would be able to track this branch as well. If any team members or contractors have pushed to that remote repo, we would be able to pull their work from there as well.

Branch Strategy

Up to this point, our work has been done directly on the master branch, which, for us, means the "production" code. Ideally, our flow would be to *never* work directly on the master branch, but to work on a dev branch first, or better yet, work on topic branches which then get merged into development for review, and then get merged into the master branch when they're finally ready to go live.

That is the extent of what we've needed at Masuga Design—we've never had a team larger than five people at any time, and only 2-3 of those people on the same project at once. I know other shops have development, staging, production, and who knows what other branches, as well as far more complex deployment strategies, but we don't add layers or overcomplicate anything if we don't have to. If we don't "need" a staging server in addition to a development server, then we are certainly not going to introduce one "just because."

If you're a freelancer, you might find it easy enough to work right on a development branch without needing to involve topic branches. In the past, I've worked on small enough sites by myself that I didn't even feel the need to keep a development branch. I worked directly on master. The method and flow you ultimately end up with will depend on your team and your clients and what works best for all involved.

As I briefly touched on earlier, our Git workflow most closely resembles a Feature Branch Workflow as outlined on Atlassian's *Getting Git Right*[113] page. We have a master branch that represents the production site, a development branch that is the dev site (where the client typically reviews

updates before they go live) and topic branches, which is where most of the actual work takes place. Everyone has a feature branch or two going on for most projects.

The Feature Branch Workflow works pretty well for us, but even a small team can sometimes start tripping over one another with merge conflicts when too many features are going on at once and they need to find their way back into a common branch. We currently have a project with three active topic branches and all three of them do something different to our header template. Getting all those changes merged back into the site without wiping out each others' work will come down to communication. In any case, we have something that works, but I'm always open to learn new and better ways to improve our version control workflow!

Master/Development Branches

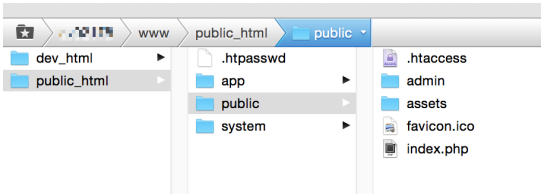


Fig. 5.26 The development site living alongside (not in) the production site.

A typical setup for us is to have a development site sitting alongside the production site. In **Fig. 5.26**, we're looking at a typical client site on an Arcustech server. We probably should have changed the name of our "admin" directory to something a little harder to guess, but we do have htpasswd authentication on it. The development site is accessed at a subdomain like "dev.theclienturl.com" and the files for that live *outside* the main site.

Many times when you're setting up your own subdomains on a server that has cPanel, you can change the default location of the subdomains you create. If you don't have that control, ask the host to set it up for you so that at least your development subdomain (or any other subdomains you'll be deploying to) is living outside of the main site. This makes it easy to set things up with a deployment service like DeployHQ so that your

development branch is deployed directly to dev_html, and your master branch is deployed to public_html. They're separated, and you're not deploying one "inside" the other and potentially messing things up in a bad way.

To create our initial development branch (in Tower) we make sure our master branch is up-to-date and then click on the word "master" and drag it on top of the word "Branches" in the left sidebar. You'll see a label pop up that reads "New Branch." Fill out the Name field, and you now have a development branch.

At this point, your development branch still exists only locally because you haven't published it to your remote (origin) yet. To do that, right click on it in the left sidebar and select "Publish "development"…" **(Fig. 5.27)**

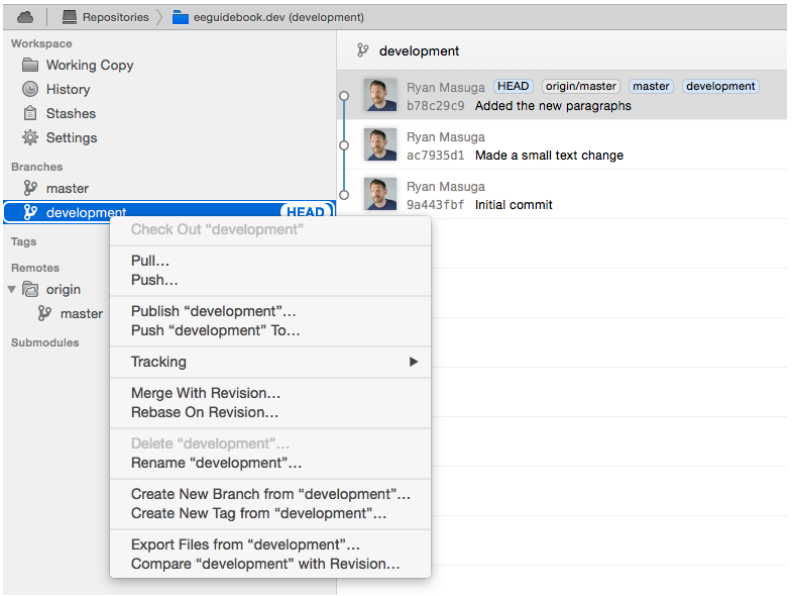


Fig. 5.27 Publishing the "development" branch to the remote repo.

You can leave the fields pre-populated because everything is as it should be by default: our local branch is development, the remote where we want to publish it to is "origin," the branch name we want on origin is "development" (I have yet to find a compelling reason to name the branch that is out on origin something other than what the branch name is locally), and you want

to track this branch so you can push changes to it and pull changes from it. Once you click "Publish Branch," you'll see it listed under the Remotes section of Tower's left sidebar **(Fig. 5.28)**.

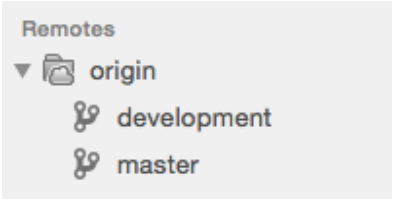


Fig. 5.28 The two main branches listed in the remote repository.

Topic Branches

> "Small teams using Git: how do you handle topic branches when many may affect the same template? Would love to hear topic branch merge tips."
> *—@masuga*
>
> "@masuga merge and hope"
> *—@ryanirelan*
>
> *—A brief Twitter conversation that sums things up quite well, March 9, 2015*[114]

Our flow involves working primarily on topic branches which are merged into development for client review, and then ultimately merged into master to go live. We try to keep dev's state relatively close to that of the master branch, but sometimes it's simply not possible and there is code on dev that just seems to hang out forever, never making its way to the main site.

We have one site where we pushed a bunch of changes to development for review and the client said, "Everyone stay tuned on final deployment. Should be sometime next week." That was in early September 2014, going on a year ago now as I write this. Will that particular area of the site ever make it live?

Who knows? But if and when they *do* finally give us the go-ahead to put this work on the live site (master branch), it will be relatively easy because we will finally merge our *topic branch* of this work into master.

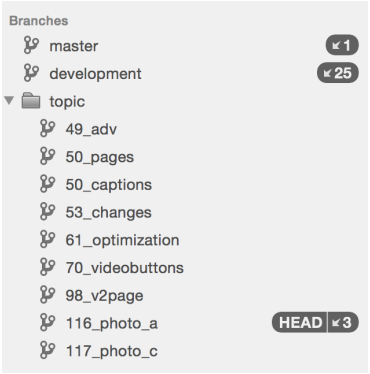


Fig. 5.29 Numerous topic branches going at once.

This is a current look at a busy site we work on (looks like I need to pull to get the things that other team members have been working on!). This client always has us developing numerous things at any given time. Those numbers correspond to task numbers in our project management application for easy reference **(Fig. 5.29)**.

Introducing topic branches is where things *can* start to get complicated. As just mentioned, we have a project with three active topic branches, all three of which affect one template, and who knows if any small "hotfix" type tweaks will be made to it in the meantime, while these other branches are being developed? It *is* possible to take changes from any branch and apply them to other branches, but we would only do that if absolutely necessary.

For our small team, we don't have exceptionally deep knowledge of Git (we didn't get into this business to be version control experts), so communication is key. When merging a topic branch into master and reviewing a merge conflict, you might hear something like "Hey so-and-so, just confirming that you did indeed want that social media junk in the footer removed?" I'm positive there are better ways to accurately combine disparate work, but I don't think you'll ever "solve" merge conflicts. There are numerous decent references out there about topic branch workflow.[115]

Despite possible (or inevitable!) merge conflicts, our ideal topic branch flow goes something like this: the client has a need for us to develop Project X. I pull master and dev just to make sure I have everything up to date, then make a topic branch (there are "opinions" about whether it's smarter to branch your topic off of development or master—but let's not worry about that at the moment and assume I go off of master) and start working on Project X there. I make a number of commits and am satisfied with the work. I merge it into development, which is deployed to the dev site for the client to review. The client likes what they see and gives us the go-ahead. I then merge my Project X topic branch into master ("squashing" the commits[116] to keep the master history clean) and deploy that. After that has been done, we delete the topic branch.

Deploying Updates

Now that we've made a few changes, committed them, and pushed them, how do we deploy that work to our server for the world to see? There are a few ways to do it, such as keeping your website as a working copy, deploying files via SFTP, or using an automation tool like Capistrano[117]. We've used two of those methods, which I'll outline here.

The Website as a Working Copy (Git Repo)

We were going this route back in 2010 with devot:ee. Our site was a working copy, which meant every time I wanted to put changes out on the live site, I'd have a Terminal window open and have to do a "git pull" to update the site. This method also requires you to have Git installed on the server where your site resides.

Living with your site as an actual working copy is dangerous, and adds problems that you don't need.

For example, your .git directory will be sitting out there with a complete file history in it. If you haven't taken measures to protect access to the .git directory with a rule in your .htaccess file[118] or by tweaking the Apache config, all of that information might be available to someone smart enough to snoop for it. It's easy to google for articles on how to exploit sites with .git directories sitting in the root of the site.[119]

It also means deployments can be tough, particularly if you've included directories where users upload (and change) content frequently. It seemed like it was easy for devot:ee to get out of whack because sometimes something minor like a file permission would change and Git would see that as an update, so we would either have to commit that or discard it before getting the changes.

One of the largest sites we've worked on (in terms of number of files) was a working copy when we inherited it. The .git directory was in the public_html directory, so it was necessary to have the following rule in the .htaccess file to keep anyone from accessing it:

```
RedirectMatch 403 /\.git.*$
```

The other issue with this site (which I've referenced earlier) is that the repo is already 1.8GB in size because none of the user upload directories were ignored in the .gitignore file. Users and admins are always adding images, so this repo was just growing and growing. Every time we wanted to push changes to the live site, we had to SSH into the server, commit all the images they uploaded, pull that update down to our local machine, and only then could we push our updates. Needless to say, this was a PITA. (Not only that, but I don't *want* all the content images on my local machine. This site alone takes up 2GB of my hard drive!)

Why go to this trouble? Is there any real advantage to the production site being a working copy? We didn't think there was, so this led us to find a different way to deploy our sites, this time via SFTP.

The Website as a "Collection of Files" Deployed Over SFTP

There are services out there that will deploy to your server over SFTP. We use DeployHQ, but there are others, like dploy.io[120] or ftploy.com[121]. If your code is hosted at Beanstalk[122], it can be deployed from there as well. One nice thing about Beanstalk is that in addition to deploying automatically or manually, you can trigger a deployment simply by adding a tag to any commit or push comment[123]. For example, you could trigger an automatic deployment to your dev environment by adding "[Deploy: development]."

The basic idea behind our deployment flow is that we push our changes to our main "central" repository at Bitbucket, and then we use DeployHQ to deploy those changes to the server **(Fig. 5.30)**. You can preview your changes before they're deployed, so it is easy to confirm exactly which files are getting uploaded, modified, or removed. If nothing else, this method means I would no longer have to log in via SSH to the live box and do a "git pull" all the time.

You can set it up with your repository so that deployments are automatic (we usually do this with the development sites, but not with the production sites). As mentioned back when I discussed the `/config` directory, we can also set up files to be ignored in the deployment, so we're not copying files out to our server that don't need to be there.

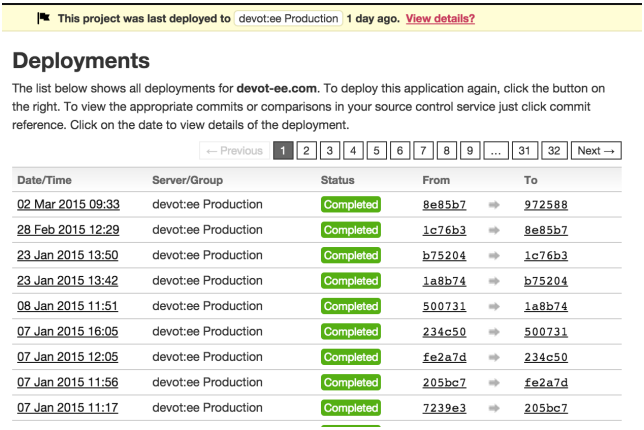


Fig. 5.30 Here is a sample of the deployment history to devot-ee.com, as seen in DeployHQ.

Using a service like DeployHQ also makes it easy to deploy different branches to different servers or locations. We automatically deploy development branches to our development sites, and manually deploy master branches to our production sites.

Syncing the Database

The subject of syncing the database is the elephant in the room when it comes to talking version control and ExpressionEngine. This is really a general issue with MySQL-based apps that no one has a perfect solution for yet—it's not limited to ExpressionEngine. In any case, it doesn't have to be an issue if you can sit back and go with the "flow."

As I mentioned back in the first chapter, you do *not* want to use a remote database when developing locally—particularly the production database. Confession: I actually used to do this. (Don't tell any clients I was doing work for in 2009.) Back then I would occasionally be working on local files while my database config was pointed to the production MySQL server. Even if I was pointed at a remote development database, why would I want to slow my work down like that?

In addition to being slow, this is just bone-headed. Why would you risk messing with your client's production data while you're monkeying around, experimenting with add-ons in your local sandbox?

Keep your environments separate. In most cases, it's not too much trouble to keep separate databases for every environment, including a copy for your local sandbox.

Data should flow "down" from production to your dev sites. Template and web changes should flow "up" to your production site. See slides 82 and 83[124] from Erik Reagan's talk at EECI 2011, *Environments and Version Control in ExpressionEngine*.[125]

> *It's OK if your dev/local site isn't up-to-date with the live site.*

Get used to it.

Having dev/local sites completely synced with production may not be entirely optimal anyway, depending on the security needs of your application. For example, if your production database holds a lot of sensitive data, it might not make sense to have all that data replicated in dev, staging, and local databases—as well as any databases you might need

to give to contractors. It might make sense to only have enough data in the non-production DB's so that you can accurately test your code. Database size might also be a concern. If you're working on a site whose database is gargantuan, it can be prohibitively difficult to have to sync and work with that locally. We have worked with clients whose databases were a few gigabytes in size, and even with the command line, it was inconvenient to copy that database to a local environment with any regularity.

Sometimes you have to manually update the various databases. For example, adding a new field to a channel can be a pain because you want to ensure that the field ID is the same across all your databases. Same with new items like Low Variables. If you add a new variable to one database and modify a template to display that new variable, any other developer working on the site with a now outdated database is going to see a tag like `{new_low_variable_I_dont_have}` rather than the content stored in that variable.

Mark J. Reeves posted a response on ExpressionEngine Answers[126] that encapsulates very well how we handle our databases when working on client sites. Going almost parallel with what he said there, we approach databases during development like this:

- We start development locally. With a config file that's geared for multiple environments, everyone can get set up on their local machine very easily. At the *very* beginning, someone (usually me) takes the reins and installs most of the usual add-ons and gets basic user accounts set up.
- The database is moved off that user's machine to a central machine in the office that everyone can access. Everyone uses this dev database, while working with the files on their machine.
- When it's time for the client to take a look, we set up a subdomain and move the dev database out to that server so the review site can function. We periodically re-upload the database to this dev server as we do more work on the DB at the office.

- If we let the client start making changes before launch, then that development database becomes the "master" DB. We know anything we do on the office database now will not stick around. The client might start uploading content, so now we go the other way and periodically copy that database down locally.
- Dev continues this way through…forever. That database either became the production DB or was copied to the production DB, so now it's the Real Deal®, and we copy it down as needed either to the office server or to our local machine.

Just like Mark and Clearbold, we also believe "the database is always a one-way update. We add new fields or install new add-ons in the master database, and then copy it down to dev to start working with it."

This data flow and data syncing issue is something I don't believe anyone has "solved," and I'm not sure with ExpressionEngine's database schema the way it is if it can be solved with a magic bullet.

I've seen some solutions suggested or hinted at, such as the ExpressionEngine DB Trace Module[127] that states, "The general concept is to trace your local db changes and store them into a release file. Release files can be deployed through FTP/GIT/SVN/Etc, after deploy you can do an install of the release, pushing all your db changes." I don't think any modules like this have been thoroughly battle-tested, so I can't speak to how well they would work. Feel free to try them out if you come across them, but *always* backup your database first!

6 Updating Your ExpressionEngine Install

> Upgrading #eecms sites now days is an incredible exercise in patience for me.
> —*Anna Brown, @mediagirl via Twitter*[128]

In my opinion, updating ExpressionEngine is truly the Achilles Heel of the system. I'll admit there are ExpressionEngine builds we've done out there that should be updated, but we haven't done so because it can be so tedious when factoring in add-ons. If you wait too long, though, the process definitely becomes a mountain out of a molehill. We know this from experience. With devot-ee.com, upgrading from ExpressionEngine 1 to ExpressionEngine 2 took weeks of planning and the better part of a weekend in the execution. Upgrading it from EE 2.2 to 2.5 was no small feat, either. Let's say that the site isn't running on any flavor of 2.9 at this point—and might not for quite some time.

There are articles being written all the time on how to make the upgrade process less painful. A recent example by Kristin Grote is *The 20-Minute ExpressionEngine Upgrade*[129]. Twenty minutes? In some CMS circles where there are native 1-click updates, twenty minutes would mean that something probably went wrong. But for ExpressionEngine, that's not half bad. Don't get me wrong, some sites upgrade just fine, but some whoppers (like devot:ee) will try your patience.

The process isn't terrible, but again, when add-ons are factored in, it can be tough. There was an article written recently by Seth Giammanco titled *The Technology Debt of CMS Add-ons*[130] that touches on this subject. In it he states:

> "From my experience, the time and effort required as well as the

> risk/challenge to upgrade grows exponentially with the number of installed add-ons. One, maybe two add-ons along with core is quick and clear to assess time and effort to upgrade. However, as those add-on numbers grow, the complexity and effort curves up notably."

In 2012, DevDemon released Updater[131], which was designed to make the task of updating ExpressionEngine sites much less painful. It also can update add-ons (at least those that are packaged in such a way that Updater can work with them). It was so popular that it won 2012 New Module of the Year in the AcademEE Awards[132]. That said, it's not something we've ever used. In my opinion, it can be hard to reconcile using 1-click updating anything with a version control workflow. I might be a little conservative in this regard, but I'm wary of 1-click updating when I don't know what is going on under the hood.

Fortunately, I don't have to fake or pretend to upgrade a site because while I was writing this guide, ExpressionEngine 2.10.1 was released, so I can take the 2.9.2 EE Guide site we've worked with and upgrade it here.

General Updating Steps

One thing that is nice about having the site in a repo is that when you make changes to it, you can see exactly what changed in each file before you commit anything. You can do your whole update locally (and on a new branch if you prefer) and test before ever putting those updated files on a live site. Being able to see these changes makes it easy to reapply and core hacks that might have been overwritten when you brought in the new files.

EllisLab has update instructions at https://ellislab.com/expressionengine/user-guide/installation/update.html and we'll be following along with those for the most part.

Step 1: Get an up-to-date version of the database copied down locally.

Make sure you have a copy of the database. This isn't always easy if you're working on a site that has a massive database, but most of the time it's easy enough to make a backup—so you should absolutely do so.

Step 2: Get a copy of the latest version of ExpressionEngine.

We'll be upgrading 2.9.2 to 2.10.1, which came out as I was writing this chapter.

Step 3: Follow the EllisLab upgrade directions (sort of).

I usually have EllisLab's docs open in one browser window so I can always refer back to the steps.

I unzipped the 2.10.1 directory, and am ready to overwrite things in my 2.9.2 install. I won't worry about clearing any caches that ExpressionEngine does natively because we haven't enabled any of those. When you're using version control, you don't really have to "Back-up all your ExpressionEngine files and directories" as it states in the docs.

As the docs mention, let's update the following:

- admin.php
- index.php
- system/
- themes/

First of all, "admin.php" is now "/control/index.php" as we have it set up. So drag admin.php into the control folder, delete the index.php that was there, and rename admin.php to index.php.

Next, simply replace the index.php file in the root of your site. You can also go ahead and replace the `/system` directory because we have modified so little in it.

For the `/themes` directory, I do rename the original to something like "themes_old", and drag the new themes directory in right next to it. Then I copy over the "third_party" directory (at this stage we have nothing in there, but normally you would have some third-party themes installed if you had any add-ons installed on your site). I might also copy out any alternate

control panel themes located in the "cp_themes" directory before deleting the original themes directory.

At this point we can check our work in Tower. We'll be able to easily revert any changes we don't want to keep (or keep any hacks that we need), and keep any files and directories from getting back into our install that we don't want in there (like the wiki themes or Agile Records stuff we removed earlier).

If I go back into Tower, I see there are 550 files changed (added, removed, or modified) in our upgrade from 2.9.2 to 2.10.1. Highlighting any modified file will show you what's changed in it. In this particular upgrade, I see in a lot of these files that only the copyright date at the top has been updated. Looking at an upgrade file by file like this can be very interesting, and give you a better understanding of what is changing between releases.

Another thing you can do to glimpse what is happening between releases is to look at the "ud_" files in the `/system/installer/updates` directory. There you'll find an update file associated with every ExpressionEngine version. For example, we know there were no database updates for ExpressionEngine 2.9.2 because the `do_update()` function in the ud_292.php file is empty.

```
class Updater {

  var $version_suffix = '';

  /**
   * Do Update
   *
   * @return TRUE
   */
  public function do_update()
  {
    return TRUE;
  }
}
```

Going back a bit to ExpressionEngine 2.9.0 (see the ud_290.php file), you'll notice that this upgrade has five changes involved.

```
public function do_update()
{

  ee()->load->dbforge();
```

```
    $steps = new ProgressIterator(
      array(
          '_update_template_routes_table',
          '_set_hidden_template_indicator',
          '_ensure_channel_combo_loader_action_integrity',
          '_convert_template_conditional_flag',
          '_warn_about_layout_contents'
      )
    );

    foreach ($steps as $k => $v)
    {
      $this->$v();
    }

    return TRUE;
}
```

Each of those lines in the array corresponds to a function farther down in the file. Sometimes these functions do things other than change the database. Here, the `_set_hidden_template_indicator` function adds or updates a variable in the config file. The `_update_template_routes_table` clearly affects the database, though, because it's adding a column to the template_routes table.

If you want to see an example where ExpressionEngine had a lot of work to do during an update, look at all 2,167 lines of the ud_200.php file. This is when ExpressionEngine made the jump from the EE1 branch to EE2.

In our case, because we're jumping from 2.9.2 to 2.10.1, the upgrade will go through a couple files. The ud_2_10_00.php file has five functions in it, including a couple that make database changes. These are the type of updates where you'll want to make sure you've made a database backup before getting things rolling.

Back to our specific upgrade, I clicked the "Status" column in Tower to group like types together. Here I can see all the items that aren't in Git all together at the top of the list **(Fig. 6.1)**.

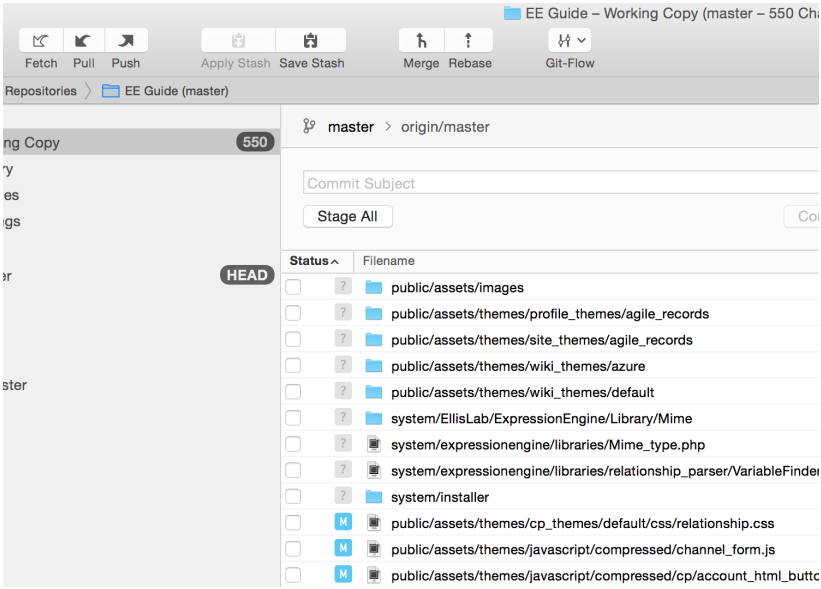


Fig. 6.1 All the untracked files are shown together at the top of the list.

We see that the wiki themes and the agile_records themes are trying to make their way back into the repo, so I'll right click on those four items and select "Move to Trash." I also see the `system/installer` directory there. We will leave that alone until after the upgrade is done. Make sure not to include that in the upgrade commit!

The changed files list also reflects moving our images from the main directory into the assets directory.

Keep in mind that there are a couple files in the `/system/expressionengine/config` directory that we had previously modified: config.php and database.php, and we just *threw those modifications away* when we overwrote the system folder **(Fig. 6.2)**. Now they're both totally blank (as they should be in a fresh install of ExpressionEngine). Let's (easily) bring the contents of those files back using Tower. First we need to find them in the list of modified files.

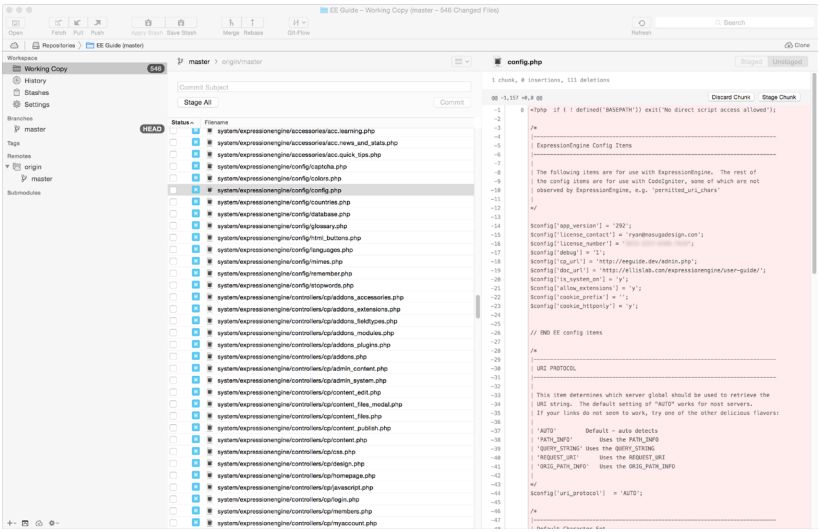


Fig. 6.2 Looks like our system folder overwrite blew away every line in the config file in the system folder. Quick…discard local changes!

When I've located them, I'll right-click on both those files and select "Discard Local Changes…" which will bring them back to where we had them. Boom, done.

Step 4: Run the Upgrade

We don't have to commit all the other changed files before running the upgrade. Ideally you have a database backup (and as we saw above, the upgrade to 2.10.1 involves a couple database changes, by way of the updates made in 2.10.0), so we can run the upgrade locally and check things out first. If things look great in testing, then we can figure out exactly what we're going to commit.

Go to your system directory. Because our system directory is safely out of webroot, we visit our "masked" admin at http://eeguide.dev/control/. With a new version of ExpressionEngine installed, the system will automatically redirect you to the "ExpressionEngine Installation and Update Wizard" page of the installer. Don't worry if the graphics don't load correctly. They never seem to load correctly if you move your themes folder or use a masked control panel. This is something we've learned to look past—it doesn't matter.

Click the "Click here to begin!" button, and if all is well, you should see a message like the one I'm seeing.

> *We have detected that you are currently running ExpressionEngine version 2.9.2.*
> *If you are ready to update ExpressionEngine to version 2.10.1 click the link below:*
> *Click here to update ExpressionEngine to Version 2.10.1*

Clicking the link brings up a browser prompt about backing up your database and files (thanks for the reminder). Agree to the license and click "Submit." If all went as planned, you will see the system make a few upgrade steps, first to 2.9.3, then to 2.10.0, and finally to 2.10.1. Your mileage may vary if you're not following along with the exact versions I'm working with.

Next, delete the installer directory.

Step 5: Test Your Site

This can be easier said than done, depending on the size and scope of your site. Testing should take about 15 seconds for the site we're working with here, but could entail many people over a few hours if there are a lot of custom add-ons or core hacks.

I tested that the pages still work and that I can still log into the control panel and navigate without issue. Now it's time to make our update commit.

Step 6: Commit the Updated Files

Here's where you want to be mindful of any core hacks that might have come before. You will want to make sure that you don't commit the new files that override your hacks unless, in your testing, you found that you don't need them any longer.

I like to comb through the files that need to be committed and do them one by one so I can understand what's changed. I know I can commit all the files in the system folder en masse because we already changed our config and database files back to the way we had them. However, notice that the config file has changed after the upgrade **(Fig. 6.3)**:

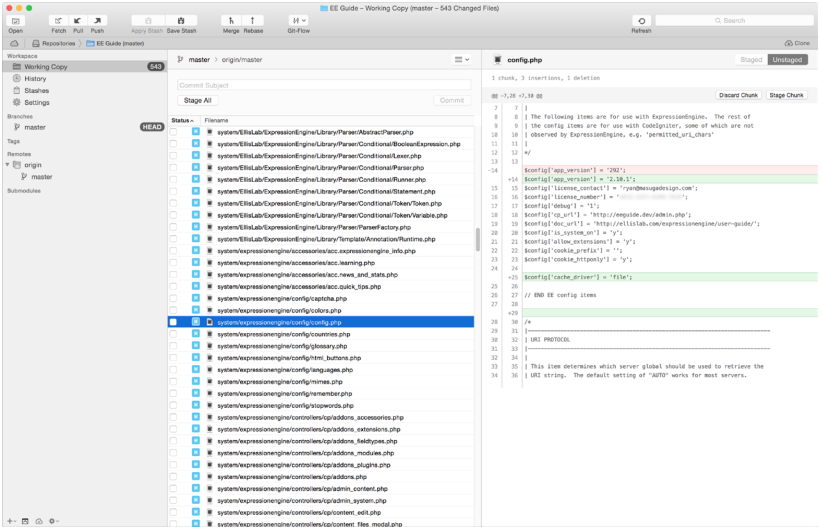


Fig. 6.3 Our config file has changed a bit after running the upgrade.

The version number has been updated and a new config variable has been added. What I could do is leave the `app_version` change alone and move the new `cache_driver` out of this file. I'd move it into BASE-config (so other people working on the repo could get this default value, too), and add it to my env_config.php file as well. For simplicity, we'll leave it right where it is and move forward by committing this file with the changes we see.

Now that I have all the files staged, I type a descriptive commit message like "Upgraded to EE 2.10.1" and click "Commit." Then I push it to the remote repo that I set up earlier.

Updating ExpressionEngine on the Server

Everything we just did is all well and good for your local copy, but how do you get those changes on the live site? The following is the process we use (although I'm willing to bet someone, somewhere, might have an even better way to do it). We'll assume that you've already got a live site up and running.

You've tested everything locally and you're certain you're ready to update the live site, so put your live site into maintenance mode. ExpressionEngine comes with a very basic "offline.html" template located at `/system/expressionengine/utilities/offline.html`. Duplicate this to the public directory and rename it "index.html." Your visitors should see that page instead of the index.php file, but that might depend on your server setup. The directory index order might not be looking for index.html before index.php. Some servers might not allow the DirectoryIndex setting in your .htaccess file. If that's the case and it isn't working, rename your main index.php file to anything other than index.php (e.g. index.phpMAIN) and rename offline.html to index.php. Now your visitors will definitely see your offline message

Back up your production database. Via SFTP, upload the `/system/installer` directory (remember, we're not keeping this directory in version control). We'll remove it right after we're done with the update.

Deploy your changed/updated files. Using a service like DeployHQ makes this simple. We just log in and tell it to deploy our latest commit, which has information about all the updated, added, or removed files. I would *highly* suggest utilizing a service like this. The time you can save dragging and dropping files in an FTP program will more than make up for the cost. Not only that, but a deployment service can make it easy for you to revert commits, too.

If you don't have a deployment method like DeployHQ or dploy.io, you'll have to upload the files manually. You could highlight the commit and look at Tower's rightmost column as a reference to what's changed if you're being careful—otherwise you can essentially update your whole install (but don't overwrite the index.php file we just changed!)

Now that all the changed files are out there, you can run the installer. The live database should be in the same state as your local copy. You can now delete the installer directory from the remote server.

At this point I'll set the `$config['is_system_on']` variable to 'n' which ensures that when we change our index file back the site will only be available to Super Admins[133]. Now I'll get rid of the index.php file in the public folder (which is actually our renamed offline.html file) and revert the renamed index.php file back to its original state.

Now we can test the site out, while it's still inaccessible to the general public. You may have to login to the control panel as a Super Admin first to ensure that you can see the site when it is offline. If everything behaves as you'd expect, set `$config['is_system_on']` to 'y' and your site is back in business on the latest version of ExpressionEngine.

Now let's look at customizing ExpressionEngine, which can really help a client feel at home, and help set you apart as an ExpressionEngine developer.

7 Customization

Customizing an ExpressionEngine control panel doesn't take much effort and can really set your implementation apart from the rest of the pack **(Fig. 7.1)**.

Fig. 7.1 Getting fancy with the CP login screen doesn't take much effort.

We always do some level of customization to the control panel. I've never been a fan of the default look of the ExpressionEngine 2 control panel. The overly-rounded corners and carnival pink are off-putting to me, and there has even been an "anti-pink" movement out there ever since this version of the control panel was released. I'm always taken aback when I come across a site that we didn't build that doesn't have a theme installed and remember what the control panel actually looks like in its native state. We *always* install

a theme to add the necessary polish to the back-end before handing it off to a client **(Fig. 7.2)**.

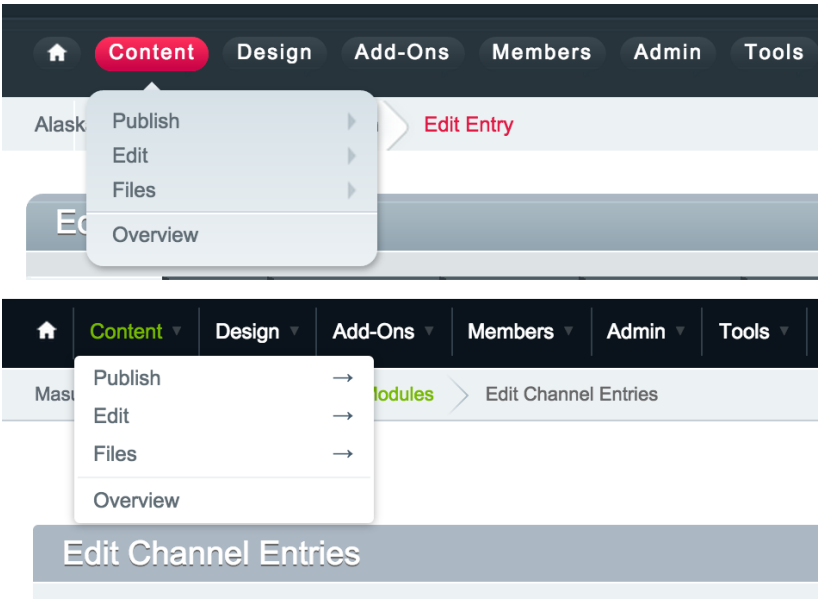


Fig. 7.2 To me, a less rounded and less pink look makes the control panel "feel" better. Default CP nav look (top), and same CP nav with a theme (bottom).

Applying a theme to the control panel in ExpressionEngine 2.9+ is easy to do, and allows you a lot of control without hacking core files by utilizing the view and CSS overrides. Before ExpressionEngine made it easy to override control panel views, updating the control panel login screen and installing a control panel theme were two separate things—and modifying the control panel login pretty much required hacking core files. Fortunately, those days are over and it's easy to utilize a control panel theme to handle both.

If you're going to create a lot of custom functionality, it can also be useful to create a "utility" add-on that can control numerous things (which I'll talk about shortly). Of course, with the ease of theming the control panel in recent versions of ExpressionEngine, we can also add *functionality* to our theme that we used to only get by installing various other add-ons.

Because themes are where it's at nowadays, let's talk about those.

Control Panel Themes

As I just mentioned, updating the control panel login screen and installing a control panel theme used to be two separate things, but with the ability to override views, we do everything with a custom theme. Part of a function of the theme is overriding some visual elements and colors, but going further, we can actually add utility and enhance certain areas of the control panel.

Early on, the only game in town was to install Newism's (their add-on division is now known as "EE Garage") Override CSS [134], which makes it easy to pick a default color for the control panel, and adds a nice look to the menus and main navigation.

Now you can search devot:ee to find a number of control panel themes[135]. The only other theme we have tested from this group is the popular Nerdery theme[136], the look and feel of which hint at Override CSS, and it's free.

We have found that no theme has been 100% effective at removing the default pink. I'm not sure if this is due to the way the core CSS is written or because of something else, but there always seems to be some element or another tucked away on a less-visited settings page that displays pink. It also seems that some third-party add-ons must have the pink color hard-coded in them, or their markup written in such a way that the theme stylesheet isn't able to effectively target all of the elements it needs to.

We've recently made our own control panel theme that takes elements from both the Nerdery and EE Garage themes and adds our own touches. We call it Eclipse. With the ability to fully override layouts in ExpressionEngine 2.9, this theme is becoming more and more powerful, because we can not only do basic things like change link colors, but we can add features we use all the time like Hover Intent, Superfish, and character counters. These are things that you can currently only get by installing separate extensions. We're even doing some cool advanced things in the templates by checking for the presence of certain popular and commonly-used add-ons, or adding useful links to various screens.

It's still a little early to tell if including some of the functionality that was previously in an Extension is better to have in a theme. We don't see much of a downside at this point.

We've installed Eclipse on our sites for the past half year and it's been working great, and it really helps facilitate customization of the control panel for our clients.

Short of installing a complete theme, let's just cover how simple it is to modify the login and forgot password pages.

Customizing the Control Panel Login and Forgot Password Screens

Before we started using a theme to handle all the control panel modifications, we used to at least customize the "login" and "forgot password" screens for all of our client sites. This used to be a somewhat difficult task because you had to hack core files to make it happen, but now you can simply override a couple view files, which is possible with a theme. If you want to get a taste of what modifying the control panel login used to be like, there is a method that was written up by Carl Crawley back in 2010[137]. His method required fewer hacks to core files than our old method did, but it also required jQuery.

Here we'll create a basic theme and override the control panel screens you see before you've logged in: the login page itself and the forgot password screen. These instructions are very similar to the instructions for customizing a control panel theme[138] found in the ExpressionEngine docs.

Duplicate the 'default' theme

We can start by duplicating the default theme to give us a head start. Mine is at `/Users/masuga/Sites/eeguide.com/_site/public/assets/themes/cp_themes/default`. I duplicated this folder and named it "custom." Now we need to tell ExpressionEngine that we prefer to use this theme, and there is a config variable that we can add to our env_config.php file: `$config['cp_theme']`. We'll set ours to this:

```
$config['cp_theme'] = 'custom';
```

We'll keep the images folder in our new theme, but we only need to keep CSS and PHP files that we'll modify. This means we can get every .css file out of the duplicated css folder, and put a single "override.css" file in it instead **(Fig 7.3)**.

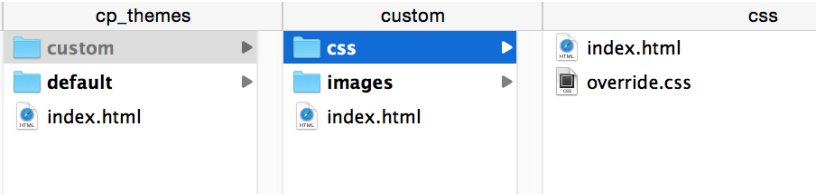


Fig. 7.3 Our new "custom" theme.

To quickly test if our new theme/override.css file is working as expected, I'll copy a couple style rules into override.css from the `/default/css/global.css` file and change them to some radical colors:

```
html {
   background-color: red; /* was #3e4c54 */
}

body {
   background-color: green; /* was #fff */
}

#branding {
   background: none;
}

#mainMenu {
   background-color: blue; /* was #27343C */
  color: yellow; /* was #fff */
}

#mainWrapper {
   background-color: orange; /* was #fff */
}
```

You can see our incredible result in **Fig. 7.4**.

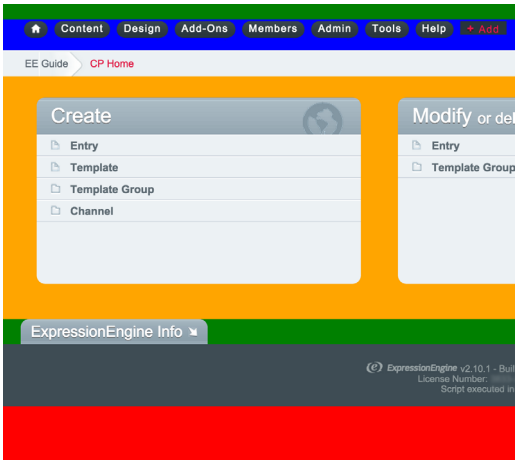


Fig. 7.4 I'd say those override styles are working. This is not a sneak preview of ExpressionEngine 3.

Create a 'views' folder

The views folder is where you can really feel the power of customizing the control panel. From the ExpressionEngine docs:

> If you would like to override any of […the default theme's PHP files…] in your own theme, copy them to your own theme's directory in a views folder, making sure that the copied files maintain the same directory structure in your theme as they do in the views directory.

First, create the "views" folder in the custom theme. Make sure to add the .htaccess file to this directory as outlined in the ExpressionEngine documentation because you don't want anyone accessing these templates directly.

To override the login and forgot password screens, first navigate all the way back out to `system/expressionengine/views` and copy the account/forgot_password.php and account/login.php files to `/public/assets/themes/cp_themes/custom/views/account` **(Fig. 7.5)**.

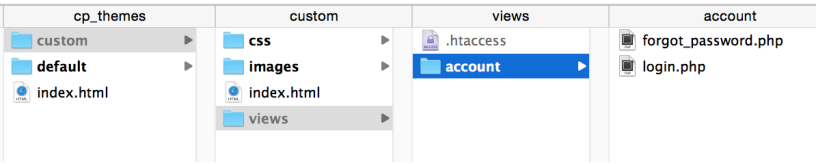


Fig. 7.5 Overriding views in our custom theme.

To quickly test this, replace the contents on login.php with anything else and then visit your site in the browser. I set login.php to this basic message **(Fig. 7.6)**.

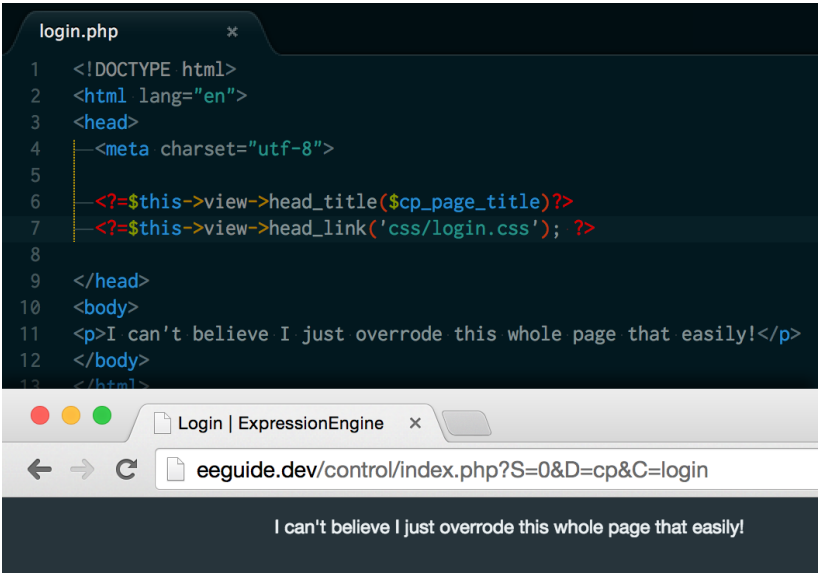


Fig. 7.6 It's that easy to override the login screen in ExpressionEngine without hacking any core files.

Now I'll show you how we modify our login/forgot password screens to use no images except for the client logo.

The goal is to turn the default login screen into something branded and inviting **(Fig. 7.7)**. You don't have to get fancy with the background gradients. Even changing the background color can set the tone for the client.

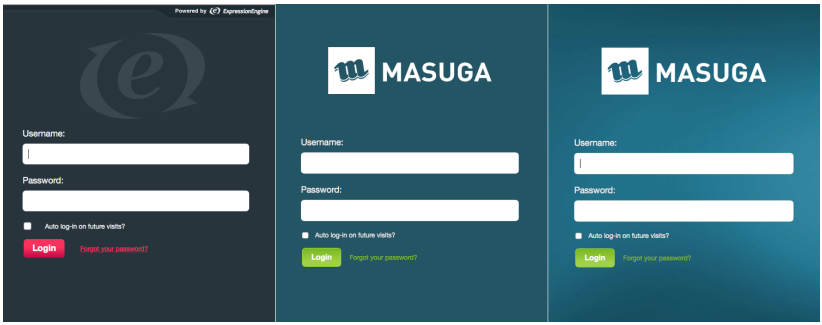


Fig. 7.7 Turning the default login screen into something more branded, with different levels of tweaking.

We already added the two account view override files, but the CSS file you'll need to modify is at `/public/assets/themes/cp_themes/default/css/login.css`. Copy this to `/public/assets/themes/cp_themes/custom/css/login.css` and we'll come back to it in a minute.

The first thing we did was to create the ExpressionEngine logo as SVG[139]. This allows us to easily change its color or size without having to go into a graphic program. We can use CSS to modify any of its properties. The "issue" with that is that the resulting HTML takes up a lot of space, as you can see in **Fig. 7.8**. So inside our "account" directory, we created a new "_branding.php" file. In that file, we took the single line out of the top of both login.php and forgot_password.php, and moved it to the _branding.php file.

We took this:

```
<div id="branding"><a href="http://ellislab.com/"><img src="<?=PATH_CP_GBL_
IMG?>ee_logo_branding.gif" width="250" height="28" alt="<?=lang('powered_
by')?> ExpressionEngine" /></a></div>
```

and moved it to _branding.php. Then we replaced the img tag with our SVG, like this:

```
<div id="branding"><a href="http://ellislab.com/"><svg>[a lot of code]</
svg></a></div>
```

Fig. 7.8 The markup for the ExpressionEngine logo SVG is long, but so flexible.

As a simpler alternative to the SVG, you could still wipe out the `img` tag and put text in the link instead. That eliminates your need to bother with an extra file and an include. That might look like:

```
<div id="branding"><a href="http://ellislab.com/">Powered by Expres-
sionEngine&amp;reg;</a></div>
```

You might need to do the simple text version on servers that don't have short open tags enabled. If that setting isn't enabled, you might get errors because the include statement won't work correctly **(Fig 7.9)**.

I believe the ExpressionEngine license states that the "Powered by" message has to be on the login screen somewhere and link to EllisLab, but I don't think there are any rules that say it has to be text, a rasterized graphic, or SVG[140].

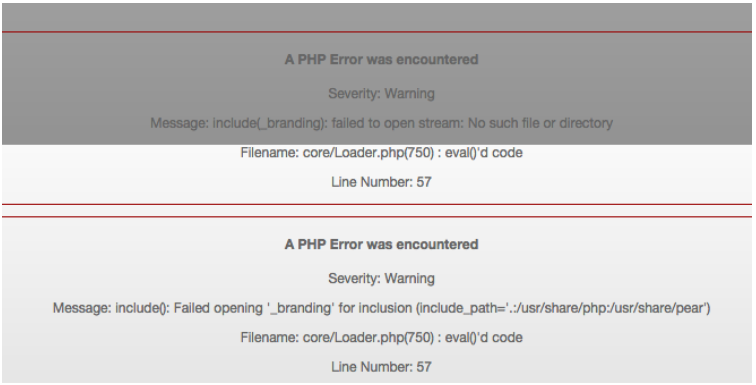


Fig. 7.9 This error crops up when trying to use an include on servers without short tags enabled.

As a side note, all the links to EllisLab.com *should* be over SSL, but as of 2.10.1, the URLs in the default views are still http. You could go the extra mile and add the 's' to the URL yourself while you're in here.

Once the SVG is in place, we add a couple things to the bottom of both the login and forgot_password templates. Above the last closing div, add a "push" div element, and below it add an include to our branding file. For example, the bottom of login.php goes from:

```
  </div>
<?php
if (isset($script_foot))
{
  echo $script_foot;
}
?>

</body>
</html>
```

to:

```
    <div class="push"></div>
  </div>

<?php include '_branding.php'; ?>

<?php
if (isset($script_foot))
{
  echo $script_foot;
```

```
}
?>

</body>
</html>
```

If the includes don't work on your server, you can just go the simple text link route instead:

```
   <div class="push"></div>
   </div>

<div id="branding"><a href="http://ellislab.com/">Powered by Expres-
sionEngine&reg;</a></div>

<?php
if (isset($script_foot))
{
   echo $script_foot;
}
?>
</body>
</html>
```

The only other thing we do is make the form inputs go 100% instead of 90%. The content element on this page is about 400px wide, so if we create a new custom logo at 400px wide, those form inputs will look off-centered.

As for the forgot_password.php file, we do nearly the same thing as we did on login.php. Move branding to the bottom, and add the "push" div. The *extra* step here is to remove all the CSS at the top of the file (I have no idea why that is there, when there is a default login.css file whose styles apply here already). I make the top of this template the exact same as the login.php file, down to the closing head tag. Both files should be uniform, like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
   <?=$this->view->head_title($cp_page_title)?>
   <?=$this->view->head_link('css/login.css'); ?>
   <?=$this->view->script_tag('jquery/jquery.js')?>
</head>
```

The rest is creating a custom logo (ideally a transparent PNG) and modifying the CSS. Because our login background is dark, I created a quick reverse (white) PNG logo, named it "login_logo.png" and put it in the images folder.

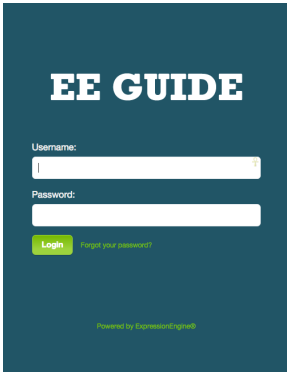


Fig. 7.10 Our EE Guide login screen.

Here are the basic styles needed to get the login screen looking like **Fig. 7.10**. I'm doing the "powered by" message as text to keep it simple.

```
html, body { height: 100%;}

body {
   background-color: #215566;
   font-family: "Helvetica Neue", Arial, Helvetica, Geneva, sans-serif;
   text-align: center;
   font-size: 12px;
   color: #efefef;
   margin: 0;
   padding: 0;
}

*,*:before,*:after { box-sizing: border-box;}

#content {
   text-align: left;
   padding: 230px 20px 0;
   margin: 0px auto;
   width: 440px;
   min-height: 100%;
   margin: 0 auto -100px; /* the bottom margin is the negative value of the
branding and push element heights */
   background: transparent url('../images/login_logo.png') center 0px no-re-
peat;
```

```
}

/* ExpressionEngine Logo (Branding, footer)
---------------------------------------- */
#branding,
.push {
   text-align: center;
   height: 100px;
}
/* ---------------------------------- */

a:link,
a:visited {
   color: #76b900;
   text-decoration: none;
}

a:hover {
   color: #9dd33d;
   text-decoration: underline;
}

dt {
   font-size: 16px;
   line-height: 24px;
   margin-bottom: 5px;
}

dd {
   margin: 0 0 15px 0;
   font-size: 11px;
   line-height: 24px;
   color: #666;
}

input {
   outline: 0;
   font-size: 14px;
   border: 1px solid #EEF2F5;
   border-radius: 6px;
   padding: 10px;
   color: #333;
}

input[type=checkbox] { width: auto; }

span { margin-left: 10px; }

input.submit {
  padding: 7px 16px 9px 16px;
  color: #fff;
  font-weight: bold;
  border: 0;
  width: auto;
```

```
  background: #76b900;
  background: -webkit-linear-gradient(#9dd33d, #76b900);
          background: linear-gradient(#9dd33d, #76b900);
  border: 1px solid #9dd33d;
  box-shadow: 0 1px 0 rgba(0, 0, 0, 0.2);
   text-shadow: 0 1px 2px rgba(0,0,0,0.2);
}

input.submit:hover,
input.submit:active {
  cursor: pointer;
  background: #9dd33d;
  background: -webkit-linear-gradient(#76b900, #9dd33d);
          background: linear-gradient(#76b900, #9dd33d);
}

.success,
.error {
   font-size: 14px;
   color: #007822;
   background: #e9fdd7 url('../images/success.png') no-repeat 8px 10px;
   border: 1px solid #bce99a;
   padding: 3px 15px 3px 30px;
   margin: 40px 0 20px 0;
   border-radius: 6px;
}

.error {
   color: #ce0000;
   background: #fdf5b2 url('../images/error.png') no-repeat 8px 10px;
}

.success p,
.error p { margin: 8px 0; }
```

Have Fun

The login and forgot password screens are the tip of the iceberg. You can go to town on the rest of the control panel after this. Changing colors, styles and so on—even adding functionality, all without installing any extra extensions.

I feel that taking a bit of time to tweak the control panel login is one of those things that can help set you apart from other development shops. It goes a long way with a client when they can see a control panel login that looks like their own branded entryway to the area where they can effortlessly create their content, rather than a billboard reminding them the name of the tool they're using.

The "Utility" Add-on

On every site, we end up making at least one "utility" add-on, which is sort of like a Swiss army-knife of functionality. It's usually a way to consolidate the functions of various plugins into one place. This Utility add-on is generally a module and plugin combination, and deals mostly with front-end manipulation, but can also handle some modifications to the control panel.

This practice came about after I hired my first real PHP developer. I had previously done most of the coding (PHP and front-end) on devot:ee myself in the first couple years. There was a ton of PHP in the templates, and things were pretty messy. Our goal became to get all the PHP out of the templates and into plugins, and we ended up making a single add-on to house all of these functions.

You can guess the types of things we consolidated by reading some of the function names: member_picture, force_https, addon_download_links, can_edit, addon_count, and so on.

On devot-ee.com we ended up with two utility add-ons: devot:ee Utilities and devot:ee Forum Utilities **(Fig. 7.11)**. The support forums are a big enough beast that we decided to put all the functions pertaining to them in their own add-on. Both of these utility add-ons have the same structure:

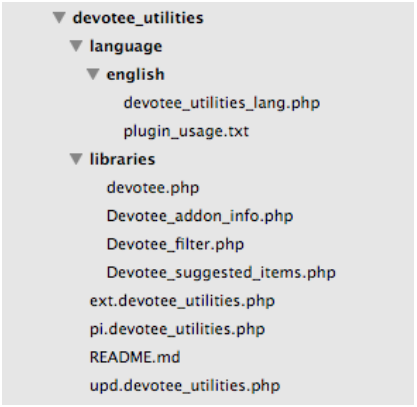


Fig. 7.11 The files for the devot:ee Utilities add-on.

Anything remotely complex can be added into the libraries directory. Note the plugin_usage.txt file. This is a file dedicated to explaining exactly what tags can be used, where these might be used, and why. This ties into something fundamental that I explained way back in the Common Sense chapter: *document your code*.

If I need to know if this utility does a particular thing, or we hire a new developer and want to get them up to speed quickly, a document like this saves a ton of time and therefore a ton of money. We don't have to scan through the code for comments to try and piece together what is happening.

"CP Mods" Add-on

Sometimes we need to make modifications to the control panel, and we like to house them all in a single "CP Mods" add-on, which is generally an extension. An example of something we do with this is to append the channel name to the CP page header **(Fig. 7.12)**. This information isn't on an edit screen before EE 2.10.0, but at least there is an easy way to modify the template to make it happen. For this modification, we use the `cp_js_end` hook, which allows us to add javascript to every control panel page. With it, we load a view file that contains some JS. In our view file we have:

```
$(document).ready(function() {
var eeChannels = new Array();

   <?php foreach ($channels as $id => $name): ?>
      eeChannels[ <?php echo $id; ?> ] = "<?php echo $name; ?>";
   <?php endforeach; ?>

   // Append channel name to the CP page header
   if ( $('.contents .heading h2').html() == 'Edit Entry' ) {
      var channel_id = $('input[name="channel_id"]').val();
      $('.contents .heading h2').append(' - '+eeChannels[channel_id]);
   }
});
```

Now when the client is editing an entry, it will display the channel name after the Edit entry text, like "Edit Entry—Quarterly Highlights." This is particularly nice when there are a lot of channels and some of them share a field group or otherwise look very similar. Because this is a generic "CP

Mods" add-on, we can store a number of similar tweaks in this one add-on rather than installing a bunch of small extensions.

Fig. 7.12 This is our example of using a CP Mods add-on to enhance the user experience by appending the channel name after "Edit Entry."

This was a big site with a lot of similar channels, so this little touch—adding a clear indication of what channel the current entry belongs to—was a big deal to the client. One glance and…"Yep, I'm in the Resources channel… whew, I'm not getting fired today! Thanks, Masuga Design!"

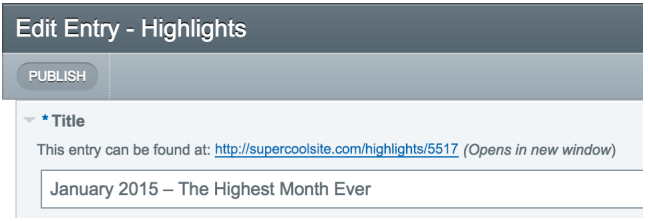


Fig. 7.13 Adding helpful info and links doesn't take too much effort.

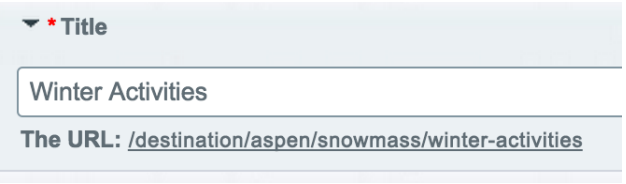


Fig. 7.14 We will frequently append useful information like a clickable URL to title fields (especially on channels that have non-standard URL setups).

As I've touched on numerous times: anything to make the experience better for the client is desirable, even (and maybe especially) the little things **(Fig. 7.13-7.14)**.

8 Must-Have Add-Ons

Ah, the subject of many a blog post. There are must have add-on lists all over the place, and everyone certainly has their opinions. (Go ahead and Google "top expressionengine addons." I'll be here when you get back.) I have my opinions, too, and seeing that you've read this far, I think I'll share them. Take any list with a grain of salt, though. There are so many ways of working with ExpressionEngine, and so many requirements specific to each project, that there simply can be no "one size fits all" magic solution for what to install when core isn't cutting it.

I'm listing our most commonly used add-ons here, and the reason we use them. In many cases, a third-party add-on is replicating native functionality (or was there before the functionality was added to core) and the third-party option is simply better executed. There are add-ons that are certainly not on this list that might deserve to be here such as Assets or CE Image, for example. This list is more about those unsung heroes, or those add-ons that literally get installed on every site. This list is in alphabetical order to be as fair as possible.

Note that the majority of these "must-haves" are for the back-end. They are control panel enhancements for the client, not necessarily for you! They could all conceivably be rolled into the core product at some point.

Deviant

A simple add-on that makes the control panel more usable by allowing you to choose where to go after entries are added or updated. The default page one lands on after editing an entry is baffling to say the least. It sort of outputs the content you just entered in a rather unformatted way, and doesn't serve much purpose or help the user. Deviant solves this.

Small. Simple. Works with Zenbu (also in this list). Free.

http://dvt.ee/deviant

(Zoo) Flexible Admin

You will always have content editors. They won't always need to access every area of your control panel. There are native controls and restrictions in place, but the main navigation stays the same for everyone (short of someone adding a link for themselves with the "+Add" link. With Flexible Admin, you can completely control the main navigation in the control panel for an entire member group.

http://dvt.ee/flxadmn

In

This add-on, in addition to having the most generic name possible, is something I've always wanted: a lightweight alternative to snippets and embeds. The best thing is that it can also behave like a snippet or an embed depending on how you want to use it. You can use In to Insert (inserts a template as an early-parsed global variable or snippet) and/or Include (like an embed that is parsed as an exp tag, instead of much later in the parse order) template files. You can pass variables to an Include, just like an embed.

This add-on is by Aaron Waldon, of CausingEffect.com, who has a proven track record of solid coding practices and fast support. It's free, and both helps us easily version control "snippets" and save on queries and overhead by avoiding embeds where we can.

http://dvt.ee/in

Lamplighter

This was formerly the "devot:ee Monitor." I'll try to avoid going into too much self-promotion here. This add-on looks at version information from devot:ee for all the add-ons you have installed and informs you as to what is out of date. It is simply the best way to get a look at this information. If you have a Lamplighter.io account, you can see this information in a centralized way, across all of your sites, in your Lamplighter dashboard.

Another useful thing that the Lamplighter add-on helps you do is export a list of everything installed, which is great when you're trying to debug an add-on issue with a developer and they ask you, "What else do you have installed?" This list shows the EE version, PHP version, some cookie and session information, and a list of add-ons with version numbers (and the number of the most recent version of that add-on if the one you have installed is out of date), and details as to what type(s) they are: module, extension, accessory, etc.

By clicking "Site debug info" link at the bottom of the accessory, you're presented with a handy list that you can copy and paste into a support thread or email. Here's an example:

```
General Information
ExpressionEngine : 281
PHP : 5.3.27
DB Driver : mysql
Server Software : Apache
Browser : Chrome 35.0.1916.153

Session Settings
User Session Type : c
Admin Session Type : c
Cookie Domain : .yoursitedomain.com
Cookie Path :
CP Cookie Domain :
CP Cookie Path :
CP Session TTL : 604800

Installed Add-ons
Deviant 1.0.6 [ EXT ]
Editor 3.2.0 (current: 3.2.3) [ MOD EXT FLD ]
Lamplighter 1.1.2 (current: 1.1.4) [ MOD ACC ]
MD Entry Limit 1.0 [ EXT PLG ]
Snippet Sync (Developer license) 1.3.5 [ EXT ]
Stash 2.5.1 (current: 2.5.3) [ MOD EXT ]
Superfish EE Control Panel 0.1 [ EXT ]
```

```
Switchee 2.1 (current: 2.1.1) [ PLG ]
Zenbu 1.9.1 [ MOD EXT ]
Zoo Flexible Admin 1.8 (current: 1.82) [ MOD EXT ]
```

http://dvt.ee/lamplite

Low Variables

There is almost always a need for "one-off" content—those items that just don't fit into the idea of a channel, or even a static page. Low Variables is practically the only game in town to solve this. It's so well done, it almost doesn't need competition because what could a similar add-on do better?

In 2009, when ExpressionEngine 1.6 was the current stable version and Low Variables was only a few weeks from hitting the scene, I kept this kind of data in a "structural" channel that had a single entry. I wrote an article on devot:ee titled "*Give Your Client More Control by Utilizing a Single-Entry Structural Channel*"[140] that explained how to set up a single-entry channel and put all your content and "controls" in various fields in this entry. It was basically a primitive version of Low Variables. I'm not sure how we'd build a site without this add-on today.

http://dvt.ee/lowvar

MD Utilities

As mentioned earlier in this guide, we have a pretty common set of stuff that we collect into a single add-on so that we don't have to install a ton of different add-ons. This isn't something we always install, because it depends on how complex the site is. The minute things start getting tricky and we need to start getting custom, we look at installing a Utilities plugin.

Minimee

This is a newer addition to our installs. I'm a fan of compiling our CSS and JS *before* it goes out to the production site, but one thing that always seems to cause issues is versioning the final, compressed and compiled CSS and JS. Our small team would always have merge conflicts with them in Git, so it's easier to not commit the compressed files. Short of having to install

something on every client server that could combine and compress our files for us upon successful deployment (or adding a build script to our deployment process), we went with the next (much) easier thing: Minimee. This really is a fantastic add-on that just works. It's free—but don't let that scare you. The support is great.

http://dvt.ee/minimee

Mo' Variables

Mo' Variables was created by Rob Sanchez, who, in my opinion, is one of the best developers in and around the ExpressionEngine world. This does what it says on the tin: "Add more early-parsed, global variables to your EE installation." Once you get used to using it and having all these variables available, you might wonder how you got along without it. It is something we use so often that if I come across a site that doesn't have it installed, I'll scratch my head wondering why variables like `{segments_from_1}` or `{post:your_key}` don't work.

http://dvt.ee/movars

Resource Router

(formerly Template Routes)

Resource Router is another winner from Rob Sanchez. EllisLab recently added some basic template routing as of ExpressionEngine 2.8.0 and called it Template Routes[141], hence the add-on's name change to Resource Router. As of this writing, we find that this third-party implementation is better and more robust than the core version. We would much rather control this sort of thing in our config file than in a settings page buried in the control panel.

Resource Router allows you to control your URLs by remapping URI routes to a specific HTTP response, using CodeIgniter-style routing rules. The reasons for using something like this are pretty easy to understand:

- You need to break out of the template_group/template_name paradigm of traditional EE routing

- You need to nest URLs on a Structure/Pages URI (pagination, categories, etc.)
- You need to point multiple URL patterns to a single template
- You need custom JSON/HTML endpoints that do not warrant a full EE template
- You want to remove excess conditional logic in your templates

http://dvt.ee/resrtr

Stash

ExpressionEngine development has changed entirely since Mark Croxton released Stash in early 2011. It's hard to convey how useful this add-on is, and I know that I've barely scratched the surface of what it can do. On that note, it's also hard to convey how dangerous it is if you don't know what you're doing with it.

We mostly use it for setting variables in channel:entries loops that we can use later on in template processing, allowing us to avoid having two (or more!) channel:entries loops on a page. Even that basic usage can help keep you sane on certain site builds, but you can use Stash as an embed replacement and for a complete template partials approach to assembling your templates.

EllisLab recently added some basic template layouts[142] to ExpressionEngine 2.8.0, but I know many ExpressionEngine developers have been using the Stash partials method for a long time. As of this writing, we have used the ExpressionEngine layouts on four or five sites, but don't have much experience with Stash partials. Stash has been out for a long time, so there are plenty of resources out there for best practices when using Stash partials, some going back as fas as 2012.

Please, though, be careful before you go crazy with this one. With great power comes great responsibility, and a potentially difficult build to maintain down the line.

http://dvt.ee/stash

Superfish

It's too bad this sort of functionality isn't built into the default control panel experience, but thank goodness for extensions. This is a simple one, but makes the control panel menus so much easier to use. You'll know what I mean when you have a site with numerous channels and you need to edit one by navigating to Content > Edit and then having to hover way down to a channel at the bottom of the list and boom…you hover off the edge of menu by one pixel accidentally and it disappears, forcing you to start over. Superfish includes Hover Intent so you can safely mouse off the edge of the menu for roughly a quarter second and it won't disappear on you. Sometimes, it really is the little things. Anything to make the experience easier and more usable for your content editors is a good thing!

As of ExpressionEngine 2.9, we skip installing this extension and add this functionality with our own Eclipse control panel theme, so if you're using Eclipse, Superfish is baked in, and there is no need for the add-on!

http://dvt.ee/sprfsh

Switchee (and IfElse)

This is another free offering from Mark Croxton. Switchee allows you to use switch/case control logic in your templates. It's great for allowing one template to act as a "controller" to show multiple things (lists of entries with pagination, single entry, and more) rather than having to use multiple templates in a template group.

Before version 2.9, each condition in ExpressionEngine's if/else advanced conditionals was parsed before being removed at the end of the template parsing process. So, if your template had a lot of tags within each condition, it would run unnecessary queries and functions which could have a direct impact on the page load times. Switchee removes unmatched conditions before they're parsed, so you can rest easy knowing that any queries not needed to render the page are not being run.

Switchee also allows you to use regular expressions to perform advanced matching on case conditions, and supports defaults. This is a free add-on, and (like Stash) will likely change the way you construct your templates.

There is a similar plugin from Mark called "IfElse" that forces early parsing of If/Else advanced conditionals. Depending on your use case, this might work better for you than Switchee, but using one or the other of these will certainly help keep your query counts down.

The way ExpressionEngine's template engine has been rewritten to handle conditionals in 2.9 (which was released while this guide was being written—and the template engine is faster, simpler, and has better error reporting for syntax errors) means IfElse may no longer be necessary, and Switchee's advantage is diminished. As of this writing, we still use them out of habit.

http://dvt.ee/switchee

http://dvt.ee/ifelse

Zenbu

You will always have to view entry lists and sort the edit screen. The native list view isn't nearly as flexible as what you can do with Zenbu. The second your client needs to add columns here (ours frequently do) or do any sort of advanced filtering, you'll want to be using Zenbu.

You can set up views per member group, add and sort of custom fields, save searches, and you can even customize the output of your column content. Not only that, it's extendable—the developer has added a number of hooks for other developers to utilize if they need to make Zenbu do even more. On devot:ee, we've needed enough tweaks to Zenbu that we created an extension called MD Zenbu Mods.

http://dvt.ee/zenbu

Honorable Mention: Dashee

I mention this one not because it's necessarily *great*, but the idea sure is great. This one is a little rough around the edges, but what it does is the important thing: it makes the control panel landing page configurable and useful. I feel that the control panel landing page is one of the most useless pages in the system, rivaled only by the default page a user is taken to after editing an entry (and see Deviant, outlined previously, for a way to banish that page). If I've ever clicked a link on the control panel landing page that wasn't in the main navigation, it was almost certainly by accident.

Dashee solves this problem by allowing you to create completely customized dashboards, and to create, configure, and install "widgets" on them. Creating custom widgets is easy, and you can tailor them for the site content and the editors to give each site's default control panel default screen a personality and usefulness that you could not easily do otherwise.

Dashee was recently updated to 2.0 which cleaned up some of the rough edges I mentioned, and brought the ability to have multiple dashboards and interactive widgets. This is well worth checking out, and can be useful on every site.

http://dvt.ee/dashee

9 Conclusion

ExpressionEngine is a secure, customizable, and robust content management system that is well-suited for version control. By taking the time to make security-minded modifications, sensibly organize your directories, add a few simple customizations, and version your work, you can easily set yourself apart from other ExpressionEngine developers.

Separate Yourself From the Pack

At the end of the day, your client isn't just buying an ExpressionEngine site—they're buying you. So what separates you from any other ExpressionEngine shop? It's the details. Every well-structured template. All those useful comments and documentation you added in the code. Each thorough field instruction for the content editors. A secure, maintainable, version controlled site that the client can be proud of and with which they can feel like they have a machine that solves their business problem and that is worth every penny they paid.

> "I'm beginning to think our past developer might as well have been a plumber. Or a veterinarian."
> *—A client, March 2015*

Paying attention to details and taking advantage of ExpressionEngine's strengths will not go unnoticed or unrewarded, either by you or the client.

For your part, the site will be easy to update or upgrade, and client requests will not be seen as a chore because the site was sensibly put together and well documented.

For the client's part, they won't complain about having to add or update content, and they will feel at home and understand where things are. Your conversations will revolve around adding new features and improving existing ones to move their business forward, rather than spinning wheels discussing how to fix this bug or that problem area.

You will be solving business problems, not CMS problems. This means your clients won't be spending time hunting for another developer like they did before they found you—they will want you to stick around.

We hear very positive comments all the time from clients who can't believe the difference between what we offer as an ExpressionEngine consultant and what the "previous developer" offered. We also hear comments along the lines of "this used to be confusing, but it's easy now." If you take the time to read the ExpressionEngine documentation to understand how it works, pay attention to the client's needs, and balance your knowledge and the project requirements as you complete the build there's no reason clients won't say the same sort of nice things about you.

Make More Money With ExpressionEngine

In our experience, having a system, or "method to our madness," keeps us consistent, efficient, and profitable. Attention to small details makes a huge difference in the quality of the site you deliver, and this has a direct and vital impact on your relationship with your client that translates into dollars.

If you can competently and repeatedly produce a solid product and sell your ExpressionEngine skills with confidence, you're going to be able to charge more money for your services. Our rate has tripled over the last few years, and I believe after reading this guide you'll feel confident enough to charge what you're worth.

I believe that introducing version control to your workflow and utilizing any (or all!) of the methods in this guide will help you be a more successful ExpressionEngine developer. It has, and continues, to work for us—at the time I finish this book, our services are in higher demand than ever before. I quite literally can't keep up with requests for work, and that is a great problem to have.

Now It's Your Turn!

I've given you everything I have, but your success is going to depend on you. ExpressionEngine is just a tool. It's the hammer, and your client's business need is not a nail…it's the hole. Their website is a nail. There is no one way to hit the nail with the hammer to create the hole, but some ways are much smarter and more efficient than others. My goal was to show you how to set up ExpressionEngine in such a way that you're hitting the nail squarely with the head of the hammer…and not the handle.

Get In Touch

I'd love to know how this guide helped you. For that matter, I'd love to know where I might have dropped the ball, too. Was there something that I glossed over? Is there a method you've found for doing something or other that's better than what I've written about here? I'd love to hear it. Any revisions I make to this guide that incorporate your suggestions will be credited—particularly if I test something out and decide to go with it! Email any questions or comments to me at ryan@masugadesign.com.

If this guide helped you in any measurable way, and you'd like to write a quick blurb or testimonial, that would be much appreciated and I will make sure to post it where others can see it and become aware of you and your business in the process. Of course, reviews, social shares (use the #eecmsguide hashtag on Twitter), or blog posts about the Guide would be appreciated too. Make sure to let me know if you've written something up.

Thank you so much for your support! I wish you much success!

- Ryan Masuga

Resources

As I mentioned way back in "What This Guide Is Not"—this guide was not one to deal with how you actually build the front end of your ExpressionEngine site, but is focused on the back end, security, folder structure, common sense, and version control. There are other resources out there to help you with the front-end aspect of developing with ExpressionEngine. In fact, some of those would work quite well alongside the things discussed in this guide. Here are a few:

Mijingo / Ryan Irelan

Founded by Ryan Irelan, and billed as "Step-by-step video courses for web professionals," Mijingo **(Fig. 10.1)** has courses spanning many subjects, but their series on "Learning ExpressionEngine[143]" has been around a long time and is a go-to for people new to ExpressionEngine.

Fig. 10.1 Mijingo. Great training materials from Ryan Irelan.

Mijingo offers other ExpressionEngine-related resources as well. For example, if you want to learn how to build add-ons *the right way,* without question you'll want to immerse yourself in Low's "Building an ExpressionEngine Add-on"[144] **(Fig. 10.2)** screencast that is offered by Mijingo.

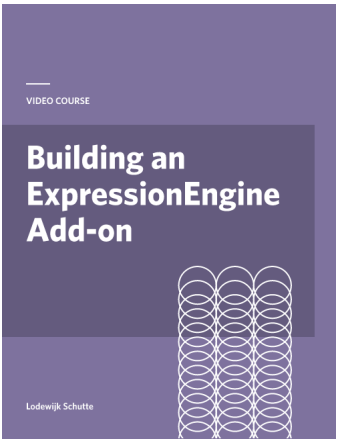


Fig. 10.2 If you want to build an add-on for EE, start with Low.

Low builds some of the best add-ons out there and is one of the top-selling commercial add-on developers on devot:ee—he knows what he's doing!
https://mijingo.com

Creat//ee

Creat//ee offers online courses and tutorials for ExpressionEngine, from beginner to advanced.
http://www.creat-ee.com/classes

Train-ee

Train-ee offer books, screencasts, and classes. Their main book is "Building an ExpressionEngine 2 Site—Small Business" and has been updated for ExpressionEngine 2.9.
http://www.train-ee.com

Other Resources

There are plenty of other resources out there for all aspects of ExpressionEngine.

devot:ee

The #1 unofficial resource for ExpressionEngine and the one place you need to go to find, research, and purchase third-party add-ons. Going strong since 2009.

https://devot-ee.com

ExpressionEngine® Answers

Alternatively known as the ExpressionEngine Stack Exchange[145]. Years ago the place to go for EE answers was the EllisLab Forums[146], but EESE quickly became the de facto place to go for Q&A regarding all things ExpressionEngine-related.

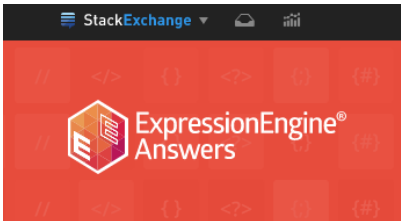


Fig. 10.3 This has become **the** place to get your EE questions answered.

Some developers even use it as their main form of support. I don't know if the official EE Forums get much traffic these days because I haven't been there in forever, myself. If you need quick answers, EE Stack Exchange might be your best bet. It's the first place I look when a technical question comes up.

http://expressionengine.stackexchange.com/

director-ee.com

This directory makes it easy to search for ExpressionEngine developers, and even helps narrow the search geographically. They also have a job board. If you're an EE developer, you might want to add your profile here. I frequently direct potential clients here whom we can't work with or can't help.

http://director-ee.com

#eecms (Twitter)

There are a number of helpful people using the #eecms[147] hashtag on Twitter every day. You can ask questions and hashtag them with #eecms, and you're very likely to get an answer.

The ExpressionEngine Slack Channel

This is a recent addition to the places one can go to chat with fellow ExpressionEngine developers. I've been in there a bit myself lately, and have even gotten some decent support from other users. Ask to be added and say "Hi!"

https://eecms.slack.com/

The ExpressionEngine URL Schematic

This is incredible just for the amount of time it must have taken to test and put together. Well done!

http://www.jamessmith.co.uk/articles/expressionengine_url_schematic

Fig. 10.4 This schematic must have taken an age to put together.

ExpressionEngine Conference (and workshops)

These have been going on for a number of years, and I've been to so many of them, I've lost count (I *think* six…I've got all the conference badges somewhere). They are always a great time. If you're interested in ExpressionEngine, it would behoove you to get to one of these conferences.
http://www.expressionengineconference.com/

ExpressionEngine Meetups

ExpressionEngine Meetups are happening all over the world. The easiest place to see where they're taking place is on meetup.com[149].

Blogs

At this time, there are not a lot of blogs or sites that I'm aware of that *regularly* update content in and around ExpressionEngine. Probably the longest-standing and most reputable site is EE Insider, and even that site (run by Ryan Irelan, of Mijingo, mentioned above) doesn't post content as regularly as it used to.
http://eeinsider.com/

Show-EE

Show-EE is a showcase for recent ExpressionEngine sites that have been launched.
http://show-ee.com/

Git

There are any number of incredible resources out there for using Git, and I've mentioned quite a few of them over the course of this guide. I'll list those here for your convenience, but keep in mind they're nearly endless.

git - the simple guide
"just a simple guide for getting started with git. no deep shit"
http://rogerdudler.github.io/git-guide/

Git Tower's "Learn Version Control with Git"
A step-by-step course for the complete beginner.
http://www.git-tower.com/learn/ebook/command-line/appendix/why-git

Atlassian's "Getting Git Right"
Getting Git right, with tutorials, news, and tips.
https://www.atlassian.com/git/

The Git Documentation
The book, videos, and huge link resource.
http://git-scm.com/documentation

References

Chapter 1

1 https://twitter.com/sandwich_hlp/status/565973992029179904

2 https://ellislab.com/expressionengine

3 http://www.mamp.info/en/mamp-pro/

4 http://www.git-tower.com/

5 http://www.sourcetreeapp.com/

6 http://www.sublimetext.com/

7 https://github.com/Etsur/EE-ST2

8 Package Manager for Sublime Text: https://sublime.wbond.net/

9 https://github.com/mrw/ExpressionEngine2-Sublime-Text-Bundle

10 https://github.com/fcgrx/ExpressionEngine2-Sublime-Text-3-Bundle

11 http://www.sequelpro.com/

12 This was true with the initial version of Tower, but Tower 2 was released and I've successfully made commits that had many thousands of files involved without issue.

13 https://devot-ee.com

14 See: https://devot-ee.com/articles/item/organizing-an-expressionengine-add-on-collection

15 https://ellislab.com/expressionengine/user-guide/development/extension_hooks/global/core/index.html#core-template-route

16 https://devot-ee.com/hooks/ee/core-template-route

17 https://subversion.apache.org/

18 http://mercurial.selenic.com/

Chapter 2

19 https://ellislab.com/expressionengine/user-guide/cp/design/templates/template_access.html

20 http://stackoverflow.com/questions/876089/who-wrote-this-programing-saying-always-code-as-if-the-guy-who-ends-up-maintai

21 https://twitter.com/erwinheiser/status/606212873442979840

22 March 16, 2015. https://twitter.com/mithra62/status/577671332420149248

23 https://devot-ee.com/add-ons/ce-cache

24 https://devot-ee.com/add-ons/static-page-caching

25 https://www.varnish-cache.org/about

26 See the brief Twitter conversation here: https://twitter.com/masuga/status/537362504499097600

27 This is a common thing to do, and there is a section in the ExpressionEngine docs outlining when and how you would need to do this: https://ellislab.com/expressionengine/user-guide/add-ons/channel/channel_entries.html#start-on

28 See the docs about PHP in Templates: https://ellislab.com/expressionengine/user-guide/templates/php.html

29 Read up here on EE's URL Structure https://ellislab.com/expressionengine/user-guide/urls/url_structure.html

30 https://ellislab.com/expressionengine/user-guide/urls/404pages.html

31 More info on the dynamic parameter: https://ellislab.com/expressionengine/user-guide/add-ons/channel/channel_entries.html#dynamic

32 March 12, 2014 https://twitter.com/wordlust/status/443610029347074048

33 http://www.404notfound.fr/

34 https://developers.google.com/analytics/devguides/collection/gajs/asyncTracking

35 https://developers.google.com/analytics/devguides/collection/analyticsjs/

36 https://twitter.com/johnwbaxter/status/585740293238087681

37 On devot:ee at https://devot-ee.com/add-ons/structure

38 http://devot-ee.com/add-ons/matrix

39 http://devot-ee.com/add-ons/playa

40 https://github.com/focuslabllc/ee-master-config

41 For the record, that bug was where the site index template would be rendered after the 404 template, making 404 pages look incorrect. https://support.ellislab.com/bugs/detail/20471/site-index-template-is-rendered-after-the-404-template

42 Wygwam, Wyvern or Expresso are all great alternatives that use ckEditor. Editor is a third-party fieldtype that uses Redactor.

43 From a conversation with former CEO EllisLab Les Camacho, NYC 2010.

44 http://slickplan.com/

Chapter 3

45 https://ellislab.com/expressionengine

46 http://ellislab.com/expressionengine/user-guide/development/guidelines/security.html

47 http://mijingo.com/products/ebooklets/securing-expressionengine-2/

48 http://en.wikipedia.org/wiki/Security_through_obscurity

49 https://exp-resso.com/blog/post/2011/08/securing-your-expressionengine-website-with-https

50 See this EllisLab blog post for more info about mod_spdy: https://ellislab.com/blog/entry/ssl-everywhere-at-ellislab

51 https://ellislab.com/expressionengine/user-guide/development/guidelines/security.html

52 See: https://ellislab.com/expressionengine/user-guide/development/guidelines/security.html#general-security-practice

53 https://ellislab.com/expressionengine/user-guide/general/system_configuration_overrides.html#allow-member-registration

54 A few popular ones are Zoo Visitor http://dvt.ee/zoovstr, Profile:Edit http://dvt.ee/adeYJKo, and User http://dvt.ee/ad480wk.

55 This example comes from metasushi.com: http://metasushi.com/blog/easy-custom-plugins/

56 https://ellislab.com/expressionengine/user-guide/templates/php.html

57 See https://ellislab.com/expressionengine/user-guide/general/system_configuration_overrides.html#debug and https://ellislab.com/expressionengine/user-guide/cp/admin/output_and_debugging_preferences.html#debug-preference

Chapter 4

58 https://www.owasp.org/index.php/Full_Path_Disclosure

59 http://en.wikipedia.org/wiki/Symbolic_link

60 https://ellislab.com/expressionengine/user-guide/installation/installation.html#set-file-permissions

61 https://github.com/quickshiftin/set-ee-perms

62 See this thread from July 2007: https://ellislab.com/forums/viewthread/56203/ The repo is still out there on Github: https://github.com/mdesign/md.hide_smileys.ee_addon

63 https://devot-ee.com/add-ons/in

64 https://www.deployhq.com/

65 http://beanstalkapp.com/

66 See: https://ellislab.com/expressionengine/user-guide/general/system_configuration_overrides.html

67 Our basic multi-environment config: https://gist.github.com/mdesign/e3d6e9fd97d22ce68b4c

68 New Relic settings in the EE docs: https://ellislab.com/expressionengine/user-guide/general/system_configuration_overrides.html#use-newrelic

69 New Relic is, according to their site, a platform that "...empowers developers, IT/Ops, and business leaders to transform their business using real-time data directly from production software." We've used it before, and it is pretty powerful. http://newrelic.com/

70 See this answer on Stack Overflow: http://stackoverflow.com/a/10895944/1860400. There is another thread titled 'how safe is $_SERVER["HTTP_HOST"]?' that explains further: http://stackoverflow.com/questions/10350602/how-safe-is-serverhttp-host/10350718#10350718
http://www.skeletonscribe.net/2013/05/practical-http-host-header-attacks.html

71 https://ellislab.com/expressionengine/user-guide/general/system_configuration_overrides.html#gzip-output

72 https://ellislab.com/expressionengine/user-guide/templates/globals/user_defined.html

73 http://dvt.ee/LnkVlt

74 See: https://ellislab.com/expressionengine/user-guide/general/system_configuration_overrides.html

75 http://dvt.ee/lowvar

76 http://dvt.ee/in

77 http://devot-ee.com/add-ons/snippet-sync-developer-license

78 http://gruntjs.com/

79 http://ellislab.com/expressionengine/user-guide/templates/templates_as_files.html

80 http://dvt.ee/minimee

81 http://dvt.ee/ceimg

82 http://ellislab.com/expressionengine/user-guide/cp/design/templates/global_template_preferences.html#strict-urls

83 http://devot-ee.com/add-ons/resource-router

84 http://ellislab.com/expressionengine/user-guide/templates/hidden_templates.html

Chapter 5

85 http://www.git-tower.com/learn/ebook/command-line/appendix/why-git

86 http://git-scm.com/docs/git-revert

87 http://git-scm.com/

88 http://en.wikipedia.org/wiki/Git_%28software%29

89 http://git-scm.com/about

90 https://www.atlassian.com/git

91 https://www.atlassian.com/git/workflows#!workflow-feature-branch

92 http://git-scm.com/downloads

93 Article from 2010: http://nvie.com/posts/a-successful-git-branching-model/

94 You can read all about these commands and many more in the Git docs. http://git-scm.com/docs

95 http://git-scm.com/docs/git-commit

96 http://git-scm.com/docs/git-push

97 http://git-scm.com/docs/git-pull

98 http://git-scm.com/docs/git-branch

99 http://git-scm.com/docs/git-merge

100 http://git-scm.com/docs/git-cherry-pick

101 http://git-scm.com/docs/git-stash

102 http://www.git-tower.com/

103 http://www.sourcetreeapp.com/

104 http://git-scm.com/download/gui/linux

105 Path Finder is a Finder alternative I've used for many years. http://www.cocoatech.com/pathfinder/

106 Here is a blog post about how to set up a couple commands in Terminal: http://ianlunn.co.uk/articles/quickly-showhide-hidden-files-mac-os-x-mavericks/. There are apps, too. A quick Google search will turn up a number of them.

107 https://ellislab.com/expressionengine/user-guide/templates/hidden_templates.html

108 http://git-scm.com/docs/gitignore

109 https://www.gitignore.io/

110 http://expressionengine.stackexchange.com/questions/563/what-do-you-put-in-your-gitignore-for-expressionengine-sites. I borrowed from my own answer there to flash out this .gitignore section.

111 http://macrabbit.com/espresso/

112 `git rm --cached` is useful if you want to remove a file from git source control but not delete it. You're not "removing" the file, but you will no longer be tracking it. It still lives in the history, though. http://stackoverflow.com/q/7602106/1860400

113 See: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

114 https://twitter.com/masuga/status/575127947528519681 and https://twitter.com/ryanirelan/status/575140341491236864

115 See, for example: https://github.com/dchelimsky/rspec/wiki/Topic-Branches#basic-topic-branch-workflow or http://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows#Topic-Branches

116 A couple advantages of squashing are because your changes belong to the same "logical changeset" (or topic), and you can easily git revert it if you don't want those changes in the repo. See: https://robots.thoughtbot.com/git-interactive-rebase-squash-amend-rewriting-history

117 http://capistranorb.com/documentation/overview/what-is-capistrano/

118 See http://biasedbit.com/hide-git-folder-with-htaccess/ or http://serverfault.com/questions/128069/how-do-i-prevent-apache-from-serving-the-git-directory/325841

119 http://resources.infosecinstitute.com/hacking-svn-git-and-mercurial/

120 http://dploy.io/

121 http://ftploy.com/

122 http://beanstalkapp.com/

123 See http://support.beanstalkapp.com/customer/portal/articles/75805-what-are-beanstalk%E2%80%99s-deployment-tools-

124 Slides on Speakerdeck: https://speakerdeck.com/erikreagan/environments-and-version-control-in-ee-the-why-and-how?slide=82

125 Post: http://focuslabllc.com/journal/eeci2011-environments-and-version-control-in-expressionengine

126 http://expressionengine.stackexchange.com/questions/5956/move-expressionengine-from-development-to-live-same-server/5971#5971

127 https://github.com/fccotech/ee-db-trace

Chapter 6

128 https://twitter.com/mediagirl/status/502157791142752256

129 https://www.kristengrote.com/blog/articles/the-20-minute-expressionengine-upgrade

130 http://mod-lab.com/blog/the-technology-debt-of-cms-add-ons

131 http://devot-ee.com/add-ons/updater

132 See: http://devot-ee.com/academee-awards/2012

133 https://ellislab.com/expressionengine/user-guide/general/system_configuration_overrides.html#is-system-on

Chapter 7

134 http://ee-garage.com/override-css

135 You can filter for control panel themes here: https://devot-ee.com/add-ons/filter?t=cpt

136 https://devot-ee.com/add-ons/nerdery

137 http://www.madebyhippo.com/blog/view/customising_your_ee2_cp_login

138 https://ellislab.com/expressionengine/user-guide/development/cp_styles/#customizing-the-control-panel-theme

139 http://en.wikipedia.org/wiki/Scalable_Vector_Graphics

140 See the user guide about Display of Copyright Notices: https://ellislab.com/expressionengine/user-guide/about/license.html#display-of-copyright-notices and EllisLab's Trademark Use Policy: https://ellislab.com/about/trademark-use-policy

Chapter 8

141 See: http://devot-ee.com/articles/item/give-your-client-more-control-utilizing-a-single-entry-structural-channel

142 See the blog announcement here: https://ellislab.com/blog/entry/template-routes-in-expressionengine-2.8. Docs: https://ellislab.com/expressionengine/user-guide/urls/template_routes.html

143 Blog post: https://ellislab.com/blog/entry/template-layouts-in-expressionengine-2.8. Layouts docs: https://ellislab.com/expressionengine/user-guide/templates/layouts.html

Resources

144 https://mijingo.com/products/screencasts/expressionengine-tutorial/

145 https://mijingo.com/products/screencasts/how-to-develop-expressionengine-add-on/

146 http://expressionengine.stackexchange.com/

147 https://ellislab.com/forums/

148 https://twitter.com/search?q=%23eecms

149 Try searching: http://www.meetup.com/find/?allMeetups=true&keywords=expressionengine&radius=Infinity

About the Author

Ryan Masuga owns a web consultancy (masugadesign.com) and has used ExpressionEngine professionally since 2006. He founded devot:ee (devot-ee.com) in 2009, which is the #1 unofficial resource for ExpressionEngine add-on information and sales.

He studied fine art in college, receiving a BFA in painting from the University of Michigan in 1995. After living the "starving artist lifestyle" in New York City for a couple years in the late 90's, he moved back to Michigan where, by chance, he was thrown into web development.

Ryan enjoys reading, playing super-hero with his kids, exercising, playing guitar, and Michigan football. He can often be found wasting precious time on Twitter @masuga.

He lives in West Michigan with his wife and three children.