Best Practices for Web Developers v1.01

by Kate Rhodes (masukomi at masukomi dot org)

May 18, 2007

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License. http://creativecommons.org/licenses/by-nc-nd/3.0/

Contents

1 Let's get a few things straight **3**

2 Tools, tools, tools **4**

3 Getting version control right **5**

3.1 You have a production branch. 6

3.2 Emergency bug fixes start with production. 6

3.3 Exploring new functionality starts with a new branch. 6

3.4 You have a main development branch. 7

3.5 You are not using a version control system that makes branching and merging a pain. 7

3.6 You are not using CVS anymore. 7

3.7 You have started using a version control system that's actually good. 8

4 Testing **10**

4.1 Basic testing concepts 11

4.1.1 Unit testing 11

4.1.2 Functional / Acceptance testing 11

4.1.3 Integration testing 12

4.1.4 System testing 12

4.2 Heuristic Driven Development 13

4.2.1 Triage 13

4.2.2 When things break. 14

4.2.3 Your development environment 15

5 Optimization & load tests **16**

5.1 Triage based on predicted usage. 16

5.2 Create test data early in realistic amounts. 17

5.3 Mirror your production environment. 17

5.4 Learn to use profiling software. 18

5.5 Set some targets. 18

5.6 Use diversionary tactics. 18

5.7 Load testing rules of thumb. 19

6 Application deployment **20**

6.1 Rules of thumb 20

6.1.1 Your live webapp is never in a state that can't be rolled back to in the future. 20

6.1.2 You have copies of *all* your server / client specific configuration files in version control. 21

6.1.3 Updates are always done atomically. 21

6.1.4 All instances of your webapp are in the same state. 22

6.2 Automated application deployment 22

Chapter 1

Let's get a few things straight

You're cheating.

You're doing all your development in trunk. You're screwed if a major bug surfaces while you're in the middle of exploring some new functionality because you've already touched half the code. Your version of "going live" is some variation on just copying files from here to there. Maybe you're all fancy and doing "svn update" to get your code there, but lets be honest, there's always a bit of crossing your fingers and hoping things don't break. You know what unit tests are, but you've got some lame excuse for the fact that you don't actually have any, or at least not enough to be actually worth anything. Or maybe you want unit tests but just "haven't gotten around to it."

If this isn't you feel free to stop reading here. If this *is* you then welcome to the club. We suck. For entirely too long now you and I have been too lame to actually get off our asses and do things right. But guess what. We've taken the first step. You and I, and everyone else reading this, are going to stop acting like inexperienced college grads and start acting like the professionals we are (even if you happen to also be an inexperienced college grad).

Ready to start doing things right?

Good. Let's start off with a definition:

> Heuristic
> Pronunciation: hyû'ristik
> Definition:
> 1. [n] a commonsense rule (or set of rules) intended to increase the probability of solving some problem
> 2. [adj] of or relating to a general formulation that serves to guide investigation
> 3. [adj] (computer science) relating to or using a heuristic rule

Got it? That's how we're going to approach things here. We're going to take a heuristic approach to version control, unit testing, and app deployment. The zealots are going to hate us. But we'll be getting things done while they're worrying about getting test coverage on that last 10% of their code-base.

Chapter 2

Tools, tools, tools

Listen, everyone's got their opinion on what tools are best, but the longer you hang around the more you come to realize that it just doesn't matter. Pick something that works for you. If you find something better, switch. I'm going to be recommending a few atypical tools here, but don't worry about it; you're smart. You can extrapolate what I'm talking about doing to your favorite tool, or use it to choose some other tool you like better. What's important is that you get tools that make it easy for you and your coworkers to get your job done easily and right.

Chapter 3

Getting version control right

This is where we are probably the most lame. Some of us are just more lame than others. So let's just lay out how we *should* be doing things and then tackle why and what we need to do to actually get there. If you're not using version control now the rest of us are just going to pretend we didn't know that and you are going to stop reading this and go install a version control system (VCS) the second, nay, nanosecond you finish this chapter. We shall never talk of this again.

By the time we're done getting our asses into shape all of these statements should be true:

1. You have a production branch.
2. Every time you have to fix some catastrophic bug that showed up on the live site you do so in another branch based on production.
3. Every time you want to explore some new functionality you do so on *a new branch* and that branch is always based off of the production branch.
4. You have a main development branch. When you're done speccing out that new functionality you merge it into here for all the other developers to share and enjoy.
5. You are not using a version control system that makes branching and merging a pain. If it's a pain you'll avoid it. If you avoid it, like you have been, you're a dork and *you* are most definitely not a dork, now are you?
6. You are not using CVS anymore.
7. You have started using a version control system that's actually good.

I admit, some of these seem a bit excessive. Especially all that branching. But there are good reasons for each of those statements. So let's explore why:

3.1 You have a production branch.

This is probably the biggest one. Some of us get this one right at least, but *far* too many of us are still doing all our work in the trunk. You need a production branch for the simple reason that you are fallible and some of those dorks you have to work with are *even more* fallible. It is just a matter of time before some bug sneaks onto the live site. When it does you'll need to fix it ASAP and you'll need to know that your patch doesn't introduce some new bugs from that code you were working on. If you've got a production branch you can know with certainty exactly what's on the server *right now* and you'll have a pristine starting point from which to start tracking down and killing that bug.

The only time this branch doesn't reflect exactly what's on your live site is in the short time between when you add new code to it and when you deploy it to the live site. Everything that gets deployed to the live site from here gets a sensible tag that you don't have to look up in some document when you need it.

3.2 Emergency bug fixes start with production.

We pretty much just covered why it's good to start from the production branch but there's a little more to it. People are counting on your site. You can't afford to be "pretty sure" you haven't touched anything else in the system when patching the bug. Reading through old check-in logs to see what we've changed is tedious and boring, and lets be honest with ourselves here: we're not going to do it anyway.

Before you touch that bug you're going to make a branch from production to work on it. Remember number five? "You are not using a version control system that makes branching and merging a pain". You are going to have the benefit of version control on your changes no matter how small they are. And remember 1b? That means no check-ins to production until they are ready to go and you've still got to check it into the dev branch too. Are you seriously considering trying to merge the live branch into the dev branch? That's just asking for pain. So you start with a branch of production, write the test, fix the bug, merge it in to the dev branch, merge it into the production branch, and deploy. And no, as you've probably guessed I'm not a unit testing freak but you just had a bug make it to the live site that was so severe it couldn't wait until the next push. You can't afford that to happen again so you write a test for it.

3.3 Exploring new functionality starts with a new branch.

Most people will want to skip this rule and just work on the dev branch. But, most of the time this is because branching and merging suck in most version control systems. Exploring new functionality in a branch has a lot of benefits:

- "OMFGWTFBBQ that feature is sooooo cool." It's so cool you've decided to push it live ahead of all those other whozeywhatsits you've been working on. This is a non-issue if you've worked it out in it's own branch. Just merge it with production when you're ready to go and you've got no worries about other things coming along for the ride.

- "Ok that sucked." You thought it would be cool but it just ended up being lame, or breaking more things than you feel like dealing with, or whatever. You're not keeping it. If you worked it out on it's own branch you're done. It's not messing with anyone else's code. It's not in the trunk. Just walk away. If you were working on the trunk you'd have to roll back all those changes.
- "It's good but it can wait." Sometimes this means wait till the next push, in which case you just merge it into the dev branch. Sometimes this means you need to shelve it for a couple months and wait for some other major newness, in which case you'll be quite relieved you weren't working this out the development branch. Again, you're done. Just walk away. A couple months from now when you're ready to use it you can merge it into dev. Yeah that merge may be a pain because of all the other changes that will take place in the interim but it's less of a pain then trying to juggle keeping it in the dev branch and needing to work around it.

3.4 You have a main development branch.

Duh.

3.5 You are not using a version control system that makes branching and merging a pain.

Branching is easy. It's so easy it's trivial. It's so trivial it's less work to branch than it is to drill down to a particular file and open it.

"But Kate," (that's me) you say, "branching and merging *is* a pain."

No. It's just a pain in that lame ass VCS you're using now. Move on.

If you keep using something which makes you want to avoid branching you'll eventually give up and go back to just doing all your work in development. And *you* are not a dork. You are going to start doing things right and than means setting yourself up with tools that do what you need them to do in such a way that you don't mind actually doing it.

3.6 You are not using CVS anymore.

CVS is for dinosaurs. CVS was *written* by dinosaurs.... literally. Ok, maybe not literally, but it's roughly as old as dirt. For some reason I'll never understand it got to a point where it was workable and then nobody bothered to actually make it good even though tens of thousands of developers were using it every day. There are a number of other good systems out there and they're all relatively easy to install for a geek like you. We're practically in the middle of a version control renaissance these days.

3.7 You have started using a version control system that's actually good.

The main criteria here being that whatever system you decide on it must: make branching and merging feel almost trivial to you, have a command set that's easy and flexible, and allow multiple people to work on the same file at the same time. If your system can do all that and you and your coworkers are happy with it then don't change.

I'm going to stick my neck out here and suggest that you really ought to be using a distributed version control system. I don't care if you have a development team of one or one thousand. Traditional version control systems that work like CVS and Subversion are good. But, you're missing out on some truly great features if you stick with them.

The leaders, at the moment seem to be Darcs `http://www.darcs.net`, Git `http://git.or.cz/`, Mercurial `http://www.selenic.com/mercurial/`, and Bazaar `http://bazaar-vcs.org/`. Darcs has the advantage of a very well thought out set of commands, each with an corresponding undo. Git is being used to manage the Linux kernel and has some significant performance benefits. Mercurial is being used for the open source version of the Java Development Kit. Bazaar is being used to manage the Samba project but not much else of note. They're all conceptually fairly similar but each has taken a very different approach in their respective implementations. Darcs is written in Haskell, Git is written in C and leverages a variety of shell scripts, which means it doesn't work very well under Windows, and Mercurial and Bazaar are written in Python. All have Eclipse plug-ins in various stages of development.

It was my intent to discuss all the reasons that distributed version control systems are great here, but it turns out I've already discussed the feature that I find most advantageous above: branching. And, Mark Reinhold[2] has summarized the high-points quite well with this:

- A branch is just another repository, not some state in a central database. You can create branches at will to explore new ideas and, equally easily, simply delete them if things don't work out.
- You don't need to be connected to a server - or even be on the network - in order to get work done [and still take full advantage of revision control].
- You don't need to set up and manage a central [VCS] host with sufficient disk space, compute power, bandwidth, and backup to support the concurrent [VCS] operations of your entire development community.

The one thing Mark doesn't mention is the huge advantage a distributed VCS has for open source projects. If Mozilla has taught us anything it's that you're *not* going to get thousands of developers working on your project. So, while huge numbers of developers is something that distributed VCS handle exceedingly well it's pretty much a moot point. In my experience, a developer interested in some OSS (Open Source Software) project will download the source from a traditional VCS, poke around to understand it, and, if they're smitten with the idea, start customizing it for their needs. But, they are forced to either work without the benefits of version control or they have to check it into their own personal VCS. If they check it into their own the projects main developers can pretty

much forget about ever getting patches from them because they're no longer able to sync with your tree and it would be way more work than they generally want to do to get synced and give you a patch that was useful. If they work without VCS (because they don't have commit rights to your VCS) they *may* send in a patch but the work to get there has been like climbing a rock face without lines and harnesses.

If the projects developers were to use a distributed VCS each developer would be working off of their own personal copy of it that could be synced at any point in time no matter how many changes, revisions, or commits they have made. They're generally not going to check it into their own VCS system because that's work that doesn't get them any real benefit because they already have version control via their checkout and the fact that it's a distributed system. The end result is that they don't need to work without the safety and security of an VCS and they will generally always be working on a system they can easily send you changes from when they're ready.

Chapter 4

Testing

I don't even want to think about how much heat I'm going to get for what I write here but hopefully the testing zealots will agree with me that a heuristic approach to testing is better than a half-assed, or non-existent approach to testing. So, without further ado...

I'm going to assume that you're being good about maintaining a strict MVC (Model View Controller) separation in your app. If not please imagine that I have just walked up beside you, smacked you up-side the head and told you to "Stop being stupid!" If you don't know what MVC is Wikipedia has a good article on it. `http://en.wikipedia.org/wiki/Model-view-controller`

If you Google around you'll see that the prevailing opinion is that you should test everything and that you should be striving for 100% test coverage. And, while this is good, and a laudable goal, for most developers this just isn't realistic for a variety of reasons both personal and environmental. The practice of test driven development (TDD) , or the more recent behavior driven development (BDD) is a great idea, and if you can manage manage it I highly recommend it. But, as much as we'd like to promise ourselves that we'll start doing this right away, most of us know deep down inside that that's a pile of bull. So, instead of an all or nothing approach I'm going to advocate the rarely mentioned heuristically driven development (HDD) approach. It's rarely mentioned because I just made it up. But, HDD, like any other methodology, depends on you understanding its basic concepts, which happen to be the same as the other testing methodologies. So, before we get on with what HDD really means (see section 4.2 on page 13) we need to cover those basic concepts, which you're hopefully already familiar with.

If you have the discipline, or your boss is forcing you, to do TDD development you can skip this chapter entirely. Many congratulations to you for doing things the best possible way. If you're like me, keep reading...

4.1 Basic testing concepts

4.1.1 Unit testing

> In computer programming, unit testing is a procedure used to validate that individual modules or units of source code are working properly.
>
> More technically one should consider that a unit is the smallest testable part of an application. In a Procedural Design a unit may be an individual program, function, procedure, web page, menu etc. But in Object Oriented Design, the smallest unit is always a Class; which may be a base/super class, abstract class or derived/child class.
>
> A unit test is a test for a specific unit. Ideally, each test case is independent from the others; mock objects can be used to assist testing a module in isolation. Unit testing is typically done by the developers and not by end-users.[3]

What some people fail to grasp is the distinction between unit testing, functional testing, integration testing, and system testing. Far too frequently what people call "unit" tests are really "integration" tests. The distinction is important because frequently if you set out to write a unit test but find that you can't without first booting up half of the other components in your app it's a safe bet that you've broken encapsulation and are too tightly coupled to other pieces of the system. In general you should be able to write any unit test with basic language objects or the help of a few mock objects `http://en.wikipedia.org/wiki/Mock_Object`.

4.1.2 Functional / Acceptance testing

> Acceptance testing generally involves running a suite of tests on the completed system. Each individual test, known as a case, exercises a particular operating condition of the user's environment or feature of the system, and will result in a pass or fail boolean outcome. There is generally no degree of success or failure. The test environment is usually designed to be identical, or as close as possible, to the anticipated user's environment, including extremes of such. These test cases must each be accompanied by test case input data or a formal description of the operational activities (or both) to be performed - intended to thoroughly exercise the specific case - and a formal description of the expected results.[3]

Now, since we're taking a heuristic approach you can skip anything "formal" and be realistic. You should have some predefined starting and ending conditions, and some, hopefully automated, way of testing them. But, long before you get to the automated testing part you should have yourself a sit-down with whoever you're developing this thing for and walk them through it, or a mock-up of it. You'll find nine times out of ten that when developing something for a specific group or company it doesn't matter how well it was described it on paper. They'll still say something to the effect of "Oh, well that's nice, but we need it to [insert thing they never mentioned before]."

Once you've gotten something close to what they *actually* wanted (as opposed to what they claimed they wanted) you can start putting together some automated tests for it. I really don't recommend

spending too much time writing functional tests before this point because it's usually easier with tools like Selenium and MaxQ to just record a correct sequence than it is to tweak an existing one.

Most of the time you'll find that the line between Functional tests and System tests (see below) tends to blur, and that's ok... most of the time, but you should always try and decouple the components of your application as much as possible. If you successfully decouple everything you can use mock objects to represent all the interactions with remote machines or applications and just test the local functionality. For example. Imagine your app sends e-mails. Wouldn't it be better if you could test your system without having to have a working SMTP server, appropriate accounts and permissions, and net access? Just use a mock SMTP server and you can test all your e-mail sending routines without having to worry about potential network failures, processing time for your e-mails as they pass through overloaded spam filters, etc.

4.1.3 Integration testing

> Integration testing ... is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing.
>
> Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.[3]

A real world example would be a test that makes sure that the models you're loading from the database and saving to it actually do so correctly. You're testing the integration of your Object Relational Mapping (ORM) system and your persistent storage system. Fortunately for most of us those two components were written by the authors of our favorite web framework and have already been thoroughly tested. Testing them would be redundant and wasteful. What we do need to test though, is the integration of our controllers and models as well as our web service APIs.

4.1.4 System testing

> System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of Black box testing, and as such, should require no knowledge of the inner design of the code or logic.
>
> As a rule, System testing takes, as its input, all of the "integrated" software components that have successfully passed Integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of Integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.[3]

In short, you boot up your app and test that when everything is put together and you start using it like a real person it doesn't go "boom."

4.2 Heuristic Driven Development

If heuristic testing is based on "a commonsense rule (or set of rules) intended to increase the probability of solving some problem". We need to first ask ourselves what the "problem" we're trying to solve is. "Things going boom" is one answer, but lets be honest with ourselves. We're actually ok with *some things* going "boom". If we weren't we'd be working for NASA. Every other development house I know of regularly releases software with bugs in it. As long as nothing too important breaks and nothing breaks in a way that leaves you looking like an idiot there's a good chance you're willing to deal with it. So, now that we've admitted the truth to ourselves, we can start triaging our app.

4.2.1 Triage

> Triage is a system used by medical or emergency personnel to ration limited medical resources when the number of injured needing care exceeds the resources available to perform care so as to treat those patients in most need of treatment who are able to benefit first.
>
> The word triage is a French word meaning "sorting", which itself is derived from the Latin tria, meaning "three". The term literally means sorting into three categories.[3]

My recommendation is to triage your app while you write it because I have yet to meet the person who found writing tests for an existing app to be anything but a chore. If you've come into a project that's already got a lot of untested code this approach still works just as well, if not more so, but do yourself a favor and write tests *while* you're writing the code and it's still fresh in your mind. You and I both know that it just feels like work to have to go back and write them later.

In medical triaging there are three states "minor", "delayed", "immediate" and "morgue". Yeah, I don't know either. Maybe the French didn't want to think about the dead ones... Anyway, your app can be triaged into the same categories, although you'd probably be better off calling them something like "minor", "complex", "critical". and "morgue" .

"Minor" is well, minor. Mostly this is code that you have high confidence in, is so simple that the odds of it breaking are low, and if it does break it's more of an inconvenience than anything else.

"Complex" code is not necessarily mission critical but it is more prone to breaking than a simple loop. If you can refactor it into a set of simple routines do so. If not, you should probably test it because the more complicated something is the greater the chance that you, or your coworkers can screw it up. We're human. We screw stuff up; accept it and take measures to prevent it.

"Critical" code is "mission critical". If that code breaks your app is screwed. Your customers will start brandishing hatchets and leave your site in droves. Your investors will reconsider participating in your next round of financing. This code is *critical*. There is no excuse for not testing this. You don't need to imagine me smacking you up-side the head for this one. Some real person will do it for me when they find out you just crossed your fingers and hoped that "it looked like it was working", actually meant "i know for a fact that it's working."

"Morgue" would be for old dead code that should really be culled. Either it no longer works, it's been commented out, or it's no longer touched by any part of the system. Get rid of it. If you ever need it again it's in your version control system. Leaving it in only makes your system harder to maintain or your code harder to read (because you're skipping over all the commented out sections). In the instances where you are culling code from a working class you may want to leave yourself a note that, "removed code that used to do foo. " and include some reference to the revision / tag / whatever in your VCS where it was last seen.

So, before you write the first line of the function, take a second to ask yourself if you've got your head wrapped around what you're doing enough that you could write the unit test for it right now, or if writing the unit test would help you to understand what you need to code in the function. If you answer yes to either of these then go write the test, then write the function. But, I know you. You're going to skip this at least half the time. Bad developer! No cookie! So, with TDD being skipped ("just this once, really. I promise") for this function we need to triage what we write. If you finish this and move on to anything else without first triaging it I shall, once again, smack you up-side the head. If I'm not around you're going to have to promise to do it yourself. TRIAGE IS NOT OPTIONAL. Got it?

If your triage comes up critical you write the test for it *now*. No checking your e-mail. No surfing. *Do it Now*. If you have to leave, get someone else to write it now or take other measures to guarantee it will be written before you touch another line of code. Do whatever it takes. This code is critical to your apps survival.

If your triage comes up complex you really ought to write a test for it. Putting it off is a bad idea because that introduces the possibility it won't get done.

If your triage comes up minor it's a judgment call. How confident are you about it? Are you sure it'll be trivial if it does break?

4.2.2 When things break.

I don't care if you have 1000% code coverage, even NASA with their incredible attention to detail and quality code with tons of peer review and documentation still releases code with unforeseen bugs. So, it's really just a matter of time before things do go wrong. With that in mind your path going forward is simple:

1. Isolate the bug.
2. Write a test for it. It should, obviously, fail.
3. Fix the bug.
4. Rerun the test.
5. Check fixes into the appropriate branch(es).

If this is a bug on your live site you need to create a new branch of the code-base before step one.

4.2.3 Your development environment

Your development environment is the same environment your app will be running in on the production server. You need at least one main developer working this way. This means that your computer has the exact same versions of your database, whatever language(s) you develop in, and operating system. If you don't do this then how do you know that when you put your site live some missing library or version mismatch or operating system quirk isn't going to bring it all crashing down. These things all need to be part of your system tests, but if you're developing in the same environment your app will be running in on the server then you are constantly doing a small, but very important form of system test.

Chapter 5

Optimization & load tests

Clinton Forbes has a great blog post on this subject `http://clintonforbes.blogspot.com/2007/04/on-shell-be-right-mate-approach-to.html` and much of the writing in this chapter is based on the ideas he set forth.

The common wisdom is that you should never pre-optimize, instead you should wait until you actually discover something is slow and then optimize it. The problem with that isn't the advice but the fact that developers rarely do what it takes to find out if something is slow or not before deploying it. Some people also go the other way and spend time trying to optimize everything, which is generally just a waste of time and money because you spend your time optimizing things that just don't need it.

I wish claim credit for the following list of tips but honestly, it's a paraphrased version of Clinton's:

1. Triage based on predicted usage.
2. Create test data early in realistic amounts.
3. Mirror your production environment.
4. Learn to use profiling software.
5. Set some targets.
6. Use diversionary tactics.

5.1 Triage based on predicted usage.

In most webapps "features" will be equivalent to "pages". Just like we were triaging code to decide what code to test we need to triage our features / pages based on which ones we predict will get the most usage and which of those require the most work or time on the back end.

If we stick with the same triage terms of minor, complex, critical, and morgue in this context they would work out like this:

"minor" features are either really fast or not slow and rarely used. Keep going.

"complex" features are generally going to be edge cases. If you've got the time, or suspect it may be a problem, write a load test.

"critical" features are going to be hit constantly. Ones that require a good amount of work on the back end too are even more critical.

"morgue" features would be those things that are just going to be slow no matter what you do (see section 5.6).

In the beginning "predicted usage" is all you'll have to go on. But, once your site is live be sure to have hit counters that track the usage of specific features. One page may result in one call to feature A and four to feature B. Over time that may be a bigger issue than you initially thought. Or maybe people just don't use the site the way you thought they would. Don't guess when it's so easy enough to track real usage.

5.2 Create test data early in realistic amounts.

> This is one of the very first lessons that I learned as a junior developer many years ago. I wrote a lovely but complex SQL query that ran in 0.01s on our development server but took over a minute to execute on the production database which stored pricing data for Australia's largest retailer... Had the team been working against a full copy of the production database I would have discovered my inefficient code early, while I was actually writing it, rather than later when were were in user testing. - Clinton Forbes

Real data is always best if you can get your hands on it but if you can't just write a script to fill up your database with a realistic amount of varied data. Don't just use it when developing either. Use it in your automated tests too. Putting it in your tests has two advantages. 1) It will provide more edge cases in the data which means your tests will be proving that things work not just in a handful of cases but in a significantly varied set of cases. 2) If your automated tests start taking forever to run on a decent computer you know you need to start looking for things to optimize.

5.3 Mirror your production environment.

This *should* be obvious. Sadly, for many people, it isn't. The best case scenario is to have the development platform match the production environment not just in software (see section 4.2.3) but in hardware too. Unfortunately the best case scenario is frequently not affordable or even possible. Imagine if every one of Google's developer had their own Google infrastructure to test on. In most cases though, it's possible to have a test server set aside that is either identical to the production box or pretty close. It's also important to replicate any supporting boxes. If your database is on a different box than your web server you need to have a test environment that replicates that setup. Otherwise, you'll never see issues brought on by network latency.

This is incredibly important because otherwise you will never be able to get a realistic idea of how your app performs under a repeatable load. Most of us have mediocre development boxes and fairly sweet servers. We could run a load test on our dev boxes but the data we got from it would be almost meaningless. The only thing you could be confident about is that the production box would do better. Which is fine if your load tests are running great on your dev box but can only really tell you which parts are slower relative to others if the load tests don't run great on it.

5.4 Learn to use profiling software.

There are two types of profiling software that come into play here: code profiling, and load testing. The former is specific to your chosen language or platform and a little Googling will go a long way towards finding the best profilers for your environment (assuming there are any). The short short definition is that a code profiler tests the performance of various pieces of your code. They can be terribly useful, especially when you've written nicely modularized code but they're a bit out of scope for this chapter.

When it comes to webapps though, load tests don't have to be tied to your webapp's language or platform at all. There are a number of great open source load testing tools written Java, many of which use Jython. Perl, having been so integral to the web's development has a bunch too. But really it doesn't matter. Look for the tools that will make it easiest to get your tests written. If you can find a tool whose tests can be kicked off by your tests suite it's even better, but since load tests generally take a long time to run they're not something you want to have in every test run.

5.5 Set some targets.

You can't know if your app is "fast enough" if you haven't defined "enough". Is 200 simultaneous users "enough"? It is for most webapps but it certainly isn't for sites like Slashdot. And, don't forget to consider bandwidth issues. If your users are all in the US or the EU you can generally assume they've got broadband connections. If they're scattered across China... well, you may be lucky if they get a 56.6k modem connected to the net for half an hour a day.

5.6 Use diversionary tactics.

> If you can't make your software faster, make it *seem* faster. Developers who create desktop GUI applications have been doing this for years. Keep the user's mind busy with cute little animations and barber-shop style progress bars. 'AJAX' web-applications often retrieve data no faster than 'Web 1.0' applications that refresh the entire page. However the AJAX application may seem faster because the user is not presented with a boring white page but is instead given an animated GIF to pass the time. - Clinton Forbes

I'm not sure that applications actually seem faster when you do this but there have been a number of tests performed over the years that confirm that users are much more accepting of slow procedures if it *looks* as if the computer is doing something as opposed to just sitting there with no feedback.

5.7 Load testing rules of thumb.

Load testing methods and strategies for webapps could fill a book. In fact, it already fills a number of them plus at least one book on testing SOA and Web Services `http://www.amazon.com/gp/product/0123695139`

The short short short version is that there are two basic types of load tests that you'll need to put together. The first test simulates real users and includes significant pauses between page hits that reflect how long you think an anverage user would spend on each of the pages. The second simply pounds your site into submission by repeatedly hitting a feature or set of features until they can't handle any more. These tests, while entirely unrelistic let you know where your app's breaking point is.

It's important to remember when simulating users that you have to stagger their hits because it's highly unlikely that your maximum expected number of users are all going to hit your site in the exact same instant. Even better, randomize the pauses and starting point for each of those users. Then rerun the tests a few times to get an average.

Chapter 6

Application deployment

I'm going to lay out some rules of thumb for deploying webapps and then go into the why they're important. But first I should note that while all of these are very important, when you start using an automated tool like Capistrano you stop having to worry about these, because you just take care of them all in the beginning when you write your deployment script.

6.1 Rules of thumb

1. Your live webapp is never in a state that can't be rolled back to in the future.
2. You have copies of *all* your server / client specific configuration files in version control.
3. Updates are always done atomically.
4. All instances of your webapp are in the same state.

6.1.1 Your live webapp is never in a state that can't be rolled back to in the future.

The basic idea here is that at any given point in time your site is probably in a working state, but it's only a matter of time before some hideous bug rears it's head. Frequently that hideous bug was introduced in a recent update to the site. Obviously you want to just fix the bug and update the site, but sometimes the bug can require far more time to fix than you are willing to allow your live site to be broken for. So, you have to roll back to the last known working version of your site until you can get a patch ready and tested. If you've got your site deployed on multiple boxes you need to be able to roll *all* of them back.

I've seen, and been responsible for, far too many sites where when a bug was found a fix would be made, but since we didn't have good branching habits, because we had tools that made it a pain, we would make that fix in the development branch, in the middle of half-finished new functionality.

We obviously couldn't roll that out so we'd check in the patch and then upload just the affected files to the server(s). Ignoring the obvious problem that the dev branch might have been so altered that a fix there might not fix the bug on the server, we have the problem that, assuming our patch works well, when we go to roll out the new features in a week or so there is no way we'll be able to roll the server back to that patched, and working, state if we need to because it was an essentially random collection of files from various points in development.

The solution is simple, but depends on you having good branching habits. You have your production branch, you make your changes elsewhere and when they're done merge them into the production branch. Then, if the merging process didn't already do this, you tag this newly patched version of the production branch in such a way that you can easily remember/find it should you ever need to roll back to it. And, of course, you only push code from the production branch to the live server(s).

6.1.2 You have copies of *all* your server / client specific configuration files in version control.

Most of the sites I've worked on that require different configuration settings on the live site than they do on the dev boxes, or that have multiple live sites (each with their own settings), tend to have the conf files in place on the server and nowhere else. The problem is that the same file has to be in different states for each server. I've also seen the same thing happen when instead of server specific settings they were customer specific settings, like customer name and logo for a customer specific installation. But not backing these up in version control is just crazy. What if your server goes up in flames?

This is an incredibly easy problem to fix. You just make a folder for each type of conf file, naming it, and them in some sensible way. For example: let's say you're using Apache and need a different http.include file for each server. Just make an http_includes folder and fill it with files named server_foo.http.include, server_bar.http.include, etc.. If you're using an automated deployment tool you can just configure it to copy the appropriate file out of there, stick it in the appropriate place for the server to find it, and rename it accordingly. And, since you're almost never going to have to actually touch those files, it's not a big deal to do the same thing manually. You can do the exact same thing for customer specific files.

6.1.3 Updates are always done atomically.

Murphy, is alive and well. His law pervades our industry. Do you really want to risk the welfare of your live site on the hope that your net connection won't go out, that your ethernet cable won't get yanked, or that your favorite deity won't choose today to screw with you in some other way while you're trying to update the site? I don't either. Along the same lines you don't want a user hitting your site while you're in the middle of an update and thus getting some Frankenstinian combination of new and old code.

Upload the new version of the site to a temp directory on the production box. If you can run a test suite against this temp version without negatively affecting the live version do so. If the upload worked and your tests passed you have two options:

1. Rename the temp dir to something final and repoint the server at it. This will probably result in the least downtime. Just make sure none of your users will end up in a halfway state or that part of their session will be tied to the old install, or anything like that.

2. Down your site and put up a maintenance notice. Hopefully you're only doing this during a scheduled downtime that you've let your users know about in advance, but we all know that's not always the case. Move the old app to a backup location, move in the new version and take down the notice.

If you have to perform any updates to the database, or anything else that would potentially leave the site in an funky trasitional state, you don't really have a choice about putting up a maintenance notice during that time.

The only real exception being when the update is just a matter of replacing one or two files that you know for a fact won't have any ripple effects, like a CSS file and can be moved in without having to down the site, but you should still be sure to upload to a temp directory first and then move it into the live site (after backing up the old one) because imagine how bad it would be if the main CSS file for your site got truncated by a bad connection mid-upload.

6.1.4 All instances of your webapp are in the same state.

If you do an update in one place you do it everywhere. It's simple. That way when customer A reports a bug you won't have to say, "Did we update their server with patch foo?" Because you know. Either all the apps have it or none of the apps have it. I know, this is an obvious rule. But people violate it all the time because we're human, and thus don't want to do boring work. This is yet another reason you need an automated deployment system.

6.2 Automated application deployment

Automated application deployment is a simple solution to all of these problems. With a tool like Capistrano `http://manuals.rubyonrails.com/read/book/17` you can write a simple deployment script that will log into the server(s), check out the latest (or specified), version of your code, make backups of each deployment for emergency rollbacks, and restart your server or execute any other commands you need it to (tests for example). If updating the site fails it will automatically go back to the last version. The short amount of time you'll spend writing the deployment script will pay for itself in spades by giving you confidence that all of the rules mentioned above have been addressed and by making it all happen with one reliable command.

As of this writing Capistrano seems to be the leader in the field of webapp deployment tools. But all of the docs on it are currently Rails specific, even though you could use it for deploying anything remotely. Rails developers should definitely be using it. Everyone else should either spend the time to translate the Railsisms in it's docs to something aprropriate or be ransacking it for ideas to make a new build system.

When considering using, or building, an automated deployment system you should keep a few things in mind:

- Make sure it is integrated with your version control system. You need to be able to tell it to deploy a specific version of your app.
- Make sure it can execute commands on the remote server. You'll want to be able to down the running app, copy the new files in place, and restart the app if nothing else.
- Make sure it can run your unit tests before deploying.
- It would be good if it had an easy way to roll back the live app(s) when/if something goes wrong.

Bibliography

[1] Clinton Forbes, On the "She'll be right, Mate" approach to software optimization http://clintonforbes.blogspot.com/2007/04/on-shell-be-right-mate-approach-to.html 11 Apr. 2007.

[2] Mark Reinhold, Source-control management for an open JDK http://blogs.sun.com/mr/entry/openjdk_scm 26 Sept. 2006.

[3] Wikipedia, various authors, various dates. Relevant pages: Unit Testing http://en.wikipedia.org/wiki/Unit_testing, Functional Testing http://en.wikipedia.org/wiki/Functional_testing, Integration Testing http://en.wikipedia.org/wiki/Integration_testing, System testing http://en.wikipedia.org/wiki/System_testing, and Triage http://en.wikipedia.org/wiki/Triage.