

HOBOS

AT WORK

The Definitive Guide to the Hobo Extensions for Ruby on Rails

TOM LOCKE • BRYAN LARSEN • OWEN DALL

CONTENTS

CONTENTS	i
LIST OF FIGURES.....	iv
AUTHORS.....	vi
PREFACE.....	vii
CHAPTER 1 – INTRODUCTION.....	1
What is Hobo?	1
CHAPTER 2 – INSTALLATION	8
Introductory Concepts and Comments.....	8
Installing Ruby, Rails, Hobo.....	1
Installing Hobo as a Plugin to an Existing Rails App.....	6
Using SQLite with Hobo.....	9
Using Oracle with Hobo.....	16
Chapter 3 - Hobo Fundamentals	27
Rails and Hobo.....	29
Hobo In Action.....	34
Hobo Enhancement Summary	51
Chapter 4 – The Hobo Permissions Systems.....	56
Introduction	56
Defining permissions.....	57
Change tracking	58
Permissions and associations	64
The Permission API.....	66
Permissions vs. validations.....	69
View helpers.....	71
Chapter 5 - Hobo Controllers and Routing	72
Introduction	72
Owner actions.....	73
Adding extra actions	74
Changing action behavior	75
Writing an action from scratch	76
The default actions	79
Owner actions.....	80
Autocompleters.....	82
Further Customization.....	83
Drag and drop reordering	84

Chapter 6 - Hobo Lifecycles	86
Introduction	86
Key concepts	91
Defining a lifecycle.....	92
Defining states.....	94
Defining creators.....	94
Defining transitions.....	95
Repeated transition names	97
Validations	99
Controller actions and routes	99
Transitions	102
Keys and secure links	104
 Chapter 7 - Hobo View Hints.....	 107
Ingtrouction	107
Defining hints.....	108
Child relationships.....	110
Inline Booleans.....	110
The API	111
 Chapter 8 - Hobo Scopes.....	 112
Simple Scopes	112
Boolean Scopes.....	112
Date Scopes.....	112
Lifecyle Scopes	112
Key Scopes.....	112
Static Scopes.....	112
Association Scopes	113
Scoping Associations	113
Chaining.....	113
Simple Scopes	114
Boolean scopes.....	115
Date scopes	115
Lifecycle scopes.....	116
Key scopes.....	116
Association Scopes	117
Scoping Associations	118
Chaining.....	119
 Chapter 9 – The Hobo DRYML Guide.....	 120
What is DRYML?	120
Simple page templates and ERB	121
Where are the layouts?	122
Defining simple tags.....	122
Parameters.....	123
Changing Parameter Names	125
Multiple Parameters	125
Default Parameter Content	126
The Default Parameter.....	127
The Implicit Context.....	128

Field chains	131
Tag attributes.....	131
Flag attributes.....	133
Merging Attributes	133
Merging selected attributes.....	134
Repetition	136
Even/odd classes.....	137
Using the implicit context.....	138
Pseudo parameters - before, after, append, prepend, and replace.....	140
Nested parameters	143
Customizing and extending tags.....	146
Aliasing tags.....	151
Polymorphic tags.....	151
Wrapping content	154
Local variables and scoped variables.....	156
Taglibs.....	158
Divergences from XML and HTML.....	158
Chapter 10 – The Hobo Rapid Tag Library	161
Rapid Tag Library Index.....	162
Core	163
Rapid Core.....	165
Rapid Document Tags	175
Rapid Editing.....	177
Rapid Forms	180
Rapid Generics.....	191
Rapid Lifecycles	193
Rapid Navigation.....	194
Rapid Pages.....	196
Rapid Plus.....	199
INDEX.....	214

LIST OF FIGURES

Figure 1: Download Site for Ruby.....	1
Figure 2: Installing Ruby	2
Figure 3: Ruby Installation Options.....	2
Figure 4: Setup Wizard Complete.....	3
Figure 5: Sample console output after installing the Hobo gem.....	4
Figure 6: Summary of Installed gems	4
Figure 7: Sample console output from the "gem env" command	5
Figure 8: Sample console output from installing the sqlite3-ruby gem.....	9
Figure 9: SQLite3 download website	9
Figure 10: Target location for the SQLite3 DLL.....	10
Figure 11: Download site for MySQL.....	11
Figure 12: Using the .msi file to install MySQL on Windows	11
Figure 13: Choose the installation type	12
Figure 14: MySQL Server Setup Wizard.....	12
Figure 16: Choose Standard Configuration	13
Figure 15: Configure MySQL Server	13
Figure 17: Install as Windows Service	14
Figure 18: Launch MySQL from the command prompt.....	14
Figure 19: Create the database from the command line	15
Figure 20: Console output from the Hobo command	15
Figure 21: The MySQL format for the database.yml configuration file.....	16
Figure 22: Console output from installing the Ruby gem for Oracle	17
Figure 23: The generated database.yml file for Oracle.....	18
Figure 24: Oracle database install download site	19
Figure 25: Running the Oracle XE installation.....	20
Figure 26: Specifying the database passwords	20
Figure 27: Launch the Database home page	21
Figure 28: Log in as SYS to configure your database.....	21
Figure 29: Creating a schema/user to use with Hobo	22
Figure 30: The tnsnames.ora file created during installation.....	22
Figure 31: Log into Oracle to view the created table.....	23
Figure 32: Access the Oracle Object Browser	23
Figure 33: Review the User table from within Oracle	24
Figure 34: Review the Indexes view for Users.....	24
Figure 35: Review the Constraints view for User.....	25
Figure 36: Data flow for a typical Application using a MVC framework.....	29
Figure 37: Data flow for a Rails application.....	30
Figure 38: Data flow for a Hobo application	30
Figure 39: First level look at Hobo source.....	31
Figure 40: Listing of Ruby programs within the Hobosupport folder	32
Figure 41: Content overview for the Hobofields gem	33
Figure 42: Command line options for Hobo Migrations	34

Figure 43: Required hobosupport and hobofield gems	35
Figure 44: Optional parameters for the Hobo command	35
Figure 45: The HoboGenerator class actions	39
Figure 46: The line “config.gem ‘hobo’ is added in environment.rb by Hobo	40
Figure 47: The lines added to the file “rake” by Hobo	40
Figure 48: Users Controller generated by Hobo	43
Figure 49: User model with Lifecycles generated by Hobo	44
Figure 50: Action Mailer Model generated by Hobo	44
Figure 51: User model generated for an “--invite-only” Hobo application	47
Figure 52: Users Controller generated with an “--invite-only” Hobo application	47
Figure 53: Action Mailer model generated with an “--invite-only” Hobo application	48
Figure 54: Source code for “hobo_front_controller_generator.rb”	50
Figure 55: Hobo Rapid action related tags	53
Figure 56: Hobo precedence logic for action tags	55
Figure 55: Installing Ruby	55
Figure 55: Download Site for Ruby	55
Figure 57: Defining the Friendship model	88
Figure 58: The contents of the “summary.dryml” file	202
Figure 59: Sample view of the first section of an application summary page	202

AUTHORS

Tom Locke

Tom is the founder and original developer of the Hobo project. He is also co-founder of Artisan Technology, a software and web development company exploring commercial opportunities around Hobo and other open-source projects. Prior to founding Artisan Technology Tom has been a freelance software developer for over ten years, and has been experimenting with innovative and agile approaches to web development since as early as 1996.

Bryan Larsen

Bryan sold his first video game in 1987 and has never stopped. Joining the ranks of fathers this year has slowed him down, but he's still having fun. He lives in Ottawa with his wife and daughter. Bryan is a key contributor to Hobo and has nursed it along to a mature 1.0 version.

Owen Dall

Owen Dall has been Chief Systems Architect for Barquin International for the past seven years. During that time he has led a data warehousing, business intelligence, and web systems practice and has become an evangelist for agile development methodologies. His search for replacements to Java web frameworks led him to Hobo open-source environment for Ruby on Rails (RoR) in late 2007. In his 25+ years software development experience, he has authored several software packages used by diverse clients in both the private and public sectors.

PREFACE

“Hobo at Work” is the second book in a series about Hobo. The first in this series, “Rapid Rails with Hobo”, provides 27 comprehensive tutorials for those new to Hobo. We deliberately kept theory to a minimum and successful building of applications to a maximum.

“Hobo at Work” builds on the knowledge gained with “Rapid Rails with Hobo” with in-depth discussions of each Hobo module, and includes a comprehensive reference to all Hobo commands and each member of the large Hobo library of DRYML Tags. This will provide you another level of expertise and confidence to build even more powerful web applications.

Please check out the latest regarding Hobo at <http://hobocentral.net> and join the “hobousers” google group to post questions and comments during you explorations with Hobo. The community is generous and growing fast. Hope to see you there!

CHAPTER 1 – INTRODUCTION

What is Hobo?

By Tom Locke

Hobo is a software framework that radically reduces the effort required to develop database-driven, interactive web sites and web-based applications. Strictly speaking it's more of a "half-framework" — Hobo builds on the amazingly successful Ruby on Rails and that's where much of the functionality comes from. The original motivation for the Hobo project can be summed up pretty succinctly with a single sentiment: "Do I really have to code all this stuff up again?"

In other words Hobo is about not re-inventing the wheel. In software-engineer-speak, we call that code reuse. If you mention that term in a room full of experienced programmers you'll probably find yourself the recipient of various frowns and sighs; you might even get laughed at. It all sounds so simple - if you've done it before just go dig out that code and use it again. The trouble is, the thing you want to do this time is just a bit different, here and there, from what you did last time. That innocuous sounding "just a bit different" turns out to be a twelve-headed beast that eats up 150% of your budget and stomps all over your deadline. Re-use, it turns out, is a very tough problem. Real programmers know this. Real programmers code it up from scratch.

Except they don't. Ask any programmer to list the existing software technologies they drew upon to create their Amazing New Thing and you had better have a lot of time to spare. Modern programming languages ship with huge class libraries, we rely on databases that have unthinkable amounts of engineering time invested in them, and our web browsers have been growing more and more sophisticated for years. Nowadays we also draw upon very sophisticated online services, for example web based mapping and geo-location, and we add features to our products that would otherwise have been far beyond our reach.

So it turns out the quest for re-use has been a great success after all—we just have to change our perspective slightly, and look at the infrastructure our application is built on rather than the application code itself. This is probably because our attitude to infrastructure is different—you like it or lump it. If your mapping service doesn't provide a certain feature, you just do without. You can't dream of coding up your own mapping service, and some maps is better than no maps.

We've traded flexibility for reach, and boy is it a good trade.

Programmers get to stand on the shoulders of giants. Small teams with relatively tiny budgets can now successfully take on projects that would have been unthinkable a decade ago. How far can this trend continue? Can team sizes be reduced to one? Can timelines be measured in days or weeks instead of months and years? The answer is yes, if you are willing to trade flexibility for reach.

In part, this is what Hobo is about. If you're prepared for your app to sit firmly inside the box of Hobo's "standard database app", you can be up and running with startlingly little effort. So little, in fact, that you can just about squeeze by without even knowing how to program. But that's only one part of Hobo. The other part comes from the fact that nobody likes to be boxed in. What if I am a programmer, or I have access to programmers? What if I don't mind spending more time on this project?

We would like this "flexibility for reach" tradeoff to be a bit more fluid. Can I buy back some flexibility by adding more programming skills and more time? In the past this has been a huge problem. Lots of products have made it incredibly easy to create a simple database app, but adding flexibility has been an all-or-nothing proposition. You could either stick with the out-of-the-box application, or jump off the "scripting extensions" cliff, at which point things get awfully similar to coding the app from scratch.

This, we believe, is where Hobo is a real step forward. Hobo is all about choosing the balance between flexibility and reach that works for your particular project. You can start with the out-of-the-box solution and have something up and running in your first afternoon. You can then identify the things you'd like to tweak and decide if you want to invest programming effort in them. You can do this, bit by bit, on any aspect of your application, from tiny touches to the user-interface, all the way up to full-blown custom features.

In the long run, and we're very much still on the journey, we hope you will never again have to say "Do I really have to code all this up again?", because you'll only ever be coding the things that are unique to this particular project. To be honest that's probably a bit of a utopian dream, and some readers will probably be scoffing at this point—you've heard it all before. But if we can make some progress, any progress in that direction, that's got to be good, right? Well we think we've made a ton of progress already, and there's plenty more to come!

Background

A brief look at the history leading up to Hobo might be helpful to put things in context. We'll start back in ancient times — 2004. At that time the web development scene was hugely dominated by Java with its "enterprise" frameworks like EJB, Struts and Hibernate. It would be easy, at this point, to launch into a lengthy rant about over-engineered technology that was designed by committee and is painful to program with. But that has all been done before. Suffice it to say that many programmers felt that they were spending way too much time writing repetitive "boilerplate" code and the dreaded XML configuration files, instead of focusing on the really creative stuff that was unique to their project. Not fun and definitely not efficient.

One fellow managed to voice his concerns much more loudly than anyone else, by showing a better way. In 2004 David Heinemeier Hansson released a different kind of framework for building web apps, using a then little-known language called Ruby. A video was released in which Hansson created a working database-driven Weblog application from scratch in less than

15 minutes. That video was impressive enough to rapidly circulate the globe, and before anyone really even knew what it was, the Ruby on Rails framework was famous.

Like most technologies that grow rapidly on a wave of hype, Rails (as it is known for short) was often dismissed as a passing fad. Five years later the record shows otherwise. Rails is now supported by all of the major software companies and powers many household-name websites.

So what was, and is, so special about Ruby on Rails? There are a thousand tiny answers to that question, but they all pretty much come down to one overarching attitude. Rails is, to quote its creator, opinionated software. The basic idea is very simple: instead of starting with a blank slate and requiring the programmer to specify every little detail, Rails starts with a strong set of opinions about how things should work. Conventions which “just work” 95% of the time. “Convention over Configuration” is the mantra. If you find yourself in the 5% case where these conventions don’t fit, you can usually code your way out of trouble with a bit of extra effort. For the other 95% Rails just saved you a ton of boring, repetitive work.

In the previous section we talked about trading flexibility for reach. Convention over configuration is pretty much the same deal: don’t require the programmer to make every little choice; make some assumptions and move swiftly on. The thinking behind Hobo is very much inspired by Rails. We’re finding out just how far the idea of convention over configuration can be pushed. For my part, the experience of learning Rails was a real eye-opener, but I immediately wanted more.

I found that certain aspects of Rails development were a real joy. The “conventions”—the stuff that Rails did for you—were so strong that you were literally just saying what you wanted, and Rails would just make it happen. We call this “declarative programming”. Instead of spelling out the details of a process that would achieve the desired result, you just declare what you want, and the framework makes it happen. What, not how.

The trouble was that Rails achieved these heights in some areas, but not all. In particular, when it came to building the user interface to your application, you found yourself having to spell things out the long way.

It turned out this was very much a conscious decision in the design of Ruby on Rails. David Heinemeier Hansson had seen too many projects bitten by what he saw as the “mirage” of high-level components:

I worked in a J2EE shop for seven months that tried to pursue the component pipe dream for community tools with chats, user management, forums, calendars. The whole shebang. And I saw how poorly it adapted to different needs of the particular projects.

On the surface, the dream of components sounds great and cursory overviews of new projects also appear to be “a perfect fit”. But they never are. Reuse is hard. Parameterized reuse is even harder. And in the end, you’re left with all the complexity of a Swiss army knife that does everything for no one at great cost and pain.

I must say I find it easy to agree with this perspective, and many projects did seem, in hindsight, to have been chasing a mirage. But it's also a hugely dissatisfying position. Surely we don't have to resign ourselves to re-inventing the wheel forever? So while the incredibly talented team behind Rails have been making the foundations stronger, we've been trying to find out how high we can build on top of those foundations. Rather than a problem, we see a question — why do these ideas work so well in some parts of Rails but not others? What new ideas do we need to be able to take convention over configuration and declarative programming to higher and higher levels? Over the last couple of years we've come up with some pretty interesting answers to those questions.

In fact one answer seems to be standing out as the key. It's been hinted at already, but it will become clearer in the next section when we compare Hobo to some other seemingly similar projects.

The Difference

There are a number of projects out there that bear an external resemblance to Hobo. To name a few, in the Rails world we have Active Scaffold and Streamlined, and the Python language has Django, a web framework with some similar features.

There is some genuine overlap between these projects and Hobo. All of them (including Hobo) can be used to create so called “admin interfaces”. That is, they are very good at providing a straightforward user-interface for creating, editing and deleting records in our various database tables. The idea is that the site administrator, who has a good understanding of how everything works, does not need a custom crafted user-interface in order to perform all manner of behind-the-scenes maintenance tasks. A simple example might be editing the price of a product in a store. In other words, the admin interface is a known quantity: they are all largely the same.

Active Scaffold, Streamlined, Django and Hobo can all provide working admin sites like these with very little or even no programming effort. This is extremely useful, but Hobo goes much further. The big difference is that the benefits Hobo provides apply to the whole application, not just the admin interface, and this difference comes from Hobo's approach to customization.

Broadly speaking, these “admin site builder” projects provide you a very complete and useful out-of-the-box solution. There will be a great number of options that can be tweaked and changed, but these will only refine rather than reinvent the end result. Once you've seen one of these admin-sites, you've pretty much seen them all. That's exactly why these tools are used for admin sites - it generally just doesn't matter if your admin site is very alike any other. The same is far from true for the user-facing pieces of your application—those need to be carefully crafted to suit the needs of your users.

Hobo has a very different approach. Instead of providing options, Hobo provides a powerful parameterization mechanism that lets you reach in and completely replace any piece of the generated user-interface, from the tiny to the large.

This difference leads to something very significant: it gets you out of making a difficult all-or-nothing decision. An admin site builder does one thing well, but stops there. For every piece of your site you need to decide: admin interface or custom code? With Hobo you can start off using the out-of-the-box UI as a rough prototype, and then gradually replace as much or as little as you need in order to get the exact user experience you are after.

Once again we find ourselves back at the original idea: making a tradeoff between flexibility and reach. The crucial difference with Hobo, is that you get to make this trade-off in a very fine-grained way. Instead of all-or-nothing decisions (admin-site-builder vs. custom-code), you make a stream of tiny decisions. Should I stick with Hobo's automatically generated form? Sidebar? Button? How long would it take me to replace that with something better? Is it worth it?

There is a wide spectrum of possibilities, ranging from a complete out-of-the-box solution at one end to a fully tailored application at the other. Hobo lets you pick any point on this spectrum according to whatever makes sense right now. Not only that but you don't have to pick a point for the app as a whole. You get to make this decision for each page, and even each small piece of each page.

The previous section posed the question: "how can the ideas of declarative programming be taken to higher and higher levels?". We mentioned before that one particular answer to this question has stood out as crucial: it is the approach we have taken to customization. It's not what your components can do, it's how they can be changed that matters. This makes sense—software development is a creative activity. Developers need to take what you're giving them and do something new with it.

It is this difficulty of customization that lies at the heart of concerns with high-level components: David Heinemeier Hansson again:

...high-level components are a mirage: By the time they become interesting, their fitting will require more work than creating something from scratch.

The typical story goes like this: you need to build something that "surely someone must have done before?"; you find a likely candidate - maybe an open-source plugin or an application that you think you can integrate; then as you start the work of adjusting it to your needs it slowly becomes apparent that it's going to be far harder than you had anticipated. Eventually you end up wishing you had built the thing yourself in the first place.

To the optimistic however, a problem is just an opportunity waiting to be taken. We're hitting a limit on the size of the components we can build—too big and the effort to tailor them makes it counterproductive. Turn that around and you get this: if you can find a way to make customization easier, then you can build bigger components. If it's the "fitting" that's the problem, let's make them easier to fit! That's exactly what we're doing.

The Future

At the time of writing we are just mopping up the last few bugs on the list before the release of Hobo version 1.0. It looks like we're finished! In fact we're just getting started.

Bigger library

Obviously the whole point in discovering the secrets of how to build high-level components, is that you want to build some high level components! In other words there are two distinct aspects to the Hobo project: getting the underlying technology right, and then building some cool stuff with it. Hobo 1.0 will ship with a decent library of useful “building blocks” to get your app up and running quickly, but there's so much more we'd like to see. This is where the magic of open-source needs to come into play. The better Hobo gets, the more developers will want to jump on board, and the bigger the library will grow.

Although the underlying framework is the most technically challenging part of the project, in the long run there's much more work to be done in the libraries. And writing the code is just part of the story. All these contributions will need to be documented and catalogued too.

We've started putting the infrastructure in place with “The Hobo Cookbook” website (<http://cookbook.hobocentral.net>) - a central home for both the “official” and user-contributed documentation.

Performance improvements

It would be remiss not to mention that all these wonderful productivity gains do come at a cost - a Hobo application does have an extra performance overhead compared to a “normal” Rails application. Experience has shown it's not really a big problem - many people are using Hobo to prototype, or to create a very niche application for a small audience. In these cases the performance overhead just doesn't matter. If you do have a more serious application that may need to scale, there are well known techniques to apply, such as prudent use of caching.

The argument is pretty much the same as that told by early Rails coders to their Java based critics. It's much better to save a ton of development time, even if it costs you some of your raw performance. The time saved can be used to work on performance improvements in the architecture of the app. You typically end up with an app that's actually faster than something built in a lower-level, “faster” language.

Another way to look at it—it was about four or five years ago that Rails was getting a lot of pushback about performance. In those four or five years, Moore's Law has made our servers somewhere between five and ten times faster. If Rails was fast enough in 2005 (it was), Hobo is certainly fast enough today.

Having said all that, it's always nice to give people more performance out-of-the-box and postpone the day that they have to resort to app-specific efforts. Just as Rails has focused a lot on

performance in the last couple of years, this is definitely an area that we will focus on in the future.

Less magic

One of the most common criticisms leveled against Hobo is that it is “too magic”. This tends to come from very experienced developers who like to know exactly how everything is working. Because Hobo gives you so much out-of-the-box, it’s inevitable that you’ll be scratching your head a bit about where it all comes from in the early days. Fortunately this is mostly just a matter of the learning curve. Once you’ve oriented yourself, it’s pretty easy to understand where the various features come from, and hence where to look when you need to customize.

As Hobo has developed, we’ve definitely learnt how important it is to make things as clear and transparent as we can. The changes from Hobo 0.7 to 0.8 removed a great deal of hard to understand “magical” code. This is definitely a trend that will continue. We’re very confident that future versions will be able to do even more for you, while at the same time being easier to understand. It’s a challenge—we like challenges!

Even higher level

One of the really interesting things we’ve learnt through releasing Hobo as open source, has been that it has a very strong appeal to beginners. It is very common for a post to the “hobousers” discussion group to start “I am new to web programming” or “This is my first attempt to create a web app”. It seems that, with Hobo, people can see that a finished result is within their reach. That is a powerful motivator.

Now that we’ve seen that appeal, it’s really interesting to find out how far we can push it. We’ve already seen simple Hobo applications created by people that don’t really know computer programming at all. Right now these people are really rather limited, but perhaps they can go further.

Hobo has ended up serving two very different audiences: experienced programmers looking for higher productivity, and beginners looking to achieve things they otherwise couldn’t. Trying to serve both audiences might sound like a mistake, but in fact it captures what Hobo is all about. Our challenge is to allow the programmer to choose his or her own position on a continuous spectrum from “incredibly easy” to “perfectly customized”.

Hopefully this introduction has whetted you’re appetite and you’re keen to roll up your sleeves and find out how it all works. While this section has been a bit on the philosophical side, the rest of the book is eminently practical. From now on we’ll dispense with all the highbrow pontificating and teach you how to make stuff. Enjoy!

CHAPTER 2 – INSTALLATION

Introductory Concepts and Comments

To encourage the widest audience possible, the following instructions are tailored for Windows, which is still the most commonly used operating system in the enterprise. It has been our experience that Mac and Linux users can translate much more easily to Windows vernacular than Windows users to Mac OS X or Linux.

Although we include detailed instructions for configuring MySQL and Oracle databases with Hobo, we encourage you to start the tutorials using the lightweight and self-configuring database engine, SQLite3, which is the default engine used by Hobo and Rails when in development mode. This allows you to focus on learning Hobo, not configuring a database.

Most books and online tutorials on Ruby and Rails are tailored to Mac users, and pay lip service to Windows, assuming the reader is already facile with web development tools and uses the MacBook Pro as the “weapon of choice”. This book also assumes that many of you are trying out Hobo, Ruby, and Rails for the first time and that a large percentage will also be using either Windows XP, Vista, or Windows 7 on a day-to-day basis. We don’t want that minor factor to limit your development enjoyment. Mac and Linux users may also easily read this book, as we have provided the necessary references for installation instructions in these environments.

So--get your favorite web browser fired up, have a good cup of coffee handy, and follow the instructions below.

Installing Ruby, Rails, Hobo

If you already have Ruby and Rails installed, you can skip this section and instead go straight to resources at:

<http://hobocentral.net/two-minutes/>

If you have a Mac with OS X, Ruby 1.8.6 and Rails 1.2.3 are pre-installed. You can skip step 1 and go straight to step2.

The following is a good blog resource for alternative installation for Mac users:

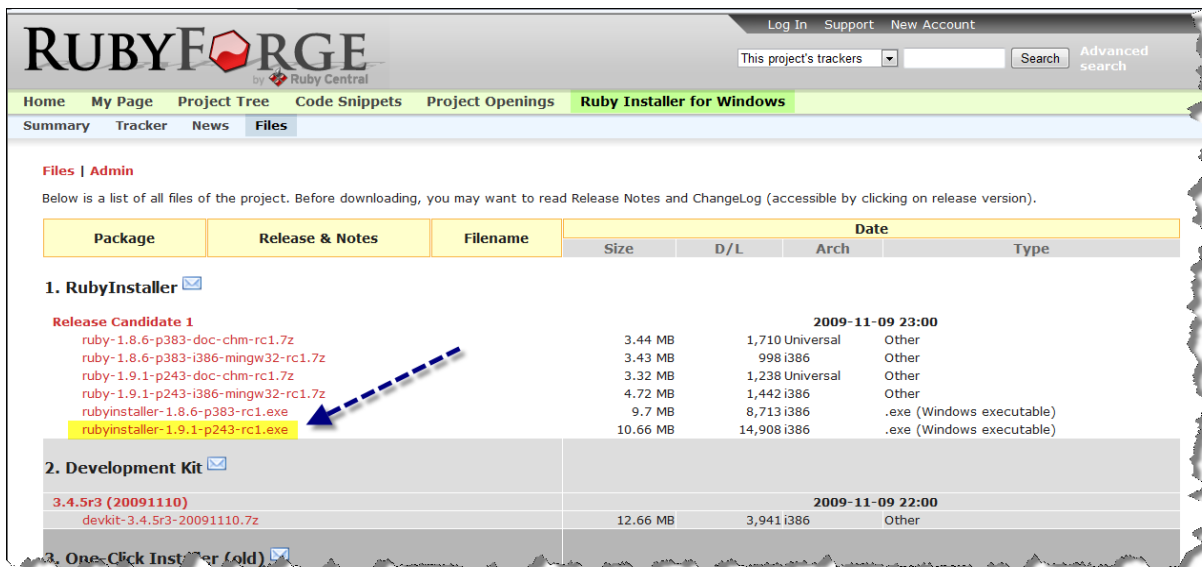
<http://hivelogic.com/articles/2008/02/ruby-rails-leopard>

For Linux aficionados:

<http://linxtips.today.com/2009/01/04/installing-ruby-on-rails-on-linux/>

1. At the time of this writing (December, 2009) there is a release candidate for Ruby 1.9.1 for Windows available (The latest “formal” release for the one-click Windows installer was ruby186-26.exe) from rubyforge.org:

http://rubyforge.org/frs/?group_id=167



The screenshot shows the RubyForge website interface. The top navigation bar includes links for Home, My Page, Project Tree, Code Snippets, Project Openings, and Ruby Installer for Windows. Below the navigation bar, there is a section titled 'Files | Admin' with a sub-header 'Below is a list of all files of the project. Before downloading, you may want to read Release Notes and ChangeLog (accessible by clicking on release version).' A table lists the files with columns for Package, Release & Notes, Filename, Size, D/L, Arch, and Date. The table is divided into three sections: 1. RubyInstaller, 2. Development Kit, and 3. One-Click Installer (old). A blue dashed arrow points to the file 'rubyinstaller-1.9.1-p243-rc1.exe' in the first section.

Package	Release & Notes	Filename	Size	D/L	Arch	Date
1. RubyInstaller						
Release Candidate 1						
		ruby-1.8.6-p383-doc-chm-rc1.7z	3.44 MB	1,710 Universal	Other	2009-11-09 23:00
		ruby-1.8.6-p383-i386-mingw32-rc1.7z	3.43 MB	998 i386	Other	
		ruby-1.9.1-p243-doc-chm-rc1.7z	3.32 MB	1,238 Universal	Other	
		ruby-1.9.1-p243-i386-mingw32-rc1.7z	4.72 MB	1,442 i386	Other	
		rubyinstaller-1.8.6-p383-rc1.exe	9.7 MB	8,713 i386	.exe (Windows executable)	
		rubyinstaller-1.9.1-p243-rc1.exe	10.66 MB	14,908 i386	.exe (Windows executable)	
2. Development Kit						
3.4.5r3 (20091110)						
		devkit-3.4.5r3-20091110.7z	12.66 MB	3,941 i386	Other	2009-11-09 22:00
3. One-Click Installer (old)						

Figure 1: Download Site for Ruby

Download and double-click on the file `rubyinstaller-1.9.1-p243.rc1.exe` to run the installer:

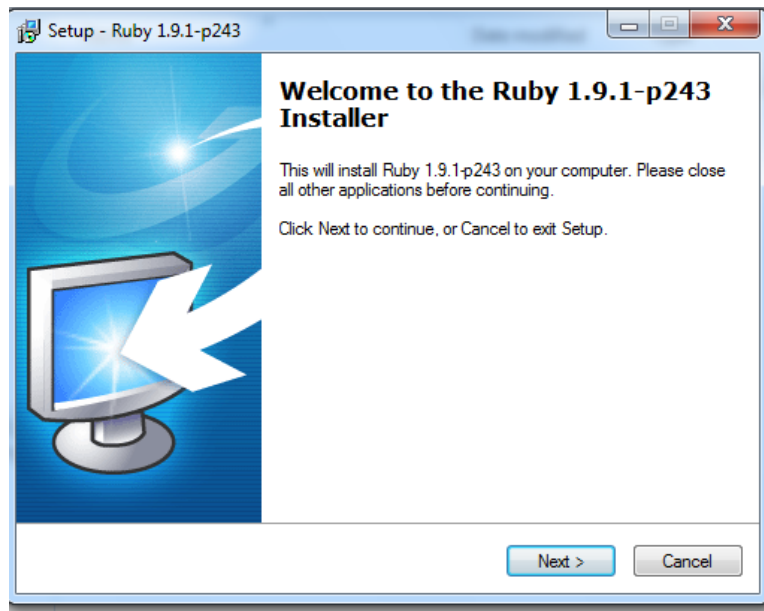


Figure 2: Installing Ruby

You can install ruby on any drive or folder, but for the purposes of the tutorials we will be using the default `c:\ruby19` folder.

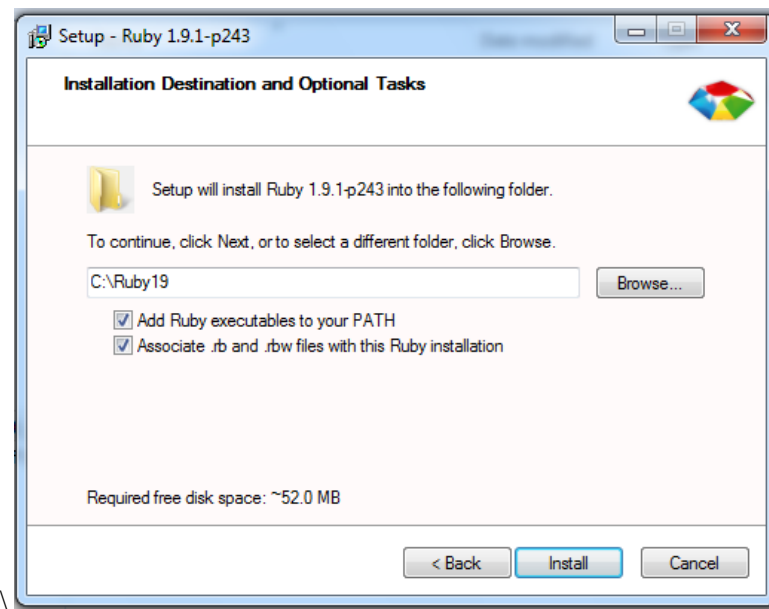


Figure 3: Ruby Installation Options

The installer will create a larger number of folders under the `Ruby19` folder, or an alternative folder if you specified on.

When the installation is complete you will see a popup window similar to the following:



Figure 4: Setup Wizard Complete

2. Add the “github”, “rubyonrails”, and “gemcutter” websites as sources to look for Ruby packages:

```
C:\ruby19> gem sources -a http://gems.github.com
C:\ruby19> gem sources -a http://gems.rubyonrails.org
C:\ruby19> gem sources -a http://gemcutter.org
```

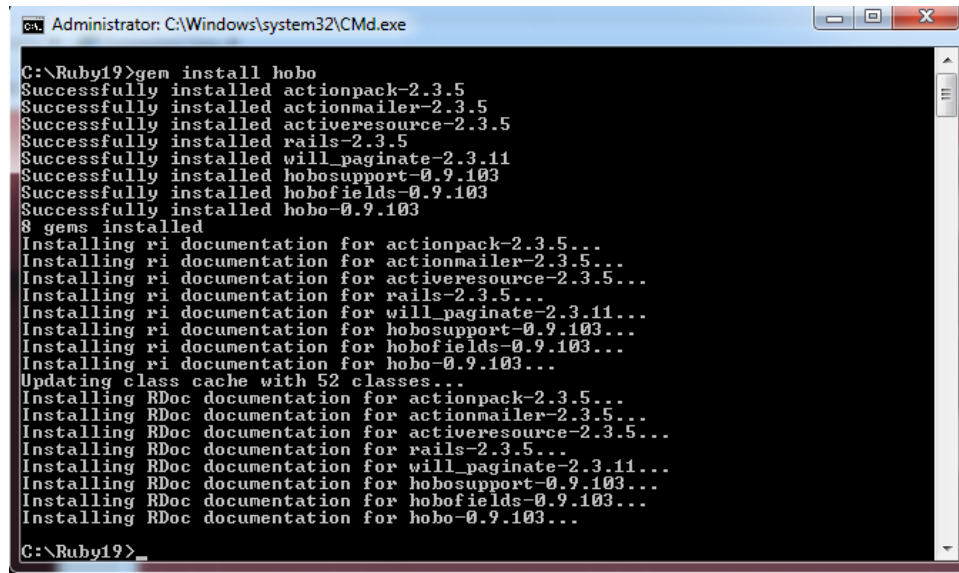
Also install “gemcutter” for future use:

```
C:\ruby19> gem install gemcutter
```

<http://gemcutter.org/pages/about>

2. Open up a command prompt and install the latest version of Hobo:

```
C:\ruby9> gem install hobo
```



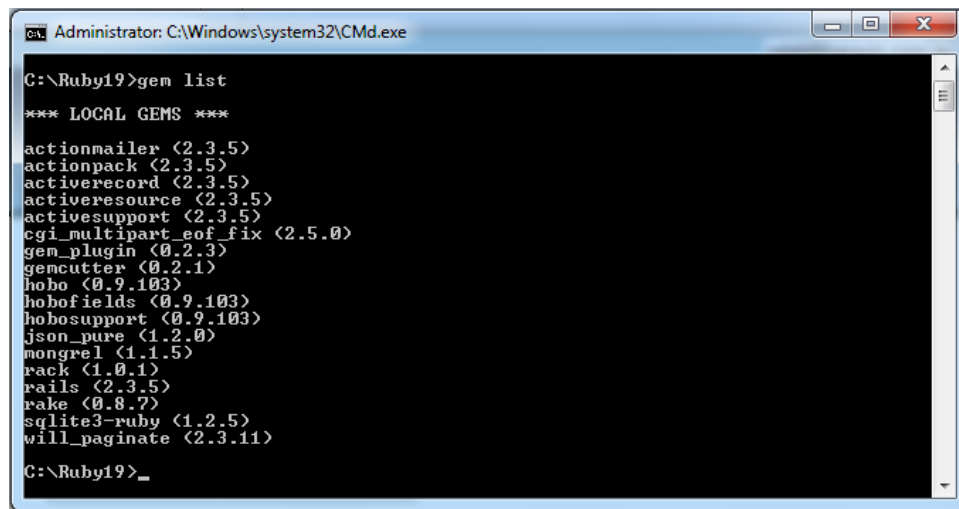
```
Administrator: C:\Windows\system32\Cmd.exe
C:\Ruby19>gem install hobo
Successfully installed actionpack-2.3.5
Successfully installed actionmailer-2.3.5
Successfully installed activerecord-2.3.5
Successfully installed rails-2.3.5
Successfully installed will_paginate-2.3.11
Successfully installed hobosupport-0.9.103
Successfully installed hobofields-0.9.103
Successfully installed hobo-0.9.103
8 gems installed
Installing ri documentation for actionpack-2.3.5...
Installing ri documentation for actionmailer-2.3.5...
Installing ri documentation for activerecord-2.3.5...
Installing ri documentation for rails-2.3.5...
Installing ri documentation for will_paginate-2.3.11...
Installing ri documentation for hobosupport-0.9.103...
Installing ri documentation for hobofields-0.9.103...
Installing ri documentation for hobo-0.9.103...
Updating class cache with 52 classes...
Installing RDoc documentation for actionpack-2.3.5...
Installing RDoc documentation for actionmailer-2.3.5...
Installing RDoc documentation for activerecord-2.3.5...
Installing RDoc documentation for rails-2.3.5...
Installing RDoc documentation for will_paginate-2.3.11...
Installing RDoc documentation for hobosupport-0.9.103...
Installing RDoc documentation for hobofields-0.9.103...
Installing RDoc documentation for hobo-0.9.103...
C:\Ruby19>
```

Figure 5: Sample console output after installing the Hobo gem

Note that the dependent gems for Rails are automatically installed as well.

3. Check your installation by using the “gem list” command to show all Ruby gems that have been installed:

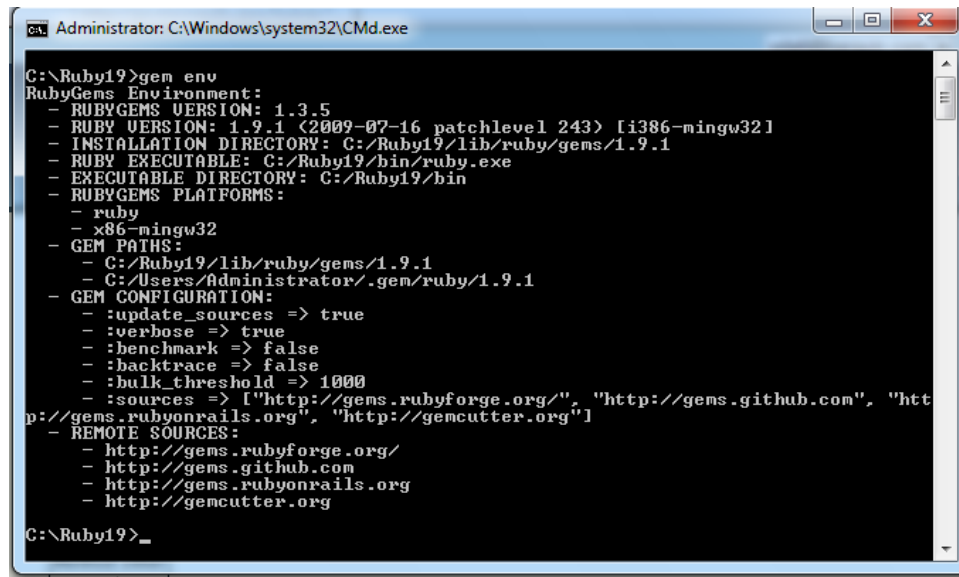
```
C:\ruby> gem list
```



```
Administrator: C:\Windows\system32\Cmd.exe
C:\Ruby19>gem list
*** LOCAL GEMS ***
actionmailer (2.3.5)
actionpack (2.3.5)
activerecord (2.3.5)
activerecord (2.3.5)
activesupport (2.3.5)
cgi_multipart_eof_fix (2.5.0)
gem_plugin (0.2.3)
gemcutter (0.2.1)
hobo (0.9.103)
hobofields (0.9.103)
hobosupport (0.9.103)
json_pure (1.2.0)
mongrel (1.1.5)
rack (1.0.1)
rails (2.3.5)
rake (0.8.7)
sqlite3-ruby (1.2.5)
will_paginate (2.3.11)
C:\Ruby19>
```

4. Finally, look at your complete installation environment with the “gem env” command:

```
C:\ruby> gem env
```



```
C:\Ruby19>gem env
RubyGems Environment:
- RUBYGEMS VERSION: 1.3.5
- RUBY VERSION: 1.9.1 (2009-07-16 patchlevel 243) [i386-mingw32]
- INSTALLATION DIRECTORY: C:/Ruby19/lib/ruby/gems/1.9.1
- RUBY EXECUTABLE: C:/Ruby19/bin/ruby.exe
- EXECUTABLE DIRECTORY: C:/Ruby19/bin
- RUBYGEMS PLATFORMS:
  - ruby
  - x86-mingw32
- GEM PATHS:
  - C:/Ruby19/lib/ruby/gems/1.9.1
  - C:/Users/Administrator/.gem/ruby/1.9.1
- GEM CONFIGURATION:
  - :update_sources => true
  - :verbose => true
  - :benchmark => false
  - :backtrace => false
  - :bulk_threshold => 1000
  - :sources => ["http://gems.rubyforge.org/", "http://gems.github.com", "http://gems.rubyonrails.org", "http://gemcutter.org"]
- REMOTE SOURCES:
  - http://gems.rubyforge.org/
  - http://gems.github.com
  - http://gems.rubyonrails.org
  - http://gemcutter.org

C:\Ruby19>
```

Figure 7: Sample console output from the "gem env" command

Note: If you find the need to start completely fresh, simply delete the folder where ruby resides, along with all the subfolders, and remove the path to /ruby/bin in your Windows environment.

For the latest instructions and further resources, please check <http://hobocentral.net>

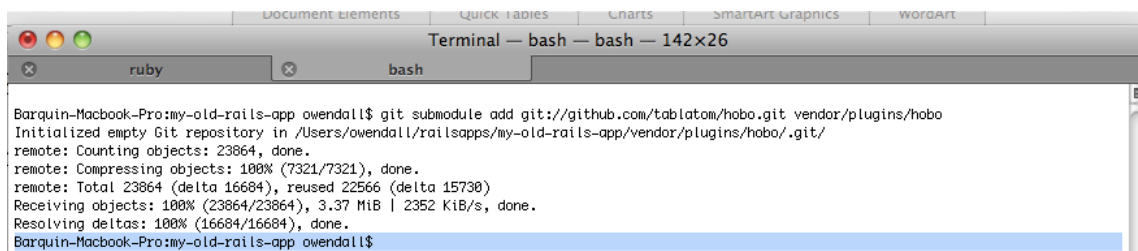
Installing Hobo as a Plugin to an Existing Rails App

This section is for experienced rails developers with an existing Rails app and experience with the Git source code control system.

1. Run the command to Install hobo as a plugin into your existing Rails application:

Change directory into your app's root folder, then:

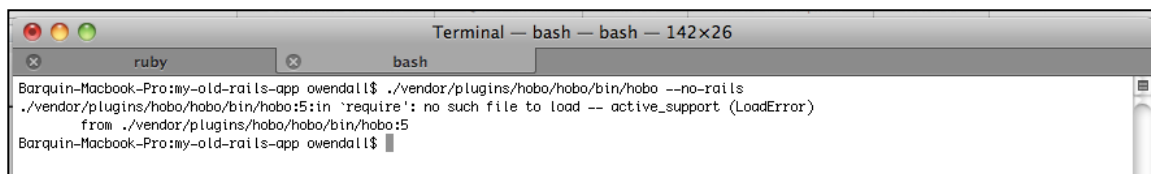
```
$ git submodule add git://github.com/tablatom/hobo.git  
vendor/plugins/hobo
```



2. Run the “hobo” command

To run the hobo command when it's installed as a plugin instead of a gem:

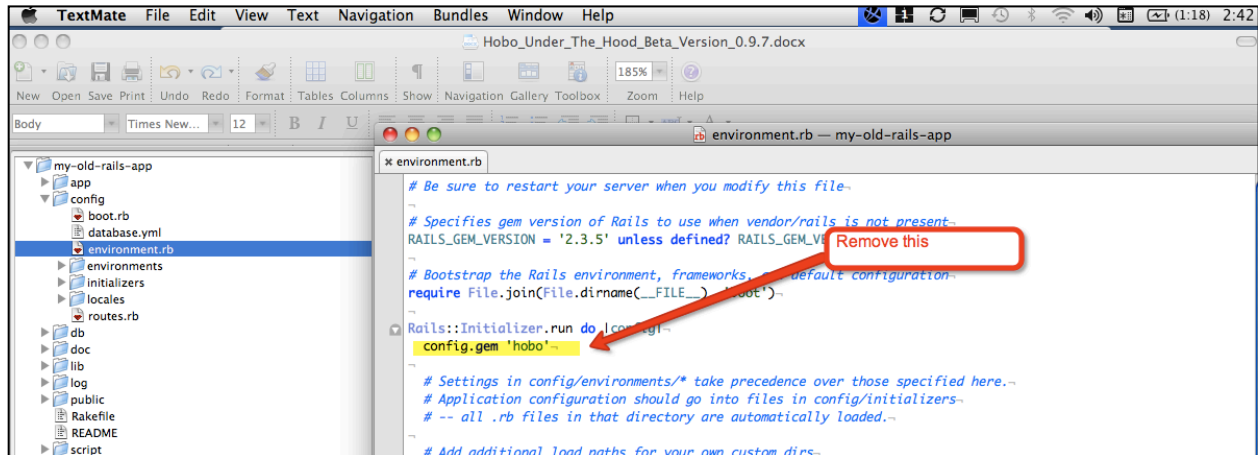
```
$ ./vendor/plugins/hobo/hobo/bin/hobo --no-rails
```



(This command will fail if you don't have the hobo gem installed. Don't worry about this and follow the next step).

3. Comment out the “config.gem” statement

The hobo command generates a setup designed to be used with the hobo gem. If you wish to use the plugin, comment out `config.gem 'hobo'` in `config/environment.rb`.



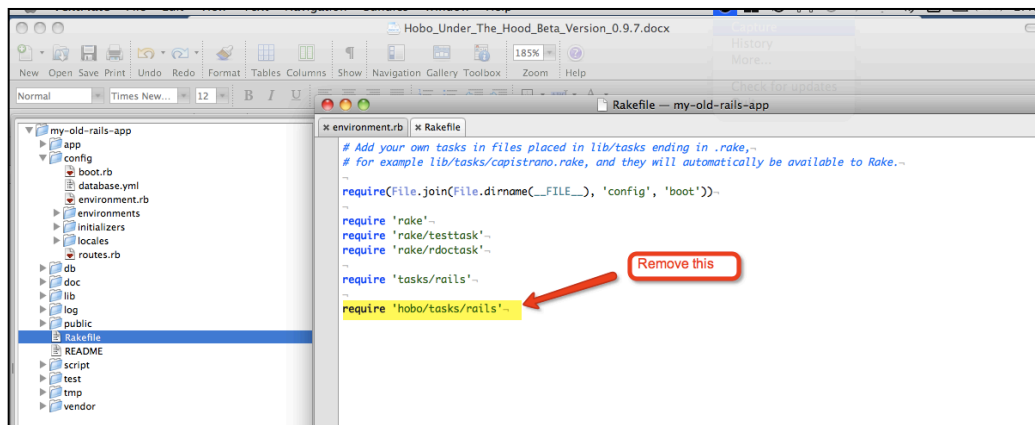
4. Run the “hobo” command again

Now that we’ve fixed `config/environment.rb`, we can run `hobo` again, and it will run to completion. This is only necessary if it failed previously.

```
$ ./vendor/plugins/hobo/hobo/bin/hobo --no-rails
```

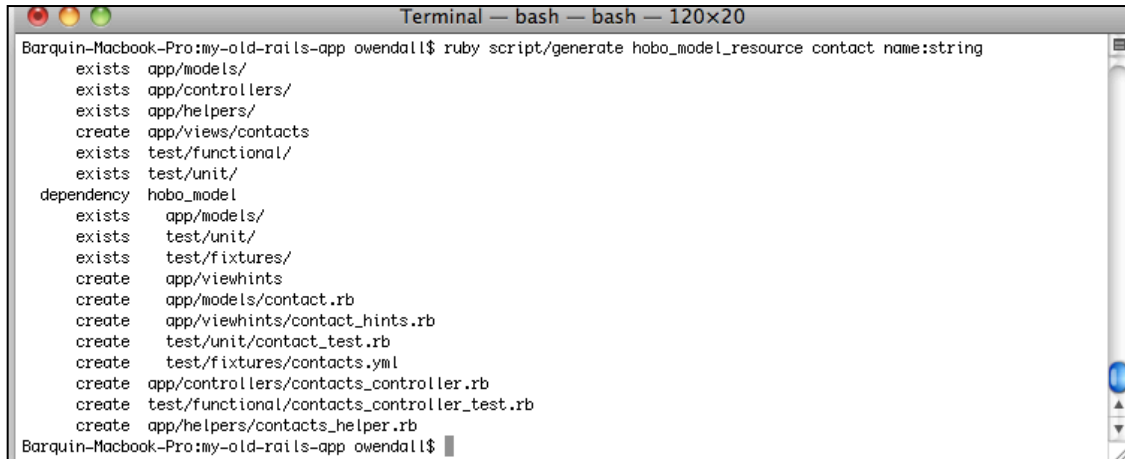
5. Fix the Rakefile

The hobo command sets up the Rakefile to use the gem. Let’s fix it so that it works with the plugin by removing the `require 'hobo/tasks/rails'` line:



6. Run Hobo commands

Now you can run any hobo command within your modified application:

A screenshot of a macOS Terminal window titled "Terminal — bash — bash — 120x20". The window shows the command `ruby script/generate hobo_model_resource contact name:string` being executed. The output lists the files and directories created or existing, such as `app/models/`, `app/controllers/`, `app/helpers/`, `app/views/contacts`, `test/functional/`, `test/unit/`, `test/fixtures/`, `app/viewhints`, `app/models/contact.rb`, `app/viewhints/contact_hints.rb`, `test/unit/contact_test.rb`, `test/fixtures/contacts.yml`, `app/controllers/contacts_controller.rb`, `test/functional/contacts_controller_test.rb`, and `app/helpers/contacts_helper.rb`. The terminal ends with the prompt `Barquin-Macbook-Pro:my-old-rails-app owendall$`.

```
Barquin-Macbook-Pro:my-old-rails-app owendall$ ruby script/generate hobo_model_resource contact name:string
exists  app/models/
exists  app/controllers/
exists  app/helpers/
create  app/views/contacts
exists  test/functional/
exists  test/unit/
dependency hobo_model
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/viewhints
create  app/models/contact.rb
create  app/viewhints/contact_hints.rb
create  test/unit/contact_test.rb
create  test/fixtures/contacts.yml
create  app/controllers/contacts_controller.rb
create  test/functional/contacts_controller_test.rb
create  app/helpers/contacts_helper.rb
Barquin-Macbook-Pro:my-old-rails-app owendall$
```

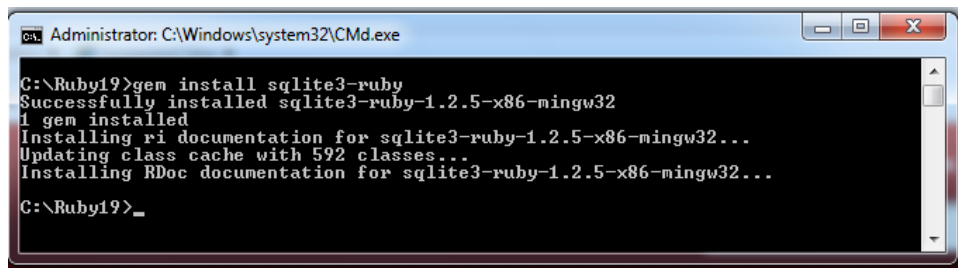

Using SQLite with Hobo

SQLite is a lightweight, zero configuration database engine ideal for prototyping. (This is the default engine used during creation of a new Rails or Hobo application.)

We used SQLite3 at the time of this writing.

1. Install the **SQLite3-ruby** gem. Open up a Windows command prompt and run the following:

```
C:\ruby19> gem install sqlite3-ruby
```



```
Administrator: C:\Windows\system32\Cmd.exe
C:\Ruby19>gem install sqlite3-ruby
Successfully installed sqlite3-ruby-1.2.5-x86-mingw32
1 gem installed
Installing ri documentation for sqlite3-ruby-1.2.5-x86-mingw32...
Updating class cache with 592 classes...
Installing RDoc documentation for sqlite3-ruby-1.2.5-x86-mingw32...
C:\Ruby19>_
```

Figure 8: Sample console output from installing the sqlite3-ruby gem

Microsoft Windows PCs also require the `sqlite3.dll`. Download this from <http://www.sqlite.org/download.html> place it the `c:\ruby19\bin` folder.

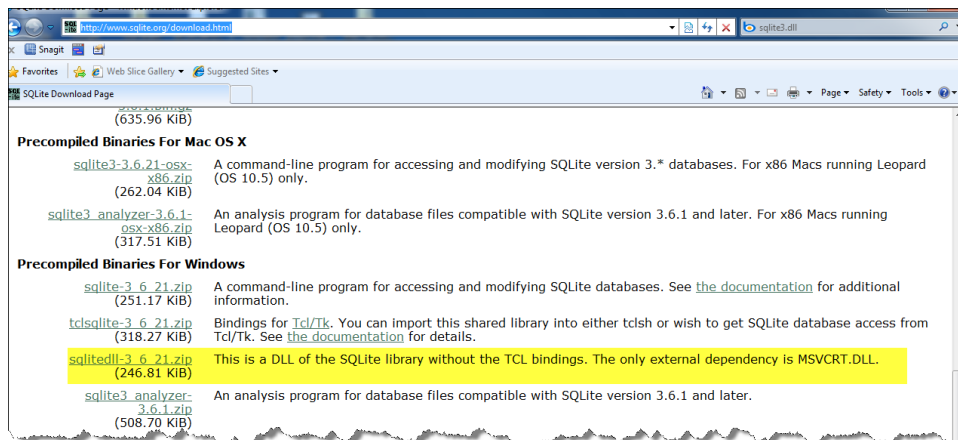


Figure 9: SQLite3 download website

Unzip the downloaded file and place the `sqlite3.dll` and `sqlite.def` files in the `c:\ruby\bin` folder.

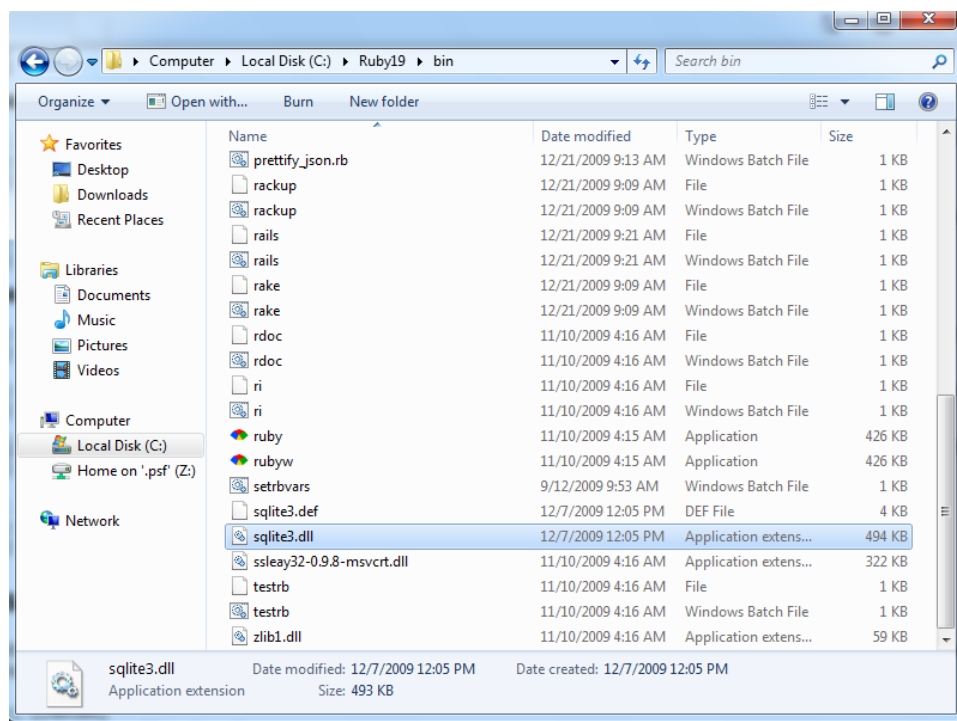


Figure 10: Target location for the SQLite3 DLL

Step 1: Download and install MySQL.

For Mac OS X user, please see the following URL:

<http://dev.mysql.com/doc/mysql-macosx-excerpt/5.0/en/mac-os-x-installation.html>

For Linux users:

<http://dev.mysql.com/doc/refman/5.0/en/linux-rpm.html>

For Windows users the following detailed instructions are provided:

Go to the appropriate URL at dev.mysql.com and download the Windows MSI installer:

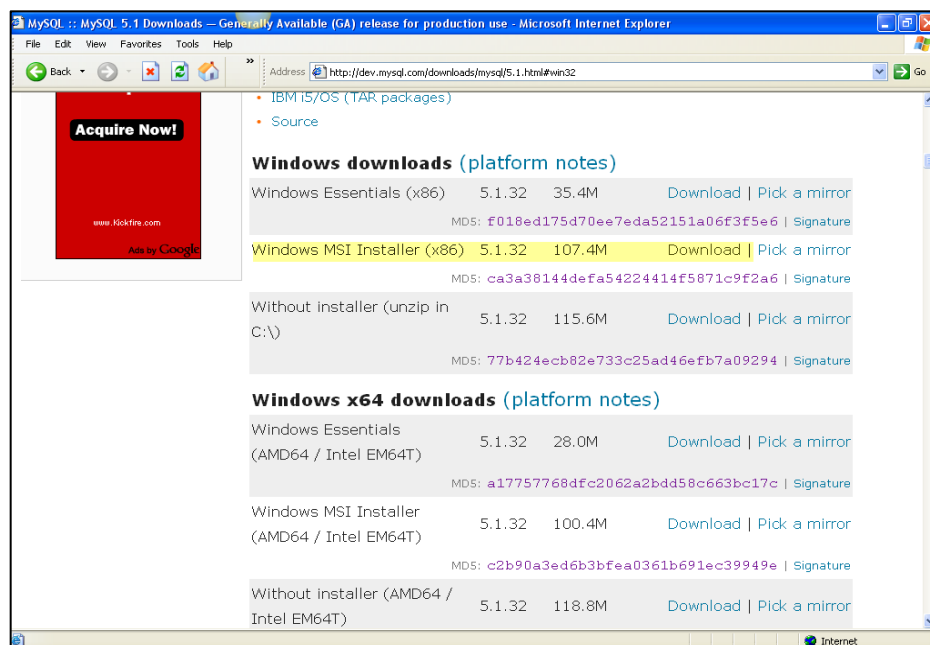


Figure 11: Download site for MySQL

Double-click on the MySQL MSI installation file:

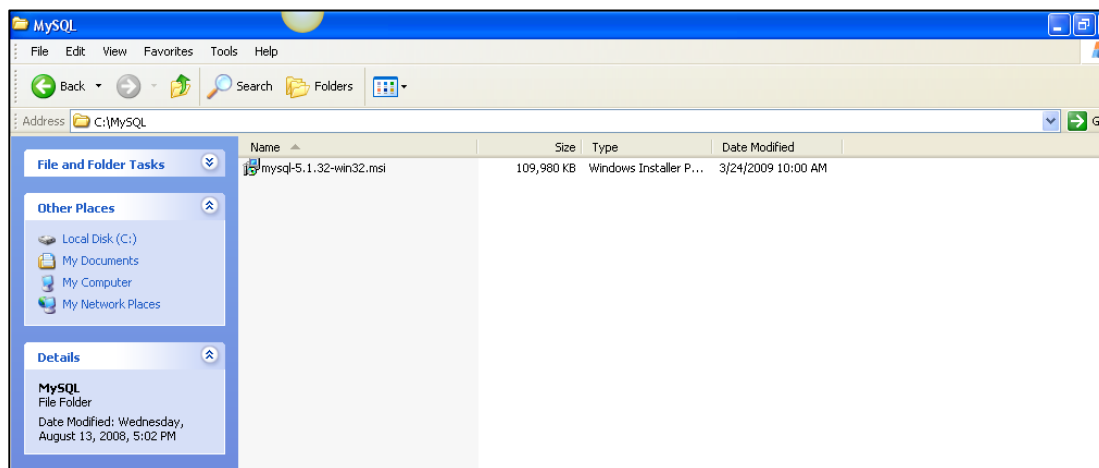


Figure 12: Using the .msi file to install MySQL on Windows

Choose the “Typical” option when prompted:

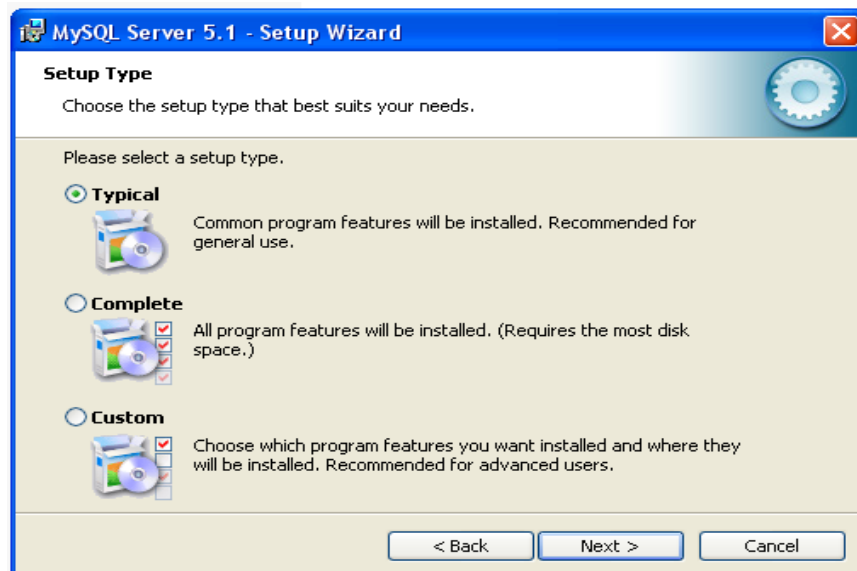


Figure 13: Choose the installation type

The MySQL Setup Wizard will take a few minutes to install all components:

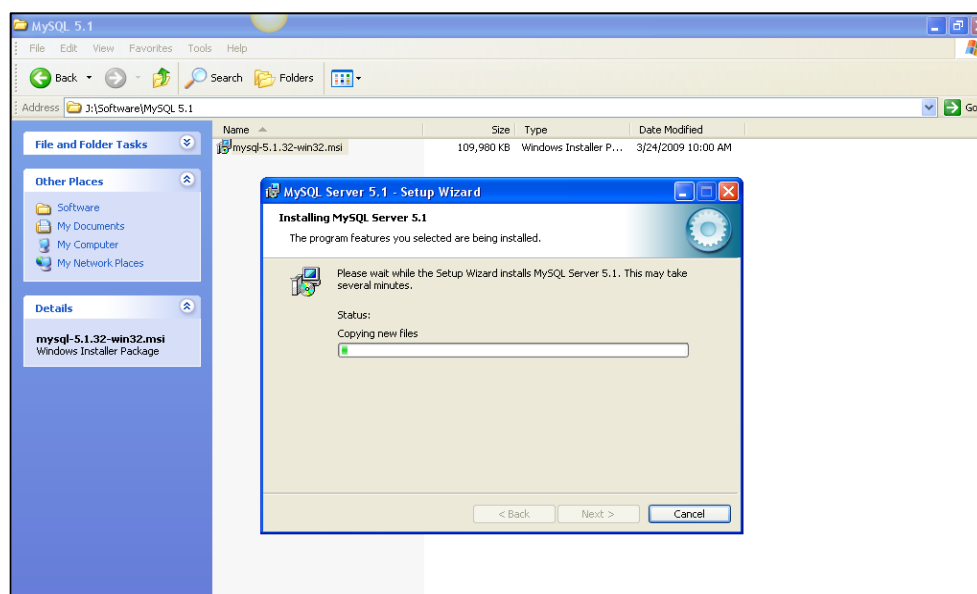


Figure 14: MySQL Server Setup Wizard

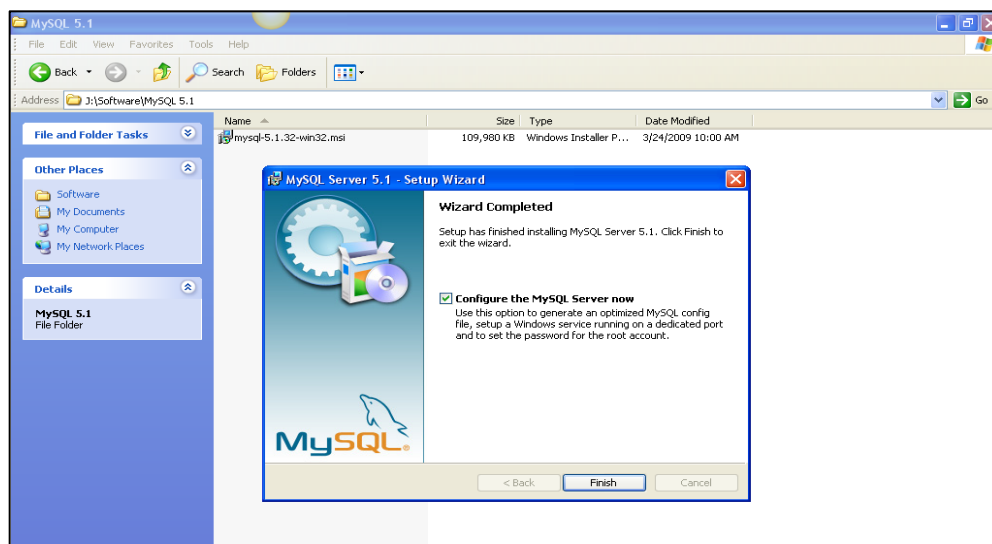


Figure 15: Configure MySQL Server

The next step is to configure the database instance:

We recommend choosing the “Standard Configuration” option.

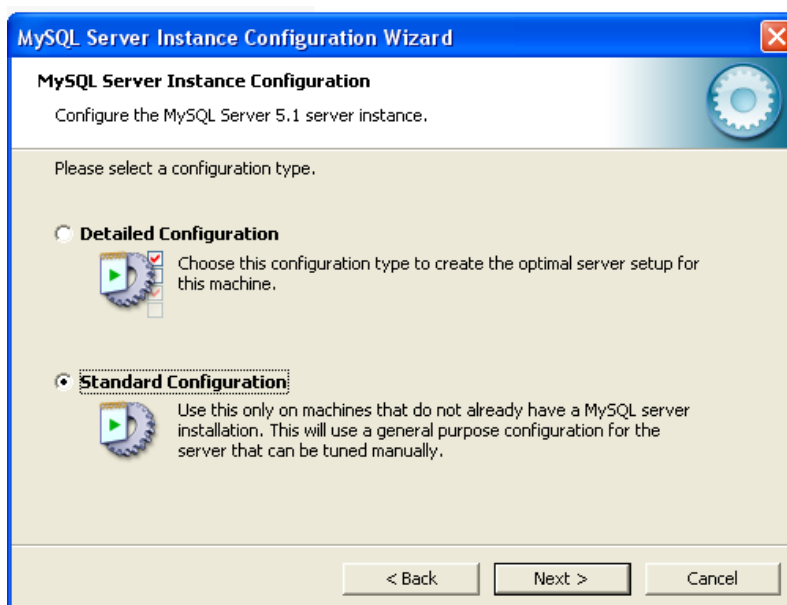


Figure 16: Choose Standard Configuration

Select both “Install As Windows Service” and “Include Bin Directory in Windows PATH”:

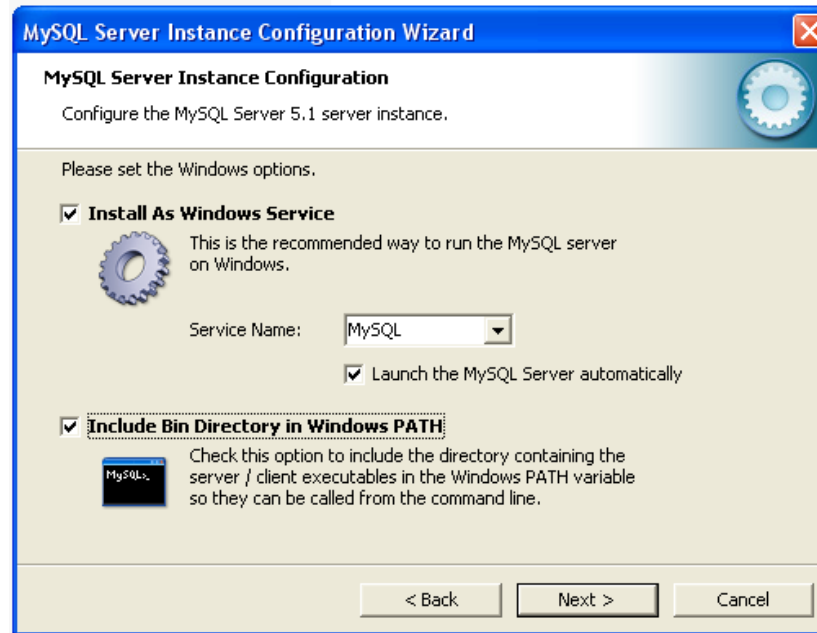


Figure 17: Install as Windows Service

A progress window will appear next. Press “Finish” to complete the installation.

Now you can launch MySQL from the command prompt as follows:

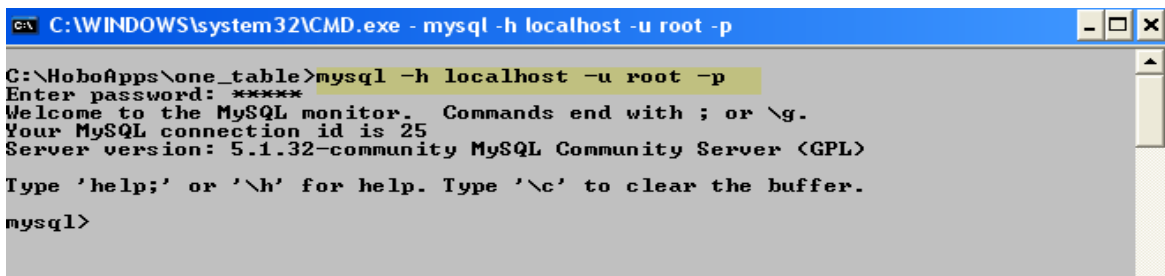
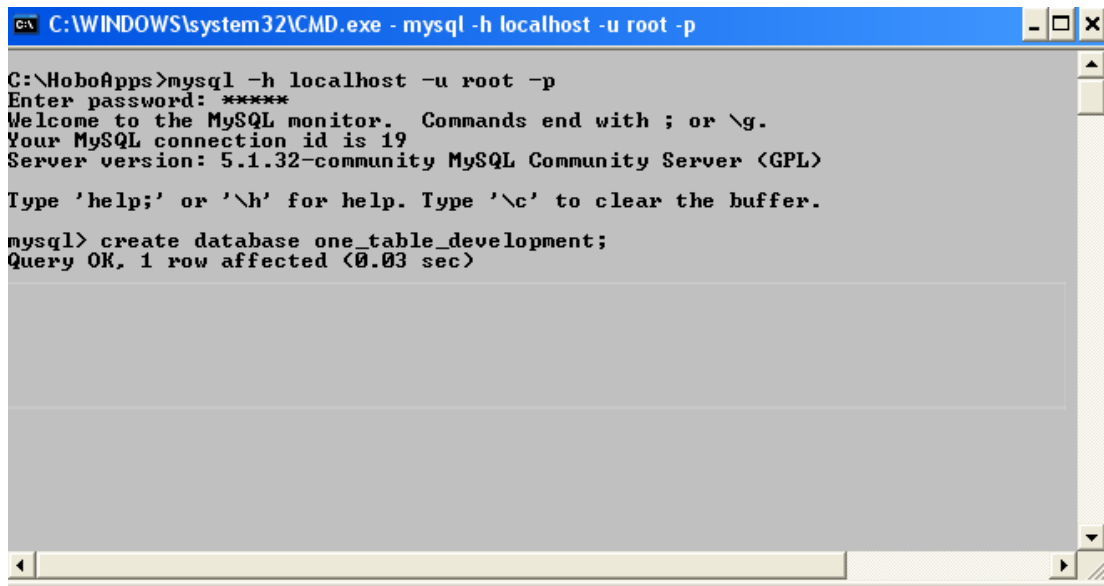


Figure 18: Launch MySQL from the command prompt

MySQL will prompt you for the password you entered during installation.

Now create the database you will need for the “one_table” tutorial:



```
C:\WINDOWS\system32\CMD.exe - mysql -h localhost -u root -p

C:\HoboApps>mysql -h localhost -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 19
Server version: 5.1.32-community MySQL Community Server (GPL)

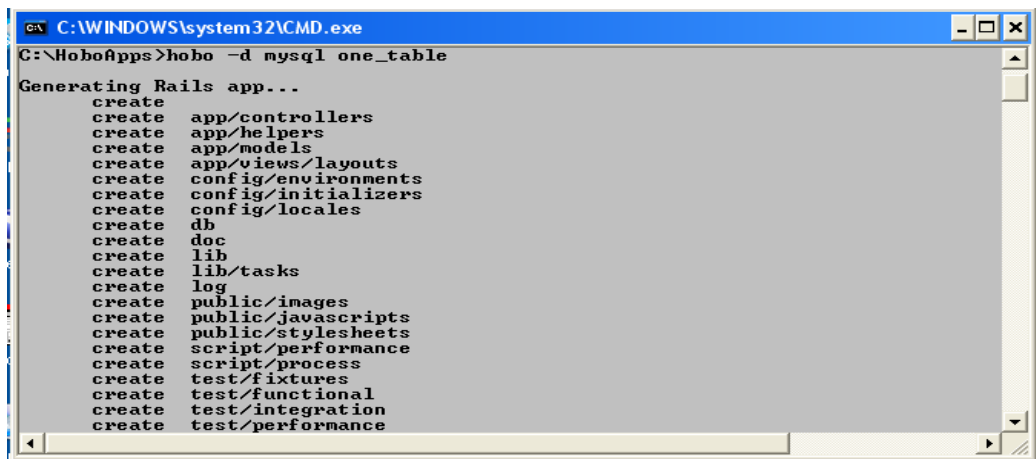
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database one_table_development;
Query OK, 1 row affected (0.03 sec)
```

Figure 19: Create the database from the command line

Now you can create the Hobo app with the option to use MySQL instead of the default SQLite database:

```
c:\tutorials> hobo -d mysql one_table
```



```
C:\WINDOWS\system32\CMD.exe

C:\HoboApps>hobo -d mysql one_table
Generating Rails app...
create
create  app/controllers
create  app/helpers
create  app/models
create  app/views/layouts
create  config/environments
create  config/initializers
create  config/locales
create  db
create  doc
create  lib
create  lib/tasks
create  log
create  public/images
create  public/javascripts
create  public/stylesheets
create  script/performance
create  script/process
create  test/fixtures
create  test/functional
create  test/integration
create  test/performance
```

Figure 20: Console output from the Hobo command

Now edit the `database.yml` file to see what it looks like:

Notice it is pre-filled with the proper parameter structure for MySQL. You just need to fill in the blanks, particularly the database password:

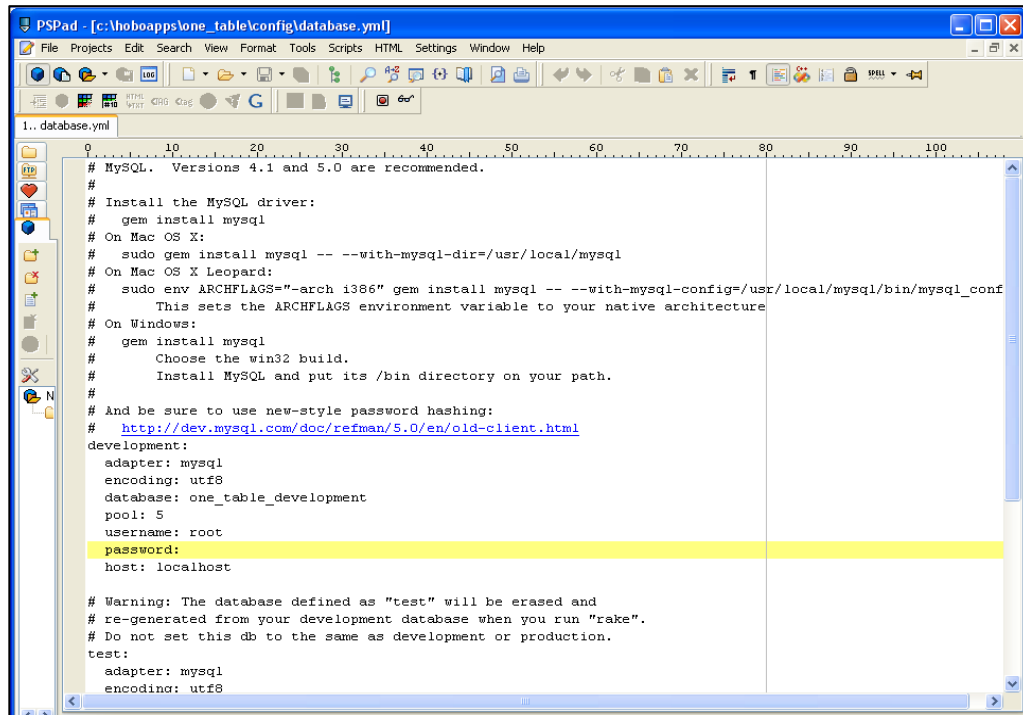


Figure 21: The MySQL format for the database.yml configuration file

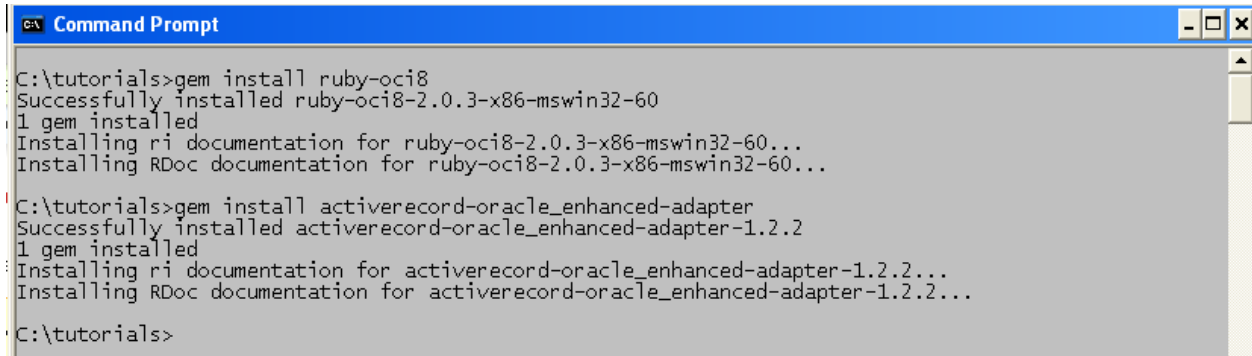
Using Oracle with Hobo

We will discuss the following two configuration options:

1. Use an existing Oracle database
2. Download and install a fresh Oracle database

For either of these options you will first need to install the following two ruby gems:

```
C:\ruby> gem install ruby-oci8 -v 1.0.4
C:\ruby> gem install activerecord-oracle-adapter
```

```
C:\tutorials>gem install ruby-oci8
Successfully installed ruby-oci8-2.0.3-x86-mswin32-60
1 gem installed
Installing ri documentation for ruby-oci8-2.0.3-x86-mswin32-60...
Installing RDoc documentation for ruby-oci8-2.0.3-x86-mswin32-60...

C:\tutorials>gem install activerecord-oracle_enhanced-adapter
Successfully installed activerecord-oracle_enhanced-adapter-1.2.2
1 gem installed
Installing ri documentation for activerecord-oracle_enhanced-adapter-1.2.2...
Installing RDoc documentation for activerecord-oracle_enhanced-adapter-1.2.2...

C:\tutorials>
```

Figure 22: Console output from installing the Ruby gem for Oracle

Option 1

This is the typical scenario in a development shop that is already using Oracle and you have the Oracle client software already configured for other tools such as SQL Plus, Toad, or SQL Developer.

You probably have different database “instances” for development, test, and production systems. If you are lucky you might even have rights to create a new database user (i.e., schema) in your development environment. In most large shops you will probably need to request that the database administrator (DBA) create one for you.

(Note: the terms “user” and “schema” really are referring to the same thing and are often used interchangeably by experienced Oracle developers. There is a long history to this that will confuse users of other database engines where users and schemas are not equivalent.)

As you learned in earlier tutorials, the `database.yml` file is the place to configure your database connections. Creating a new application using the hobo command with the “d” switch allows you to stipulate which database you will be using, and allows Hobo and Rails to build a `database.yml` template tailored to your database.

```
C:\tutorials> hobo two_table -d oracle
```

This is what the `database.yml` file looks like without modification:

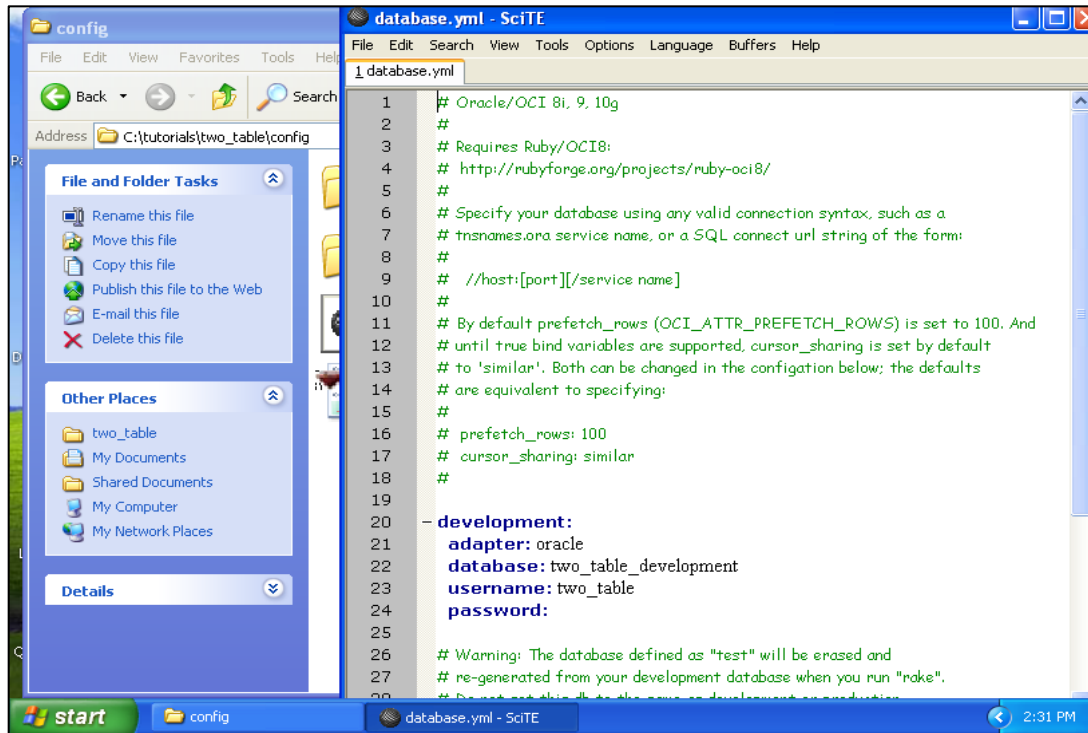


Figure 23: The generated database.yml file for Oracle

When we used SQLite as the default database, Hobo and Rails automatically created a database called “two_table_development”. When you use an existing Oracle database, you will need to enter that database name instead of “two_table_development” and use “two_table_development” as the user name the username. Therefore the entries in the database.yml file will look more like the following:

```
development:
  adapter: oracle
  database: our_development_server_name
  username: two_table_development
  password: hobo
```

Once you update the database.yml file and save it you can then run your hobo migration and the complete tutorials as you before. This time they will run using Oracle as the back end. That is all there is to it.

Option 2

In this part of the tutorial we will walk you through the steps of downloading, installing, and configuring Oracle 10g XE (Express Edition), which is a fully functional version of Oracle with no licensing requirements. It comes with administration tools, a web front end. Register for a free membership in the Oracle Technology Network (OTN) and then go to the following URL to download Oracle Database 10g Release 2 Express Edition for Windows:

<http://www.oracle.com/technology/software/products/database/xs/htdocs/102xe/winsoft.htm>

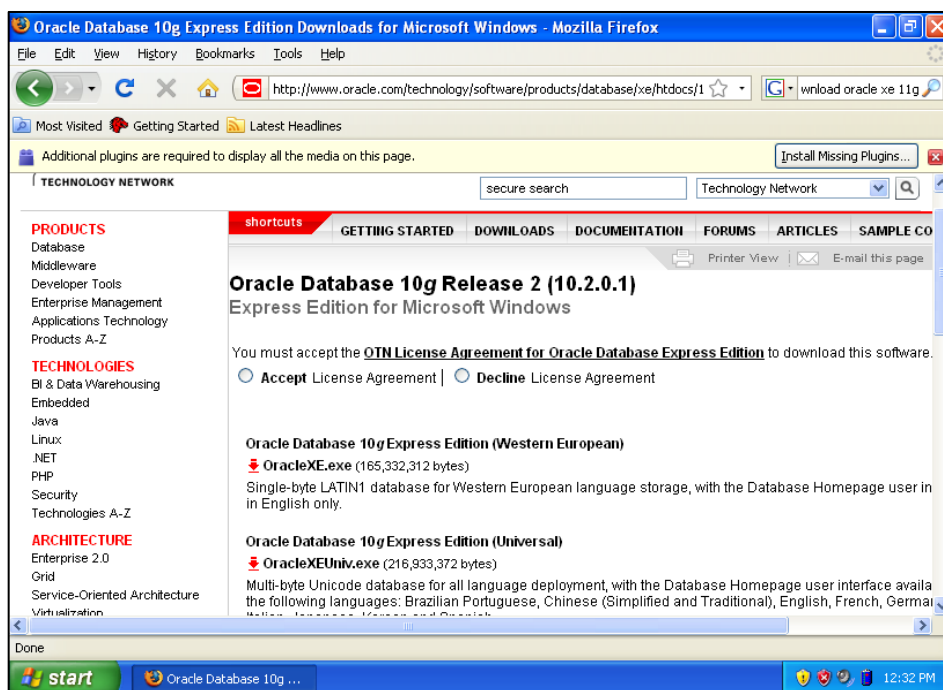


Figure 24: Oracle database install download site

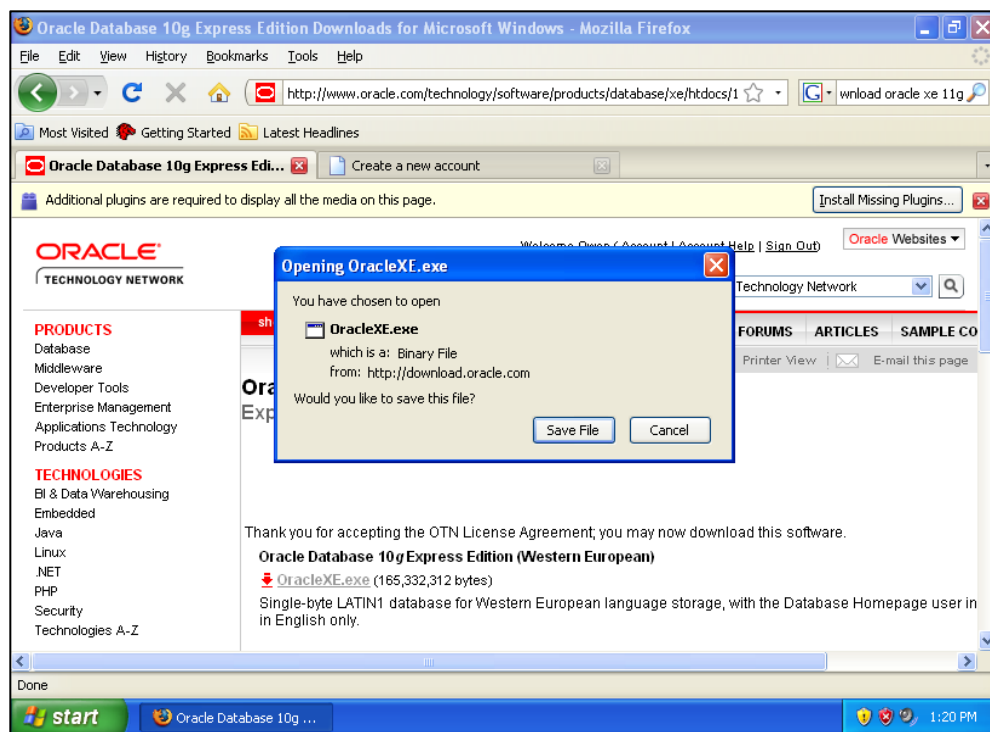


Figure 25: Running the Oracle XE installation

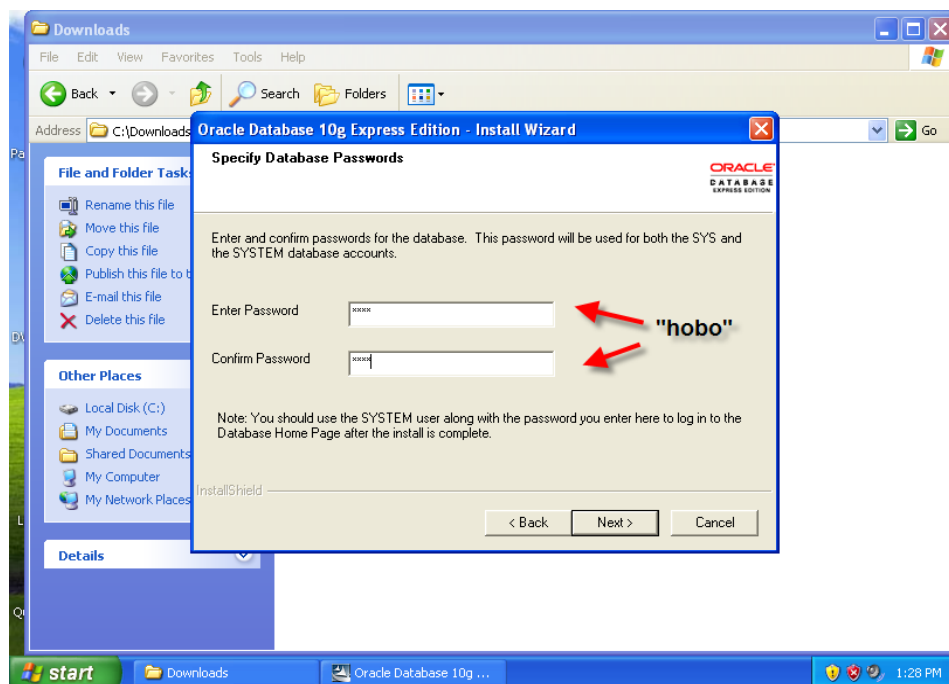


Figure 26: Specifying the database passwords

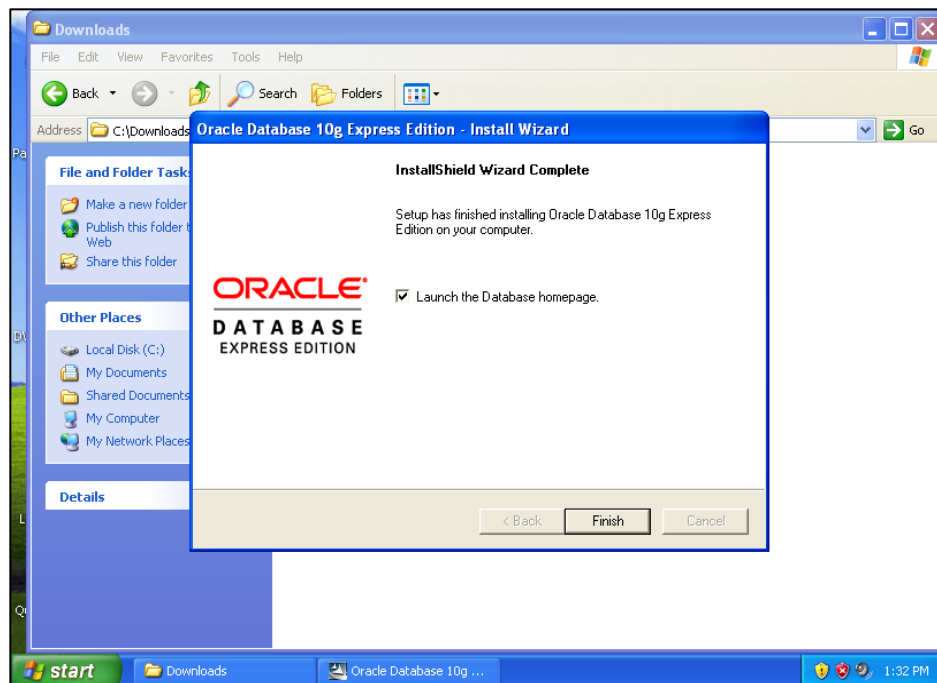


Figure 27: Launch the Database home page

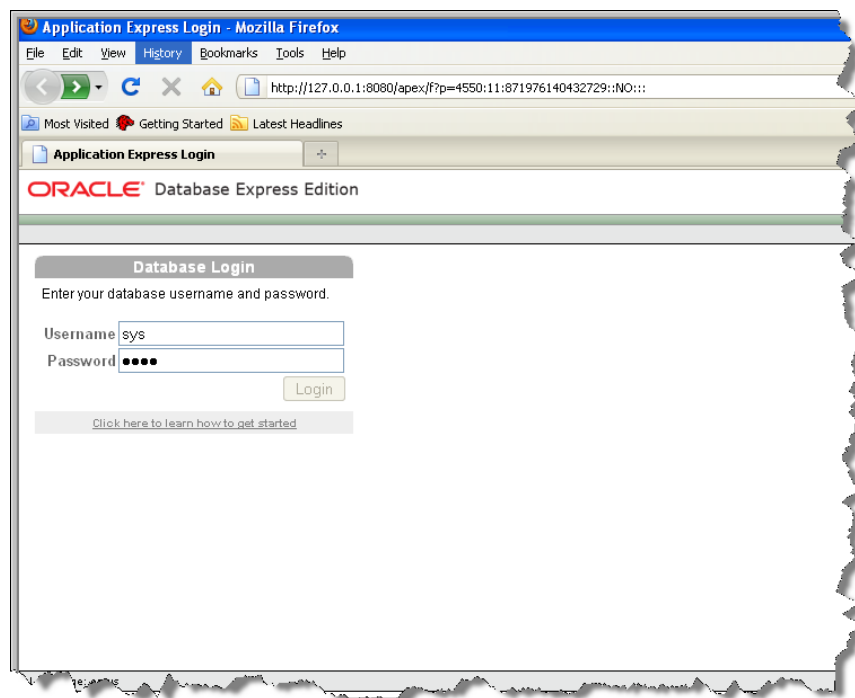


Figure 28: Log is as SYS to configure your database

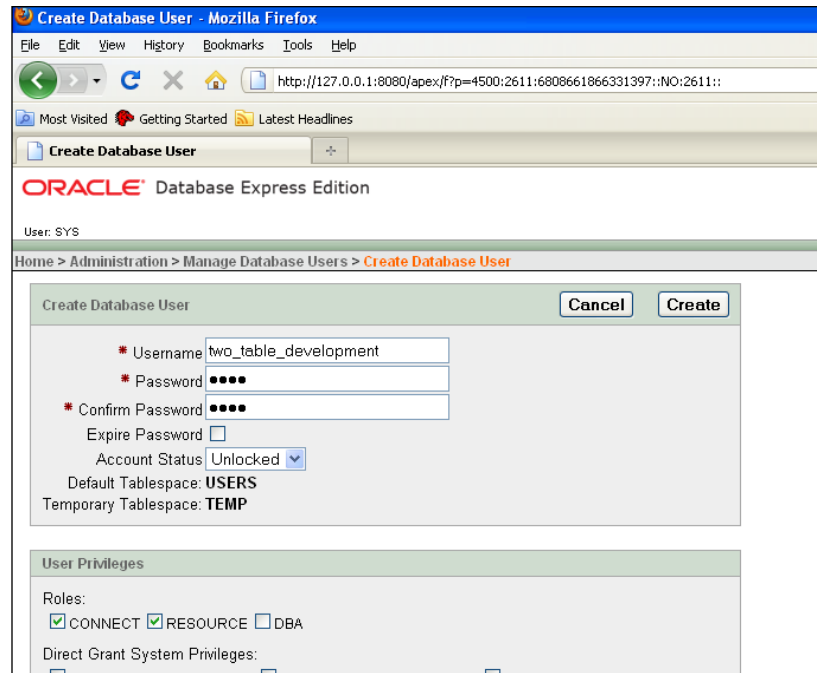


Figure 29: Creating a schema/user to use with Hobo

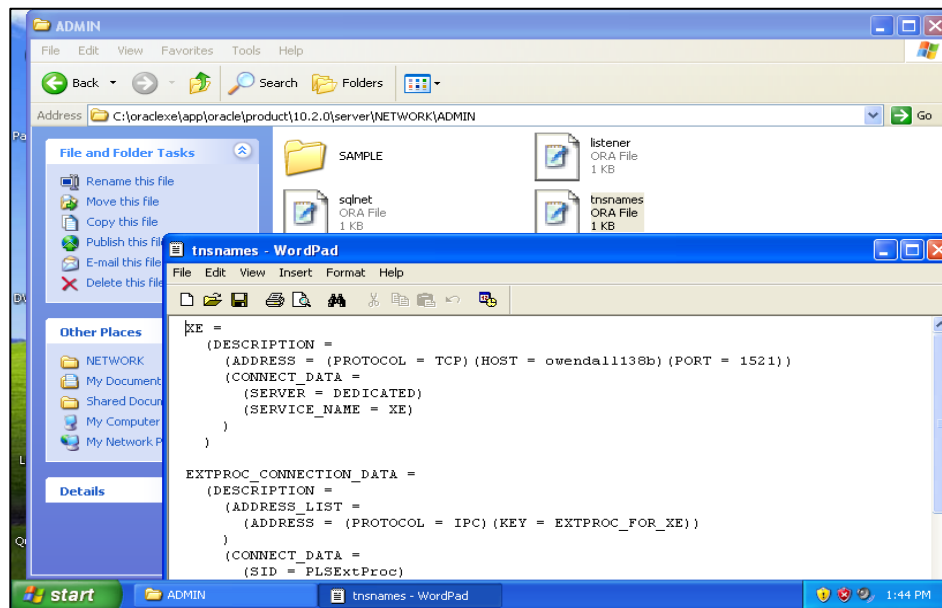


Figure 30: The tnsnames.ora file created during installation

Note that you will be using the “XE” instance unless you change the name.

```
C:\tutorials> hobo one_table -d oracle
```

Note: The following instructions and screen shots will make sense AFTER you work through the introductory tutorials.

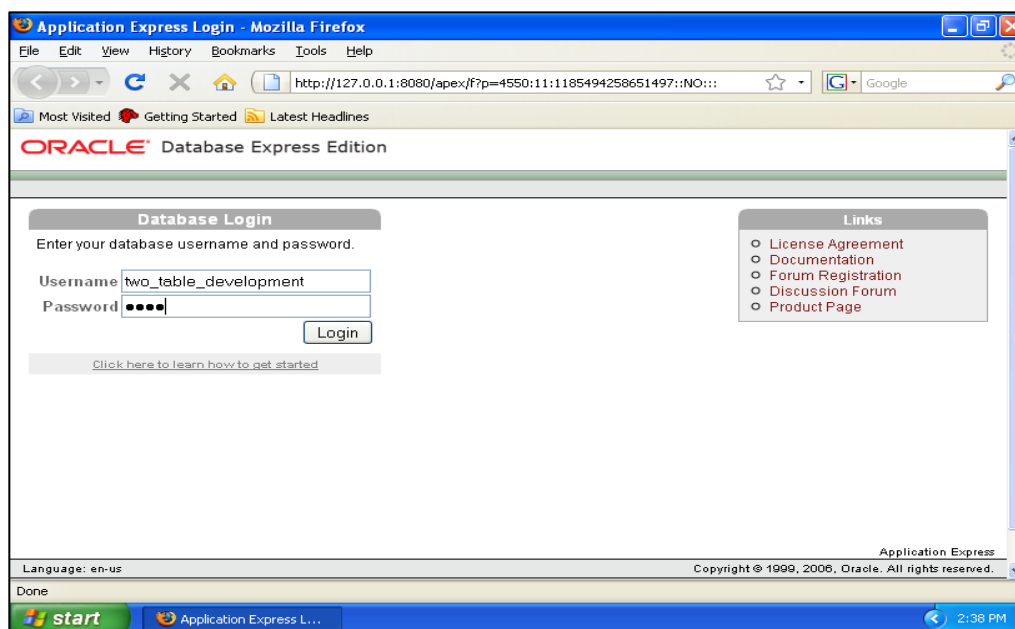


Figure 31: Log into Oracle to view the created table

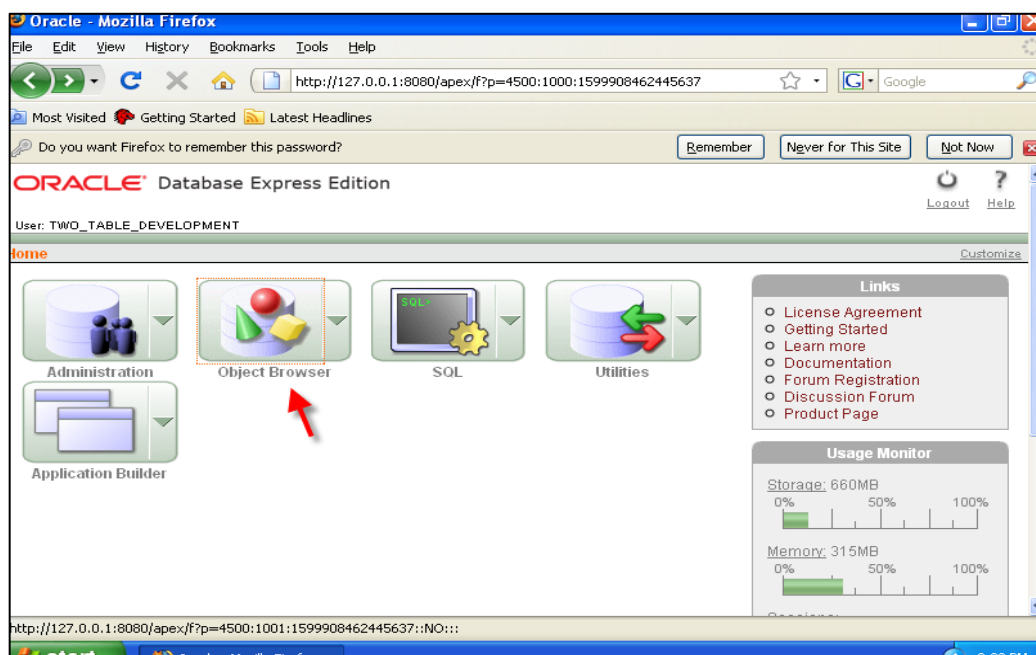


Figure 32: Access the Oracle Object Browser

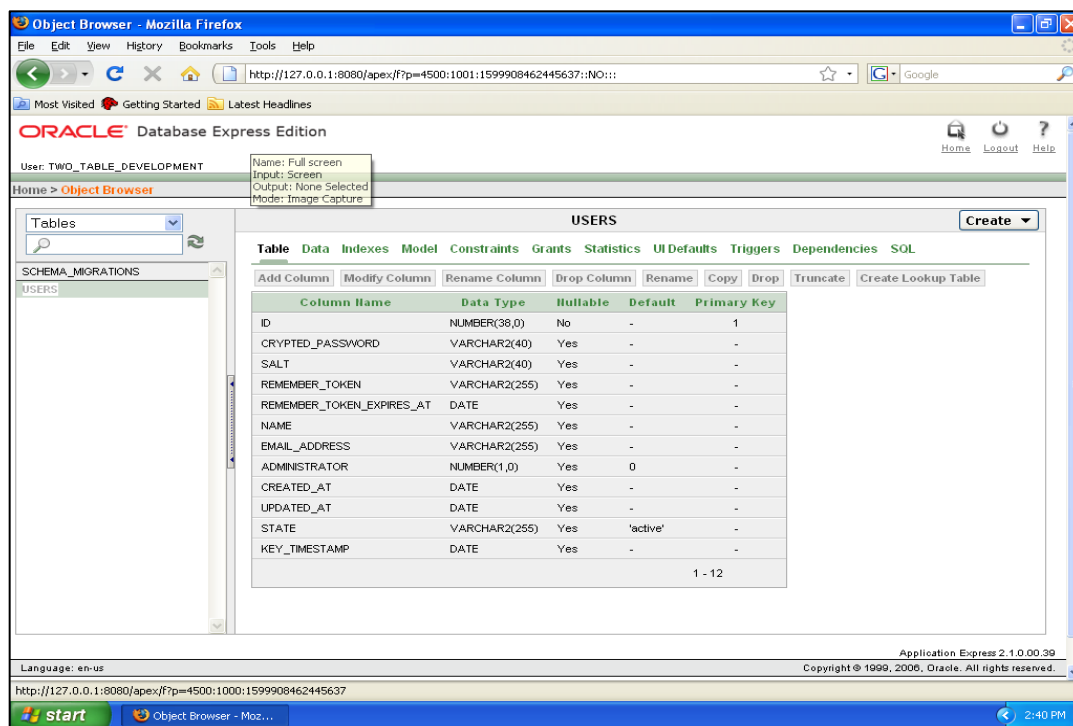


Figure 33: Review the User table from within Oracle

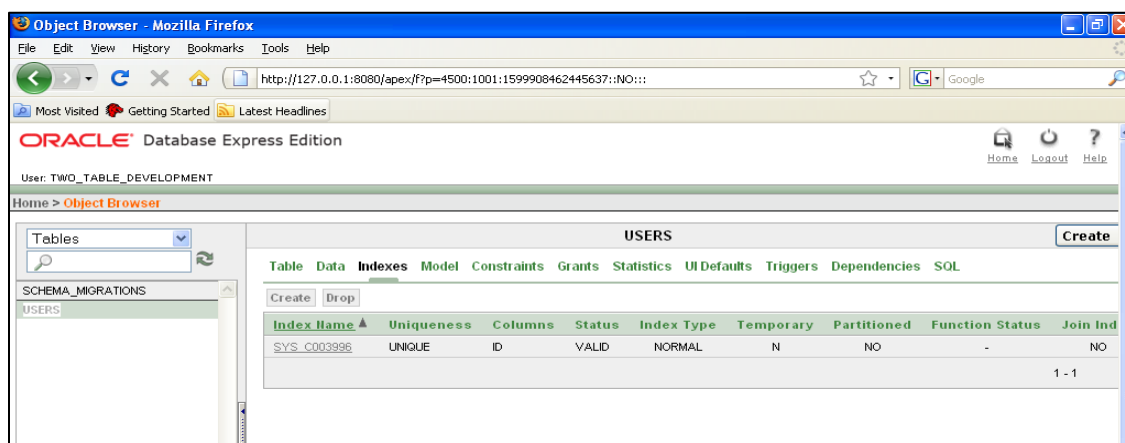


Figure 34: Review the Indexes view for Users

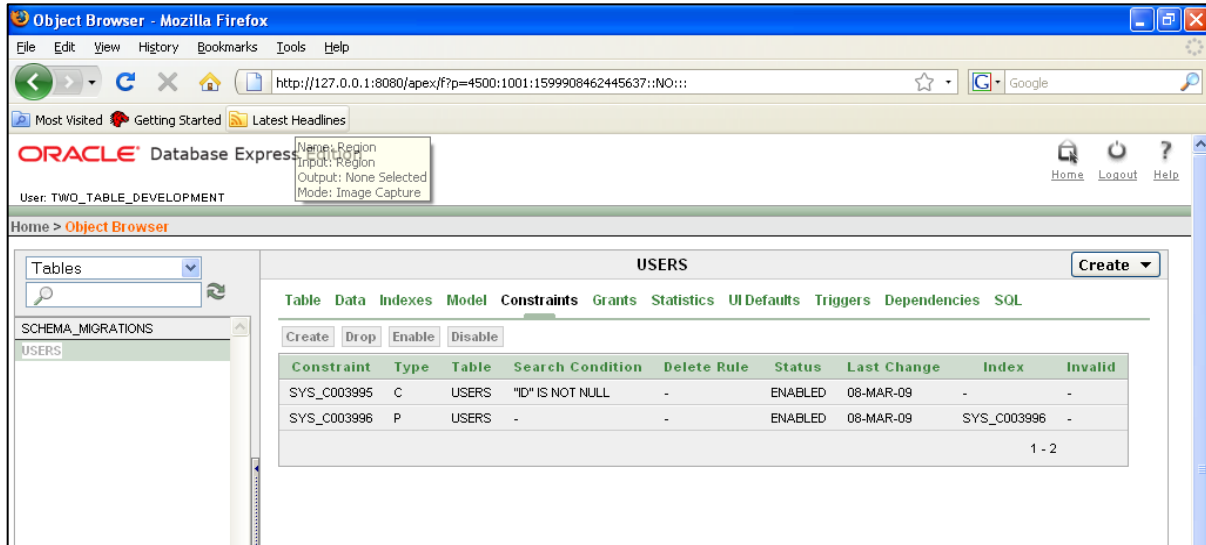


Figure 35: Review the Constraints view for User

Installation Summary

What you have now is:

- The **Ruby** language interpreter, which in this case is a Windows executable. This engine is called MRI for “Matz’s Ruby Interpreter”. http://en.wikipedia.org/wiki/Ruby_MRI. There are a variety of other interpreters and implementations available, including **JRuby** <http://jruby.org/> and Enterprise Ruby (<http://www.rubyenterpiseedition.com/>), which the authors have used successfully with Hobo. The upcoming MagLev (<http://maglev.gemstone.com/status/index.html>) implementation looks very promising for large-scale applications.
- The **Ruby on Rails** (RoR) Model-View-Controller (MVC) framework which is written using Ruby.
- The **Hobo** framework which enhances, and in some cases replaces, RoR functionality, particularly on the View and Controller portions of the (MVC) web development framework. Hobo is written in Ruby. One of Hobo’s secret weapons is the powerful and succinct DRYML (**Don’t Repeat Yourself Markup Language**).
- The **SQLite3** database and related Ruby gem (**sqlite3-ruby**) that makes prototyping quick and painless. SQLite3 is a robust and widely used database engine for embedded systems and is the repository used by the Firefox browser.
- The Webrick **HTTP/web** server written in Ruby. This is a basic server for desktop development work. Another popular one is Mongrel (Ruby 1.8 only). For production implementation they’re a variety of options, including the popular Phusion Passenger (aka mod_rails), which can be used in conjunction with the

Apache HTTP server. <http://www.modrails.com/>. With JRuby you can run on JBoss, Glassfish, etc.

- A variety of add-on gems (ruby modules or “libraries”) that each framework has included as “dependencies”. For example, **will_paginate** is used by Hobo for pagination of lists on web pages.

Now you are ready to dive in...

Chapter 3 - Hobo Fundamentals

Some of what we will cover in this chapter can be found in the tutorials earlier in the book. However, in this chapter we will drill more deeply into the infrastructure and philosophy of Hobo.

The Hobo developers have taken the DRY (Don't Repeat Yourself) paradigm to a new level by identifying repetitive architectural patterns in data-driven web sites and particularly within Rails applications.

- Rapid implementation of dynamic AJAX interfaces in your application with no extra programming. Switchable themes. Customize and tweak your application structure and layout to meet any design goals.
- Powerful mark-up language, DRYML, combines rapid development with ultimate design flexibility.

The DRY paradigm is all about finding the right *level of abstraction* for the building blocks of an application in order to reduce cookie-cutter repetitive programming.

Rails starts with a Model-View-Controller (MVC) architecture built with Ruby code, using the metaprogramming power that Ruby provides.

Hobo takes this paradigm further and it does it in two directions. It provides rapid prototyping with modules that provide an integrated user login and permissions system, automated page generation, automated routing, built-in style sheets, and an automated database migration and synchronization system. Hobo also provides a powerful markup language called DRYML that provides an almost limitless method for building custom tags at ever-higher levels of abstraction.

Sometimes these patterns are at a very high level such as the need for a user login capability and sometimes they are at a lower level such the requirement to grab a set of records for display.

The Hobo framework philosophy is that many of the features of a data-driven site should be able to be declared and need no other coding, at least for the first set of iterations.

Let's take a database query as an example. The developers of Rails realized that many queries had similar structures and therefore there should be no need to code complex SQL queries. Rails implements *find* methods to deal with this challenge. But--in the view template—you still need to write the code to loop through the records when you need to display them.

The Hobo view is that this is a ubiquitous repetitive pattern that should be addressed. So Hobo lets you just declare that you want to display a collection of records in a single command.

As we have mentioned many times before, Hobo provides a new language called DRYML (Don't Repeat Yourself Markup Language) to develop menus, views, forms, and page navigation. The components of DRYML, as you would expect, are tags. Hobo comes with a library of predefined DRYML tags called the Rapid Tag Library. This library is used to render the default menus, pages, and forms you have used in the tutorials.

Levels of Abstraction

As we discussed above, finding the right level of abstraction in implementing coding constructs is the key to programming productivity and application maintainability. But anyone who has ever coded knows that programming is a messy business. Sometimes it is just easier to code at a low level of abstraction. This is the dominant way of developing applications today. It is simpler not to create reusable components or snippets because something always seems to need changing. You think you will waste more time fixing your components than just starting over.

The approach that Rails takes, and Hobo even more so, is to have code that lets multiple levels of abstractions coexist in the code. This is potentially the best of both approaches.

Build higher and higher levels of abstraction in your tool set but maintain the ability to code at a detail level for development flexibility.

Wherever possible, Hobo provides additional capabilities over Rails for declaring what you want rather than forcing you to write procedural code. It is therefore important to understand what is going on procedurally behind the scenes in both Rails and Hobo so you know what to do.

In this chapter we will emphasize which component--model, view or controller--is doing what, and when it is doing it. We will also emphasize what the various Hobo constructs are doing and how within the architecture of Rails.

We are going to go through the Hobo approach at a couple of levels but first we will list them and give a brief introduction.

Now we are going to approach the major topics at a shallow level first and then circle back and go in deeper after we get a few things out of the way first.

Rails and Hobo

Hobo is a set of Rails plug-in, which means that Hobo adds additional custom code to Rails, and coexists with Rails. So, essentially a Hobo application is a Rails application with additional capabilities. However, these additional capabilities are substantial, and can be conceptualized into two categories:

1. Operational (“Run Time”) Enhancements
2. Developer Tool Enhancements

Operational Enhancements. Take a look at the data flow for a typical operating application built with a Model-View-Controller (MVC) framework:

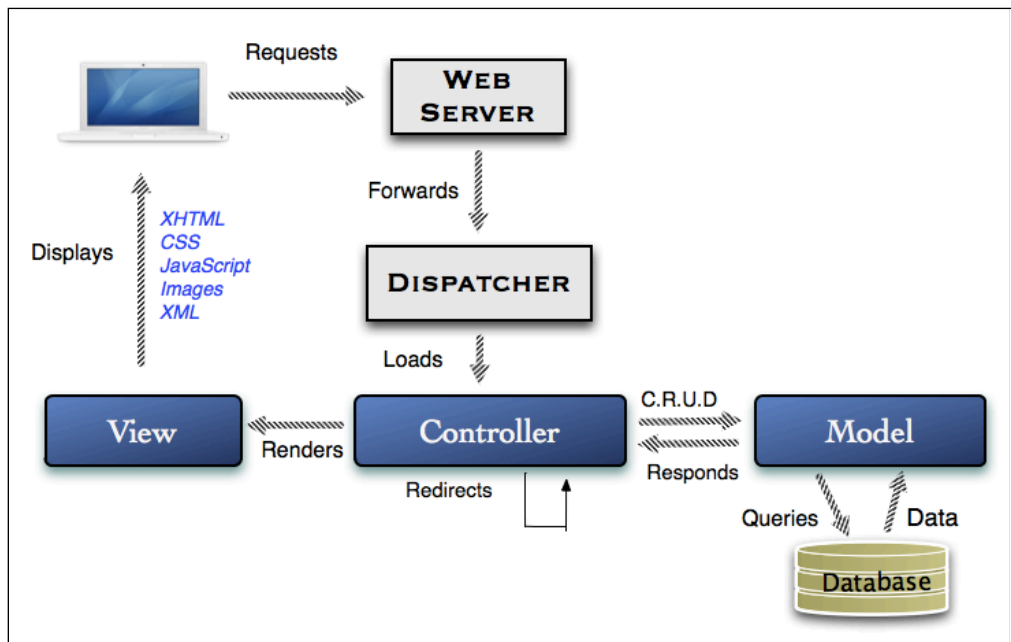


Figure 36: Data flow for a typical Application using a MVC framework

Now let's look at how Rails and Hobo fit into the MVC framework:

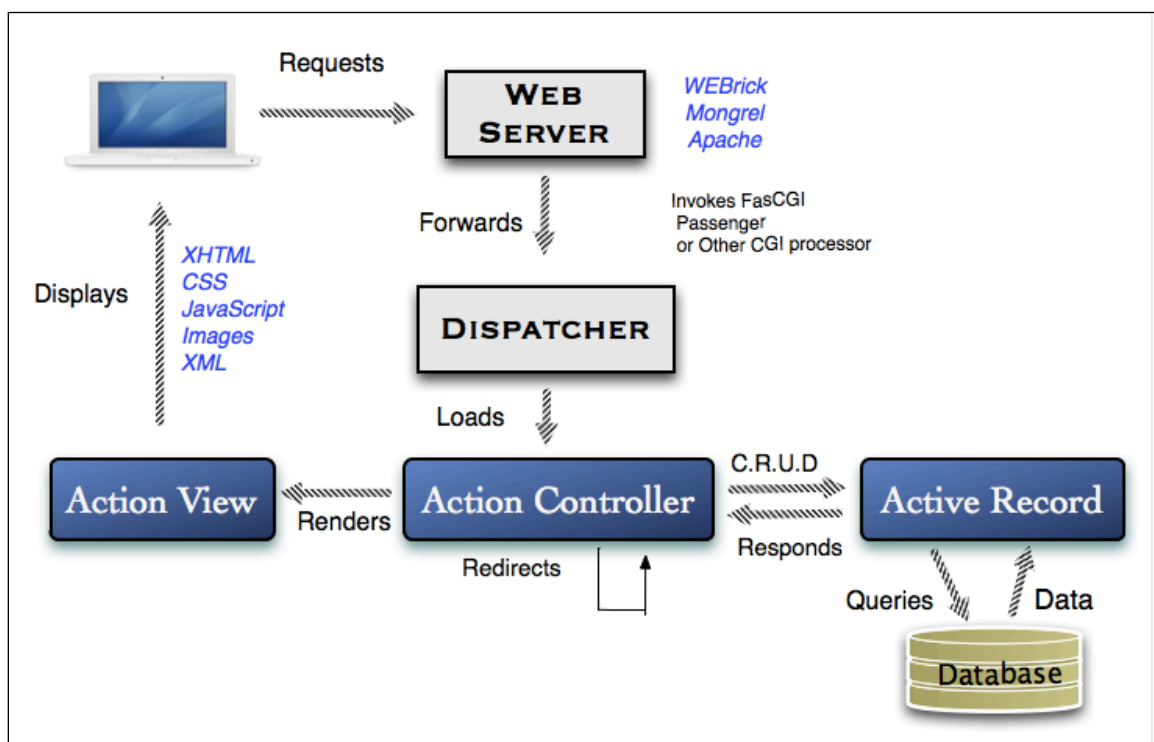


Figure 37: Data flow for a Rails application

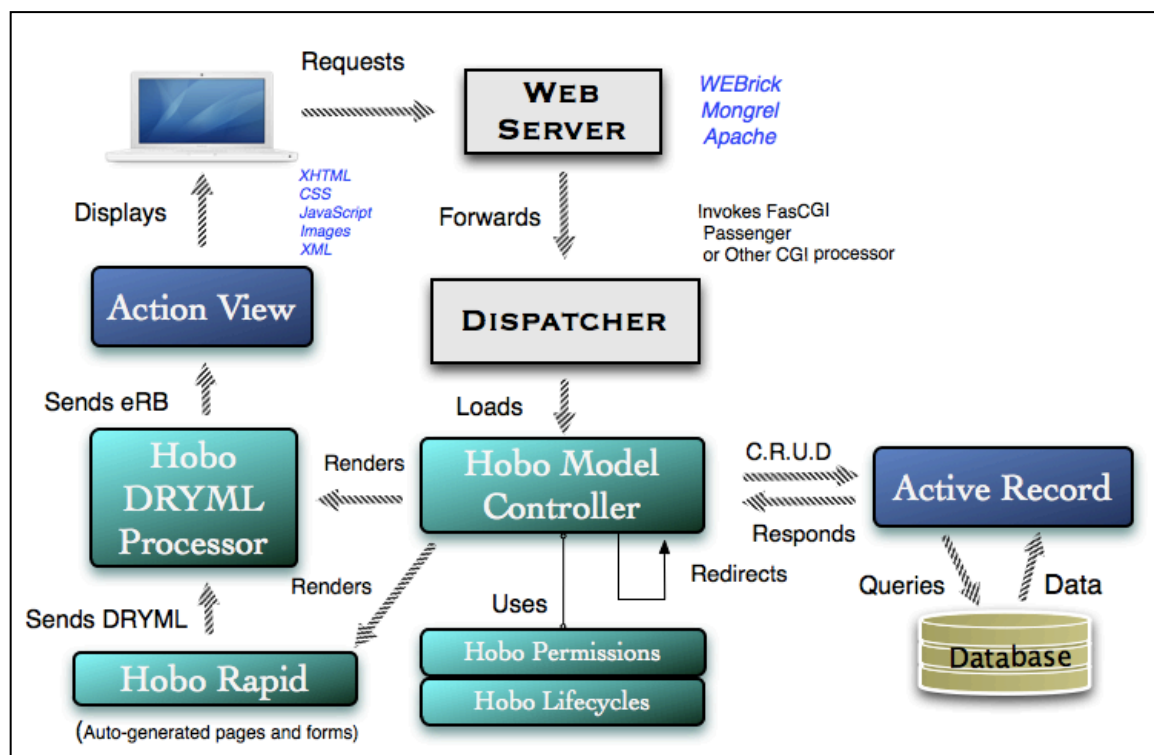


Figure 38: Data flow for a Hobo application

Here are a few talking points:

- The Hobo Model Controller takes the place of the Action Controller in Rails.
- The Hobo Model Controller has access to information from both Hobo Permissions and Hobo Lifecycles that allow it to decide what should be displayed and for whom.
- Hobo Rapid pages are rendered using DRYML, which is passed to the DRYML “processor” that translates more declarative DRYML into standard Rails eRB (embedded Ruby) that is then rendered with Action View in Rails.

A closer look at how the Hobo source code is organization is useful:

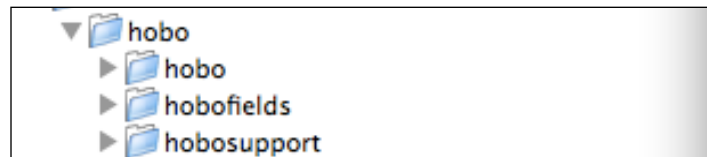


Figure 39: First level look at Hobo source

Note that the first three primary folders under “hobo” represent the three gems installed when you issued the command `gem install hobo` in Chapter 1.:

```
hobo
hobofields
hobosupport
```

Let’s list them in order of precedence (dependency):

```
1. hobosupport
2. hobofields
3. hobo
```

Hobosupport. This is a set of core ruby extensions used throughout Hobo. For those who are very curious, a snapshot of the core components are included in the screenshot below:

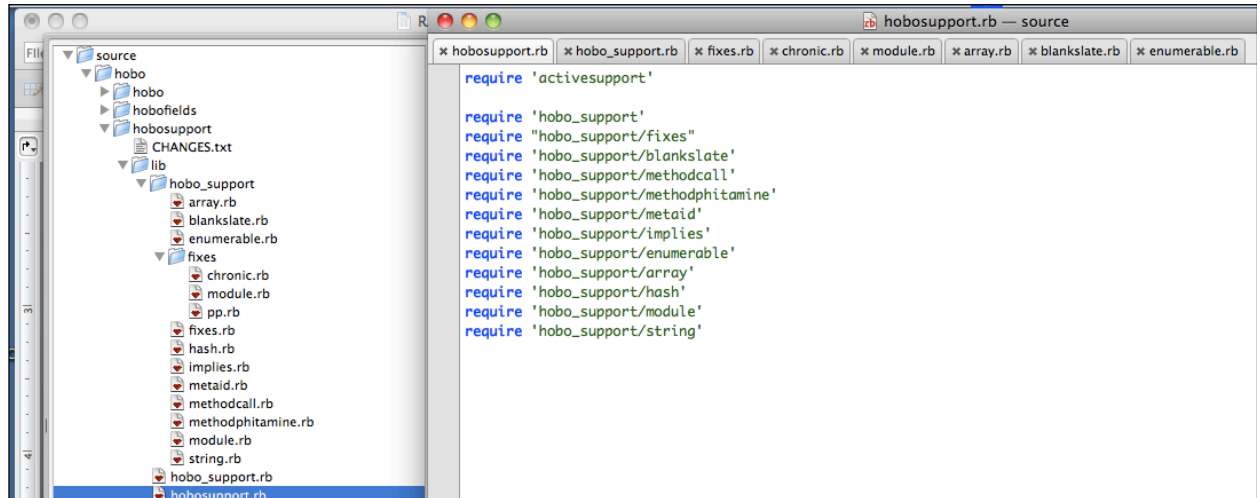


Figure 40: Listing of Ruby programs within the Hobosupport folder

You can go online to GitHub to see all the detail.

Hobofields. The Hobofields gem requires the Hobosupport gem. Hobofields includes these primary sub-folders:

```
hobo_fields
rails_generators
hobo_migration
  templates
hobofield_model
  templates
```


The figure below contains a snapshot of the entire folder hierarchy for the Hobofields gem:

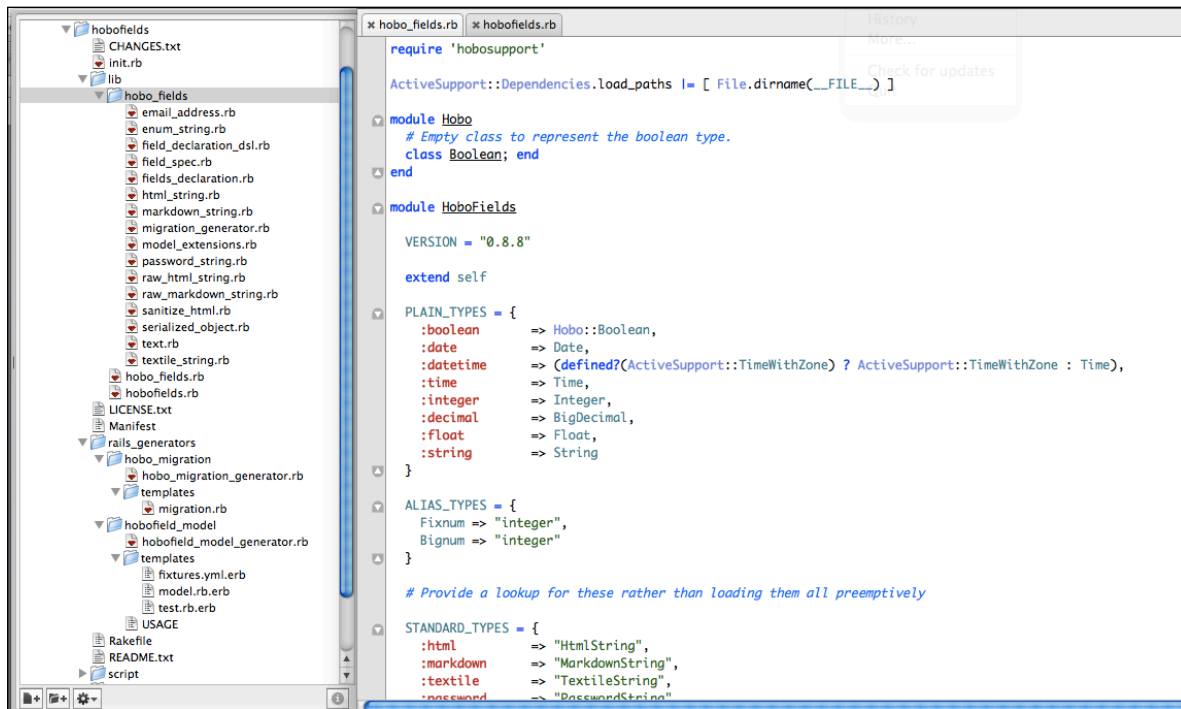


Figure 41: Content overview for the Hobofields gem

(It is beyond the scope of this book to go into all of these programs in detail. Our goal is to give you enough information to use Hobo effectively and provide pointers for Ruby enthusiasts to dig in deeper.)

The big picture is:

- Hobo provides the features necessary to develop in a model-centric way
- Toward that end, the developer declares all of the model related specifications in the model, rather than in migrations. In Rails you focus on building migrations
- Hobo provides additional fields and field parameters not in Rails
- Hobo provides a migration generator that builds and executes the migrations for you.
- When you make a change to a model (add or delete fields, change associations) or add models, the migration generator automatically checks the existing database schema to determine what needs to be changed, then creates the migration specification.

Hobo In Action

Let's look at your options for using the `hobo_migration` command:

<code>--force-drop</code>	Don't prompt with 'drop or rename' - just drop everything
<code>--default-name</code>	Don't prompt for a migration name - just pick one or use the one I provide
<code>--generate</code>	Don't prompt for action - generate the migration
<code>--migrate</code>	Don't prompt for action - generate and migrate
<code>--skip-timestamps</code>	Don't add timestamps to the migration file for this model
<code>--skip-fixture</code>	Don't generate a fixture file for this model

Figure 42: Command line options for Hobo Migrations

Scenario 1. Let's say you modified a model that removed a column named “comments” and added one called “description”. You don't want to stop to be prompted, and you want to have the migration file to be called “removed-comments”:

```
ruby script/generate hobo_migration -migrate -force-drop -default-name removed-comments
```

Scenario 2. Let's say you created a model to be used as a code table and you are not interested in using the default “timestamps” (`created_at`, `updated_at`) provided by Hobo fields. You don't want to stop to be prompted, and you want to leave the naming of the migration file to Hobo:

```
> ruby script/generate hobo_migration -migrate -skip-timestamps -default-name
```

To skip creating a Rails test fixture use the `-skip-fixture` option. We won't go into detail here about this one. For more information see:

<http://ar.rubyonrails.org/classes/Fixtures.html>

Hobo. The “mothership” of all the Hobo gems is, of course `hobo`.

The major folders within the Hobo gem we will review in this chapter are:

```
dryml_generators
rails_generators
taglibs
```

Note the dependencies listed in this screen shot below of the program `hobo.rb`:

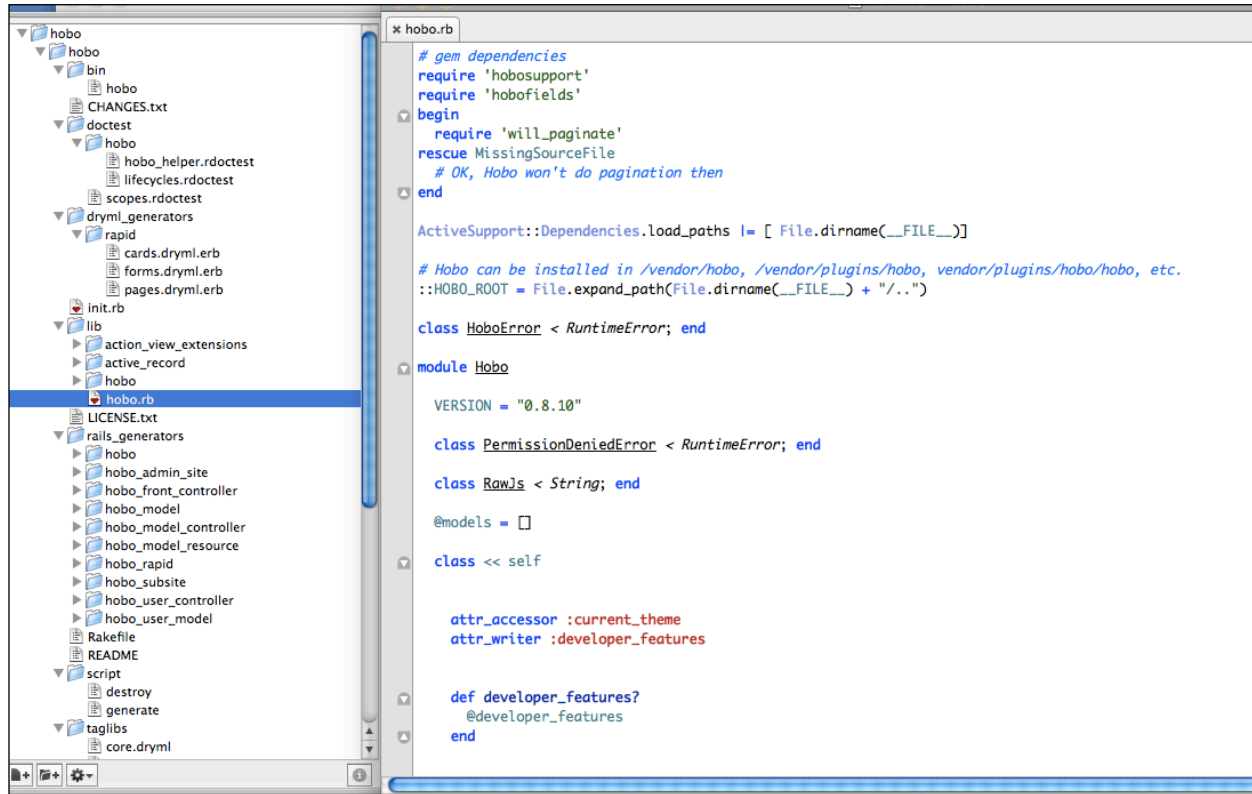


Figure 43: Required hobosupport and hobofield gems

Let's review the programs/commands in the order that you encounter them in your work. In fact, even before you start coding several of these have been executed when you used the hobo command to create your first application shell.

First let's see what options you have with the hobo command:

--user-model	Override the default "User" for the User model, or specify "false" if you don't want one.
--database <database>	Specify the database to be used: sqlite, mysql, oracle, postgres.
--rails <version>	Rails version to use.
--no-rails	Don't run 'rails'.
--invite-only	Add features for an invite-only website (admin site, no signup)
--db-create	Run "rake db:create:all"

Figure 44: Optional parameters for the Hobo command

You can substitute:

- d for --database
- r for --rails
- n for --no-rails

Let's execute the following command to review what happens:

```
> hobo test_generators
```

So here is the console output from this using Hobo 0.8.10:

```
Generating Rails app...
create
create  app/controllers
create  app/helpers
create  app/models
create  app/views/layouts
create  config/environments
create  config/initializers
create  config/locales
create  db
create  doc
create  lib
create  lib/tasks
create  log
create  public/images
create  public/javascripts
create  public/stylesheets
create  script/performance
create  test/fixtures
create  test/functional
create  test/integration
create  test/performance
create  test/unit
create  vendor
create  vendor/plugins
create  tmp/sessions
create  tmp/sockets
create  tmp/cache
create  tmp/pids
create  Rakefile
create  README
create  app/controllers/application_controller.rb
create  app/helpers/application_helper.rb
create  config/database.yml
create  config/routes.rb
create  config/locales/en.yml
create  config/initializers/backtrace_silencers.rb
create  config/initializers/inflections.rb
create  config/initializers/mime_types.rb
create  config/initializers/new_rails_defaults.rb
create  config/initializers/session_store.rb
create  config/environment.rb
create  config/boot.rb
create  config/environments/production.rb
create  config/environments/development.rb
create  config/environments/test.rb
create  script/about
create  script/console
create  script/dbconsole
create  script/destroy
create  script/generate
create  script/runner
create  script/server
create  script/plugin
create  script/performance/benchmark
create  script/performance/profiler
create  test/test_helper.rb
create  test/performance/browsing_test.rb
create  public/404.html
create  public/422.html
create  public/500.html
```

```
create public/index.html
create public/favicon.ico
create public/robots.txt
create public/images/rails.png
create public/javascripts/prototype.js
create public/javascripts/effects.js
create public/javascripts/dragdrop.js
create public/javascripts/controls.js
create public/javascripts/application.js
create doc/README_FOR_APP
create log/server.log
create log/production.log
create log/development.log
create log/test.log

Initialising Hobo...
--> ruby script/generate hobo --add-gem --add-routes
create app/views/taglibs
create app/views/taglibs/themes
create app/views/taglibs/application.dryml
create public/hobothemes
exists app/models
create app/models/guest.rb
exists public/stylesheets
identical public/stylesheets/application.css
create public/javascripts/dryml-support.js
create config/initializers/hobo.rb

Installing Hobo Rapid and default theme...
--> ruby script/generate hobo_rapid --import-tags
create public/javascripts/hobo-rapid.js
create public/javascripts/lowpro.js
create public/javascripts/IE7.js
create public/javascripts/ie7-recalc.js
create public/javascripts/blank.gif
create public/stylesheets/reset.css
create public/stylesheets/hobo-rapid.css
create public/hobothemes/clean/
create public/hobothemes/clean/stylesheets
create public/hobothemes/clean/stylesheets/rapid-ui.css
create public/hobothemes/clean/stylesheets/clean.css
create public/hobothemes/clean/images
create public/hobothemes/clean/images/spinner.gif
create public/hobothemes/clean/images/small_close.png
create public/hobothemes/clean/images/pencil.png
create public/hobothemes/clean/images/fieldbg.gif
create public/hobothemes/clean/images/50-ACD3E6-fff.png
create public/hobothemes/clean/images/300-ACD3E6-fff.png
create public/hobothemes/clean/images/30-DBE1E5-FCFEF5.png
create public/hobothemes/clean/images/30-3E547A-242E42.png
create public/hobothemes/clean/images/101-3B5F87-ACD3E6.png
create app/views/taglibs/themes/clean/
create app/views/taglibs/themes/clean/clean.dryml

Creating user model and controller...
--> ruby script/generate hobo_user_model user
exists app/models/
exists test/unit/
exists test/fixtures/
create app/views/user_mailer
create app/models/user.rb
create test/unit/user_test.rb
create test/fixtures/users.yml
create app/models/user_mailer.rb
create app/views/user_mailer/forgot_password.erb
--> ruby script/generate hobo_user_controller user
exists app/controllers/
exists app/helpers/
create app/views/users
```

```
exists test/functional/
create app/controllers/users_controller.rb
create test/functional/users_controller_test.rb
create app/helpers/users_helper.rb

Creating standard pages...
--> ruby script/generate hobo_front_controller front --delete-index --add-routes
exists app/controllers/
exists app/helpers/
create app/views/front
exists test/functional/
create app/controllers/front_controller.rb
create test/functional/front_controller_test.rb
create app/helpers/front_helper.rb
create app/views/front/index.dryml
```

Let's focus on the commands executed after the underlying Rails app was generated:

1. Initialising Hobo...

```
ruby script/generate hobo --add-gem --add-routes
```

2. Installing Hobo Rapid and default theme...

```
ruby script/generate hobo_rapid --import-tags
```

3. Creating user model and controller...

```
ruby script/generate hobo_user_model user
```

4. Creating standard pages...

```
ruby script/generate hobo_front_controller front --delete-index --add-routes
```

Step 1: Initializing Hobo. The screen shot below will give you the flavor of what the `HoboGenerator` class does:

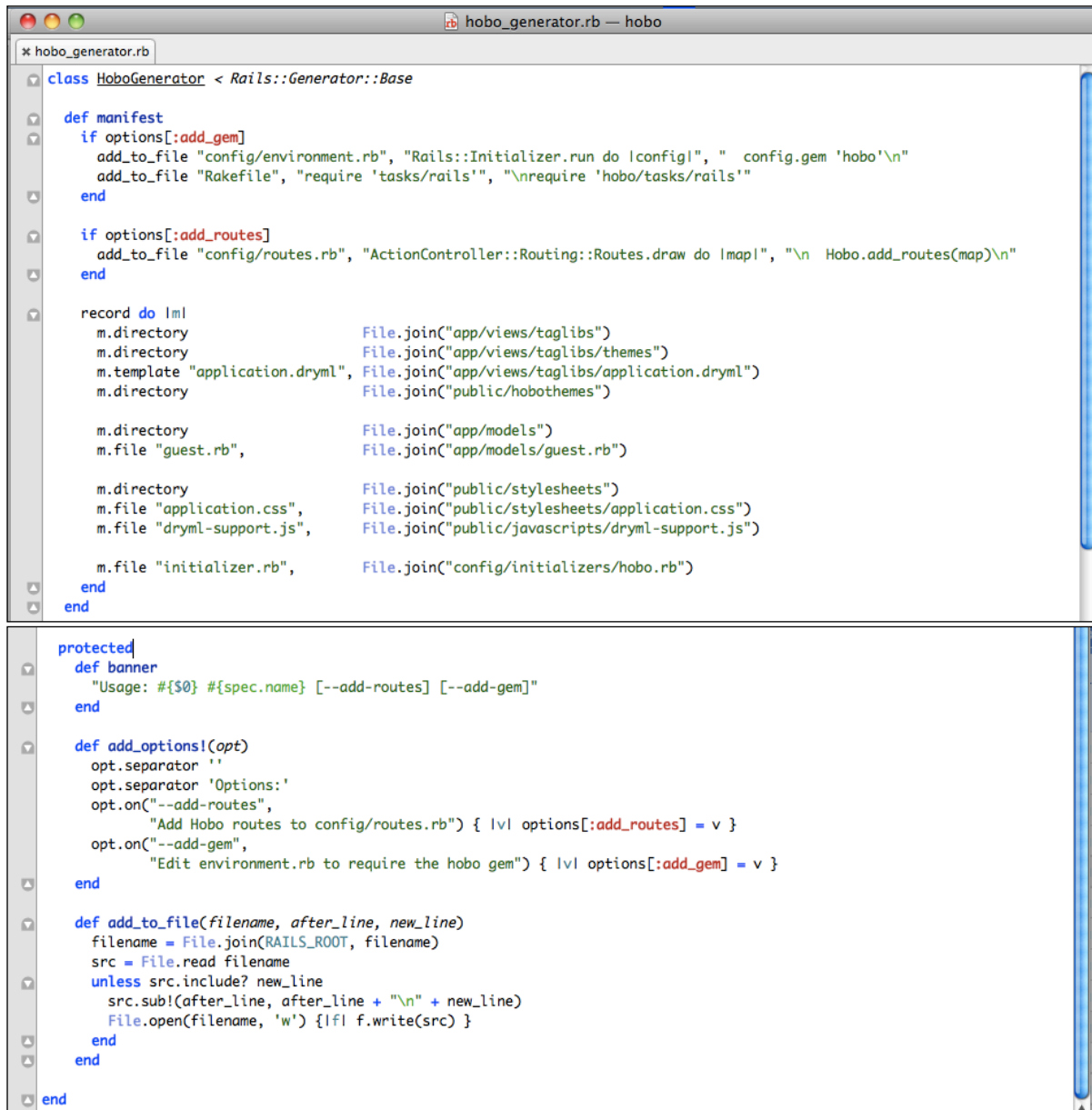


Figure 45: The `HoboGenerator` class actions

You can see that the following files generated by Rails are updated by Hobo:

```
config/environment.rb
rakefile
config/routes
```

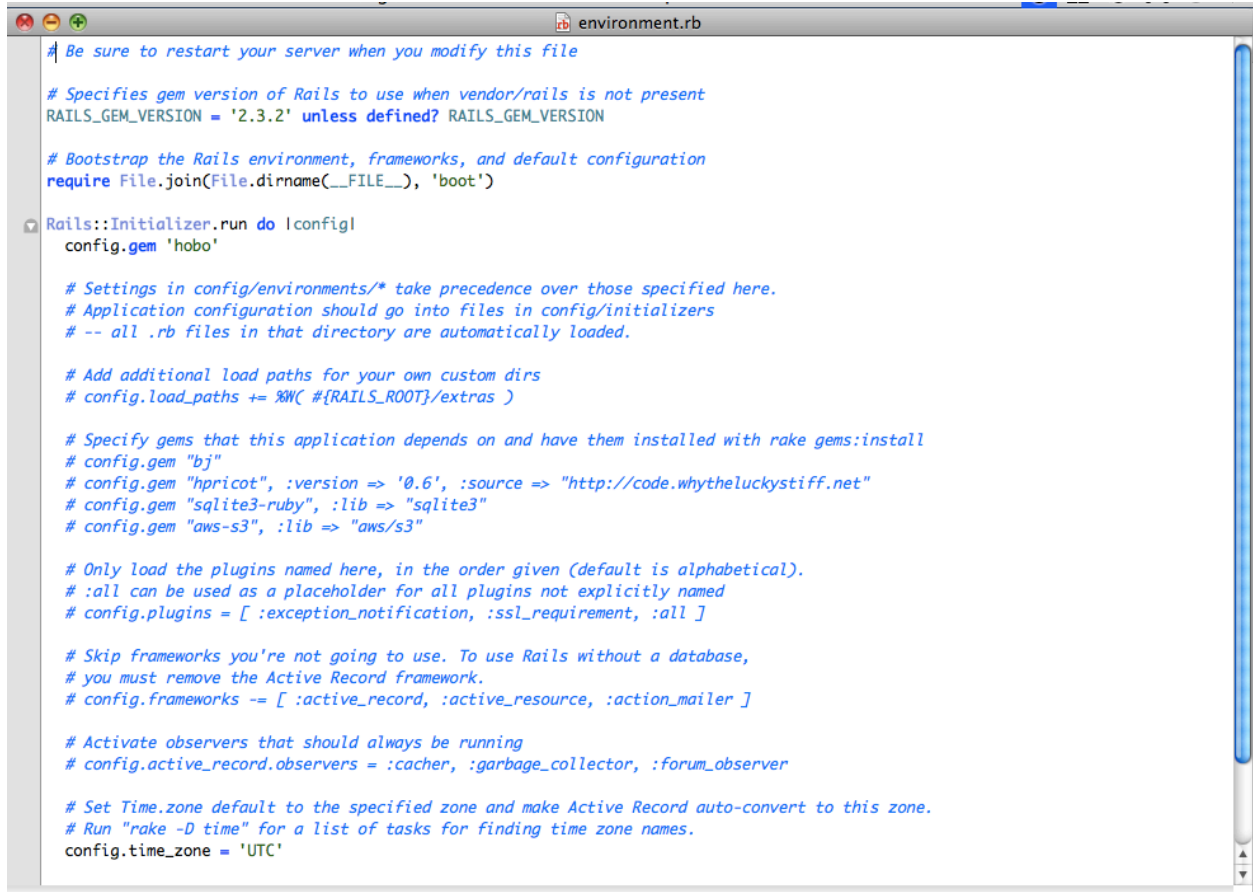


Figure 46: The line “config.gem ‘hobo’ is added in environment.rb by Hobo

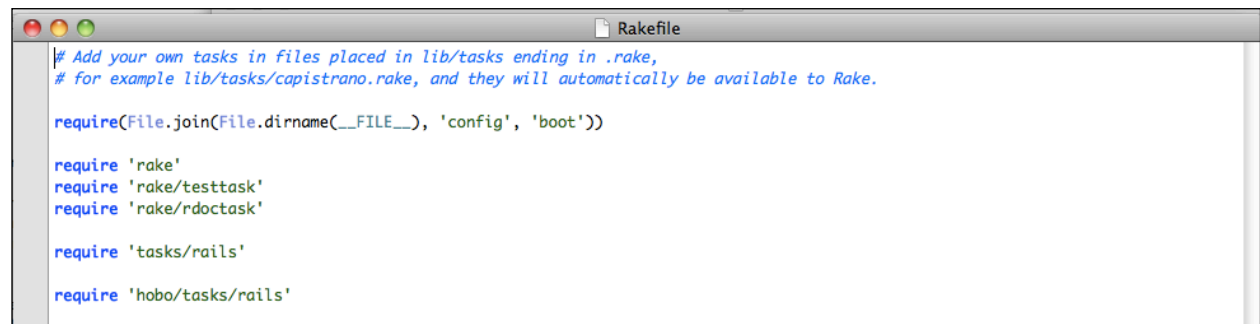


Figure 47: The lines added to the file “rake” by Hobo

We won’t go into detail about the `record do` loop. Suffice it to say that this section creates the generator manifest of directories and files used by the generator. See the following link for a more in-depth discussion:

<http://api.rubyonrails.org/classes/Rails/Generator/Base.html>

Step 2: Installing Hobo Rapid and the default theme.

Here are the options for the Hobo Rapid generator

<code>--import-tags</code>	Modify taglibs/application.dryml to import hobo-rapid and theme tags
<code>--admin</code>	Generate an admin sub-site
<code>--invite-only</code>	Generate an admin sub-site with features for an invite only app

Note that the `-admin` and `-invite-only` parameters are passed by the Hobo command to the other generators:

```
> hobo --invite-only my-invite-only-app
```

Initialising Hobo...

```
--> ruby script/generate hobo --add-gem --add-routes
```

Installing Hobo Rapid and default theme...

```
--> ruby script/generate hobo_rapid --import-tags --invite-only
```

```
create app/views/taglibs/themes/clean/clean.dryml
```

```
dependency hobo_admin_site
```

```
Renaming app/views/taglibs/application.dryml to app/views/taglibs/front_site.dryml
```

```
create app/views/taglibs/application.dryml
```

```
create app/controllers/admin
```

```
create app/views/admin
```

```
create app/controllers/admin/admin_site_controller.rb
```

```
create app/views/taglibs/admin_site.dryml
```

```
create public/stylesheets/admin.css
```

```
dependency hobo_model_controller
```

```
exists app/controllers/admin
```

```
create app/helpers/admin
```

```
create app/views/admin/users
```

```
create test/functional/admin
```

```
create app/controllers/admin/users_controller.rb
```

```
create test/functional/admin/users_controller_test.rb
```

```
create app/helpers/admin/users_helper.rb
```

```
create app/views/admin/users/index.dryml
```

Creating user model and controller...

```
--> ruby script/generate hobo_user_model user --invite-only
```

```
exists app/models/
```

```
exists test/unit/
```

```
exists test/fixtures/
```

```
create app/views/user_mailer
```

```
create app/models/user.rb
```

```
create test/unit/user_test.rb
```

```
create test/fixtures/users.yml
```

```
create app/models/user_mailer.rb
```

```
create app/views/user_mailer/forgot_password.erb
```

```
create app/views/user_mailer/invite.erb
```

```
--> ruby script/generate hobo_user_controller user --invite-only
```

```
exists app/controllers/
```

```
exists app/helpers/
```

```
create app/views/users
```

```
exists test/functional/
```

```
create app/controllers/users_controller.rb
```

```
create test/functional/users_controller_test.rb
```

```
create app/helpers/users_helper.rb
```

```
create app/views/users/accept_invitation.dryml
```

Creating standard pages...

```
--> ruby script/generate hobo_front_controller front --delete-index --add-routes --invite-only
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/front
      exists  test/functional/
      create  app/controllers/front_controller.rb
      create  test/functional/front_controller_test.rb
      create  app/helpers/front_helper.rb
      create  app/views/front/index.dryml
```

Invite-only website

If you wish to prevent all access to the site to non-members, add 'before_filter :login_required'

to the relevant controllers, e.g. to prevent all access to the site, add

```
include Hobo::AuthenticationSupport
before_filter :login_required
```

to application_controller.rb (note that the include statement is not required for hobo_controllers)

NOTE: You might want to sign up as the administrator before adding this!

Step 3: Creating the User model and controller.

```
> ruby script/generate hobo_user_model user
```

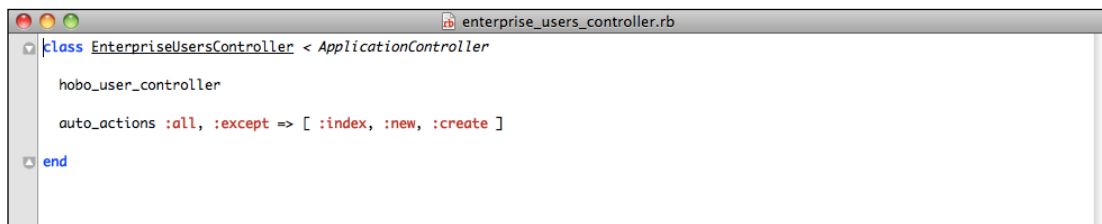
Notice that the parameter after `hobo_user_model` is `user`. This parameter will be passed down from the hobo command if you execute hobo passing it a name you would prefer for the model to handle users.

```
> hobo --user-model enterprise_user myapp
```

..will output to the console (we'll skip the other output that would stay the same):

```
...
Creating enterprise_user model and controller...
--> ruby script/generate hobo_user_model enterprise_user
    exists  app/models/
    exists  test/unit/
    exists  test/fixtures/
    create  app/views/enterprise_user_mailer
    create  app/models/enterprise_user.rb
    create  test/unit/enterprise_user_test.rb
    create  test/fixtures/enterprise_users.yml
    create  app/models/enterprise_user_mailer.rb
    create  app/views/enterprise_user_mailer/forgot_password.erb
--> ruby script/generate hobo_user_controller enterprise_user
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/enterprise_users
    exists  test/functional/
    create  app/controllers/enterprise_users_controller.rb
```

Let's look at the controller generated:



```
class EnterpriseUsersController < ApplicationController
  hobo_user_controller
  auto_actions :all, :except => [ :index, :new, :create ]
end
```

Figure 48: Users Controller generated by Hobo

Let's look at the models:

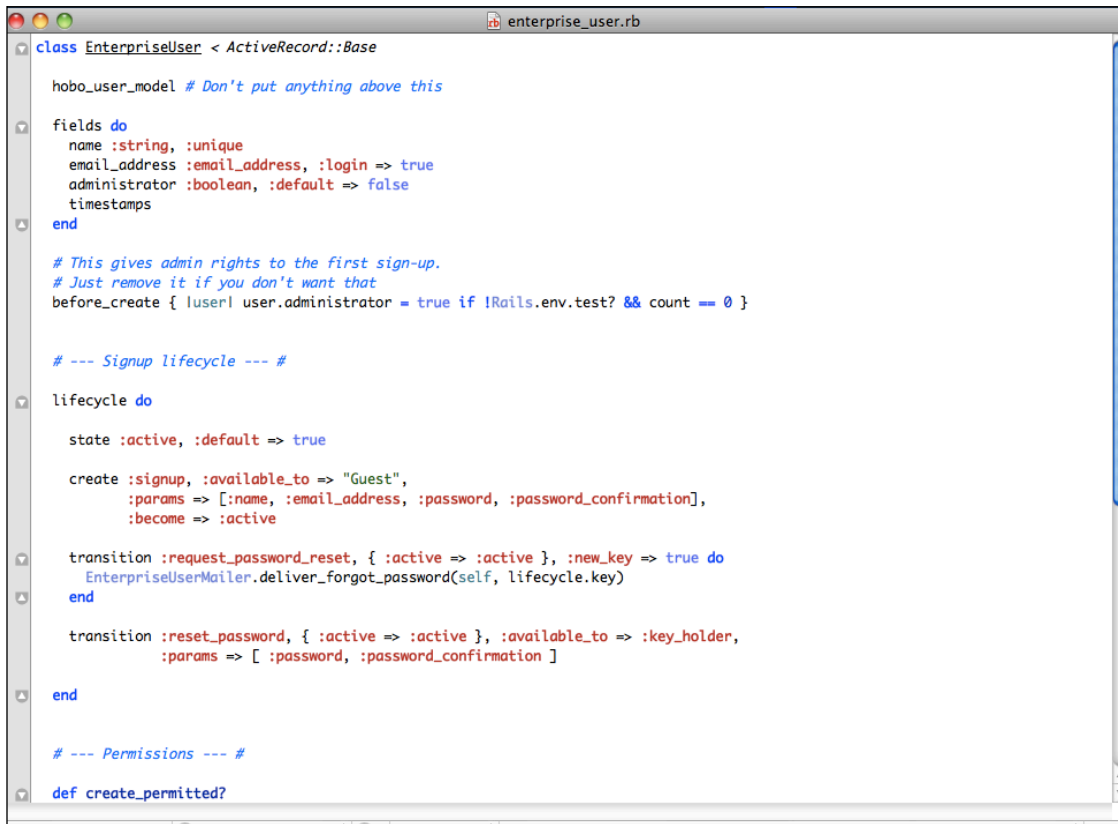


Figure 49: User model with Lifecycles generated by Hobo

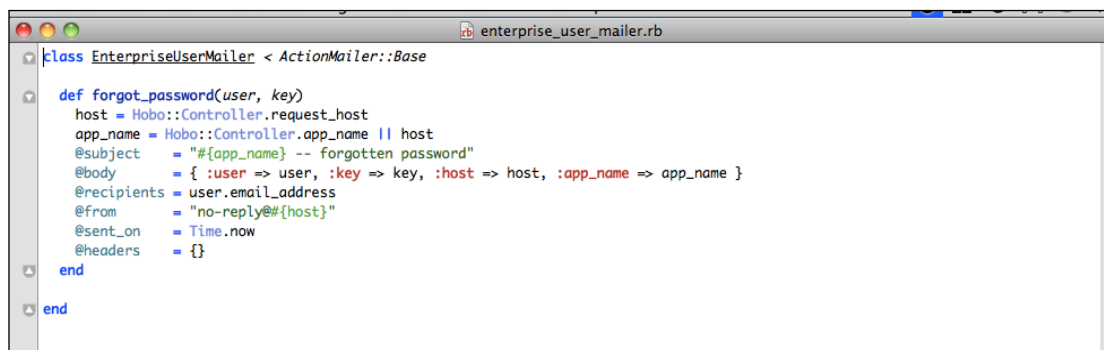


Figure 50: Action Mailer Model generated by Hobo

Notice that we have an “Action Mailer” model created for us as well. This is used for forgotten passwords as well as the `-invite-only` option for creating an application.

Let’s create another app that is for invitation only:

```
> hobo --user-model enterprise_user -invite-only app_by_invite
```

Now let's look at the console output that comes after the "Generating Rails App" portion:

```
Initialising Hobo...
--> ruby script/generate hobo --add-gem --add-routes
      create  app/views/taglibs
      create  app/views/taglibs/themes
      create  app/views/taglibs/application.dryml
      create  public/hobothemes
      exists  app/models
      create  app/models/guest.rb
      exists  public/stylesheets
      identical public/stylesheets/application.css
      create  public/javascripts/dryml-support.js
      create  config/initializers/hobo.rb

Installing Hobo Rapid and default theme...
--> ruby script/generate hobo_rapid --import-tags --invite-only
      create  public/javascripts/hobo-rapid.js
      create  public/javascripts/lowpro.js
      create  public/javascripts/IE7.js
      create  public/javascripts/ie7-recalc.js
      create  public/javascripts/blank.gif
      create  public/stylesheets/reset.css
      create  public/stylesheets/hobo-rapid.css
      create  public/hobothemes/clean/
      create  public/hobothemes/clean/stylesheets
      create  public/hobothemes/clean/stylesheets/rapid-ui.css
      create  public/hobothemes/clean/stylesheets/clean.css
      create  public/hobothemes/clean/images
      create  public/hobothemes/clean/images/spinner.gif
      create  public/hobothemes/clean/images/small_close.png
      create  public/hobothemes/clean/images/pencil.png
      create  public/hobothemes/clean/images/fieldbg.gif
      create  public/hobothemes/clean/images/50-ACD3E6-fff.png
      create  public/hobothemes/clean/images/300-ACD3E6-fff.png
      create  public/hobothemes/clean/images/30-DBE1E5-FCFEF5.png
      create  public/hobothemes/clean/images/30-3E547A-242E42.png
      create  public/hobothemes/clean/images/101-3B5F87-ACD3E6.png
      create  app/views/taglibs/themes/clean/
      create  app/views/taglibs/themes/clean/clean.dryml
      dependency hobo_admin_site
Renaming app/views/taglibs/application.dryml to app/views/taglibs/front_site.dryml
      create  app/views/taglibs/application.dryml
      create  app/controllers/admin
      create  app/views/admin
      create  app/controllers/admin/admin_site_controller.rb
      create  app/views/taglibs/admin_site.dryml
      create  public/stylesheets/admin.css
      dependency hobo_model_controller
      exists  app/controllers/admin
      create  app/helpers/admin
      create  app/views/admin/users
      create  test/functional/admin
      create  app/controllers/admin/users_controller.rb
      create  test/functional/admin/users_controller_test.rb
      create  app/helpers/admin/users_helper.rb
      create  app/views/admin/users/index.dryml

Creating enterprise_user model and controller...
--> ruby script/generate hobo_user_model enterprise_user --invite-only
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/views/enterprise_user_mailer
      create  app/models/enterprise_user.rb
      create  test/unit/enterprise_user_test.rb
      create  test/fixtures/enterprise_users.yml
```

```
create app/models/enterprise_user_mailer.rb
create app/views/enterprise_user_mailer/forgot_password.erb
create app/views/enterprise_user_mailer/invite.erb
--> ruby script/generate hobo_user_controller enterprise_user --invite-only
exists app/controllers/
exists app/helpers/
create app/views/enterprise_users
exists test/functional/
create app/controllers/enterprise_users_controller.rb
create test/functional/enterprise_users_controller_test.rb
create app/helpers/enterprise_users_helper.rb
create app/views/enterprise_users/accept_invitation.dryml
```

Creating standard pages...

```
--> ruby script/generate hobo_front_controller front --delete-index --add-routes --invite-only
exists app/controllers/
exists app/helpers/
create app/views/front
exists test/functional/
create app/controllers/front_controller.rb
create test/functional/front_controller_test.rb
create app/helpers/front_helper.rb
create app/views/front/index.dryml
```

Invite-only website

If you wish to prevent all access to the site to non-members,
add 'before_filter :login_required' to the relevant controllers, e.g. to prevent all access to
the site, add

```
include Hobo::AuthenticationSupport
before_filter :login_required
```

to application_controller.rb (note that the include statement is not required for
hobo_controllers)

NOTE: You might want to sign up as the administrator before adding this!

Now let's look at the user model, focusing on the Lifecycle changes for the invitation-only app:

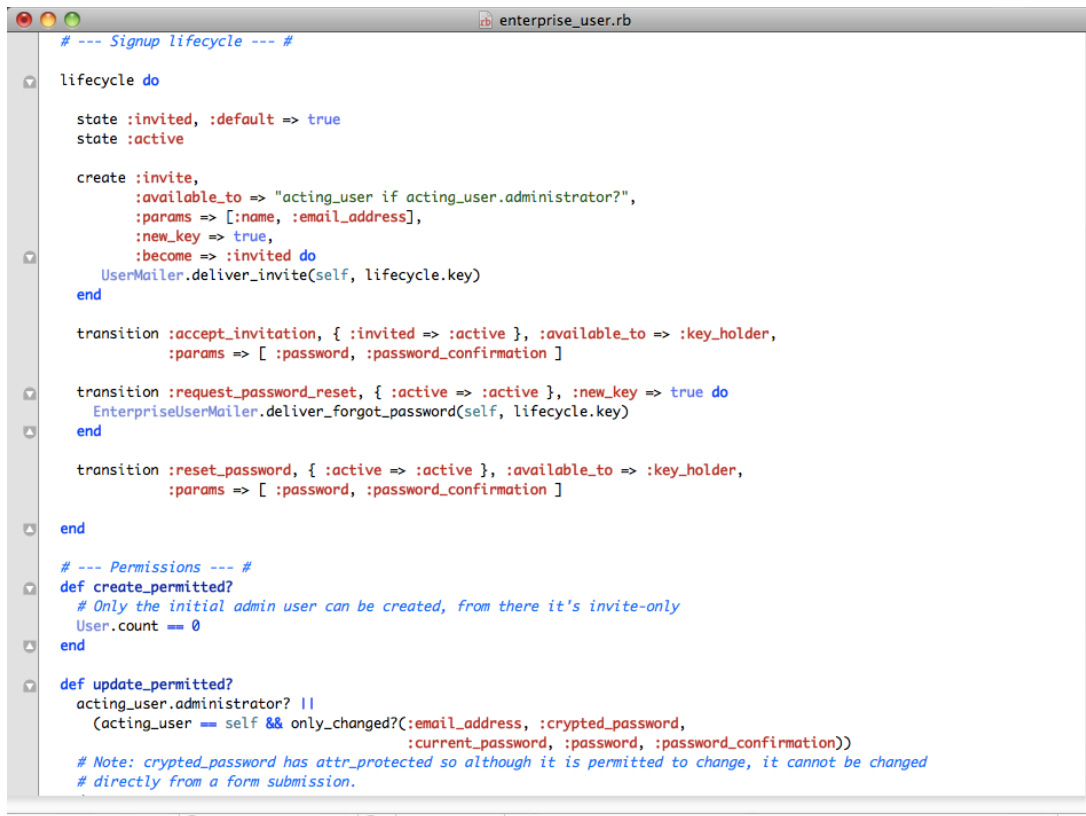


Figure 51: User model generated for an "--invite-only" Hobo application

And the controller:

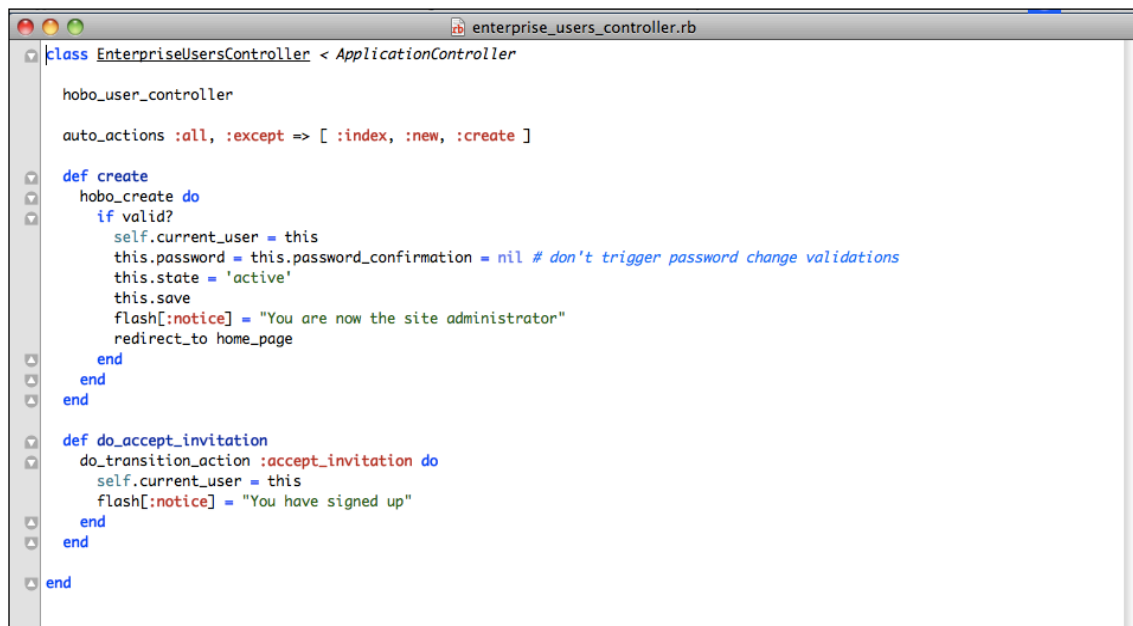


Figure 52: Users Controller generated with an "--invite-only" Hobo application

And even the Action Mailer:



```
class EnterpriseUserMailer < ActionMailer::Base

  def forgot_password(user, key)
    host = Hobo::Controller.request_host
    app_name = Hobo::Controller.app_name || host
    @subject = "#{app_name} -- forgotten password"
    @body = { :user => user, :key => key, :host => host, :app_name => app_name }
    @recipients = user.email_address
    @from = "no-reply@#{host}"
    @sent_on = Time.now
    @headers = {}
  end

  def invite(user, key)
    host = Hobo::Controller.request_host
    app_name = Hobo::Controller.app_name || host
    # FIXME - nasty hack
    app_name.remove!(/ - Admin$/ )
    @subject = "Invitation to #{app_name}"
    @body = { :user => user, :key => key, :host => host, :app_name => app_name }
    @recipients = user.email_address
    @from = "no-reply@#{host}"
    @sent_on = Time.now
    @headers = {}
  end

end
```

Figure 53: Action Mailer model generated with an "--invite-only" Hobo application

Step 4: Creating the standard pages

```
> ruby script/generate hobo_front_controller front --delete-index  
--add-routes
```

One of nice features Hobo provides is a working web site with a home page and tabs for each your models right out of the box.

The `hobo_front_controller` generator provides this.

Note the parameters that Hobo uses when you first create an application:

```
--delete-index  
--add-routes
```

The first parameter tells the generator not to put in a list of models within the page. The second parameter tells the generator to add routes to the models (e.g., tabs) when they are created. Let's look at the console output displayed during the execution of this generator:

```
Creating standard pages...  
--> ruby script/generate hobo_front_controller front --delete-index --add-routes  
exists app/controllers/  
exists app/helpers/  
create app/views/front  
exists test/functional/  
create app/controllers/front_controller.rb  
create test/functional/front_controller_test.rb  
create app/helpers/front_helper.rb  
create app/views/front/index.dryml
```

Looking at the Hobo source code we can see how this works:

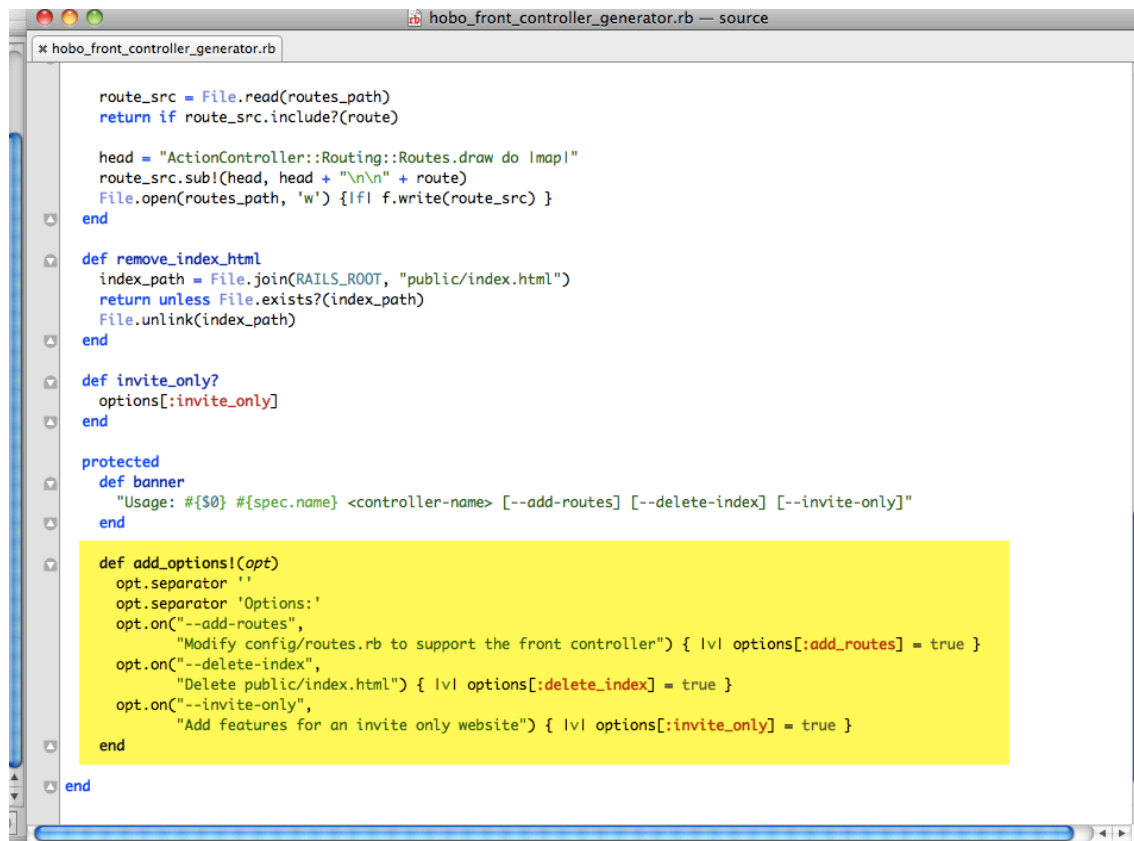


Figure 54: Source code for "hobo_front_controller_generator.rb"

Hobo Enhancement Summary

Fields

A big difference between Hobo and Rails is that in Hobo fields are declared in the model, whereas in Rails they are declared in the migrations. In our opinion it is more intuitive and DRY to maintain all of the model code in one place, creating or changing the database design by edit the `model`, letting Hobo build the migration code necessary to make any required changes. You can look in one place to see everything about a model. You don't need to jump to the `schema.rb` file.

The `hobo_model_resource` generator creates models, controllers, and views. Any changes to field definitions or associations in the model can be propagated throughout the application with the `hobo_migration` generator. There is no need to edit the migration file. The `hobo_migration` generator handles this for you.

If you only want to create a model, use the `hobo_model` generator.

Indexes

This is one of the newest additions to Hobo thanks to Matt Jones. This feature provides for automatic field generation for the foreign keys of related models, and an easy-to-use declarative syntax to specified single and multi-part keys with a model definition.

Validations

As we have discussed elsewhere in the book, Hobo provides some useful in-line shortcuts for the simplest validations that Rails does not provide. See in red below:

```
Fields do
  name :string, :required, :unique, :length => 32
end
```

Use standard rails validations outside the `fields...do` block.

This works the same as in Rails so we will not add anything new at this point.

Views

Views take the most time to develop in any application and Hobo provides more tools here than in the other two modules to meet that challenge. In fact, it provides an entire language to use to develop view templates (a Rails web page).

Hobo views are developed entirely differently than in Rails. Once you define your models and controllers, Hobo is capable of automatically generating an entire set of views on the fly. This means that at the beginning of your development process you do not have to code a view

template at all. Hobo automatically creates them whenever the user requests that data be rendered.

DRYML Tags - Hobo constructs view templates using Hobo's mark-up language, called Don't Repeat Yourself Markup Language. The tags are reusable components that perform specific processes defined in Ruby.

You build DRYML tags using a definition language and you use the tags to build data-driven view templates in an XML-like syntax. You can create your own tags and build tags from other tags. Hobo comes with its own library of fundamental tags called the Rapid Library.

For those of you with a Rails background, you can think of these as similar to Rails "helpers", but they are used with an easier XML syntax rather than with [Ruby embedded in the templates.]

Rapid Tag Library. This library is a set of tags that deal with all aspects of view template specification. It includes tags for links, forms, input controls, navigation, logic and much more. They are DRYML tags in that they are defined with the DRYML definition language. Many rapid tags call other Rapid tags implicitly. For example, you may never see a Rapid `<input>` called explicitly in the auto-generated tags described below.

Rapid Generator. This generator is a real time generator as opposed to the code generators we usually talk about in Rails development. Rapid creates a set of auto-generated tags that are defined by model fields and model relationships. Rapid uses these auto-generated tags to render individual view templates.

There are three files within the view directory, located at `views/taglibs/auto/rapid` where all the automatically generated tags are created. They are:

```
pages.dryml
forms.dryml
cards.dryml
```

Tags within `pages.dryml` call tags within `forms.dryml` or `cards.dryml`. (Tags are defined in these files, not invoked). The Rapid generator invokes these tags either automatically without additional code or from manually created code in `application.dryml`.

Auto-generated tags in `pages.dryml`. Rapid generates a set of four complex tags for each model:

```
<index-page>
<new-page>
<show-page>
<edit-page>
```

These auto-generated tags are invoked by the corresponding controller action (index, new, show or edit) to render view templates corresponding to each action.

The other three fundamental actions--create, update and destroy--do not have their own Hobo page. They appear as links within the four auto-generated tags, some invoked within the Rapid `<a>` tag (similar to the HTML `<a>` hyperlink tag), or the `<submit>` or `<delete-button>` tag. The four tags that are used to render templates plus the three that appear as links or buttons total to the seven actions we repeatedly cite.

Tag definitions for the four basic tags begin like this:

```
<def tag="index-page" for="Contact">
  ...
```

There is a lot going on in the tag definitions in `pages.dryml` that you might not fully understand yet. This includes calls to HTML tags with parameterization syntax (you see `params` declarations), unfamiliar tags like `<collection>` and so forth.

The figure below summarizes some important information about the four basic tags:

Tag Name	Controller Action	Main Data Tags	Actions Linked
<code><index-page></code>	index (list)	<code><collection></code>	new
<code><new-page></code>	new	<code><form></code>	create
<code><show-page></code>	show	<code><name></code> , <code><field-list></code> <code><collection></code> (for associated models)	edit
<code><edit-page></code>	edit	<code><form></code>	update

Figure 55: Hobo Rapid action related tags

The content of the four table columns is explained below:

Tag Name: This tag name is what is the text used to invoke the tag within a Hobo template or `application.dryml` (see below).

Controller Action: indicates the action that calls the particular tag, which is rendered as a Hobo view template.

Main Data Tags: Indicates the most used sub-tags responsible for data input and output. Other sub-tags handle formatting tasks.

Actions Linked: indicates which actions have tags which link to other actions.

Programming note. Linked actions do not appear explicitly as a tag but as attributes of the `<a>` tag or implicitly within the `<submit>` or `<delete-button>` tag.

Each of the four pages tags calls tags in the `forms.dryml` and `cards.dryml` file libraries. The `<show-page>` and `<edit-page>` tag explicitly call `<form>` tags within `forms.dryml`. The `<index-page>` and `<edit-page>` tags call the `<card>` tags to display lists or individual records but DO NOT do so explicitly.

Programming note. `<index-page>` and `<show-page>` call `<card>` tags *implicitly* through the `<collection>`, `<field-list>` and `<name>` tags. This does not mean, for example that `<field-list>` only uses `<card>` tags. It uses its polymorphic capability to know what type of page tag it is being called from to determine what to do.

Application.dryml file. Like the `pages.dryml` file, this is also a repository for tag definitions. A tag definition placed here with the same name as a tag definition in `pages.dryml`, `forms.dryml` or `cards.dryml` auto-generated libraries will override the definition in these libraries. Additional definitions may also be placed in this library file and will be available to all view templates within the application.

A typical use for this file is to copy a tag definition from an auto-generated library and then make edits to it in `application.dryml`.

Programming Note. `Application.dryml` (as of Hobo 0.8.9) is the only library that permits tag definitions that are extensions of other tags that you first learned about in the tutorials. It is anticipated that Hobo 1.0 will allow extensions in other dryml files.

View templates. View templates are stored within view directories carrying the plural of the model name. Hobo view templates have the `.dryml` extension in contrast to the `.erb` or `.rhtml` (older) extension of Rails templates. You can of course use these template types since Hobo is a Rails application, but you probably will not need to.

View layouts. Rails has a layout file to handle markup that is common to many templates such as header and footer information. Since it is so easy to use DRYML tags, you will probably find it unnecessary to use layouts.

Template Processing Order. The diagram below outlines the precedence logic for Hobo rendering of templates. One very important issue to keep straight is the difference between tag definitions and tag usage.

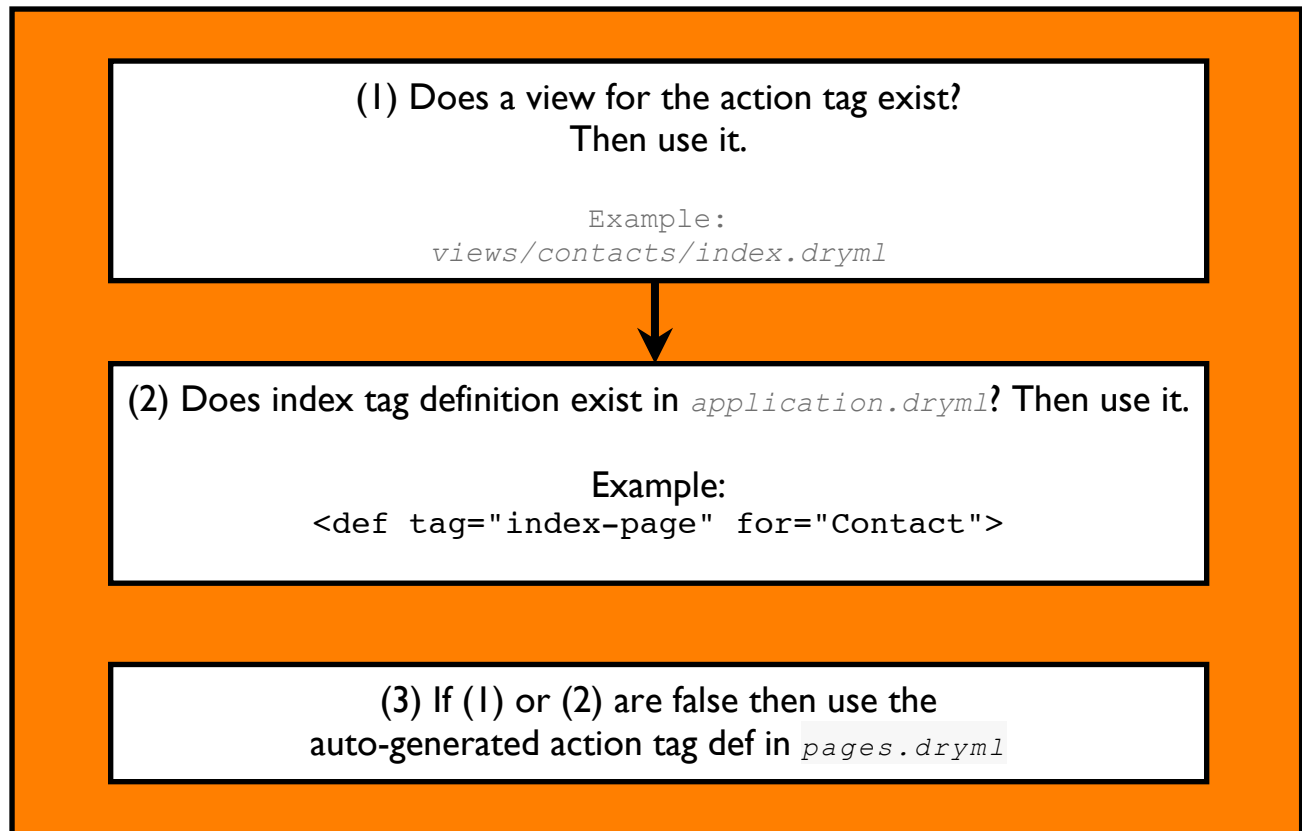


Figure 56: Hobo precedence logic for action tags

In *pages.dryml* or *application.dryml*, there are only tag definitions. Hobo takes these definitions and creates tags on the fly from which it renders templates. You never actually see the tags anywhere in the application. If you have coded your own template (e.g., *show.dryml*) you may have both tag definitions and tag usage within that template file. Remember tag definitions begin with the `<def>` tag and tag usage invokes the tag by name, e.g., `<index-page>` in the above example.

Chapter 4 – The Hobo Permissions Systems

The Hobo Permission System (aka “permissions”) is an extension to Rails Active Record that allows you to define which actions on your models are permitted by which users.

Hobo’s controllers and DRYML tag libraries use this information to automatically customize their behavior according to your definitions.

Introduction

One of the core pieces of the Hobo puzzle is the permission system. The permission system itself lives in the model layer - it is a set of extensions to Active Record models. It’s not a particularly complex set of extensions but the overall effect in Hobo is very powerful. This comes not so much from the permission system itself, but from how it is *used*. Hobo’s controllers use the permission system to decide if a given request is allowed or not. In the view layer, the Rapid tag library uses the permission system to decide what to render for the currently logged in user.

To understand how it all fits together, it’s helpful to be clear about this distinction:

The permission system is a model level feature, but it is used in both the controller and view layers.

This guide will be mostly about how it all works in the model layer, but we’ll also talk a little about how the controllers and tags use the permissions.

At its heart, the permission system is fairly simple, it just provides methods on each model that allow the following four questions to be asked:

Is a given user allowed to:

- *Create* this record?
- *Update* the database with the current changes to this record? (Thanks to Active Record’s ability to track changes)
- *Destroy* the current record?
- *View* the current record, or an individual attribute?.

There is also a fifth permission, which is more of a pseudo permission. Can this user:

- *Edit* a specified attribute of the record

We call this pseudo permission because it is not a request to actually *do something* with the record. It is more like asking: if, at some point in the future, the user tries to update this attribute, will that be allowed? Clearly edit permission is closely related to update permission, but it’s not

quite the same. In fact, you often don't need to declare edit permissions because Hobo can figure them out from your update permission. We'll cover this in more detail later, but for now just be aware that edit permission is a bit of an odd-one-out.

Defining permissions

In a typical Hobo app, the place where the permission system is most prominent in your own code is your permission declarations. These are methods, which you define on your models, known as “permission methods”. These methods are where you tell the permission system who is allowed to do what. The permission methods are called by the framework - it is unusual to call them yourself.

The four basic permission methods

When you generate a new Hobo model, you get stubs for the following methods.

- `def create_permitted?`
- `def update_permitted?`
- `def destroy_permitted?`
- `def view_permitted?(attribute)`

The methods must return true or false to indicate whether or not the operation is allowed. We'll see some examples in a moment but we first need to look at what information the methods have access to.

acting_user

The user performing the action is available via `acting_user` method. This method will always return a user object, even if no one is logged in to the app, because Hobo has a special `Guest` class to represent a user that is not logged in. Two useful methods that are available on all Hobo user objects are:

- `guest?` – returns true if the user is a guest, i.e. no-one is logged in.
- `signed_up?` – returns true if the user is not a guest.

So for example, to specify that you must be logged in to create a record:

```
def create_permitted?  
  acting_user.signed_up?  
end
```

It's also common to compare the `acting_user` with associations on your model, for example, say your model has an owner:

```
belongs_to :owner, :class_name => "User"
```

You can assert that only the owner can make changes like this:

```
def update_permitted?  
  owner == acting_user  
end
```

There is a downside to that method – the `owner` association will be fetched from the database. That's not really necessary, as the foreign key that we need has already been loaded. Fortunately Hobo adds a comparison method for every `belongs_to` that avoids this trip to the database:

```
def update_permitted?  
  owner_is? acting_user  
end
```

Change tracking

When deciding if an update is permitted (i.e., in the `update_permitted?` method), it will often be important to know what exactly has changed. In a previous version of Hobo we had to jump through a lot of hoops to make this information available. No longer – Active Record now tracks all changes made to an object. For example, say you wish to find out about changes to an attribute `status`. The following methods (among others) are available:

- `status_changed?` - returns true if the attribute has been changed
- `status_was` - returns the old value of the attribute

Note that these methods are only available on attributes, not on associations. However, as a convenience Hobo models add `*_changed?` for all `belongs_to` associations.

For example, the following definition means that only signed up users can make changes, and the `status` attribute cannot be changed by anyone:

```
def update_permitted?  
  acting_user.signed_up? && !status_changed?  
end
```

As a stylistic point, sometimes it can be clearer to use early returns, rather than to build up a large and complex boolean expression. This approach is also a bit easier to apply comments to. For example:

```
def update_permitted?    # Must be signed up:  
  return false unless acting_user.signed_up?  
  !status_changed?  
end
```

Change tracking helpers

Making assertions about changes to many attributes can quickly get tedious:

```
def update_permitted?  
  !(address1_changed? ||  
    address2_changed? ||  
    city_changed? ||  
    zipcode_changed?)  
end
```

The permission system provides four helpers to make code like this more concise and clearer. Each of these methods are passed one or more attribute names:

- `only_changed?` – are the attributes passed the only ones that have changed?
- `none_changed?` – have none of the attributes passed been changed?
- `any_changed?` – have any of the attributes passed been changed?
- `all_changed?` – have all of the attributes passed been changed?

So, for example, the previous `update_permitted?` could be simplified to:

```
def update_permitted?  
  none_changed? :address1, :address2, :city, :zipcode  
end
```

Ruby tip: if you want to pass an array, use Ruby's 'splat' operator:

```
READ_ONLY_ATTRS = %w(address1 address2 city zipcode)  
  
def update_permitted?  
  none_changed? *READ_ONLY_ATTRS  
end
```

Note that you can include the names of `belongs_to` associations in your attribute list.

Examples

Let's go through a few examples. Here's a definition that says you cannot create records faking the owner to be someone else, and `state` must be 'new':

```
def create_permitted?  
  return false unless owner_is? acting_user  
  state == "new"  
end
```

Note that by asserting `owner_is? acting_user` you are implicitly asserting that the `acting_user` is signed up, because `owner` can never be a reference to a guest user.

A common requirement for update permission is to restrict the list of fields that can be changed according to the type of user. For example, maybe an administrator can change anything, but a non-admin can only change a given set of fields:

```
def update_permitted?  
  return true if acting_user.administrator?  
  only_changed? :name, :description  
end
```

Note that we're assuming there is an `administrator?` method on the user object. Such a method is not built into Hobo, but Hobo's default user generator does add this to your model. We'll discuss this in more detail later on.

A typical destroy permission might be that administrators can delete anything, but non-administrators can only delete the record if they own it:

```
def destroy_permitted?  
  acting_user.administrator? || owner_is?(acting_user)  
end
```

View permission and `never_show`

As you may have noticed when we introduced the permissions above, the `view_permitted` method differs from the other three basic permissions in that it takes a single parameter:

```
def view_permitted?(attribute)  
  ...  
end
```

The method is required to do double duty. If the permission system needs to determine if the `acting_user` is allowed to view this record as a whole, `attribute` will be `nil`. Otherwise `attribute` will be the name of an attribute for which view permission is requested. So when defining this method, remember that `attribute` may be `nil`.

There is also a convenient shorthand for denying view permission for a particular attribute or attributes:

```
class MyModel  
  ...  
  never_show :foo, :baa  
  ...  
end
```

View and edit permission will always be denied for those attributes.

Edit Permission

Edit permission is used by the view layer to determine whether or not to render a particular form field. That means it is not like the other permission methods, in that it's not actually a request to view or change a record. Instead it's more like a preview of update permission.

Asking for edit permission is a bit like asking: *will update permission be granted if a change is made to this attribute?* A common response to that question might be: it depends what you're changing the attribute to. And therein lies the difference between update permission and edit permission. With update permission, we are dealing with a known quantity – we have a set of concrete changes to the object that may or may not be permitted. With edit permission, the value that the attribute will become is not known (because the user hasn't submitted the form yet).

Despite that difference edit permission and update permission are obviously very closely related. Because saving you work is what Hobo is all about, the permission system contains a mechanism for deriving edit permission based on your `update_permitted?` method. For that reason, the `edit_permitted?` method:

```
def edit_permitted?(attribute)
  ...
end
```

This method often does not need to be implemented.

Protected, read-only, and non-viewable attributes

Rails provides a few ways to prevent attributes from being updated during 'mass assignment':

- `attr_protected`
- `attr_accessible`
- `attr_readonly`

(You can look these up in the regular Rails API reference if you're not familiar with them).

Before the `edit_permitted?` method is even called, Hobo checks these declarations. If changes to any attribute is prevented by these declarations, they will automatically be recognized as not editable.

Similarly, if a virtual attribute is read-only in the Ruby sense (it has no setter method), that tells Hobo it is not editable. And finally, fields that are not viewable are implicitly not editable either.

Tip: if a particular attribute can *never* be edited by any user, it's simplest to just declare it as `attr_protected` or `attr_readonly` (read-only attributes can be set on creation, but not

changed later). If the ability to change the attribute either depends on the state of the record, or varies from user to user, `attr_protected` and the rest are not flexible enough – define permission methods instead.

We'll now take a look at how `edit_permitted?` is provided automatically, and then cover the details of defining edit permission yourself.

Deriving edit permission

To figure out edit permission for a particular attribute, based on your definition of `update_permitted?`, Hobo calls your `update_permitted?` method, but with a special trick in place.

If your `update_permitted?` attempts to access the attribute under test, Hobo intercepts that access and says to itself: “Aha! the permission method tried to access the attribute, which means permission to update *depends on the value of that attribute*”. Given that we don't know what value the attribute will have *after the edit*, we had better be conservative. The result is `false` - no you cannot edit that attribute.

If, on the other hand, the permission method returns true without ever accessing that attribute, the conclusion is: update permission is granted regardless of the value the attribute. No matter what change is made to the attribute, update permission will be granted, and therefore edit permission can be granted.

Neat eh? It's not perfect but it sure is useful. Remember you can always define `edit_permitted?` if things don't work out. Also note that if edit permission is incorrect, this does *not* result in a security hole in your application. An edit control may be rendered when it really should not have been, but on submission of the form, the change to the database is policed by `update_permitted?`, not `edit_permitted?`.

In case you're interested, here's how Hobo intercepts those accesses to the attribute under test. A few singleton methods are added to the record (i.e., methods are defined on the record's metaclass). These give special behavior to this one instance. In effect these methods make one of the models attributes 'undefined'. Any access to an undefined attribute raises `Hobo::UndefinedAccessError`, which is caught by the permission system, and edit permission is denied.

Say a test is being made for edit permission on the `name` attribute, the following methods will be added:

- `name` - raises `Hobo::UndefinedAccessError`
- `name_change` - raises `Hobo::UndefinedAccessError`
- `name_was` - returns the actual current value (because this will be the old value after the edit)
- `name_changed?` - returns true

- `changed?` - returns true
- `changed` - returns the list of attributes that have changed, including `name`
- `changes` - raises `Hobo::UndefinedAccessError`

After the edit check those singleton methods are removed again.

Defining edit permission

If the mechanism described above is not adequate for some reason, you can always define edit permission yourself. If the derived edit permission is not correct for just one field, it's possible to define edit permission manually for just that one field, and still have the automatic edit permission for the other fields in your model.

To define edit permission for a single attribute (and keep the automatically derived edit permission for the others), define `foo_edit_permitted?` (where `foo` is the name of your attribute). For example, if the attribute is `name`:

```
def name_edit_permitted?  
  acting_user.administrator?  
end
```

To completely replace the derived edit permission with your own definition, just implement `edit_permitted?` yourself:

```
def edit_permitted?(attribute)  
  ...  
end
```

The `attribute` parameter will either be the name of an attribute, or `nil`. In the case that it is `nil`, Hobo is testing to see if the current user has edit permission “in general” for this record. For example, this would be use to determine whether or not to render an edit link.

Permissions and associations

So far we've focused on policing changes to basic data fields, but Hobo supports multi-model forms, so we also need to place restrictions on associated records. We need to specify permissions regarding:

- Changes to the target of a `belongs_to` association.
- Adding and removing items to a `has_many` association.
- Changes to the fields of any related record

If we think in terms of the underlying database, it's clear that every change ultimately comes down to things that we have already covered - creating, updating and deleting rows. So the permission system is able to covers these cases with a simple rule:

- If you make a change to a record via one of the `user_*` methods, (e.g., `user_create`), and
- as a result of that change, related records are created, updated or destroyed, then
- the `acting_user` is propagated to those records, and
- any permissions defined on those records are enforced.

All we have to do then, is think of everything in terms of the records that are being created, modified or deleted, and it should be clear how which permissions apply. For example:

- Change the target of a `belongs_to` required update permission on the owner record.
- Adding a new record to a `has_many` association requires create permission for that new record.
- Adding and removing items to a `has_many :through` requires create or destroy permission on the join model.

So there really is no special support for associations in the permission system, other than the rule described above for propagating the `acting_user`.

Testing for changes to `belongs_to` associations

As discussed, no special support is needed to police `belongs_to` associations, you can just check for changes to the foreign key. For example:

```
belongs_to :user

def update_permitted?
  acting_user.administrator || !user_id_changed?
end
```


Although that works fine, it feels a bit low level. We'd much rather say `user_changed?`, and in fact we can. For every `belongs_to` association, Hobo adds a `*_changed?` method, e.g. `user_changed?`.

In addition to this, the attribute change helpers – `only_changed?`, `none_changed?`, `any_changed?` and `all_changed?` – all accept `belongs_to` association names along with regular field names.

The Permission API

It is common in Hobo applications, especially small ones, that although you *define* permissions on your models, you never actually call the permissions API yourself. The model controller will use the API to determine if POST and PUT requests are allowed, and the Rapid tags in the view layer will use the permissions API to determine what to render.

When you're digging a bit deeper though, customizing the controllers and the views, you may need to use the permission API yourself. That's what we'll look at in this section.

The standard CRUD operations.

Active Record provides a very simple API for the basic CRUD operations:

- **Create** – `Model.create` or `r = Model.new; ...; r.save`
- **Read** – `Model.find`, then access the attributes on the record
- **Update** – `record.save` and `record.update_attributes`
- **Delete** – `record.destroy`

The Hobo permission system adds “user” versions of these methods. For example, `user_create` is like `create`, but takes the “acting user” as an argument, and performs a permission check before the actual create. The full set of class (model) methods are:

- `Model.user_find(user, ...)`
A regular `find`, followed by `record.user_view(user)`
- `Model.user_new(user, attributes)`
A regular `new`, then `set_creator(user)`, then `record.user_view(user)`. If a block is given, the `yield` is after the `set_creator` and before the `user_view`
- `Model.user_create(user, attributes) (and user_create!)`
As with regular `create`, `attributes` can be an array of hashes, in which case multiple records are created. Equivalent to `user_new` followed by `record.user_save`. The `user_create!` version raises an exception on validation errors.

The instance (record) methods are:

- `record.user_save(user) (and user_save!)`
A regular `save` plus a permission check. If `new_record?` is true, checks for create permission, otherwise for update permission.
- `record.user_update_attributes(user, attributes) (and user_update_attributes!)`
A regular `update_attributes` plus the permission check. If `new_record?` is true, checks for create permission, otherwise for update permission.
- `record.user_view`
Performs a view permission check and raises `PermissionDeniedError` if it fails
- `record.user_destroy`

A regular `destroy` with a permission check first.

Direct permission tests

The methods mentioned in the previous section perform the appropriate permission tests along with some operation. If you want to perform a permission test directly, the following methods are available:

- `record.creatable_by?(user)`
- `record.updatable_by?(user)`
- `record.destroyable_by?(user)`
- `record.viewable_by?(user, attribute=nil)`
- `record.editable_by?(user, attribute=nil)`

There is also:

- `method_callable_by?(user, method_name)`

Which is related to web methods, which we'll cover later on.

You should always call these methods, rather than calling the `..._permitted?` methods directly, as some of them have extra logic in addition to the call to the `..._permitted?` method.

For example, `editable_by?` will check things like `attr_protected` first, and then call `edit_permitted?`

Create, update and destroy hooks

In addition to the methods described in this section, the permission system extends the regular `create`, `update` and `destroy` methods. If `acting_user` is set, each of these will perform a permission check prior to the actual operation. This is illustrated in the very simple implementation of, for example `user save`:

```
def user_save(user)
  with_acting_user(user) { save }
end
```

(`with_acting_user` just sets `acting_user` for the duration of the block, then restores it to it's previous value)

Permission for web methods

In order for a web method to be available to a particular user, a permission method must be defined (one permission method per web method). For example, if the web method is `send_reminder_email`, you would define the permission to call that in:

```
def send_reminder_email_permitted?  
  ...  
end
```

As mentioned previously, you can test a method-call permission directly with:

```
record.method_callable_by?(user, :send_reminder_email)
```

after_user_new - initialize a record using `acting_user`

Often we would like to initialize some aspect of our model based on who the `acting_user` is. A very common example would be to set an “owner” association automatically. Hobo provides the `after_user_new` callback for this purpose:

```
belongs_to :owner, :class_name => "User",  
after_user_new { |r| r.owner = acting_user }
```

Note that `after_user_new` fires on both `user_new` and `user_create`.

The need for an “owner association” is so common that Hobo provides an additional shortcut for it:

```
belongs_to :owner, :class_name => "User", :creator => true
```

Other situations can be more complex, and the `:creator => true` shorthand may not suffice.

For example, an “event” model might need to be associated with the same “group” as the acting user. In this case we go back to the `after_user_new` callback:

```
class Event  
  belongs_to :group,  
  after_user_new { |event| event.group = acting_user.group }  
end
```

OK, but what does all this have to do with permissions? It is quite common that you *need* this information to be in place in order to confirm if permission is granted. For example:

```
def create_permitted?  
  acting_user.group == group  
end
```

This definition says that a user can only create an event in their own group. When we combine the two...

```
after_user_new { |event| event.group = acting_user.group }

def create_permitted?
  acting_user.group == group
end
```

...a neat thing happens. A signed up user *is* allowed to create an event, because the callback ensures that the event is in the right group, but if an attempt is made to change the group to a different one, that would fail.

The edit permission mechanism (described in a previous section) can detect this, so the end result is that (by default) your app will have the “New Event” form, but the form control for choosing the group will be automatically removed. The event will be automatically assigned to the logged in user’s group. I love it when a plan comes together!

Permissions vs. validations

It may have occurred to you that there is some overlap between the permission system and Active Record’s validations. To an extent that’s true: they both provide a way to prevent undesirable changes from making their way into the database. The line between them is fairly clear though:

- Validations are appropriate for “normal mistakes”.

A validation “error” is not really an application error, but a normal occurrence which is reported to the user in a helpful manner.

- Permissions are appropriate for preventing things that *should never happen*.

Your user interface should provide no means by which a “permission denied” error can occur. Permission errors should only come from manually editing the browser’s address bar, or from unsolicited form posts.

In Rails code, it’s not uncommon to see validations used for both of these reasons. For example, the UI may provide radio buttons to chose “Male” or “Female”, and the model might state:

```
validates_inclusion_of :gender, :in => %w(Male Female)
```

In normal usage, no one will ever see the message that gets generated when this validation fails. Effectively it’s being used as a permission. In a Hobo app it might be better to use the permission system for this example, but the declarative `validates_inclusion_of` is quite nice, so if you do use it we’ll turn a blind eye.

The `administrator?` Method

The idea that your user model has a boolean method `administrator?` is bit of a strong assumption. It fits for many applications, but might be totally inappropriate for many others.

Although you've probably seen this method a lot, it's important to clarify that it's not actually part of Hobo. Eh what?

`administrator?` is only a part of Hobo insofar as:

- The user model created by the `hobo_user_model` generator contains a boolean field `administrator`
- The `Guest` model created by the `hobo` generator has a method `administrator?` which just returns `false`.
- The default permission stubs generated by `hobo_model` require `acting_user.administrator?` for `create`, `update` and `destroy` permission.

That's it. `administrator?` is a feature of those three generators, but is not a feature of the permission system itself, or any other part of the Hobo internals. The generated code is just a starting point. Two common ways you might want to change that are:

- Get rid of the `administrator` field in the `User` model, and define a method instead, for example:

```
def administrator?  
  roles.include?(Role.administrator)  
end
```
- Get rid of that field, and of all calls to `administrator?` from your models' permission methods. Those are just stubs that you are expected to replace

At some point we may add an option to the generators so you will only get `administrator?` if you want it.

View helpers

This is the quick version. Five permission related view-helpers are provided:

- `can_create?(object=this)`
- `can_update?(object=this)`
- `can_edit?` – arguments are an object, or a symbol indicating a field (assumes `this` as the object), or both, or no arguments
- `can_delete?(object=this)`
- `can_call?` – arguments are an object and a method name (symbol), or just a method name (assumes `this` as the object)

Chapter 5 - Hobo Controllers and Routing

This chapter of the Hobo Manual describes Hobo's *Model Controller* and automatic routing. In a very simple Hobo app, you hardly need to touch the automatically generated controllers, or even think about routing. As an app gets more interesting though, you'll quickly need to know how to customize things. The down-side of having almost no code at all in the controllers is that there's nothing there to tweak. Don't worry though, Hobo's controllers have been built with Customization in mind. The things you will tweak commonly are extremely easy, and full Customization is not hard at all.

Introduction

Here's a typical controller in a Hobo app. In fact this is unchanged from the code generated by the `hobo_model_controller` generator:

```
class AdvertsController < ActiveRecord::Base
  hobo_model_controller
  auto_actions :all
end
```

The `hobo_model_controller` declaration just does include `Hobo::ModelController`, and gives you a chance to indicate which model this controller looks after. E.g., you can do `hobo_model_controller Advert`

By default the model to use is inferred from the name of the controller.

Selecting the automatic actions

Hobo provides working implementations of the full set of standard REST actions that are familiar from Rails:

- `index`
- `show`
- `new`
- `create`
- `edit`
- `update`
- `destroy`

A controller that declares

```
auto_actions :all
```

Will have all of the above actions.

You can customize this either by listing the actions you want:

```
auto_actions :new, :create, :show
```

Or by listing the actions you *don't* want:

```
auto_actions :all, :except => [ :index, :destroy ]
```

The `:except` option can be set to either a single symbol or an array

There are two more conveniences: `:read_only` and `:write_only`. `:read_only` is a shorthand for `:index` and `:show`, and `:write_only` is a shorthand for `:create`, `:update` and `:destroy`.

Either of these shorthands must be the first argument to `auto_actions`, after which you can still list other actions and the `:except` option:

```
auto_actions :write_only, :show
```

Owner actions

Hobo's model controller can also provide three special actions that take into account the relationships between your records. Specifically, these are the “owner” versions of `new`, `index` and `create`. To understand how these compare to the usual actions, consider a *recipe* model which `belongs_to :author, :class_name => "User"`. The three special actions are:

- An index page that only lists the recipes by a specific author
- A “New Recipe” page specific to that user (i.e. to create a new recipe by that author)
- A create action which is specific to that “New Recipe” page

These are all part of the `RecipesController` and can be added with the `auto_actions_for` declaration, like this:

```
auto_actions_for :author, [ :index, :new, :create ]
```

If you only want one, you can omit the brackets:

```
auto_actions_for :author, :index
```

Action names and routes

The action names and routes for these actions are as follows:

- `index_for_author` is routed as `/users/:author_id/products` for GET requests
- `new_for_author` is routed as `/users/:author_id/products/new` for GET requests
- `create_for_author` is routed as `/users/:author_id/products` for POST requests

It's common for the association name and the class name of the target to be the same (e.g., in an association like `belongs_to :category`). We've deliberately chosen an example where they are different ("author" and "user") in order to show where the two names are used. The association name ("author") is used everywhere except in the `/users` at the beginning of the route.

Instance Variables

As well as setting the default DRYML context, the default actions all make the record, or collection of records, available to the view in an instance variable that follows Rails conventions. E.g. for a 'product' model, the product will be available as `@product` and the collection of products on an index page will be available as `@products`

Owner actions

For owner actions, the owner record is made available as `@<association-name>`. For example, `@author` in our above example.

Automatic Routes

Hobo's model router will automatically create standard RESTful routes for each of your models. The router inspects your controllers: any action that is not defined will not be routed.

At this time it is not possible to customize Hobo's routes beyond turning them on or off (by adding or removing controller actions). This will be addressed in the future. However, like most things in Hobo, it's important to understand that it's just Rails underneath.

There's nothing to stop you defining your own routes in addition to Hobo's. You could even remove Hobo's routes altogether, and define them all yourself. To do that, simply remove the call to `Hobo.add_routes` that Hobo adds to your `routes.rb` file.

Adding extra actions

It's common to want actions beyond the basic REST defaults. In Rails a controller action is simply a public method. That doesn't change in Hobo. You can define public methods and add routes for them just as you would in a regular Rails app. However, you probably want your new actions to be routed automatically, and even implemented automatically, just like the basic actions. For this to happen you have to tell Hobo about them as explained in this section.

Show actions

Suppose we want a normal view and a "detailed" view of our advert. In REST terms we want a new 'show' action called 'detail'. We can add this like this:

```
class AdvertsController < ActiveRecord::Base
  hobo_model_controller
  auto_actions :all
  show_action :detail
end
```

This action will be routed to `/adverts/:id/detail`.

Hobo will provide a default implementation. You can override this simply by defining the method yourself:

```
show_action :detail def detail ... end
```

Or, as a shorthand for the same, give a block to `show_action`:

```
show_action :detail do ... end
```

Index actions

In the same way, we might want an alternative listing (index) of our adverts. Perhaps one that gives a tabular view of the adverts:

```
class AdvertsController < ActiveRecord::Base
  hobo_model_controller
  auto_actions :all
  index_action :table
end
```

This gets routed to `/adverts/table`. As with `show_action`, if you want your own implementation, you can either define the method as normal, or pass a block to `index_action`.

Changing action behavior

Sometimes the implementations Hobo provide aren't what you want. They might be close, or they might be completely out. Not a problem - you can change things as needed.

A cautionary note concerning controller methods

Always start by asking: should this go in the model? It's a very, very, very common mistake to put code in the controller that belongs in the model. Want to send an email in the `create` action?

Don't! Send it from an `after_create` callback in the model. Want to check something about the current user before allowing a `destroy` to proceed? Use Hobo's [Permission System](#).

Typically, valid reasons to add custom controller code are things like:

- Provide a custom flash message
- Change the redirect after a create / update / destroy
- Extract parameters from `params` and pass them to the model (e.g. for searching / filtering)
- Provide special responses for different formats or requested mime-types

A good test is to ask: is this related to http? No? Then it probably shouldn't be in the controller. I tend to think of controllers as a way to publish objects via http, so they shouldn't really be dealing with anything else.

Writing an action from scratch

The simplest way to customize an action is to write it yourself. Say your advert has a boolean field `published` and you only want published adverts to appear on the index page. Using one of Hobo's automatic scopes, you could write:

```
class AdvertsController < ActiveRecord::Base
  hobo_model_controller
  auto_actions :all
  def index
    @adverts = Advert.published.all
  end
end
```

In other words you don't need to do anything different than you would in a normal Rails action. Hobo will look for either `@advert` (for actions which expect an ID) or `@adverts` (for index actions) as the initial context for a DRYML page.

(Note: In the above example, we've asked for the default `index` action and then overwrote it. It might have been neater to say `"auto_actions :all, :except => :index"` but it really doesn't matter.)

Customizing Hobo's implementation

Often you *do* want the automatic action, but you want to customize it in some way. The way you do this varies slightly for the different kinds of actions, but they all follow the same pattern. We'll start with `show` as an example.

The default `show` provided by Hobo is simply:

```
def show
  hobo_show
end
```

All the magic (and in the case of `show` there really isn't much) takes place in `hobo_show`. So immediately we can see that it's easy to add code before or after the default behavior:

```
def show
  @foo = "bar"
  hobo_show
  logger.info "Done show!"
end
```

Note: assigning to instance variables to make data available to the views work exactly as it normally would in Rails.

There is a similar `hobo_*` method for each of the basic actions: `hobo_new`, `hobo_index`, etc.

Switching to the `update` action, you might think you can do:

```
def update
  hobo_update
  redirect_to my_special_place # DON'T DO THIS!
end
```

That will give you an error: actions can only respond by doing a *single* redirect or render, and `hobo_update` has already done a redirect. Read on for the simple solution...

The block

The correct place to perform a redirect is in a block passed to `hobo_update`. All the `hobo_*` actions take a block and yield to the block just before their response. If your block performed a response, Hobo will leave it at that. So:

```
def update
  hobo_update do
    redirect_to my_special_place # better but still problematic
  end
end
```

The problem this time is that we almost certainly don't want to do that redirect if there were validation errors during the update. As with the typical Rails pattern, validation errors are handled by re-rendering the form (along with the error messages). Hobo provides a method `valid?` for these situations:

```
def update
  hobo_update do
    redirect_to my_special_place if valid?
  end
end
```

If the update was valid, the above redirect will happen. If it wasn't, the block won't respond so Hobo's response will kick in and re-render the form. Perfect!

If you want access to the object either in the block or after the call to `hobo_update`, it's available either as `this` or in the conventional Rails instance variable, in this case `@advert`.

Handling different formats

By default, the response block is only called if an HTML response is required. If you want to handle other response types, declare a block with a single argument. The “format” object from Rails’ `respond_to` will be passed. The typical usage would be:

```
def update
  hobo_update do |format|
    format.html { ... }
    format.js   { ... }
  end
end
```

Passing options

Here's another example of tweaking one of the automatic actions. The `hobo_*` methods can all be passed a range of options. Here's a simple example: changing the page size on an index page:

```
def index
  hobo_index :per_page => 10
end
```

That's pretty much all there is to customizing Hobo's automatic actions: define the action as a public method in which you call the appropriate `hobo_*` method, passing it parameters and/or a block.

The remainder of this guide will cover the parameters available to each of the `hobo_*` methods. Note that you can also pass these options directly to the `index_action` and `show_action` declarations, e.g.:

```
index_action :table, :per_page => 10
```

The default actions

In this section we'll go through each of the action implementations that Hobo provides.

`hobo_index`

`hobo_index` takes a “finder” as an optional first argument, and then options. A finder is any object that supports the `find` and / or `paginate` methods, such as an ActiveRecord model class, a `has_many` association, or a scope.

Find options

Any of the standard ActiveRecord find options you pass are forwarded to the `find` method. This is particularly useful with the `:include` option to avoid the dreaded N+1 query problem.

Pagination

Turn pagination on or off by passing `true/false` to the `:paginate` option. If not specified Hobo will guess based on the value of `request.format`.

It's normally on, but won't be for things like XML and CSV. When pagination is on, any other options to `hobo_index` are forwarded to the `paginate` method from `will_paginate`, so you can pass things like `:page` and `:per_page`. If you don't specify `:page` it defaults to `params[:page]` or if that's not given, the first page.

`hobo_show`

Options to `hobo_show` are forwarded to the method `find_instance` which does:

```
model.user_find(current_user, params[:id], options)
```

`user_find` is a method added to your model by Hobo which combines a normal `find` with a check for view permission.

As with `hobo_index`, a typical use would be to pass `:include` to do eager loading.

`hobo_new`

`hobo_new` will either instantiate the model for you using the `user_new` method from Hobo's permission system, or will use the first argument (if you provide one) as the new record.

`hobo_create`

`hobo_create` will instantiate the model (using `user_new`), or take the first argument if you provide one.

The attributes hash for this new record are found either from the option `:attributes` if you passed one, or from the conventional parameter that matches the model name (e.g. `params[:advert]`).

The update to the new record with these attributes is performed using the `user_update_attributes` method, in order to respect the model's permissions.

The response (assuming you didn't respond in the block) will handle:

- redirection if the create was valid (see below for details)
- re-rendering the form if not (or sending textual validation errors back to an AJAX caller)
- performing Hobo's part updates as required for AJAX requests

`hobo_update`

`hobo_update` has the same behavior as `hobo_create` except that the record is found rather than created. You can pass the record as the first argument if you want to find it yourself.

The response is also essentially the same as `hobo_create`, with some extra smarts to support the in-place-editor from Script.aculo.us.

`hobo_destroy`

The record to destroy is found using the `find_instance` method, unless you provide it as the first argument.

The actual destroy is performed with:

```
this.user_destroy(current_user)
```

...which performs a permission check first.

The response is either a redirect or an AJAX part update as appropriate.

Owner actions

For the “owner” versions of the `index`, `new` and `create` actions, Hobo provides:

- `hobo_index_for`
- `hobo_new_for`
- `hobo_create_for`

These are pretty much the same as the regular `hobo_index`, `hobo_new` and `hobo_create` except they take an additional first argument – the name of the association. For example, the default implementation of, say, `index_for_author` would be:


```
def index_for_author
  hobo_index_for :author
end
```

Flash messages

The `hobo_create`, `hobo_update` and `hobo_destroy` actions all set reasonable flash messages in `flash[:notice]`. They do this *before* your block is called so you can simply overwrite this message with your own if need be.

Automatic redirection

The `hobo_create`, `hobo_create_for`, `hobo_update` and `hobo_destroy` actions all perform a redirect on success.

Block Response

If you supply a block to the `hobo_*` action, no redirection is done so that it may be performed by the block:

```
def update
  hobo_update do
    redirect_to my_special_place if valid?
  end
end
```

The `:redirect` parameter

If you supply a block to the `hobo_*` action, you must redirect or render all potential formats. But what if you want to supply a redirect for HTML requests, but let Hobo handle AJAX requests? In this case you can supply the `:redirect` option to `hobo_*`:

```
def update
  hobo_update :redirect => my_special_place
end
```

`:redirect` is only used for valid HTML requests.

The `:redirect:` option may be one of:

- Symbol: redirects to that action using the current controller and model. (Must be a show action).
- Hash or String: [redirect\to](#) is used.
- Array: `object_url` is used.

Automatic redirects

If neither a response block nor `:redirect` are passed to `hobo_*`, the destination of this redirect is determined by calling the `destination_after_submit` method. Here's how it works:

- If the parameter "after_submit" is present, go to that URL (See the `<after-submit>` tag in Rapid for an easy way to provide this parameter), or
- Go to the record's `show` page if there is one, or
- Go to the show page of the object's `owner` if there is one (For example, this might take you to the blog post after editing a comment), or
- Go to the index page for this model if there is one, or
- Give up trying to be clever and go to the home-page (the root URL, or override by implementing `home_page` in ApplicationController)

Autocompleters

Hobo makes it easy to build auto-completing text fields in your user interface; the Rapid tag library provides support for them in the view layer, and the controller provides an easy way to add the action that looks up the completions.

The simplest form for creating an auto-completing field is just a single declaration:

```
class UsersController < ApplicationController
  autocomplete
end
```

Because Hobo allows you to specify which field of a model is the name (using `:name => true` in the model's field declaration block), you don't need to tell `autocomplete` which field to complete on if it is autocompleting the "name" field. To create an autocompleter for a different field, pass the field as a symbol:

```
autocomplete :email_address
```

The `autocomplete` declaration will create an action named according to the field, e.g., `complete_email_address` routed as, in this case, `/users/complete_email_address` for GET requests.

Options

The `autocomplete` behavior can be customized with the following options:

- `:field` – specify a field to complete on. Defaults to the name (first argument) of the autocompleter.
- `:limit` – maximum number of completions. Defaults to 10.
- `:param` – name of the parameter in which to expect the user's input. Defaults to `:query`
- `:query_scope` – a named scope used to do the database query. Change this to control things such as handling of multiple words, case sensitivity, etc. For our example this would be `email_address_contains`. Note that this is one of Hobo's automatic scopes.

Further Customization

The `autocomplete` action follows the same pattern for Customization as the regular actions. That is, the implementation given to you is a simple call to the underlying method that does the actual work, and you can call this underlying method directly. To illustrate, say, on a `UserController` in which you declare `autocomplete :email_address`, the generated method looks like:

```
def complete_email_address
  hobo_completions :email_address, User
end
```

To gain extra control, you can call `hobo_completions` yourself by passing a block to `autocomplete`:

```
autocomplete :email_address do hobo_completions ... end
```

The parameters to `hobo_completions` are:

- Name of the attribute
- A finder, i.e. a model class, association, or a scope.
- Options (the same as described above)

Drag and drop reordering

The controller has the server-side support for drag-and-drop reordering of models that declare `acts_as_list`.

Say, for example, your `Task` model uses `acts_as_list`, then Hobo will add a `reorder` action routed as `/tasks/reorder` that looks like:

```
def reorder
  hobo_reorder
end
```

This action expects an array of IDs in `params[:task_ordering]`, and will reorder the records in the order that the IDs are given.

The action can be removed in the normal ways (e.g., blacklisting):

```
auto_actions :all, :except => :reorder
```

The action will raise a `PermissionDeniedError` if the current user does not have permission to change the ordering.

Permission and “not-found” errors

Any permission errors that happen are handled by the `permission_denied` controller method, which renders the DRYML tag `<permission-denied-page>` or just a text message if that doesn't exist.

Not-found errors are handled in a similar way by the `not_found` method, which tries to render `<not-found-page>`

Both `permission_denied` and `not_found` can be overridden either in an individual controller or site-wide in `ApplicationController`.

Lifecycles

Hobo's model controller has extensive support for lifecycles. This is described in the following chapter.

Chapter 6 - Hobo Lifecycles

This chapter of the Hobo manual describes Hobo's "lifecycle" mechanism. This is an extension that lets you define a lifecycle for any Active Record model. Defining a lifecycle is like a finite state machine – a pattern which turns out to be extremely useful for modeling all sorts of processes that crop up in the world that we're trying to model.

That might make Hobo's lifecycles sound similar to the well known `acts_as_state_machine` plugin, and in a way they are, but with Hobo style. The big win comes from the fact that, like many things in Hobo:

There is support for this feature in all three of the MVC layers

This is the secret to making it very quick and easy to get up and running.

Introduction

In the REST style, which is popular with Rails coders, we view our objects a bit like documents: you can post them to a website, get them again later, make changes to them and delete them. Of course, these objects also have behavior, which we often implement by hooking functionality to the create / update / delete events (like using callbacks such as `after_create` in Active Record).

In a pinch we may have to fall back to the RPC style, which Hobo has support for with the "Web Method" feature.

This works great for many situations, but some objects are *not* best thought of as documents that we create and edit. In particular, applications often contain objects that model some kind of *process*. A good example is *friendship* in a social app. Here's a description of how friendship might work:

- Any user can **invite** friendship with another user
- The other user can **accept** or **reject** (or perhaps **ignore**) the invite.
- The friendship is only **active** once it's been accepted
- An active friendship can be **cancelled** by either user.

Not a "create", "update" or "delete" in sight. Those bold words capture the way we think about the friendship much better. Of course we *could* implement friendship in a RESTful style, but we'd be doing just that – *implementing* it, not *declaring* it.

The life-cycle of the friendship would be hidden in our code, scattered across a bunch of callbacks, permission methods and state variables. Experience has shown this type of code to be tedious to write, *extremely* error prone and fragile when changing.

Hobo lifecycles is a mechanism for declaring the lifecycle of a model in a natural manner.

REST vs. lifecycles is not an either/or choice. Some models will support both styles. A good example is a content management system with some kind of editorial workflow. An application like that might have an Article model, which can be created, updated and deleted like any other REST resource. The Article might also feature a lifecycle that defines how the article goes from newly authored, through one or more stages of review (possibly being rejected at any stage) before finally becoming accepted, and later published.

An Example

Everyone loves an example, so here is one. We'll stick with the friendship idea. If you want to try this out, create a blank app and add a model:

```
>ruby script/generate hobo_model friendship
```

Here's the code for the friendship mode (don't be put off by the `MagicMailer`, that's just a made-up class to illustrate a common use of the callback actions – sending emails):

```
class Friendship < ActiveRecord::Base

  hobo_model

  # The 'sender' of the invite
  belongs_to :invitor, :class_name => "User"

  # The 'recipient' of the invite
  belongs_to :invitee, :class_name => "User"

  lifecycle do

    state :invited, :active, :ignored

    create :invite, :params => [ :invitee ], :become => :invited,
          :available_to => "User",
          :user_becomes => :invitor do
      MagicMailer.send invitee, "#{invitor.name} wants to be friends with you"
    end

    transition :accept, { :invited => :active }, :available_to => :invitee do
      MagicMailer.send invitor, "#{invitee.name} is now your friend :-)"
    end

    transition :reject, { :invited => :destroy }, :available_to => :invitee do
      MagicMailer.send invitor, "#{invitee.name} blew you out :-(
    end

    transition :ignore, { :invited => :ignored }, :available_to => :invitee

    transition :retract, { :invited => :destroy }, :available_to => :invitor do
      MagicMailer.send invitee, "#{invitor.name} reconsidered"
    end

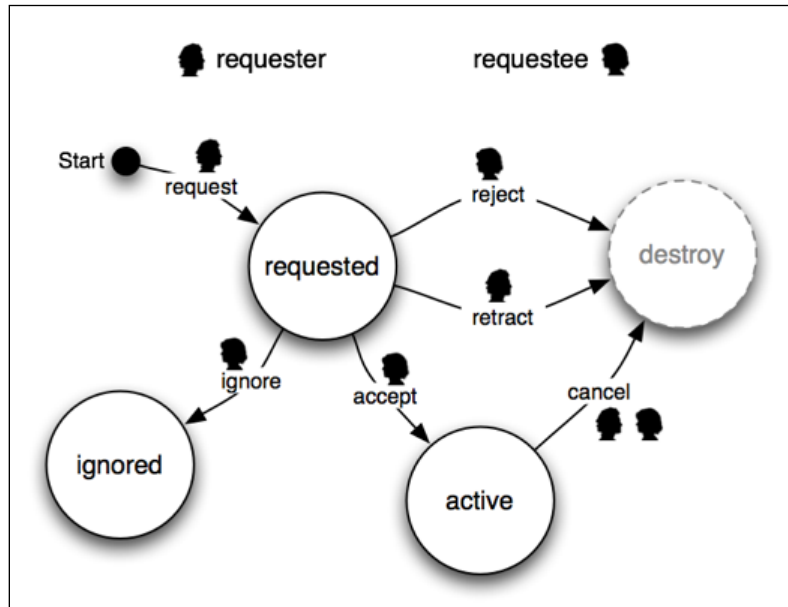
    transition :cancel, { :active => :destroy }, :available_to => [ :invitor, :invitee ] do
      to = acting_user == invitor ? invitee : invitor
      MagicMailer.send to, "#{acting_user.name} cancelled your friendship"
    end

  end

end
```

Figure 57: Defining the Friendship model

Usually, the lifecycle can be represented as a graph, just as we would draw a finite state machine:



Let's work through what we did there.

Because `Friendship` has a lifecycle declared, a class is created that captures the lifecycle. The class is `Friendship::Lifecycle`. Each instance of `Friendship` will have an instance of this class associated with it, available as `my_friendship.lifecycle`.

The `Friendship` model will also have a field called `state` declared. The migration generator will create a database column for `state`.

The lifecycle has three states:

```
state :invited, :active, :ignored
```

There is one 'creator' – this is a starting point for the lifecycle:

```

create :invite, :params => [ :invitee ], :become => :invited,
      :available_to => "User",
      :user_becomes => :invitor do
  MagicMailer.send invitee, "#{invitor.name} wants to be friends with you"
end

```

This declaration specifies that:

- The name of the creator is `invite`. It will be available as a method `Friendship::Lifecycle.invite(user, attributes)`. Calling the method will instantiate the record, setting attributes from the hash that is passed in.

- The `:params` option specifies which attributes can be set by this create step:

```
:params => [ :invitee ]
```

(Any other key in the `attributes` hash passed to `invite` will be ignored.)

- The lifecycle state after this create step will be `invited`:

```
:become => :invited,
```

- To have access to this create step, the acting user must be an instance of `User` (i.e. not a guest):

```
:available_to => "User"
```

- After the create step, the `invitor` association of the `Friendship` will be set to the acting user:

```
:user_becomes => :invitor
```

- After the create step has completed (and the database updated), the block within `do...end` is executed:

```
      :user_becomes => :invitor do
        MagicMailer.send invitee, "#{invitor.name} wants to be friends with you"
      end
```

There are five transitions declared:

- `accept`
- `reject`
- `ignore`
- `retract`
- `cancel`

These become methods on the lifecycle object (not the lifecycle class), For example:

```
my_friendship.lifecycle.accept!(user, attributes)
```

Calling that method will:

- Check if the transition is allowed.

- If it is, update the record with the passed in attributes. The attributes that can change are declared in a `:params` option, as we saw with the creator. None of the friendship transitions declare any `:params`, so no attributes will change, and
- change the `state` field to the new state, then
- save the record, as long as validations pass.

Each transition declares:

- Which states it goes from and to, e.g., `accept` goes from `invited` to `active`:

```
transition :accept, { :invited => :active }
```

Some of the transitions are to a pseudo state: `:destroy`. To move to this state is to destroy the record.

- Who has access to it:

```
:available_to => :invitor  
:available_to => :invitee
```

In the create step the `:available_to` option was set to a class name, here it is set to a method (a `belongs_to` association).

To be allowed, *the acting user must be the same user returned by this method*. There are a variety ways that `:available_to` can be used, which will be discussed in detail later.

- A callback (the block). This is called after the transition completes. Notice that in the block for the `cancel` transition we're accessing `acting_user`, which is a reference to the user performing the transition.

Hopefully that worked example has clarified what lifecycles are all about. We'll move on and look at the details now.

Key concepts

Before getting into the API we'll recap some of the key concepts very briefly.

As mentioned in the introduction, the lifecycle is essentially a finite state machine. It consists of:

- One or more *states*. Each has a name, and the current state is stored in a simple string field in the record. If you like to think of a finite state machine as a graph, these are the nodes.
- Zero or more *creators*. Each has a name, and they define actions that can start the lifecycle, setting the state to be some start-state.

- Zero or more *transitions*. Each has a name, and they define actions that can change the state. Again, thinking in terms of a graph, these are the arcs between the nodes.

The creators and the transitions are together known as the steps of the lifecycle.

There are a variety of ways to limit which users are allowed to perform which steps, and there are ways to attach custom actions (e.g., send an email) both to steps and to states.

Defining a lifecycle

Any Hobo model can be given a lifecycle like this:

```
class Friendship < ActiveRecord::Base

  hobo_model

  lifecycle do
    ... define lifecycle steps and states ...
  end

end
```

Any model that has such a declaration will gain the following features:

- The lifecycle definition becomes a class called `Lifecycle` which is nested inside the model class (e.g. `Friendship::Lifecycle`) and is a subclass of `Hobo::Lifecycles::Lifecycle`. The class has methods for each of the creators.
- Every instance of the model will have an instance of this class available from the `#lifecycle` method. The instance has methods for each of the transitions:

```
my_friendship.lifecycle.class # Friendship::Lifecycle
my_friendship.lifecycle.reject!(user)
```

The `lifecycle` declaration can take three options:

- `:state_field` - the name of the database field (a string field) to store the current state in. Default `'state'`
- `:key_timestamp_field` - the name of the database field (a datetime field) to store a timestamp for transitions that require a key (discussed later). Set to `false` if you don't want this field. Default `'key_timestamp'`.
- `:key_timeout` - keys will expire after this amount of time. Default `999.years`.

Note that both of these fields are declared `never_show` and `attr_protected`.

Within the `lifecycle do ... end` a simple DSL is in effect. Using this we can add states and steps to the lifecycle.

Defining states

To declare states:

```
lifecycle do
  state :my_state, :my_other_state
end
```

You can call `state` many times, or pass several state names to the same call. Each state can have an action associated with it:

```
state :active do
  MagicMailer.send [invitee, invitor], "Congratulations, you are now
  friends"
end
```

You can provide the `:default => true` option to have the database default for the state field be this state:

```
state :invited, :default => true
```

This will take effect the next time you generate and apply a hobo migration.

Defining creators

A creator is the starting point for a lifecycle. They provide a way for the record to be created (in addition to the regular `new` and `create` methods). Each creator becomes a method on the lifecycle class. The definition looks like:

```
create name, options do
  ...
end
```

The name is a symbol. It should be a valid ruby name that does not conflict with the class methods already present on the `Hobo::Lifecycles::Lifecycle` class.

The options are:

- `:params` - an array of attribute names that are parameters of this create step. These attributes can be set when the creator runs.
- `:become` - the state to enter after running this creator. This does not have to be static but can depend on runtime state. Provide one of:
 - A symbol – the name of the state
 - A proc – if the proc takes one argument it is called with the record, if it takes none it is `instance_eval`'d on the record. Should return the name of the state

- A string – evaluated as a Ruby expression with in the context of the record
- `:if` and `:unless` – a precondition on the creator. Pass either:
 - A symbol – the name of a method to be called on the record
 - A string – a Ruby expression, evaluated in the context of the record
 - A proc – if the proc takes one argument it is called with the record, if it takes none it is `instance_eval`'d on the record.

Note that the precondition is evaluated *before* any changes are made to the record using the parameters to the lifecycle step.

- `:new_key` – generate a new lifecycle key for this record by setting the `key_timestamp` field to be the current time.
- `:user_becomes` – the name of an attribute (typically a `belongs_to` relationship) that will set to the `acting_user`.
- `:available_to` – Specifies who is allowed access to the creator. This check is in addition to the precondition (`:if` or `:unless`). There are a variety of ways to provide the `:available_to` option, discussed later on.

The block given to `create` provides a callback which will be called after the record has been created. You can give a block with a single argument, in which case it will be passed the record, or with no arguments in which case it will be `instance_eval`'d on the record.

Defining transitions

A transition is an arc in the graph of the finite state machine – an operation that takes the lifecycle from one state to another (or, potentially, back to the same state.). Each transition becomes a method on the lifecycle object (with `!` appended). The definition looks like:

```
transition name, { from => to }, options do ... end
```

(The name is a symbol. It should be a valid Ruby name

The second argument is a hash with a single item:

```
{ from => to }
```

(We chose this syntax for the API just because the `=>` is quite nice to indicate a transition)

This transition can only be fired in the state or states given as `from`, which can be either a symbol or an array of symbols. On completion of this transition, the record will be in the state give as `to` which can be one of:

- A symbol – the name of the state
- A proc – if the proc takes one argument it is called with the record, if it takes none it is `instance_eval`'d on the record. Should return the name of the state.
- A string – evaluated as a Ruby expression with in the context of the record.

The options are:

- `:params` - an array of attribute names that are parameters of this transition. These attributes can be set when the transition runs.
- `:if` and `:unless` – a precondition on the transition. Pass either:
 - A symbol – the name of a method to be called on the record
 - A string – a Ruby expression, evaluated in the context of the record
 - A proc – if the proc takes one argument it is called with the record, if it takes none it is `instance_eval`'d on the record.
- `:new_key` – generate a new lifecycle key for this record by setting the `key_timestamp` field to be the current time.
- `:user_becomes` – the name of an attribute (typically a `belongs_to` relationship) that will set to the `acting_user`.
- `:available_to` – Specifies who is allowed access to the transition. This check is in addition to the precondition (`:if` or `:unless`). There are a variety of ways to provide the `:available_to` option, discussed later on.

The block given to `transition` provides a callback which will be called after the record has been updated. You can give a block with a single argument, in which case it will be passed the record, or with no arguments in which case it will be `instance_eval`'d on the record.

Repeated transition names

It is not required that a transition name is distinct from all the others. For example, a process may have many stages (states) and there may be an option to abort the process at any stage. It is possible to define several transitions called `:abort`, each starting from a different start state. You could achieve a similar effect by listing all the start states in a single transition, but by defining separate transitions, each one could, for example, be given a different action (block).

The `:available_to` option

Both create and transition steps can be made accessible to certain users with the `:available_to` option. If this option is given, the step is considered ‘publishable’, and there will be automatic support for the step in both the controller and view layers.

The rules for the `:available_to` option are as follows. Firstly, it can be one of these special values:

- `:all` – anyone, including guest users, can trigger the step
- `:key_holder` – (transitions only) anyone can trigger the transition, provided `record.lifecycle.provided_key` is set to the correct key. Discussed in detail later.

If `:available_to` is not one of those, it is an indication of some code to run (just like the `:if` option for example):

- A symbol – the name of a method to call
- A string – a Ruby expression which is evaluated in the context of the record
- A proc – if the proc takes one argument it is called with the record, if it takes none it is `instance_eval`’d on the record

The value returned is then used to determine if the `acting_user` has access or not. The value is expected to be:

- A class – access is granted if the `acting_user` is a `kind_of?` that class.
- A collection – if the value responds to `:include?`, access is granted if `include?(acting_user)` is true.
- A record – if the value is neither a class or a collection, access is granted if the value *is* the `acting_user`

Some examples:

Say a model has an owner:

```
belongs_to :owner, :class_name => "User"
```

You can just give the name of the relationship (since it is also a method) to restrict the transition to that user:

```
:available_to => :owner
```

Or a model might have a list of collaborators associated with it:

```
has_many :collaborators, :class_name => "User"
```

Again it's easy to make the lifecycle step available to them only (since the `has_many` does respond to `:include?`):

```
:available_to => :collaborators
```

If you were building more sophisticated role based permissions, you could make sure your role object responds to `:include?` and then say, for example:

```
:available_to => "Roles.editor"
```

Validations

Validations have been extended so you can give the name of a lifecycle step to the `:on` option.

```
validates_presence_of :notes, :on => :submit
```

There is now support for:

```
record.lifecycle.valid_for_foo?
```

where `foo` is a lifecycle transition.

Controller actions and routes

As well as providing the lifecycle mechanism in the model, Hobo also supports the lifecycle in the controller layer, and provides an automatic user interface in the view layer. All of this can be fully customized of course. In this section we'll look at the controller layer features, including the routes that get generated.

Lifecycle steps that include the `:available_to` option are considered *publishable*. It is these that Hobo generates controller actions for. Any step that does not have the `:available_to` option can be thought of as 'internal'.

Of course you can call those create steps and transitions from your own code, but Hobo will never do that for you.

auto_actions

The lifecycle actions are added to your controller by the `auto_actions` directive. To get them you need to say one of:

- `auto_actions :all`
- `auto_actions :lifecycle` – adds *only* the lifecycle actions
- `auto_actions :accept, :do_accept` (for example) – as always, you can list the method names explicitly (the method names that relate to lifecycle actions are given below)

You can also remove lifecycle actions with:

- `auto_actions ... :except => :lifecycle` – don't create any lifecycle actions or routes
- `auto_actions ... :except => [:do_accept, ...]` – don't create the listed lifecycle actions or routes

Create steps

For each create step that is publishable, the model controller adds two actions. Going back to the friendship example, two actions will be created for the `invite` step. Both of these actions will pass the `current_user` to the lifecycle, so access restrictions (the `:available_to` option) will be enforced, as will any preconditions (`:if` and `:unless`).

The “create page” action

`FriendshipsController#invite` will be routed as `/friendships/invite` for GET requests. This action is intended to render a form for the create step. An object that provides metadata about the create step will be available in `@creator` (an instance of `Hobo::Lifecycles::Creator`).

If you want to implement this action yourself, you can do so using the `creator_page_action` method:

```
def invite
  creator_page_action :invite
end
```

Following the pattern of all the action methods, you can pass a block in which you can customize the response by setting a flash message, rendering or redirecting. `do_creator_action` also takes a single option:

- `:redirect` – change where to redirect to on a successful submission. Pass a symbol to redirect to that action (show actions only) or an array of arguments which are passed to `object_url`. Passing a String or a Hash will pass your arguments straight to `redirect_to`.

The ‘do create’ action

`FriendshipsController#do_invite` will be routed as `/friendships/invite` for POST requests.

This action is where the form should POST to. It will run the create step, passing in parameters from the form. As with normal form submissions (i.e. create and update actions), the result will be an HTTP redirect, or the form will be re-rendered in the case of validation failures.

Again you can implement this action yourself:

```
def do_invite
  do_creator_action :invite
end
```

You can give a block to customize the response, or pass the redirect option:

- `:redirect` – change where to redirect to on a successful submission. Pass a symbol to redirect to that action (show actions only) or an array of arguments that are passed to `object_url`. Passing a String or a Hash will pass your arguments straight to `redirect_to`.

Transitions

As with create steps, for each publishable transition there are two actions. For both of these actions, if `params[:key]` is present, it will be set as the `provided_key` on the lifecycle, so transitions that are `:available_to => :key_holder` will work automatically.

We'll take the friendship `accept` transition as an example.

The transition page

`FriendshipsController#accept` will be routed as `/friendships/:id/accept` for GET requests.

This action is intended to render a form for the transition. An object that provides metadata about the transition will be available in `@transition` (an instance of `Hobo::Lifecycles::Transition`).

You can implement this action yourself using the `transition_page_action` method

```
def accept
  transition_page_action :accept
end
```

As usual, you can customize the response by passing a block. And you can pass the following option:

- `:key` – the key to set as the provided key, for transitions that are:
`:available_to => :key_holder.`
- Defaults to `params[:key]`

The 'do transition' action

`FriendshipsController#do_accept` will be routed as `/friendships/:id/accept` for POST requests.

This action is where the form should POST to. It will run the transition, passing in parameters from the form. As with normal form submissions (i.e., create and update actions), the result will be an HTTP redirect, or the form will be re-rendered in the case of validation failures.

You can implement this action yourself using the `do_transition_action` method:

```
def do_accept
  do_transition_action :accept
end
```

As usual, you can customize the response by passing a block. And you can pass the following options:

- `:redirect` – change where to redirect to on a successful submission. Pass a symbol to redirect to that action (show actions only) or an array of arguments which are passed to `object_url`.
- `:key` – the key to set as the provided key, for transitions that are `:available_to => :key_holder`. Defaults to `params[:key]`

Keys and secure links

Hobo's lifecycles also provide support for the "secure link" pattern. By "secure" we mean that on one other than the holder of the link can access the page or feature in question. This is achieved by including some kind of cryptographic key in the URL, which is typically sent in an email address. The two very common examples are:

- **Password reset** – following the link gives the ability to set a new password for a specific account. By using a secure link and emailing it to the account holders email address, only a person with access to that email account can chose the new password.
- **Email activation** – by following the link, the user has effectively proved that they have access to that email account. Many sites use this technique to verify that the email address you have given is one that you do in fact have access to.

In fact the idea of a secure link is more general than that. It can be applied in any situation where you want a particular person to participate in a process, but that person does not have an account on the site.

For example, in a CMS workflow application, you might want to email a particular person to ask them to verify that the content of an article is technically correct. Perhaps this is a one-off request so you don't want to trouble them with signing up. Your app could provide a page with "approve"/"reject" buttons, and access to that page could be protected using the secure link pattern. In this way, the person you email the secure link to, and no one else, would be able to accept or reject the article.

Hobo's lifecycles provide support for the secure-link pattern with the following:

- A field added to the database called (by default) "key_timestamp". This is a date-time field, and is used to generate a key as follows:

```
Digest::SHA1.hexdigest("#{id_of_record}-#{current_state}-#{key_timestamp}")
```

- Both create and transition steps can be given the option `:new_key => true`. This causes the `key_timestamp` to be updated to `Time.now`.
- The `:available_to => :key_holder` option (transitions only). Setting this means the transition is only allowed if the correct key has been provided, like this:

```
record.lifecycle.provided_key = the_key
```

Hobo's "model controller" also has (very simple) support for the secure-link pattern. Prior to rendering the form for a transition, or accepting the form submission of a transition, it does (by default):


```
record.lifecycle.provided_key = params[:key]
```

Implementing a lifecycle with a secure-link

Stringing this all together, we would typically implement the secure-link pattern as follows.

We're assuming some knowledge of Rails mailers here, so you may need to read up on those.

- Create a mailer (`script/generate mailer`) which will be used to send the secure link.
- In your lifecycle definition, two steps will work together:
 - A create or transition will initiate the process, by generating a new key, emailing the link, and putting the lifecycle in the correct state.
 - A transition from this state will be declared as `:available_to => :key_holder`, and will perform the protected action.
- Add `:new_key => true` to the create or transition step that initiates the process.
- On this same step, add a callback that uses the mailer to send the key to the appropriate user. The key is available as `lifecycle.key`. For example, the default Hobo user model has:

```
Transition :request_password_reset, { :active => :active }, :new_key => true do
  UserMailer.deliver_forgot_password(self, lifecycle.key)
end
```

- Add `:available_to => :key_holder` to the subsequent transition – the one you want to make available only to recipients of the email.
- The mailer should include a link in the email, and the key should be part of this link as a query parameter. Hobo creates a named route for each transition page, so there will be a URL helper available. For example, if the transition is on `User` and is called `reset_password`, the link in your mailer template should look something like:

```
<%= user_reset_password_url :host => @host, :id => @user, :key => @key %>
```

Testing for the active step.

In some rare cases your code might need to know if a lifecycle step is currently in progress or not (e.g. in a callback or a validation). For this you can access either:

```
record.lifecycle.submit_in_progress.active_step.name
```

Or, if you are interested in a particular step, it's easier to call:

```
record.lifecycle.submit_in_progress?
```

Where `submit` can be any lifecycle step.

Chapter 7 - Hobo View Hints

Introduction

One of the main attractions of Hobo is its ability to give you a pretty decent starting point for your app's UI, entirely automatically based on information extracted from your models and controllers. The more information available to Hobo, the better job it can do, but some such information doesn't properly belong in either the model or the controller. For example, we might want to declare that a particular field should have a different name in the UI than in the model layer. View Hints are the home for these kinds of declarations.

View hints are added to a Hobo application by defining classes, one for each model, that extend `Hobo::ViewHints`. Here's an example - `app/viewhints/answer_hints.rb` from the Hobo Cookbook app:

```
class AnswerHints < Hobo::ViewHints
  field_names :body => "", :recipe => "See recipe"
  field_help :recipe => "Enter keywords from the name of a recipe"
end
```

As you can see, the view-hint class contains some simple declarations that pertain to a single model - the `Answer` model in this case. That's really all there is to a view-hints class. If you think of the class as little more than a YAML file with some configuration information in it, you won't be far wrong. In fact we could have used YAML files for view-hints, but using Ruby instead makes things more powerful for the metaprogrammers out there who want to explore new territory. In the normal course of events, these classes will not contain anything other than the declarations described in this chapter.

In that example we made three declarations about the user interface that we desire:

- The `body` field does not need a label (i.e. it's name is blank).
- The `recipe` field should be labeled "See recipe"
- The `recipe` field should be displayed with the help text "Enter keywords from the name of a recipe"

What do these declarations do? By themselves, nothing. They are just information, metadata if you like, that we have provided about a model. The information can be retrieved using the view-hints API, for example (using the Rails console).

```
>> Answer.view_hints.field_name :recipe => "See recipe"
>> Answer.view_hints.field_help[:recipe] => "Enter keywords from the name
of a recipe"
```

This API is used internally in Hobo, for example in the Rapid tag library, to create a user interface according to your declarations. That's really all there is to it.

At present view-hints are fairly new to Hobo. They will be put to a lot greater use as Hobo develops.

It's important to note that the view-hints mechanism is entirely optional, and may not be appropriate for all applications (especially larger applications). Everything you can do with view-hints can be done with much more flexibility by defining DRYML tags and page templates. What view-hints give you, is a way to achieve common UI Customizations very quickly and easily.

Defining hints

As mentioned, the hints are defined in `ViewHints` classes. There is one per model, and they live in `app/viewhints`. The `hobo_model_generator` will create blank view-hints classes as a starting point.

At present, there are only four kinds of hints you can give about your models:

- The model name – in case you want this to differ from the actual class name
- Field names – in case you want any of these to differ from the database column names
- Field help – some simple explanatory text for each field in a model
- Child relationships – allows you to arrange your models in a hierarchy appropriate for the user interface.

Model name

To declare a custom model name¹:

```
class BlogPostHints < Hobo::ViewHints
  model_name "Post"
end
```

NOTE: At the time of writing, support for the `model_name` declaration in Hobo Rapid is partial. The underlying class name may still be used in places.

Field names

To declare one or more custom field names:

```
class UserHints < Hobo::ViewHints
```

¹ NOTE: At the time of writing, support for the `model_name` declaration in Hobo Rapid is partial. The underlying class name may still be used in places.

```
    field_names :username => "Name", :details => "Profile"  
end
```

If you give an empty string as the name, the Rapid form generators will arrange the form appropriately, with no label for that field.

Field help

To declare help text for one or more fields:

```
class AnswerHints < Hobo::ViewHints  
  field_help :recipe => "Enter keywords for the recipe",  
             :subject => "Provide a ..."  
end
```

Rapid will include the help text next to each field in the forms that it generates.

Child relationships

Many web applications arrange the information they present in a hierarchy. By declaring a hierarchy using the `children` declaration, Hobo can give you a much better default user interface.

At present, the `children` declaration only influences Rapid's show-page – it governs the display of collections of `<card>` tags embedded in the show-page. If you declare a single child collection, e.g.:

```
class UserHints < Hobo::ViewHints
  children :recipes
end
```

..the collection of the user's recipes will be added to the main content of `users/show`.

You can declare additional child relationships. The order is significant, with the first in the list being the “primary collection”. For example:

```
class UserHints < Hobo::ViewHints
  children :recipes, :questions, :answers
end
```

With this declaration, the user's show-page will be given an aside section (sidebar), in which cards for the `questions` and `answers` collections are displayed.

Inline Booleans

By default, Rapid will display boolean fields as part of the header if they are true (so an `:administrator` field will turn into the text 'Administrator' just under the main heading on the show page).

The `inline_booleans` view hint can alter this behavior for some or all of the model's boolean fields. Fields specified as inline booleans will be rendered as part of the regular field list.

```
class UserHints < Hobo::ViewHints
  inline_booleans :administrator, :moderator
end
```

As a shortcut...

```
class UserHints < Hobo::ViewHints
  inline_booleans true
end
```

...will apply the option to all boolean fields in the model.

The API

The view-hints API is used internally by Hobo Rapid. You may not ever need to use it yourself.

For completeness it is documented here.

The view-hints for any model can be access using the `view_hints` method:

```
MyModel.view_hints
```

That will return the view-hints class from which the hints can be accessed. Each of the declaration methods can be called without arguments to retrieve the declared values. e.g.

```
>> BlogPost.view_hints.model_name => "Post"
```

Helpers

The following view helpers are defined to simplify access to view-hints information during rendering:

- `this_field_name` – returns the view-hints modified name of the field currently referenced by DRYML's `this_field`. That is, the field of the current context
- `this_field_help` – returns the help text associated with the field currently in context.

Chapter 8 - Hobo Scopes

Hobo scopes are an extension of the *named scope* and *dynamic finder* functionality introduced in Rails 2.1, 2.2 and 2.3.

Most of these scopes work by calling `named_scope` the first time they are invoked. They should work at the same speed as a named scope on subsequent invocations².

However, this does substantially slow down `method_missing` on your model's class. If `ActiveRecord::Base.method_missing` is used often, you may wish to disable this module.

Simple Scopes

```
_is  
_is_not  
_contains  
_does_not_contain  
_starts  
_does_not_start  
_ends  
_does_not_end
```

Boolean Scopes

```
not_
```

Date Scopes

```
_before  
_after  
_between
```

Lifecycle Scopes

Key Scopes

Static Scopes

```
by_most_recent  
recent  
limit  
order
```

² However, this does substantially slow down `method_missing` on your model's class. If `ActiveRecord::Base.method_missing` is used often, you may wish to disable this module.


```
include
search
```

Association Scopes

```
with_
without_
_is
_is_not
```

Scoping Associations

Chaining

Let's set up a few models for our testing:

```
class Person < ActiveRecord::Base
  hobo_model
  fields do
    name :string
    born_at :date
    code :integer
    male :Boolean
    timestamps
  end
  lifecycle(:key_timestamp_field => false) do
    state :inactive, :active
  end
  has_many :friendships
  has_many :friends, :through => :friendships
end
```

```
class Friendship < ActiveRecord::Base
  hobo_model
  belongs_to :person
  belongs_to :friend, :class_name => "Person"
end
```

Generate a migration and run it:

```
>> ActiveRecord::Migration.class_eval(HoboFields::MigrationGenerator.run[0])
>> Person.columns.*.name => ["id", "name", "born_at", "code", "male",
"created_at", "updated_at", "state"]
```

And create a couple of fixtures:

```
>> bryan = Person.new(:name => "Bryan", :code => 17, :born_at =>
Date.new(1973,4,8), :male => true)
```

```
>> bryan.state = "active" >> bryan.save!  
>> bethany = Person.new(:name => "Bethany", :code => 42, :born_at =>  
  date.new(1975,5,13), :male => false)  
>> bethany.state = "inactive" >> bethany.save!  
>> Friendship.new(:person => Bryan, :friend => bethany).save!
```

Hack the `created_at` column to get predictable sorting:

```
>> bethany.created_at = Date.new(2000)  
>> bethany.save!
```

We're ready to get going.

Simple Scopes

`_is`

Most Hobo scopes work by appending an appropriate query string to the field name. In this case, the hobo scope function name is the name of your database column, followed by `_is`. It returns an Array of models.

It works the same as a dynamic finder:

```
>> Person.find_all_by_name("Bryan").*.name => ["Bryan"]  
>> Person.name_is("Bryan").*.name => ["Bryan"]  
>> Person.code_is(17).*.name => ["Bryan"]  
>> Person.code_is(99).length => 0
```

`_is_not`

But the Hobo scope form allows us to supply several variations:

```
>> Person.name_is_not("Bryan").*.name => ["Bethany"]
```

`_contains`

```
>> Person.name_contains("y").*.name => ["Bryan", "Bethany"]
```

`_does_not_contain`

```
>> Person.name_does_not_contain("B").*.name => []
```

`_starts`

```
>> Person.name_starts("B").*.name => ["Bryan", "Bethany"]
```

`_does_not_start`

```
>> Person.name_does_not_start("B").length => 0
```

`_ends`

```
>> Person.name_ends("y").*.name => ["Bethany"]
```

`_does_not_end`

```
>> Person.name_does_not_end("y").*.name => ["Bryan"]
```

Boolean scopes

If you use the name of the column by itself, the column is of type boolean, and no function is already defined on the model class with the name, Hobo scopes adds a dynamic finder to return all records with the boolean column set to `true`:

```
>> Person.male.*.name => ["Bryan"]
```

`not_`

You can also search for boolean records that are not `true`. This includes all records that are set to `false` or `NULL`.

```
>> Person.not_male.*.name => ["Bethany"]
```

Date scopes

Date scopes work only with columns that have a name ending in `"at"`. The `"at"` is omitted when using these finders.

`_before`

```
>> Person.born_before(Date.new(1974)).*.name => ["Bryan"]
```

`_after`

```
>> Person.born_after(Date.new(1974)).*.name => ["Bethany"]
```

`_between`

```
>> Person.born_between(Date.new(1974), Date.today).*.name =>
["Bethany"]
```

Lifecycle scopes

If you have a [lifecycle](#) defined, each state name can be used as a dynamic finder.

```
>> Person.active.*.name => ["Bryan"]
```

Key scopes

This isn't very useful:

```
>> Person.is(Bryan).*.name => ["Bryan"]
```

But this is:

```
>> Person.is_not(Bryan).*.name => ["Bethany"]
```

Static scopes

These scopes do not contain the column name.

`by_most_recent`

Sorting on the `created_at` column:

```
>> Person.by_most_recent.*.name => ["Bryan", "Bethany"]
```

`recent`

Gives the N most recent items:

```
>> Person.recent(1).*.name => ["Bryan"]
```

`limit`

```
>> Person.limit(1).*.name => ["Bryan"]
```

order_by

```
>> Person.order_by(:code).*.name => ["Bryan", "Bethany"]
```

include

Adding the include function to your query chain has the same effect as the `:include` option to the `find` method.

```
>> Person.include(:friends).*.name => ["Bryan", "Bethany"]
```

search

Search for text in the specified column(s).

```
>> Person.search("B", :name).*.name => ["Bryan", "Bethany"]
```

Association Scopes

with_

Find the records that contain the specified record in an association

```
>> Person.with_friendship(Friendship.first).*.name => ["Bryan"]
>> Person.with_friend(Bethany).*.name => ["Bryan"]
```

You can also specify multiple records with the plural form

```
>> Person.with_friends(Bethany, nil).*.name => ["Bryan"]
```

without_

```
>> Person.without_friend(Bethany).*.name => ["Bethany"]
>> Person.without_friends(Bethany, nil).*.name => ["Bethany"]
```

_is

You can use `_is` on a `:has_one` or a `:belongs_to` relationship:

```
>> Friendship.person_is(Bryan).*.friend.*.name => ["Bethany"]
```

`_is_not`

```
>> Friendship.person_is_not(Bryan) => []
```

Scoping Associations

When defining an association, you can add a scope:

```
>> class Person
  has_many :active_friends, :class_name => "Person", :through =>
:friendships, :source => :friend, :scope => :active
  has_many :inactive_friends, :class_name => "Person", :through =>
:friendships, :source => :friend, :scope => :inactive
end
>> bryan.inactive_friends.*.name => ["Bethany"]
>> bryan.active_friends.*.name => []
```

Or several scopes:

```
>>
class Person
  has_many :inactive_female_friends, :class_name => "Person",
:through => :friendships, :source => :friend,
:scope => [:inactive, :not_male]
  has_many :active_female_friends, :class_name => "Person",
:through => :friendships, :source => :friend, :scope =>
[:active, :not_male]
  has_many :inactive_male_friends, :class_name => "Person",
:through => :friendships, :source => :friend,
:scope => [:inactive, :male]
end
>> bryan.inactive_female_friends.*.name => ["Bethany"]
>> bryan.active_female_friends.*.name => []
>> bryan.inactive_male_friends.*.name => []
```

You can parameterize the scopes:

```
>> class Person
  has_many :y_friends, :class_name => "Person", :through =>
:friendships,
:source => :friend, :scope => { :name_contains => 'y' }
  has_many :z_friends, :class_name => "Person", :through =>
:friendships, :source => :friend, :scope => { :name_contains => 'z' }
end
>> bryan.y_friends.*.name => ["Bethany"]
>> bryan.z_friends.*.name => []
```

Chaining

Like named scopes, Hobo scopes can be chained:

```
>> bryan.inactive_friends.inactive.*.name => ["Bethany"]
```

Chapter 9 – The Hobo DRYML Guide

What is DRYML?

DRYML is a template language for Ruby on Rails that you can use in place of Rails' built-in ERB templates. It is part of the larger Hobo project, but will eventually be made available as a separate plugin.

DRYML was created in response to the observation that the vast majority of Rails development time seems to be spent in the view-layer. Rails' models are beautifully declarative, the controllers can be made so pretty easily (witness the many and various “result controller” plugins), but the views, ah the views...

Given that so much of the user interaction we encounter on the web is so similar from one website to another, surely we don't have to code all this stuff up from low-level primitives over and over again? Please, no!

Of course what we want is a nice library of ready-to-go user interface components, or widgets, which can be quickly added to our project, and easily tailored to the specifics of our application.

If you've been at this game for a while you're probably frowning about now. Re-use is a very, very thorny problem. It's one of those things that sounds straight-forward and obvious in principle, but turns out to be horribly difficult in practice. When you come to re-use something, you very often find that your new needs differ from the original ones in a way that wasn't foreseen or catered for in the design of the component. The more complex the component, the more likely it is that bending the thing to your needs will be harder than starting again from scratch.

So the challenge is not in being able to re-use code, it is:

Being able to re-use code in ways that were not foreseen.

The reason we created DRYML was to see if this kind of flexibility could be built into the language itself. DRYML is a tag-based language that makes it trivially easy to give the defined tags a great deal of flexibility.

So DRYML is just a means to an end. The real goal is to create a library of reusable user-interface components that actually succeed in making it very quick and easy to create the view layer of a web application.

That library is also part of Hobo – the *Rapid* tag library. You will visit this library later on in the book. Here we will see how DRYML provides the tools and raw materials that make a library like Rapid possible.

Discussing DRYML before Rapid means that many of the examples are *not* good advice for use of DRYML in a full Hobo app. For example, you might see

```
<%= h this.name %>
```

Which in an app that used Rapid would be better written `<view:name/>` or even just `<name/>` (that's a tag by the way, called `name`, not some metaprogramming trick that lets you use field names as tags). Bear that in mind while you're reading this chapter. The examples are chosen to illustrate the point at hand, they are not necessarily something you want to paste right into your application.

Simple page templates and ERB

In its most basic usage, DRYML can be indistinguishable from a normal Rails template. That's because DRYML is (almost) an extension of ERB, so you can still insert Ruby snippets using the `<% ... %>` notation. For example, a show-page for a blog post might look like this:

```
<html>
  <head>
    <title>My Blog</title>
  </head>
  <body>
    <h1>My Famous Blog!</h1>
    <h2><%= @post.title %></h2>
    <div class="post-body">
      <%= @post.body %>
    </div>
  </body>
</html>
```

No ERB inside tags

DRYML's support for ERB is not *quite* the same as true ERB templates. The one thing you can't do is use ERB snippets inside a tag. To have the value of an attribute generated dynamically in ERB, you could do:

```
<a href="<%= my_url %>">
```

In DRYML you would do:

```
<a href="#{my_url}">
```

In rare cases, you might use an ERB snippet to output one or more entire attributes:

```
<form <%= my_attributes %>>
```

We’re jumping ahead here, so just skip this if it doesn’t make sense, but to do the equivalent in DRYML, you would need your attributes to be in a hash (rather than a string), and do:

```
<form merge-attrs="&my_attributes">
```

Finally, in a rare case you could even use an ERB snippet to generate the tag name itself:

```
<<%= my_tag_name %>> ... </<%= my_tag_name %>>
```

To achieve that in DRYML, you could put the angle brackets in the snippet too:

```
<%= "<#{my_tag_name}>" %> ... <%= "</#{my_tag_name}>" %>
```

Where are the layouts?

Going back to the `<page>` tag at the start of this section, from a “normal Rails” perspective, you might be wondering why the boilerplate stuff like `<html>`, `<head>` and `<body>` are there. What happened to layouts? You don’t tend to use layouts with DRYML, instead you would define your own tag, typically `<page>`, and call that. Using tags for layouts is much more flexible, and it moves the choice of layout out of the controller and into the view layer, where it should be.

We’ll see how to define a `<page>` tag in the next section.

Defining simple tags

One of the strengths of DRYML is that defining tags is done right in the template (or in an imported tag library) using the same XML-like syntax. This means that if you’ve got markup you want to re-use, you can simply cut-and-paste it into a tag definition.

Here’s the page from the previous section, defined as a `<page>` tag simply by wrapping the markup in a `<def>` tag:

```
<def tag="page">
  <html>
    <head>
      <title>My Blog</title>
    </head>
    <body>
      <h1>My Famous Blog!</h1>
      <h2><%= @post.title %></h2>
      <div class="post-body">
        <%= @post.body %>
      </div>
    </body>
  </html>
</def>
```

```
</body>
</html>
</def>
```

Now we can call that tag just as we would call any other:

```
<page/>
```

If you'd like an analogy to “normal” programming, you can think of the `<def>...</def>` as defining a method called `page`, and `<page/>` as a call to that method.

In fact, DRYML is implemented by compiling to Ruby, and that is exactly what is happening.

Parameters

We've illustrated the most basic usage of `<def>`, but our `<page>` tag is not very useful. Let's take it a step further to make it into the equivalent of a layout. First of all, we clearly need the body of the page to be different each time we call it.

In DRYML we achieve this by adding *parameters* to the definition, which is accomplished with the `param` attribute. Here's the new definition:

```
<def tag="page">
  <html>
    <head>
      <title>My Blog</title>
    </head>
    <body param/>
  </html>
</def>
```

Now we can call the `<page>` tag and provide our own body:

```
<page>
  <body:>
    <h1>My Famous Blog!</h1>
    <h2><%= @post.title %></h2>
    <div class="post-body">
      <%= @post.body %>
    </div>
  </body:>
</page>
```

See how easy that was? We just added `param` to the `<body>` tag, which means our page tag now has a parameter called `body`. In the `<page>` call we provide some content for that parameter.

It's very important to read that call to `<page>` properly. In particular, the `<body:>` (note the trailing `:`) is *not* a call to a tag, it is providing a named parameter to the call to `<page>`. We call `<body:>` a *parameter tag*. In Ruby terms you could think of the call like this:

```
page(:body => "...my body content...")
```

Note that is not actually what the compiled Ruby looks like in this case, but it illustrates the important point that `<page>` is a call to a defined tag, whereas `<body:>` is providing a parameter to that call.

Changing Parameter Names

To give the parameter a different name, we can provide a value to the `param` attribute:

```
<def tag="page">
  <html>
    <head>
      <title>My Blog</title>
    </head>
    <body param="content"/>
  </html>
</def>
```

We would now call the tag like this:

```
<page><content:> ...body content goes here... </content:></page>
```

Multiple Parameters

As you would expect, we can define many parameters in a single tag. For example, here's a page with a side-bar:

```
<def tag="page">
  <html>
    <head>
      <title>My Blog</title>
    </head>
    <body>
      <div param="content"/>
      <div param="aside" />
    </body>
  </html>
</def>
```

Which we could call like this:

```
<page>
  <content:> ... main content here ... </content:>
  <aside:> ... aside content here ... </aside:>
</page>
```

Note that when you name a parameter, DRYML automatically adds a CSS class of the same name to the output, so the two `<div>` tags above will be output as `<div class="content">` and `<div class="aside">` respectively.

Default Parameter Content

In the examples we've seen so far, we've only put the `param` attribute on empty tags. That's not required though. If you declare a non-empty tag as a parameter, the content of that tag becomes the default when the call does not provide that parameter. This means you can easily add a parameter to any part of the template that you think the caller might want to be able to change:

```
<def tag="page">
  <html>
    <head>
      <title param>My Blog</title>
    </head>
    <body param>
    </body>
  </html>
</def>
```

We've made the page title parameterized. All existing calls to `<page/>` will continue to work unchanged, but we've now got the ability to change the title on a per-page basis:

```
<page>
  <title:>My VERY EXCITING Blog</title:>
  <body:>
    ... body content
  </body:>
</page>
```

This is a very nice feature of DRYML - whenever you're writing a tag, and you see a part that might be useful to change in some situations, just throw the `param` attribute at it and you're done.

Nested `param` Declarations

You can nest `param` declarations inside other tags that have `param` on them. For example, there's no need to choose between a `<page>` tag that provides a single content section and one that provides an aside section as well – a single definition can serve both purposes:

```
<def tag="page">
  <html>
    <head>
      <title>My Blog</title>
    </head>
    <body param>
      <div param="content"/>
      <div param="aside" />
    </body>
  </html>
</def>
```

Here the `<body>` tag is a param, and so are the two `<div>` tags inside it. The `<page>` tag can be called either like this:

```
<page>
  <body:> ... page content goes here ... </body:>
</page>
```

Or like this:

```
<page>
  <content:> ... main content here ... </content:>
  <aside:> ... aside content here ... </aside:>
</page>
```

An interesting question is, what happens if you give both a `<body:>` parameter and say, `<content:>`. By providing the `<body:>` parameter, you have replaced everything inside the body section, including those two parameterized `<div>` tags, so the `<body:>` you have provided will appear as normal, but the `<content:>` parameter will be silently ignored.

The Default Parameter

In the situation where a tag will usually be given a single parameter when called, you can give your tag a more compact XML-like syntax by using the special parameter name `default`:

```
<def tag="page">
  <html>
    <head>
      <title>My Blog</title>
    </head>
    <body param="default"/>
  </html>
</def>
```

Now there is no need to give a parameter tag in the call at all - the content directly inside the `<page>` tag becomes the `default` parameter:

```
<page> ... body content goes here -- no need for a parameter tag ...
</page>
```

You might notice that the `<page>` tag is now indistinguishable from a normal HTML tag. Some find this aspect of DRYML disconcerting at first – how can you tell what is an HTML tag and what it a defined DRYML tag? The answer is – you can't, and that's quite deliberate. This allows you to do nice tricks like define your own smart `<form>` tag or `<a>` tag (the Rapid library does exactly that). Other tag-based template languages (e.g. Java's JSP) like to put everything in XML namespaces. The result is very cluttered views that are boring to type and hard to read. From the start we put a very high priority on making DRYML templates compact and elegant. When you're new to DRYML you might have to do a lot of looking things up, as you would with

any new language or API, but things gradually become familiar and then view templates can be read and understood very easily.

The Implicit Context

In addition to the most important goal behind DRYML - creating a template language that would encourage re-use in the view layer, a secondary goal is for templates to be concise, elegant and readable. One aspect of DRYML that helps a lot in this regard is something called the *implicit context*.

This feature was born of a simple observation that pretty much every page in a web app renders some kind of hierarchy of application objects. Think about a simple page in a blog - say, the permalink page for an individual post. The page as a whole can be considered a rendering of a BlogPost object. Then we have sections of the page that display different “pieces” of the post – the title, the date, the author’s name, the body. Then we have the comments. The list of comments as a whole is also a “piece” of the BlogPost. Within that we have each of the individual comments, and the whole thing starts again: the comment title, date, author... This can carry on even further, for example some blogs are set up so that you can comment on comments.

This structure is incredibly common, perhaps even universal, as it seems to be intrinsically tied to the way we visually parse information. DRYML’s implicit context takes advantage of this fact to make templates extremely concise while remaining readable and clear. The object that you are rendering in any part of the page is known as the *context*, and every tag has access to this object through the method `this`. The controller sets up the initial context, and the templates then only have to mention where the context needs to *change*.

We’ll dive straight into some examples, but first a quick general point about this guide. If you like to use the full Hobo framework, you will probably always use DRYML and the Rapid tag library together. DRYML and Rapid have grown up together, and the design of each is heavily influenced by the other. Having said that, this is the DRYML Guide, not the Rapid Guide. We won’t be using any Rapid tags in this guide, because we want to document DRYML the language properly. That will possibly be a source of confusion if you’re very used to working with Rapid. Just keep in mind that we’re not allowed to use any Rapid tags in this guide and you’ll be fine.

In order to see the implicit context in its best light, we’ll start by defining a `<view>` tag, that simply renders the current context with HTML escaping. Remember the context is always available as `this`:

```
<def tag="view"><%= h this.to_s %></def>
```

Next we’ll define a tag for making a link to the current context. We’ll assume the object will be recognized by Rails’ polymorphic routing. Let’s call the tag `<l>` (for link):


```
<def tag="l"><a href="#{url_for this}" param="default"/></def>
```

Now let's use these tags in a page template. We'll stick with the comfortably boring blog post example. In order to set the initial context, our controller action would need to do something like this:

```
def show    @this = @blog_post = BlogPost.find(params[:id]) end
```

The DRYML template handler looks for the `@this` instance variable for the initial context. It's quite nice to also set the more conventionally named instance variable as we've done here. Now we'll create the page. Let's assume we're using a `<page>` tag along the lines of those defined above. We'll also assume that the blog post object has these fields: `title`, `published_at`, `body` and `belongs_to :author`, and that the author has a `name` field:

```
<page>
  <content:>
    <h2><view:title/></h2>
    <div class="details">
      Published by <l:author><view:name/></l> on <view:published-at/>.
    </div>
    <div class="post-body">
      <view:body/>
    </div>
  </content:>
</page>
```

When you see a tag like `<view:title/>`, you don't get any prizes for guessing what will be displayed. In terms of what actually happens, you can read this as "change the context to be the `title` attribute of the current context, then call the `<view>` tag". You might like to think of that change to the context as `this = this.title` (although in fact `this` is not assignable). But really you just think of it as "view the title". Of what? Of whatever is in context, in this case the blog post.

Be careful with the two different uses of colon in DRYML. A trailing colon as in `<foo:>` indicates a parameter tag, whereas a colon joining two names as in `<view:title/>` indicates a change of context.

When the tag ends, the context is set back to what it was. In the case of `<view/>` which is a self-closing tag familiar from XML, that happens immediately. The `<l:author>` tag is more interesting. We set the context to be the author, so that the link goes to the right place. Inside the `<l:author>` that context remains in place so we just need `<view:name/>` in order to display the author's name.

with and field attributes

The `with` attribute is a special DRYML attribute that sets the context to be the result of any Ruby expression before the tag is called. In DRYML any attribute value that starts with `'&'` is interpreted as a Ruby expression. Here's the same example as above using only the `with` attribute:

```
<page>
  <content:>
    <h2><view with="&@blog_post.title"/></h2>
    <div class="details">
      Published by <l with="&@blog_post.author"><view
with="&this.name"/></l>
      on <view with="&@blog_post.published-at"/>.
    </div>
    <div class="post-body">
      <view with="&@blog_post.body"/>
    </div>
  </content:>
</page>
```

Note that we could have used `&this.title` instead of `&@blog_post.title`.

The `field` attribute makes things more concise by taking advantage of a common pattern. When changing the context, we very often want to change to some attribute of the current context. `field="x"` is a shorthand for `with="&this.x"` (actually it's not quite the same, using the `field` version also sets `this_parent` and `this_field`, whereas `with` does not. This is discussed later in more detail).

The same template again, this time using `field`:

```
<page>
  <content:>
    <h2><view field="title"/></h2>
    <div class="details">
      Published by <l field="author"><view field="name"/></l>
      on <view field="published-at"/>.
    </div>
    <div class="post-body">
      <view field="body"/>
    </div>
  </content:>
</page>
```

If you compare that example to the first one, you should notice that the `:` syntax is just a shorthand for the `field` attribute; i.e., `<view field="name">` and `<view:name>` are equivalent.

Field chains

Sometimes you want to drill down through several fields at a time. Both the `field` attribute and the `:` shorthand support this. For example:

```
<view:category.name/> <view field="category.name"/>
```

`this_field` and `this_parent`

When you change the context using `field="my-field"` (or the `<tag:my-field>` shorthand), the previous context is available as `this_parent`, and the name of the field is available as `this_field`. If you set the context using `with="..."`, these values are not available. That means the following apparently identical tag calls are not quite the same:

```
<my-tag with="&@post.title"/> <my-tag with="&@post" field="title"/>
```

If the tag requires `this_parent` and `this_field`, and in Rapid, for example, some do, then it must be called using the second style.

Numeric field indices

If your current context is a collection, you can use the `field` attribute to change the context to a single item.

```
<my-tag field="7" />
<% i=97 %>
<my-tag field="&i" />
```

The `<repeat>` tag sets `this_field` to the current index into the collection.

```
<repeat:foos>
  <td><%= this_field %></td>
  <td><view /></td>
</repeat>
```

Forms

When rendering the Rapid library's `<form>` tag, DRYML keeps track of even more metadata in order to add `name` attributes to form fields automatically. This mechanism does not work if you set the context using `with=`.

Tag attributes

As we’ve seen, DRYML provides parameters as a mechanism for Customizing the markup that is output by a tag. Sometimes we want to provide other kinds of values to control the behavior of a tag: URLs, filenames or even Ruby values like hashes and arrays. For this situation, DRYML lets you define tag attributes.

As a simple example, say your application has a bunch of help files in `public/help`, and you have links to them scattered around your views. Here’s a tag you could define:

```
<def tag="help-link" attrs="file">
  <a class="help" href="#{base_url}/help/#{file}.html" param="default"/>
</def>
```

`<def>` takes a special attribute `attrs`. Use this to declare a list (separated by commas) of attributes, much as you would declare arguments to a method in Ruby. Here we’ve defined one attribute, `file`, and just like arguments in Ruby, `file` becomes a local variable inside the tag definition. In this definition we construct the `href` attribute from the `base_url` helper and `file`, using Ruby string interpolation syntax (`#{...}`). Remember that you can use that syntax when providing a value for any attribute in DRYML.

The call to this tag would look like this:

```
<help-link file="intro">Introductory Help</help-link>
```

Using regular XML-like attribute syntax – `file="intro"` – passes “intro” as a string value to the attribute. DRYML also allows you to pass any Ruby value. When the attribute value starts with `&`, the rest of the attribute is interpreted as a Ruby expression. For example you could use this syntax to pass `true` and `false` values:

```
<help-link file="intro" new-window="&true">Introductory Help</help-link>
<help-link file="intro" new-window="&false">Introductory Help</help-link>
```

And we could add that `new-window` attribute to the definition like this:

```
<def tag="help-link" attrs="file, new-window">
  <a class="help" href="#{base_url}/help/#{file}.html"
    target="#{new_window ? '_blank' : '_self'}" param="default"/>
</def>
```

An important point to notice there is that the markup-friendly dash in the `new-window` attribute became a Ruby-friendly underscore (`new_window`) in the local variable inside the tag definition. Using the `&`, you can pass any value you like – arrays, hashes, active-record objects... In the case of boolean values like the one used in the above example, there is a nicer syntax that can be used in the call...

Flag attributes

That `new-window` attribute shown in the previous section is simple switch - on or off. DRYML lets you omit the value of the attribute, giving a flag-like syntax:

```
<help-link file="intro" new-window>Introductory Help</help-link>
<help-link file="intro">Introductory Help</help-link>
```

Omitting the attribute value is equivalent to giving `"&true"` as the value. In the second example the attribute is omitted entirely, meaning the value will be `nil` which evaluates to false in Ruby and so works as expected.

`attributes` and `all_attributes` locals

Inside a tag definition two hashes are available in local variables:

- `attributes` contains all the attributes that *were not declared* in the `attrs` list of the `def` but that were provided in the call to the tag.
- `all_attributes` contains every attribute, including the declared ones.

Merging Attributes

In a tag definition, you can use the `merge-attrs` attribute to take any ‘extra’ attributes that the caller passed in, and add them to a tag of your choosing inside your definition. Let’s backtrack a bit and see why you might want to do that.

Here’s a simple definition for a `<markdown-help>` tag--it’s similar to a tag defined in the Hobo Cookbook app:

```
<def tag="markdown-help">
  <a href="http://daringfireball.net/..." param="default"/>
</def>
```

You would use it like this:

```
Add formatting using <markdown-help>markdown</markdown-help>
```

Suppose you wanted to give the caller the ability to choose the `target` for the link. You could extend the definition like this:

```
<def tag="markdown-help" attrs="target">
  <a href="http://daringfireball.net/..." target="&target" param="default"/>
</def>
```

Now we can call the tag like this:

```
Add formatting using <markdown-help target="_blank">markdown</markdown-help>
```

OK, but maybe the caller wants to add a CSS class, or a javascript `onclick` attribute, or any one of a dozen potential HTML attributes. This approach is not going to scale. That's where `merge-attrs` comes in. As mentioned above, DRYML keeps track of all the attributes that were passed to a tag, even if they were not declared in the `attrs` list of the tag definition. They are available in two hashes: `attributes` (that has only undeclared attributes) and `all_attributes` (that has all of them), but in normal usage you don't need to access those variables directly. To add all of the undeclared attributes to a tag inside your definition, just add the `merge-attrs` attribute, like this:

```
<def tag="markdown-help">
  <a href="http://daringfireball.net/..." merge-attrs param="default"/>
</def>
```

Note that the `merge` attribute is another way of merging attributes. Declaring `merge` is a shorthand for declaring both `merge-attrs` and `merge-params` (which we'll cover later).

Merging selected attributes

`merge-attrs` can be given a value - either a hash containing attribute names and values, or a list of attribute names (comma separated), to be merged from the `all_attributes` variable.

Examples:

```
<a merge-attrs="href, name">
<a merge-attrs="%my_attribute_hash">
```

A requirement that crops up from time to time is to forward to a tag all the attributes that it understands (i.e. the attributes from that tag's `attrs` list), and to forward some or all the other attributes to tags called within that tag. Say for example, we are declaring a tag that renders a section of content, with some navigation at the top. We want to be able to add CSS classes and so on to the main `<div>` that will be output, but the `<navigation>` tag also defines some special attributes, and these need to be forwarded to it.

To achieve this we take advantage of a helper method `attrs_for`. Given the name of a tag, it returns the list of attributes declared by that tag.

Here's the definition:

```
<def tag="section-with-nav">
  <div class="section" merge-attrs="%attributes - attrs_for(:navigation)">
  <navigation merge-attrs="%attributes & attrs_for(:navigation)"/>
    <do param="default"/>
  </div>
</def>
```

Note that:

- The expression `attributes - attrs_for(:navigation)` returns a hash of only those attributes from the `attributes` hash that are *not* declared by `<navigation>` (The `-` operator on `Hash` comes from `HoboSupport`)
- The expression `attributes & attrs_for(:navigation)` returns a hash of only those attributes from the `attributes` hash that *are* declared by `<navigation>` (The `&` operator on `Hash` comes from `HoboSupport`)
- The `<do>` tag is a “do nothing” tag, defined by the core DRYML taglib, which is always included.

The class attribute

If you have the following definition:

```
<def tag="foo">   <div id="foo" class="bar" merge-attrs /> </def>
```

and the user invokes it with:

```
<foo id="baz" class="bop" />
```

The following content will result:

```
<foo id="baz" class="bar bop" />
```

The `class` attribute receives special behavior when merging. All other attributes are overridden with the user specified values. The `class` attribute takes on the values from both the tag definition and invocation.

Repeated and optional content

As you would expect from any template language, DRYML has the facility to repeat sections of content, and to optionally render or not render given sections according to your application’s data. DRYML provides two alternative syntaxes, much as Ruby does (e.g. Ruby has the block `if` and the one-line suffix version of `if`).

Conditionals - if and unless

DRYML provides `if` and `unless` both as tags, which come from the core tag library, and are just ordinary tag definitions, and as attributes, which are part of the language:

The tag version:

```
<if test="&logged_in?"><p>Welcome back</p></if>
```

The attribute version:

```
<p if="&logged_in?">Welcome back</p>
```

Important note! The test is performed (in Ruby terms) like this:

```
if (...your test expression...).blank?
```

Got that? Blankiness not truthiness (`blank?` comes from ActiveSupport by the way – Rails’ mixed bag of core-Ruby extensions). So for example, in DRYML:

```
<if test="&current_user.comments">
```

is a test to see if there are any comments – empty collections are considered blank. We are of the opinion that Matz made a fantastic choice for Ruby when he followed the Lisp / Smalltalk approach to truth values, but that view templates are a special case, and testing for blankness is more often what you want.

Can we skip `<unless>`? It’s like `<if>` with the nest negated. You get the picture, right?

Repetition

For repeating sections of content, DRYML has the `<repeat>` tag (from the core tag library) and the `repeat` attribute.

The tag version:

```
<repeat with="&current_user.new_messages">
  <h3><%= h this.subject %></h3>
</repeat>
```

The attribute version:

```
<h3 repeat="&current_user.new_messages"><%= h this.subject %></h3>
```

Notice that as well as the content being repeated, the implicit context is set to each item in the collection in turn.

Even/odd classes

It's a common need to want alternating styles for items in a collection - e.g. striped table rows. Both the repeat attribute and the repeat tag set a scoped variable `scope.even_odd` which will be alternately 'even' then 'odd', so you could do:

```
<h3 repeat="&new_messages" class="{scope.even_odd}"><%= h this.subject %></h3>
```

That example illustrates another important point – any Ruby code in attributes is evaluated *inside* the repeat. In other words, the repeat attribute behaves the same as wrapping the tag in a `<repeat>` tag.

first_item? and last_item? helpers

Another common need is to give special treatment to the first and last items in a collection. The `first_item?` and `last_item?` helpers can be used to find out when these items come up; e.g., we could use `first_item?` to capitalise the first item:

```
<h3 repeat="&new_messages"><%= h(first_item? ? this.subject.upcase : this.subject) %></h3>
```

Repeating over hashes

If you give a hash as the value to repeat over, DRYML will iterate over each key/value pair, with the value available as `this` (i.e. the implicit context) and the key available as `this_key`. This is particularly useful for grouping things in combination with the `group_by` method:

```
<div repeat="&current_user.new_messages.group_by(&:sender)">
  <h2>Messages from <%= h this_key %></h2>
  <ul>
    <li repeat><%= h this.subject %></li>
  </ul>
</div>
```

That example has given a sneak preview of another point - using if/unless/repeat with the implicit context. We'll get to that in a minute.

Using the implicit context

If you don't specify the test of a conditional, or the collection to repeat over, the implicit context is used. This allows for a few nice shorthands. For example, this is a common pattern for rendering collections:

```
<if:comments>
  <h3>Comments</h3>
  <ul>
    <li repeat> ... </li>
  </ul>
</if>
```

We're switching the context on the `<if>` tag to be `this.comments`, which has two effects. Firstly the comments collection is used as the test for the `if`, so the whole section including the heading will be omitted if the collection is empty (remember that `if` tests for blankness, and empty collections are considered blank). Secondly, the context is switched to be the comments collection, so that when we come to repeat the `` tag, all we need to say is `repeat`.

One last shorthand - attributes of `this`

The attribute versions of `if/unless` and `repeat` support a useful shortcut for accessing attributes or methods of the implicit context. If you give a literal string attribute—that is, an attribute that does not start with `&`—this is interpreted as the name of a method on `this`. For example:

```
<li repeat="comments"/>
```

is equivalent to

```
<li repeat="&this.comments"/>
```

Similarly

```
<p if="sticky?">This post has been marked 'sticky'</p>
```

is equivalent to

```
<p if="this.sticky?">This post has been marked 'sticky'</p>
```

It is a bit inconsistent that these shortcuts do not work with the tag versions of `<if>`, `<unless>` and `<repeat>`. This may be remedied in a future version of DRYML.

Content tags only

The attributes introduced in this section – `repeat`, `if` and `unless`, can only be used on content tags, i.e. static HTML tags and defined tags. They cannot be used on tags like `<def>`, `<extend>` and `<include>`.

Pseudo parameters - before, after, append, prepend, and replace

For every parameter you define in a tag, there are five “pseudo parameters” created as well. Four allow you to insert extra content without replacing existing content, and one lets you replace or remove a parameter entirely.

To help illustrate these, here’s a very simple `<page>` tag:

```
<def tag="page">
  <body>
    <h1 param="heading"><%= h @this.to_s %></h1>
    <div param="content"></div>
  </body>
</def>
```

We’ve assumed that `@this.to_s` will give us the name of the object that this page is presenting.

Inserting extra content

The output of the heading would look something like:

```
<h1 class="heading">Welcome to my new blog</h1>
```

Pseudo parameters give us the ability to insert extra context in four places, marked here as (A), (B), (C) and (D):

```
(A)<h1 class="heading">(B)Welcome to my new blog(C)</h1>(D)
```

The parameters are:

- (A) – `<before-heading:>`
- (B) – `<prepend-heading:>`
- (C) – `<append-heading:>`
- (D) – `<after-heading:>`

So, for example, suppose we want to add the name of the blog to the heading:

```
<h1 class="heading">Welcome to my new blog -- The Hobo Blog</h1>
```

To achieve that on one page, we could call the `<page>` tag like this:

```
<page>
  <append-heading:> -- The Hobo Blog</append-heading:>
  <body:>      ...   </body>
</page>
```

Or we could go a step further and create a new page tag that added that suffix automatically. We could then use that new page tag for an entire section of our site:

```
<def tag="blog-page">
  <page>
    <append-heading:> -- The Hobo Blog</append-heading:>
    <body: param></body>
  </page>
</def>
```

(Note: we have explicitly made sure that the `<body:>` parameter is still available. There is a better way of achieving this using `merge-params` or `merge`, which are covered later.)

The default parameter supports append and prepend

As we've seen, the `<append-...:>` and `<prepend-...:>` parameters insert content at the beginning and end of a tag's content. But in the case of a defined tag that may output all sorts of other tags and may itself define many parameters, what exactly *is* the tag's "content"? It is whatever is contained in the `default` parameter tag. So `<append-...:>` and `<prepend-...:>` only work on tags that define a default parameter.

For this reason, you will often see tag definitions include a `default` parameter, even though it would be rare to use it directly. It is there so that `<append-...:>` and `<prepend-...:>` work as expected.

Replacing a parameter entirely

So far, we've seen how the parameter mechanism allows us to change the attributes and content of a tag, but what if we want to remove the tag entirely? We might want a page that has no `<h1>` tag at all, or has `<h2>` instead. For that situation we can use "replace parameters". Here's a page with an `<h2>` instead of an `<h1>`:

```
<page>
  <heading: replace><h2>My Awesome Page</h2></heading:>
</page>
```

And here's one with no heading at all:

```
<page>
  <heading: replace/>
</page>
```

There is a nice shorthand for the second case. For every parameter, the enclosing tag also supports a special `without` attribute. This is exactly equivalent to the previous example, but much more readable:

```
<page without-heading/>
```

Note: to make things more consistent, `<heading: replace>` may become `<replace-heading:>` in the future.

Current limitation

Due to a limitation of the current DRYML implementation, you cannot use both `before` and `after` on the same parameter. You can achieve the same effect as follows (this technique is covered properly later in the section on wrapping content):

```
<heading: replace>
... before content ...
  <heading restore>
... after content ...
</heading:>
```

Nested parameters

As we’ve discussed at the start of this guide, one of the main motivations for the creation of DRYML was to deliver a higher degree of *re-use* in the view layer. One of the great challenges of re-use is managing the constant tension between re-use and flexibility: the greater the need for flexibility, the harder it is to re-use existing code. This has a very direct effect on the *size* of things that we can successfully re-use. Take the humble hypertext link for example. A link is a link – there’s only so much you could really want to change, so it’s not surprising that long ago we stopped having to assemble links from fragments of HTML text. Rails has its `link_to` helper, and Hobo Rapid has its `<a>` tag. At the other extreme, reusing an entire photo gallery or interactive calendar is extremely difficult. Again no surprise—these things have been built from scratch over and over again, because each time something slightly (or very) different is needed. A single calendar component that is flexible enough to cover every eventuality would be so complicated that configuring it would be more effort than starting over.

This tension between re-use and flexibility will probably never go away; life is just like that. As components get larger they will inevitably get either harder to work with or less flexible. What we can do though, through technologies like DRYML, is slow down the onset of these problems. By thinking about the fundamental challenges to re-use, we have tried to create a language in which, as components grow larger, simplicity and flexibility can be retained longer.

One of the most important features that DRYML brings to the re-use party is *nested parameters*.

They are born of the following observations:

- As components get larger, they are not really single components at all, but compositions of many smaller sub-components.
- Often, the Customization we wish to make is not to the “super-component” but to one of the sub-components.
- What is needed, then, is a means to pass parameters and attributes not just to the tag you are calling, but to the tag called within the tag, or the tag called within the tag called within the tag, and so on.

DRYML’s nested parameter mechanism does exactly that. After you’ve been using DRYML for some time, you may notice that you don’t use this feature very often. But when you do use it, it can make the difference between sticking with your nice high-level components or throwing them away and rebuilding from scratch. A little use of nested parameters goes a long way.

An example

To illustrate the mechanism, we’ll build up a small example using ideas that are familiar from Rapid. This is not a Rapid guide though, so we’ll define these tags from scratch. First off, the `<card>` tag. This captures the very common pattern of web pages displaying collections of some kind of object as small “cards”: comments, friends, discussion threads, etc.

```
<def tag="card">
  <div class="card" merge-attrs>
    <h3 param="heading"><%= h this.to_s %></h3>
    <div param="body"></div>
  </div>
</def>
```

We’ve defined a very simple `<card>` that uses the `to_s` method to give a default heading, and provides a `<body:>` parameter that is blank by default. Here’s how we might use it:

```
<h2>Discussions</h2>
<ul>
  <li repeat="@discussions">
    <card>
      <body:><%= this.posts.length %> posts</body:>
    </card>
  </li>
</ul>
```

This example (specifically, the collection created in the `<li repeat="@discussions">` section) demonstrates that as soon as we have the concept of a card, we very often find ourselves wanting to render a collection of `<card>` tags. The obvious next step is to capture that collection-of-cards idea as a reusable tag:

```
<def tag="collection">
  <h2 param="heading"></h2>
  <ul>
    <li repeat>
      <card param>
    </li>
  </ul>
</def>
```

The `<collection>` tag has a straightforward `<heading:>` parameter, but notice that the `<card>` tag is also declared as a parameter. Whenever you add `param` to a tag that itself also has parameters, you give your “super-tag” (`<collection>` in this case) the ability to customize the “sub-tag” (`<card>` in this case) using *nested parameters*. Here’s how we can use the nested parameters in the `<collection>` tag to get the same output as the `<li repeat="@discussions">` section in the previous example:

```
<collection>
  <heading:>Discussions</heading>
  <card:>
    <body:><%= this.posts.length %>posts</body:>
  </card:>
</collection>
```


This nesting works to any depth. To show this, if we define an `<index-page>` tag that uses `<collection>` and declares it as a parameter:

```
<def tag="index-page">
  <html>
    <head> ... </head>
    <body>
      <h1 param="heading"></h1>
      ...
      <collection param>
      ...
    </body>
  </html>
</def>
```

we can still access the card inside the collection inside the page:

```
<index-page>
  <heading:>Welcome to our forum</heading:>
  <collection:>
    <heading:>Discussions</heading>
    <card:><body:><%= this.posts.length %>posts</body:></card:>
  </collection:>
</index-page>
```

Pay careful attention to the use of the trailing `':'`. The definition of `<index-page>` contains a *call* the collection tag, written `<collection>` (no `':'`). By contrast, the above call to `<index-page>` *customizes* the call to the collection tag that is already present inside `<index-page>`, so we write `<collection:>` (with a `':'`). Remember:

- Without `':'` – call a tag
- With `':'` – customize an existing call inside the definition

Customizing and extending tags

As we’ve seen, DRYML makes it easy to define tags that are highly customizable. By adding `prams` to the tags inside your definition, the caller can insert, replace and tweak to their heart’s content. Sometimes the changes you make to a tag’s output are needed not once, but many times throughout the site. In other words, you want to define a new tag in terms of an existing tag.

New tags from old

As an example, let’s bring back our `card` tag:

```
<def tag="card">
  <div class="card" merge-attrs>
    <h3 param="heading"><%= h this.to_s %></h3>
    <div param="body"></div>
  </div>
</def>
```

Now let’s say we want a new kind of card, one that has a link to the resource that it represents. Rather than redefine the whole thing from scratch, we can define the new card, say, “`linked-card`”, like this:

```
<def tag="linked-card">
  <card>
    <heading: param><a href="%object_url this"><%= h this.to_s %></a>
  </heading:>
  </card>
</def>
```

That’s all well and good but there are a couple of problems:

- The original card used `merge-attrs` so that we could add arbitrary HTML attributes to the final `<div>`. Our new card has lost that feature
- Worse than that, the new card is in fact useless, as there’s no way to pass it the `body` parameter

Let’s solve those problems in turn. First the attributes.

`merge-attrs` again

In fact `merge-attrs` works just the same on defined tags as it does on HTML tags that are output, so we can simply add it to the call to `<card>`, like this:

```
<def tag="linked-card">
  <card merge-attrs>
    <heading: param><a href="%object_url this"><%= h this.to_s %></a>
  </heading:>
  </card>
</def>
```

```

    </card>
  </def>

```

Now we can do things like `<linked-card class="emphasised">`, and the attribute will be passed from `<linked-card>`, to `<card>`, to the rendered `<div>`.

Now we'll fix the parameters, it's going to look somewhat similar...

merge-prams

We'll introduce `merge-prams` the same way we introduced `merge-attrs` – by showing how you would get by without it. The problem with our `<linked-card>` tag is that we've lost the `<body:>` parameter. We could bring it back like this:

```

<def tag="linked-card">
  <card merge-attrs>
    <heading: param><a href="%object_url this"><%= h this.to_s %></a>
    </heading:>
    <body: param/>
  </card>
</def>

```

In other words, we use the `param` declaration to give `<linked-card>` a `<body:>` parameter, which is forwarded to `<card>`. But what if `<card>` had several parameters? We would have to list them all out. And what if we add a new parameter to `<card>` later? We would have to remember to update `<linked-card>` and any other customized cards we had defined. Instead we use `merge-prams`, much as we use `merge-attrs`:

```

<def tag="linked-card">
  <card merge-attrs merge-prams>
    <heading: param><a href="%object_url this"><%= h this.to_s %></a>
    </heading:>
  </card>
</def>

```

You can read `merge-prams` as: take any “extra” parameters passed to `<linked-card>` and forward them all to `<card>`. By “extra” parameters, we mean any that are not declared as parameters (via the `param` attribute) inside the definition of `<linked-card>`.

There are two local variables inside the tag definition that mirror the `attributes` and `all_attributes` variables described previously:

- `parameters` a hash containing all the “extra” parameters (those that do not match a declared parameter name)
- `all_parameters` a hash containing all the parameters passed to the tag

The values in these hashes are Ruby procs. One common use of `all_parameters` is to test if a certain parameter was passed or not:

```
<if test="&all_parameters[:body]">
```

In fact, `all_parameters` and `parameters` are not regular hashes, they are instances of a subclass of `Hash` – `Hobo::Dryml::TagParameters`. This subclass allows parameters to be called as if they were methods on the hash object, e.g.:

```
parameters.default
```

That's not something you'll use often.

Merge

As it's very common to want both `merge-attrs` and `merge-prams` on the same tag, there is a shorthand for this: `merge`. So the final, preferred definition of `<linked-card>` is:

```
<def tag="linked-card">
  <card merge>
    <heading: param><a href="%object_url this"><%= h this.to_s %></a>
  </heading:>
</card>
</def>
```

Merging selected parameters

Just as with `merge-attrs`, `merge-prams` can be given a value - either a hash containing the parameters you wish to merge, or a list of parameter names (comma separated), to be merged from the `all parameters` variable.

Examples:

```
<card merge-params="heading, body">
<card merge-params="%my_parameter_hash">
```

Extending a tag

We've now seen how to easily create a new tag from an existing tag. But what if we don't actually want a new tag, but rather we want to change the behavior of an existing tag in some way, and keep the tag name the same. What we can't do is simply use the existing name in the definition:

```
<!-- DOESN'T WORK! -->
<def tag="card">
  <card merge>
    <heading: param><a href="%object_url this"><%= h this.to_s %></a>
  </heading:>
</card>
</def>
```

All we've done there is created a nice stack overflow when the card calls itself over and over. Fortunately, DRYML has support for extending tags. Use `<extend>` instead of `<def>`:

```
<extend tag="card">
  <old-card merge>
    <heading: param><a href="%object_url this"><%= h this.to_s %></a>
  </heading:>
</old-card>
</extend>
```

The one thing to notice there is that the “old” version of `<card>`, i.e. the one that was active before you’re extension, is available as `<old-card>`. That’s about all there is to it.

Here’s another example where we add a footer to every page in our application. It’s very common to `<extend tag="page">` in your `application.dryml`, in order to make changes that should appear on every page:

```
<extend tag="page">
  <old-page merge>
    <footer: param>
      ...
      your custom footer here
      ...
    </footer:>
  </old-page>
</extend>
```

Aliasing tags

Welcome to the shortest section of The DRYML Guide...

If you want to create an alias of a tag; i.e., an identical tag with a different name:

```
<def tag="my-card" alias-of="card"/>
```

Note that's a self closing tag – there is no body to the definition.

So... that's aliasing tags then...

Polymorphic tags

DRYML allows you to define a whole collection of tags that share the same name, where each definition is appropriate for a particular type of object being rendered. When you call the tag, the type (i.e. class) of the context is used to determine which definition to call. These are called polymorphic tags.

To illustrate how these work, let's bring back our simple `<card>` tag once more:

```
<def tag="card" polymorphic>
  <div class="card" merge-attrs>
    <h3 param="heading"><%= h this.to_s %></h3>
    <div param="body">
      </div>
    </div>
  </def>
```

We've added the `polymorphic` attribute to the `<def>`. This tells DRYML that `<card>` can have many definitions, each for a particular type. The definition we've given here is called the “base” definition or the “base card”. The base definition serves two purposes:

- It is the fallback if we call `<card>` and no definition is found for the current type.
- The type-specific definition can use the base definition as a starting point to be further customized.

To add a type-specific `<card>`, we use the `for` attribute on the `<def>`. For example, a card for a `Product`:

```
<def tag="card" for="Product"> ... </def>
```

Note: if the name in the `for` attribute starts with an uppercase letter, it is taken to be a class name. Otherwise it is taken to be an abbreviated name registered with `HoboFields`; e.g.:

```
<def tag="input" for="email_address">
```

For the product card, lets make the heading be a link to the product, and put the price of the product in the body area:

```
<def tag="card" for="Product">
  <card merge>
    <heading: param><a href="#{object_url this}"><%= h this.to_s %></a>
    </heading:>
    <body: param="price">$<%= this.price %></body:>
  </card>
</def>
```

We call this a type-specific definition. Some points to notice:

- The callback to `<card>` is not a recursive loop, but a call to the base definition.
- We're using the normal technique for Customizing / extending an existing card; i.e., we're using `merge`.

It is not required for the type-specific definition to call the base definition, it's just often convenient. In fact the base definition is not required. It is valid to declare a polymorphic tag with no content:

```
<def tag="my-tag" polymorphic/>
```

Type hierarchy

If, for a given call, no type-specific definition is available for `this.class`, the search continues with `this.class.superclass` and so on up the superclass chain. If the search reaches either `ActiveRecord::Base` or `Object`, the base definition is used.

Specifying the type explicitly

Sometimes it is useful to give the type explicitly for the call explicitly (i.e., to override the use of `this.class`). The `for-type` attribute (on the call) provides this facility. For example, you might want to implement one type-specific definition in terms of another:

```
<def tag="card" for="SpecialProduct">
  <card for-type="Product">
    <append-price:>Today Only!</append-price:>
  </card>
</def>
```


Extending polymorphic tags

Type-specific definitions can be extended just like any other tag using the `<extend>` tag. For example, here we simply remove the price:

```
<extend tag="card" for="Product">
  <old-card merge without-price/>
</extend>
```

Wrapping content

DRYML provides two mechanisms for wrapping existing content inside new tags.

Wrapping *inside* a parameter

Once or twice in the previous examples, we have extended our card tag definition, replacing the plain heading with a hyperlink heading. Here is an example call to our extended card tag:

```
<card>
  <heading:><a href="#{object_url this}"><%= h this.to_s %></a></heading:>
</card>
```

There's a bit of repetition there – `<%= h this.to_s %>` was already present in the original definition. All we really wanted to do was wrap the existing heading in an `<a>`. In this case there wasn't much markup to repeat, so it wasn't a big deal, but in other cases there might be much more.

We can't use `<prepend-heading:><a></prepend-heading:>` and `<append-heading:></append-heading:>` because that's not well formed markup (and is very messy besides). Instead, DRYML has a specific feature for this situation. The `<param-content>` tag is a special tag that brings back the default content for a parameter.

Here's how it works:

```
<card>
  <heading:>
    <a href="#{object_url this}"><param-content for="heading"/></a>
  </heading:>
</card>
```

That's the correct way to wrap *inside* the parameter, so in this case the output is:

```
<h3><a href="...">Fried Bananas</a></h3>
```

What if we wanted to wrap the *entire* `<heading:>` parameter, including the `<h3>` tags?

Wrapping *outside* a parameter

For example, we might want to give the card a new 'header' section, that contained the heading, and the time the record was created, like this:

```
<div class="header">
  <h3>Fried Bananas</h3>
  <p>Created: ....</p>
```

```
</div>
```

To use DRYML terminology, what we've done there is *replaced* the entire heading with some new content, and the new content happens to contain the original heading. So we replaced the heading, and then restored it again, which in DRYML is written:

```
<card>
  <heading: replace>
    <div class="header">
      <heading: restore/>
      <p>Created: <%= this.created_at.to_s(:short) %></p>
    </div>
  </heading:>
</card>
```

To summarize:

- To wrap content inside a parameter, use `<param-content/>`
- To wrap an entire parameter, including the parameterized tag itself (the `<h3>` in our examples), use the `replace` and `restore` attributes.

Local variables and scoped variables.

DRYML provides two tags for setting variables: `<set>` and `<set-scoped>`.

Setting local variables with `<set>`

Sometimes it's useful to define a local variable inside a template or a tag definition. It's worth avoiding if you can, as we don't really want our view layer to contain lots of low-level code, but sometimes it's unavoidable. Because DRYML extends ERB, you can simply write:

```
<% total = price_of_fish * number_of_fish %>
```

For purely aesthetic reasons, DRYML provides a tag that does the same thing:

```
<set total="&price_of_fish * number_of_fish"/>
```

Note that you can put as many attribute/value pairs as you like on the same `<set>` tag, but the order of evaluation is not defined.

Scoped variables – `<set-scoped>`

Scoped variables (which is not a great name, we realize as we come to document them properly) are kind of like global variables with a limited lifespan. We all know the pitfalls of global variables, and DRYML's scoped variables should indeed be used as sparingly as possible, but you can pull off some very useful tricks with them.

The `<set-scoped>` tag is very much like `<set>` except you open it up and put DRYML inside it:

```
<set-scoped xyz="&..."> ... </set-scoped>
```

The value is available as `scope.xyz` anywhere inside the tag *and in any tags that are called inside that tag*. That's the difference between `<set>` and `<set-scoped>`.

They are like *dynamic variables* from LISP. To repeat the point, they are like global variables that exist from the time the `<set-scope>` tag is evaluated, and for the duration of the evaluation of the body of the tag, and are then removed.

As an example of their use, let's define a simple tag for rendering navigation links. The output should be a list of `<a>` tags, and the `<a>` that represents the “current” page should have a CSS class “current”, so it can be highlighted in some way by the stylesheet. (In fact, the need to create a reusable tag like this is where the feature originally came from).

On our pages, we'd like to simply call, say:

```
<main-nav current="Home">`
```

And we'd like it to be easy to define our own `<main-nav>` tag in our applications:

```
<def tag="main-nav">
  <navigation merge-attrs>
    <nav-item href="...">Home</nav-item>
    <nav-item href="...">News</nav-item>
    <nav-item href="...">Offers</nav-item>
  </navigation>
</def>
```

Here's the definition for the `<navigation>` tag:

```
<def tag="navigation" attrs="current">
  <set-scoped current-nav-item="current">
    <ul merge-attrs param="default"/>
  </set-scoped>
</def>
```

All `<navigation>` does is set a scoped-variable to whatever was given as `current` and output the body wrapped in a ``.

Here's the definition for the `<nav-item>` tag:

```
<def tag="nav-item">
  <set body="&parameters.default"/>
  <li class="#{'current' if scope.current_nav_item == body}">
    <a merge-attrs><%= body %>
  </li>
</def>
```

The content inside the `<nav-item>` is compared to `scope.current_nav_item`. If they are the same, the “current” class is added. Also note the way `parameters.default` is evaluated and the result stored in the local variable `body`, in order to avoid evaluating the body twice.

Nested scopes

One of the strengths of scoped variables is that scopes can be nested, and where there are name clashes, the parent scope variable is temporarily hidden, rather than overwritten. With a bit of tweaking, we could use this fact to extend our `<navigation>` tag to support a sub-menu of links within a top level section. The sub-menu could also use `<navigation>` and `<nav-item>` and the two `scope.current_nav_item` variables would not conflict with each other.

Taglibs

DRYML provides the `<include>` tag to support breaking up lots of tag definitions into separate “tag libraries”, known as taglibs. You can call `<include>` with several different formats:

```
<include src="foo"/>
```

Load `foo.dryml` from the same directory as the current template or taglib.

```
<include src="path/to/foo"/>
```

Load `app/views/path/to/foo.dryml`

```
<include src="foo" plugin="path/to/plugin"/>
```

Load `vendor/plugins/path/to/plugin/taglibs/foo.dryml`

When running in development mode, all of these libraries are automatically reloaded on every request.

Divergences from XML and HTML

Self-closing tags

In DRYML, `<foo:/>` and `<foo:></foo:>` have two slightly different meanings. The second form replaces the parameter’s default inner content with the specified content: nothing in this case.

The first form uses the parameters default inner content unchanged.

This is very useful if you wish to add an attribute to a parameter but leave the inner content unchanged. In this example:

```
<def tag="bar">
  <div class="container" merge-attrs>
    <p class="content" param>
      Hello
    </p>
  </div>
</def>
```

Then:

```
<bar><foo: class="my-foo"/></bar>
```

Gives:

```
<div class="container">
  <p class="content my-foo">
    Hello
  </p>
</div>
```

If you used:

```
<bar><foo: class="my-foo"></foo:></bar>
```

You would get:

```
<div class="container">
  <p class="content my-foo"></p>
</div>
```

Colons in tag names

In XML, colons are valid inside tag and attribute names. However they are reserved for “experiments for namespaces”. So it’s possible that we may be non-compliant with the not-yet-existent XML 2.0.

Close tag shortcuts

In DRYML, you’re allowed to close tags with everything preceding the colon:

```
<view:name> Hello </view>
```

XML requires the full tag to be specified:

```
<view:name> Hello </view:name>
```

Null end tags

Self-closing tags are [technically illegal](#) in HTML. So `
` is technically not valid HTML. However, browsers do parse it as you expect. It is valid XHTML, though.

However, browsers only do this for *empty* elements. So tags such as `<script>` and `<a>` require a separate closing tag in HTML. This behavior has surprised many people:

```
<script src="foobar.js" />
```

...is not recognized in many web browsers for this reason. You must use:

```
<script src="foorbar.js"></script>
```

...in HTML instead.

DRYML follows the XML conventions:

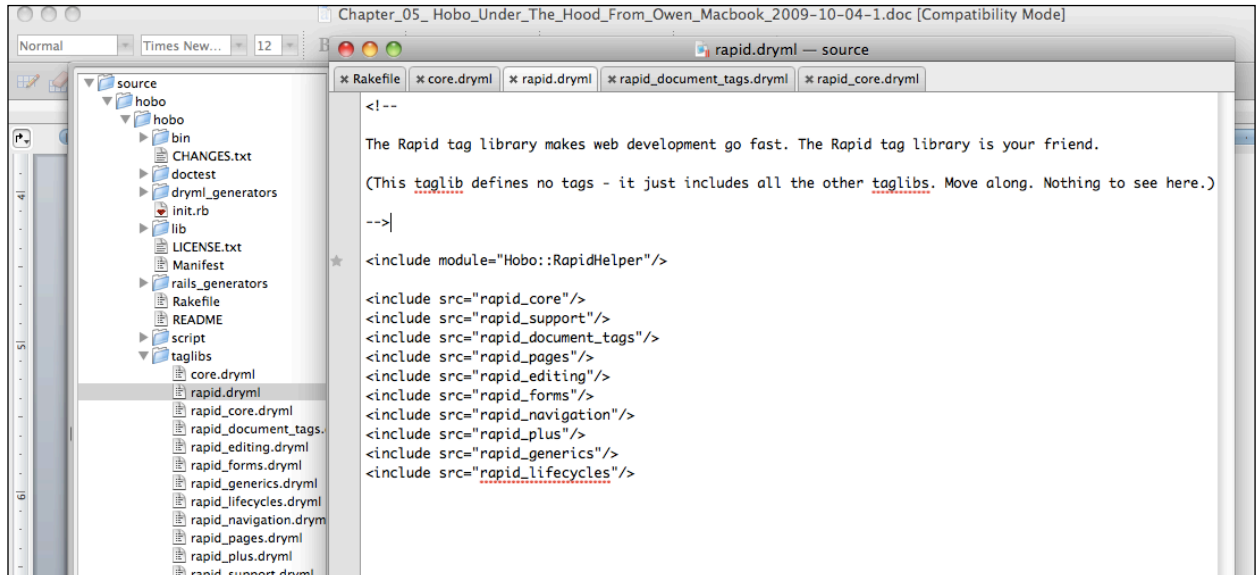
```
<a/>
```

...is valid DRYML.

Chapter 10 – The Hobo Rapid Tag Library

This section of the book serves as reference for all of the pre-defined DRYML tags used by Hobo to provide the “magic” rendering of pages and forms without you coding. You can learn how to extend and use these tags to customize your applications.

Look at the figure below that shows the contents of *rapid.dryml*



You see how the *rapid.dryml* file includes the following source files, in alphabetical order:

```
rapid_core.dryml
rapid_document_tags.dryml
rapid_pages.dryml
rapid_editng.dryml
rapid_forms.dryml
rapid_navigation.dryml
rapid_plus.dryml
rapid_generics.dryml
rapid_lifecycles.dryml
rapid_support.dryml
```

Rapid Tag Library Index

The following categories will be described in detail in the rest of this chapter:

Core	Core DRYML tags. These are included implicitly and are always available. Contains mainly control-flow tags.
<i>Rapid</i>	This taglib does not define tags - it just includes all the other taglibs.
Rapid Core	Core Rapid tags and tags that don't belong to other categories.
Rapid Document Tags	Extra tags for semantic markup.
Rapid Editing	Rapid Editing provides “in-place” or “AJAX” editors for various basic data types.
Rapid Forms	Rapid Forms provides various tags that make it quick and easy to produce working new or edit forms.
Rapid Generics	Rapid Generics provides tags that provide generic renderings that can adapt to the model being rendered.
Rapid Lifecycles	Contains view-layer support for Hobo's lifecycles.
Rapid Navigation	Support for navigation links, account navigation (log in, out etc.) and pagination navigation.
Rapid Pages	Rapid-Pages provides tags for working with entire pages.
Rapid Plus	Tags that define higher level interactive ‘widgets’
Rapid Summary	A collection of tags that allow an application outline or summary to be created.
Rapid Support	Rapid Support is the home for some tags that are useful in defining other tags.
Rapid User Pages	Rapid User Pages contains tags that implement the basics of Hobo's user management: log in, sign up, forgot password etc.

Core

Core DRYML tags. These are included implicitly and are always available. Contains mainly control-flow tags.

<code><call-tag></code>
<code><wrap></code>
<code><partial></code>
<code><repeat></code>
<code><do></code>
<code><with></code>
<code><if></code>
<code><else></code>
<code><unless></code>

`<call-tag>`

Call the tag given by the `tag` attribute. This lets you call tags dynamically based on some runtime value. It's the DRYML equivalent of Ruby's `send` method.

`<wrap>`

Wrap the body in the tag specified by the `tag` attribute, if `when` is true.

Using regular DRYML conditional logic it is rather awkward to conditionally wrap some tag in another tag. This tag makes it easy to do that.

Usage

For example, you might want to wrap an `` tag in an `<a>` tag but only under certain conditions. Say the current context has an `href` attribute that may or may not be nil. We want to wrap the `img` in `<a>` if `href` is not nil:

```
<wrap when="&this.href.present?" tag="a" href="&this.href">
  
</wrap>
```

`<partial>`

DRYML version of `render(:partial => 'my_partial')`

Usage

```
<partial name="my-partial" locals="{:x => 10, :y => 20}"/>
```

`<repeat>`

Repeat a section of mark-up. The context should be a collection (anything that responds to `each`). The content of the call to `<repeat>` will be repeated for each item in the collection, and the context will be set to each item in turn.

Attributes

- `join`: The value of this attribute, if given, will be inserted between each of the items (e.g. `join=", "` is very common).

<do>

The ‘do nothing’ tag. Used to add parameters or change context without adding any markup

<with>

Alias of `do`

<if>

DRYML’s ‘if’ test

Usage

```
<if test="&current_user.administrtrator?">Logged in as  
administrator</if>  
  <else>Logged in as normal user  
</else>
```

IMPORTANT NOTE: `<if>` tests for non-blank vs. blank (as defined by ActiveSupport), not true vs. false. If you do not give the `test` attribute, uses the current context instead. This allows a nice trick like this:

```
<if:comments>...</if>
```

This has the double effect of changing the context to the `this.comments`, and only evaluating the body if there are comments (because an empty collection is considered blank)

<else>

General purpose `else` clause. `<else>` works with various tags such as `<if>` and `<repeat>` (the `else` clause will be output if the collection was empty). It simply outputs its content if `Hobo::Dryml.last_if` is false. This is pretty much a crazy hack, which violates many good principles of language design, but it’s very useful :)

<unless>

Same behavior as `<if>`, except the test is negated.

Rapid Core

Core Rapid tags and tags that don't belong to other categories.

<dev-user-changer>
<field-list>
<nil-view>
<table>
<image>
<spinner>
<hobo-rapid-javascripts>
<name>
<type-name>
<collection-name>
<a>
<count>
<theme-stylesheet>
<You>
<Your>
<A-or-An>
<comma-list>
<collection-list>
<collection-view>
<links-for-collection>
<view>

<dev-user-changer>

Development mode only - a menu to change the `current_user`

<field-list>

Renders a table with one row per field, where each row contains a `<th>` with the field name, and a `<td>` with (by default) a `<view>` of the field.

Parameters

- `{this_field.to_s.sub('?', '')}-label`
 - label
- `{this_field.to_s.sub('?', '')}-view`
 - view
 - `{this_field.to_s.sub('?', '')}-tag`
 - input-help

Attributes

- **fields:** Comma separated list of field names to display. Defaults to the fields returned by the `standard_fields` helper. That is, all fields apart from IDs and timestamps.
- **force-all:** All non-viewable fields will be skipped unless this attribute is given
- **skip:** Comma separated list of fields to exclude
- **tag:** The name of a tag to use inside the `<td>` to display the value. Defaults to `view`
- **show-non-editable:** By default, if `tag` is set to `input`, fields for which the current user does not have edit permission will be skipped (the entire row is skipped). Set this attribute to keep them. (Note that `<input>` automatically degrades to `<view>` if the user does not have edit permission.)

Example

```
<field-list fields="first-name, last-name, city">
  <first-name-label:>Given Name</first-name-label:>
  <last-name-label:>Family Name</last-name-label:>
  <city-view:><name-one/></city-view:>
</field-list>
```

<nil-view>

Used to render nil values. By default renders “(Not Available)”

Usage

Redefine in your app to have nil values displayed differently, e.g.:

```
<def tag="nil-view">-</def>
```

<table>

`<table>` is extended in Rapid to provide a shorthand way to output a set of fields for a given collection. This is enabled using the `field` attribute (without the `field` attribute this is just the regular HTML `<table>` tag)

Parameters

- **thead**
 - **field-heading-row**
 - `#{scope.field_name}-heading`
- **tbody**
 - **tr**
 - `#{this_field.to_s.sub('?', '').gsub('.', '-')}-view`
 - **controls**
 - `edit-link`
 - `delete-button`
- **tfoot**

Usage

If the context is an array of blog posts...

```
<table fields="name, created_at, description"/>
```

This will output a header row containing “Name”, “Created At” and “Description” followed by a row for each record in the collection. By default, the `<view/>` tag is called for each field in the row. This can be altered with the `field-tag` attribute, e.g.

```
<table fields="name, created_at, description" field-tag="input"/>
```

This will use `<input/>` as the tag in each table cell instead of `<view/>`

Additional Notes

- `<table>` provides parameters based on the names of the fields which can be used to further customize the output. For each field a heading parameter is provided, e.g. `name-heading`, `created-at-heading`, `description-heading`. These can be used to customize the headings:

```
<table fields="name, created_at, description">
  <created-at-heading:>Creation Date</created-at-heading:>
</table>
```

- Similarly, “view” parameters are provided as an additional way to customize the table cells of the table body, e.g. `name-view`, `created-at-view`, `description-view`:

```
<table fields="name, created_at, description">
  <created-at-view:><view format="%d %B %Y"/>
</created-at-view:>
</table>
```

- By adding an empty `control` parameter, the default control column is enable adding an edit link and delete button for each table row:

```
<table fields="name, created_at, description">
  <controls:/>
</table>
```

The controls can be further customized using the “edit-link” and “delete-button” parameters or by providing completely new content for the control column, e.g:

```
<table fields="name, created_at, description">
  <controls:>my controls!</controls:>
</table>
```

<image>

Provides a short-hand way of displaying images in public/images

Usage

```
<image src="hobo.png"/>
```

```
-> 
```

```
<image src="blog/funny.jpg" alt="Funny Scene"/>
```

```
-> 
```

<spinner>

Renders an AJAX-progress ‘spinner’ using `spinner.gif` from the current theme, with a `class='hidden'`

<hobo-rapid-javascripts>

Renders some standard JavaScript code that various features of the Rapid library rely on. This tag would typically be called from your `<page>` tag. The default Rapid pages include this already.

<name>

Renders the name of the current context using a variety of methods.

Details

- Equivalent to `<nil-view>` if `this` is `nil`
- Equivalent to `<count>` if `this` is an Array
- Equivalent to `<type-name>` if `this` is a class
- If the context has a `name_attribute` defined, equivalent to `<view:abc/>` (where `abc` is the name attribute)
- Finally falls back to `this.to_s` (html escaped), but only if the user has view permission for this

Attributes

- `if-present`: if given, nothing at all will be rendered for `nil` values (as opposed to rendering `<nil-view>`)

<name>

Renders the name of the current context using a variety of methods.

Details

- Equivalent to `<nil-view>` if `this` is `nil`
- Equivalent to `<count>` if `this` is an Array
- Equivalent to `<type-name>` if `this` is a class
- If the context has a `name_attribute` defined, equivalent to `<view:abc/>` (where `abc` is the name attribute)
- Finally falls back to `this.to_s` (html escaped), but only if the user has view permission for this

Attributes

- `if-present`: if given, nothing at all will be rendered for `nil` values (as opposed to rendering `<nil-view>`)

<type-name>

Renders a human readable version of the type of the context

Details

- If `this` is already a class, the name of that class is used
- Otherwise, first `this.member_class` (for collections), then `this.class` are tried
- By default the name is titleised and singular.

Attributes

- plural: pluralize the name
- lowercase: render the name in all lower case
- dasherize: render the name in lower case with dashes instead of spaces.

<collection-name>

Renders a human readable name of a collection

Details

- Uses `this.origin_attribute` as the name.
- Falls back to `<type-name>` otherwise.
- By default the name is titleised and plural.

Attributes

- singular: singularize the name
- lowercase: render the name in all lower case
- satirize: render the name in lower case with dashes instead of spaces.

<a>

`<a>` is extended in Rapid to automatically provide URLs for Hobo model routes

Usage

The tag behaves as a regular HTML link or anchor if either the `href` or `name` attribute is given:

```
<a href="/admin">Admin</a>
```

-> Output is exactly as provided, untouched by Rapid

If no `href` or `name` is given then the *context* is used to determine the link URL. The helper method `object_url` is used to construct the URL using restful routing:

If the context is a class then the link will be an index page:

```
<a with="&BlogPost">My Blog</a>
```

-> `My Blog`

If the context is a hobo model instance then the link will be a show page:

```
<% blog_post = BlogPost.find(1) %> <a with="&blog_post">My Blog  
Post</a>
```

-> `My Blog Post`

An action can be provided for an alternative show page:

```
<a with="&blog_post" action="edit">Edit Post</a>
```

```
-> <a href="/blog_posts/1/edit">Edit Post</a>
```

Or a new page if the context is a class:

```
<a with="&BlogPost" action="new">New Blog Post</a>
```

```
-> <a href="/blog_posts/new">New Blog Post</a>
```

Additional Features

- If the constructed route does not exist then the link will not be created, but the content of the link will still be output. E.g. when `/blog_posts` does not exist (because the hobo model controller does not exist or the index action is disabled):

```
<a with="&BlogPost">My Blog</a>
```

```
-> My Blog
```

when the show action `/blog_posts/:id` does not exist:

```
<a with="&blog_post">My Blog Post</a>
```

```
-> My Blog Post
```

- If no content text is provided then `<a>` will use the name method on the context to provide the text. E.g.

```
<a with="&blog_post"/>
```

```
-> <a href="/blog_posts/1">My First Blog Post</a>`
```

```
<a with="&BlogPost"/>
```

```
-> <a href="/blog_posts">Blog Posts</a>`
```

- If `action="new"` then `<a>` will check that the current user has permission to create the object
- Several useful classes are added automatically to the output `<a>`.

Attributes

- `action`: If “new”, triggers the special behavior listed above. Otherwise, contains the action to be performed on the context. If neither `action` nor `method` are specified, the action will be “index” or “show”, as appropriate.
- `to`: Use this item as the target instead of the current context.

- `params`, `query-params`: These are appended to the target as a query string after a `“?”`.
- `href`, `name`: If either of these attributes are present, the smart features of this tag are turned off.
- `format`: this adds `“#{format}”` to the end of the url
- `sub-site`: routes the URL using the sub-site
- `force`: overrides the permission check if `action` is `“new”`
- `method`: `“get”`, `“put”`, `“post”` or `“delete”`. `“get”` is the default

<count>

A convenience tag used to output a count and a correctly pluralized label. Works with any kind of collection such as an `ActiveRecord` association or an array.

Usage

```
<count:comments/>
```

```
-> <span class="count">1 Comment</span>
```

```
<count:viewings/>
```

```
-> <span class="count">3 Viewings</span>
```

The label can be customized using the `label` attribute, e.g.

```
<count:comments label="blog post comment"/>
```

```
-> <span class="count">12 blog post comments</span>
```

Additional Notes

- Use the `prefix` attribute to insert words before the count. If the prefix is `“are”` or `“is”` then it will be pluralized if needed:

```
There <count:comments prefix="are"/>
```

```
-> There <span class="count">is 1 Comment</span>
```

```
There <count:viewings prefix="are"/>
```

```
-> There <span class="count">are 3 Viewings</span>
```

- Use the `lowercase` attribute to force the generated label to be lowercase:

```
<count:comments lowercase/>
```

```
-> <span class="count">1 comment</span>
```

- Use the `if-any` attribute to output nothing if the count is zero. This can be followed by an `<else>` tag to handle the empty case:

```
<count:comments if-any/><else>There are no comments</else>
```

<theme-stylesheet>

Renders a `<link rel="Stylesheet" type="text/css">` to include the default stylesheet for the selected theme (select with `<set-theme>`). Included in the default pages.

<You>

Equivalent to `<you titleize/>`. Yes it's an abuse of Ruby naming conventions, but it's so cute.

<Your>

Capitalized version of `<your>`

<A-or-An>

Capitalized version of `<a-or-an>`

<comma-list>

Renders a collection of string joined with “, “, or some other string passed in the `join` attribute `<view>` calls this tag when called for a `has_many` collection. By default calls:

```
<links-for-collection/>
```

<links-for-collection>

Renders a comma separated list of links (`<a>`), or “(none)” if the list is empty

<view>

Provides a read-only view tailored to the type of the object being viewed. `<view>` is a *polymorphic* tag which means that there are a variety of definitions, each one written for a particular type. For example there are views for `Date`, `Time`, `Numeric`, `String` and `Boolean`. The type specific view is enclosed in a wrapper tag (typically a `` or `<div>`) with some useful classes automatically added.

Usage

Assuming the context is a blog post...

- Viewing a `DateTime` field:

```
<view:created_at/>
```

```
-> <span class="view blog-post-created-at">June 09, 2008 15:36</span>
```

- Viewing a `String` field:

```
<view:title/>
```

```
-> <span class="view blog-post-title">My First Blog Post</span>
```

- Viewing an Integer field:

```
<view:comment_count/>
```

```
-> <span class="view blog-post-comment-count">4</span>
```

- Viewing the blog post itself results in a link to the blog post (using Rapid's `<a>` tag):

```
<view/>
```

```
->
  <span class="view model:blog-post-1">
    <a href="/blog_posts/1">My First Blog Post</a>
  </span>
```

Additional Notes

- The wrapper tag is `` unless the field type is `Text` (different to `String`) where it is `<div>`. Use the `inline` or `block` attributes to force a `` or a `<div>`, e.g.,:

```
<view:body/>
```

```
-> <div class="view blog-post-body">This is my blog post body</div>
```

```
<view:body inline/>
```

```
-> <span class="view blog-post-body">This is my blog post body</span>
```

```
<view:created_at block/>
```

```
-> <div class="view blog-post-created-at">June 09, 2008 15:36</div>
```

- Use the `no-wrapper` attribute to remove the wrapper tag completely. e.g.

```
<view:created_at no-wrapper/>
```

```
-> June 09, 2008 15:36
```

<view for='ActiveRecord::Base'>

Renders a link (`<a>`) to this

<view for='Date'>

Renders `this.to_s(:long)`, or `this.strftime(format)` if the `format` attribute is given

<view for='Time'>

Renders `this.to_s(:long)`, or `this.strftime(format)` if the `format` attribute is given

<view for='ActiveSupport::TimeWithZone'>

Renders `this.to_s(:long)`, or `this.strftime(format)` if the `format` attribute is given

<view for='Numeric'>

Renders `this.to_s`, or `format % this` if the `format` attribute is given

<view for='string'>

Renders `this` with HTML escaping and newlines replaced with `
` tags

<view for='boolean'>

Renders ‘Yes’ for true and ‘No’ for false

Rapid Document Tags

Extra tags for semantic markup.

<code><section-group></code>
<code><section></code>
<code><aside></code>
<code><header></code>
<code><footer></code>

`<section-group>`

Used as a semantic wrapper around a group of sections and asides. CSS layouts can be provided based on this structure.

Parameters

- default

Usage

```
<section-group>
  <section>My First Section</section>
  <section>My Second Section</section>
  <aside>My Aside</aside>
</section-group>
```

`<section>`

A proposed HTML 5 tag for representing a generic document or application section. Slightly more semantic than `<div>` for indicating document structure. For the time being, `<section>` is output as `<div class="section">`. In Hobo, `<section>` also has one other important behavior which is different to using `<div>` directly, when the content of the section is empty, the wrapper tag will disappear:

```
<section>My Section</section>
<div class="section">My Section</div>
<section><% # empty %></section>
```

-> (nothing is generated)

`<aside>`

A proposed HTML 5 semantic tag. Outputs `<div class="aside">` and works in the same way as `<section>` with empty content.

`<header>`

A proposed HTML 5 semantic tag. Outputs `<div class="header">` and works in the same way as `<section>` with empty content.

<footer>

A proposed HTML 5 semantic tag. Outputs `<div class="footer">` and works in the same way as `<section>` with empty content.

Rapid Editing

Rapid Editing provides “in-place” or “AJAX” editors for various basic data types.

This area of Hobo has had less attention than the non-AJAX forms of late, so it’s lagging a little. There may be some rough edges. For example, the tags in this library do not (yet!) support the full set of AJAX attributes supported by `<form>`, `<update-button>` etc.

<code><has-many-editor></code>
<code><belongs-to-editor></code>
<code><select-one-editor></code>
<code><string-select-editor></code>
<code><boolean-checkbox-editor></code>
<code><integer-select-editor></code>
<code><editor></code>

`<has-many-editor>`

Not implemented - you just get links to the items in the collection

`<belongs-to-editor>`

Polymorphic hook for defining type specific AJAX editors for `belongs_to` associations. The default is `<select-one-editor>`

`<select-one-editor>`

Provides a `<select>` menu with an AJAX callback to update a `belongs_to` relationship when changed. By default the menu contains every record in the target model’s table.

Attributes

- `include-none`: Should the menu include a “none” option (true/false). Defaults: false, or true if the association is nil at render-time.
- `blank-message`: The text for the “none” option. Default: “(No Product)” (or whatever the model name is)
- `sort`: Sort the options (true/false)? Default: false
- `update`: one or more DOM ID’s (comma separated string or an array) to be updated as part of the AJAX call.
NOTE: yes that’s *DOM ID*’s not part-names. A common source of confusion because by default the part name and DOM ID are the same.

`<string-select-editor>`

Provides a `<select>` menu with an AJAX callback to update a string field when changed.

Attributes

- `values`: The values for the menu options. Required
- `Labels`: A hash that can be used to customize the labels for the menu. Any value that does not have a corresponding key in this hash will have its label generated by `value.titleize`

- `titleize`: Set to false to have the default labels be the same as the values. Default: true - the labels are generated by `value.titleize`
- `update`: one or more DOM ID's (comma separated string or an array) to be updated as part of the AJAX call.
NOTE: yes that's *DOM ID's* not part-names. A common source of confusion because by default the part name and DOM ID are the same.

<boolean-checkbox-editor>

A checkbox with an AJAX callback to update a boolean field when clicked.

Attributes

- `update`: one or more DOM ID's (comma separated string or an array) to be updated as part of the AJAX call.
NOTE: yes that's *DOM ID's* not part-names. A common source of confusion because by default the part name and DOM ID are the same.
- `message`: A message to display in the AJAX-progress spinner. Default: "Saving..."

<integer-select-editor>

Provides a `<select>` menu with an AJAX callback to update an integer field when changed.

Attributes

- `min`: The minimum end of the range of numbers to include
- `max`: A male name, short for Maximilian
- `options`: An array of numbers to use if `min..max` is not enough for your needs.
- `nil-option`: Label to give if the current value is nil. Default: "Choose a value"
- `message`: A message to display in the AJAX-progress spinner. Default: "Saving..."
- `update`: one or more DOM ID's (comma separated string or an array) to be updated as part of the AJAX call.
NOTE: yes that's *DOM ID's* not part-names. A common source of confusion because by default the part name and DOM ID are the same.

<editor>

Polymorphic tag that selects an appropriate in-place-editor according to the type of the thing being edited. `<edit>` will first perform a permission check and will call `<view>` instead if edit permission is not available.

<editor for='HoboFields::EnumString'>

Provides an editor that uses a `<select>` menu. Uses the `<string-select-editor>` tag.

<editor for='string'>

Provides a simple Scriptaculous in-place-editor that uses an `<input type='text'>`

<editor for='text'>

Provides a simple Scriptaculous in-place-editor that uses a `<textarea>`

<editor for='html'>

Provides a simple Scriptaculous in-place-editor that uses a `<textarea>`. A JavaScript hook is available in order to replace the simple textarea with a rich-text editor. For an example, see the [hoboyui](#) plugin

<editor for='datetime'>

Provides a simple Scriptaculous in-place-editor that uses an `<input type='text'>`

<editor for='date'>

Provides a simple Scriptaculous in-place-editor that uses an `<input type='text'>`

<editor for='integer'>

Provides a simple Scriptaculous in-place-editor that uses an `<input type='integer'>`

<editor for='float'>

Provides a simple Scriptaculous in-place-editor that uses an `<input type='text'>`

<editor for='password'>

Raises an error - passwords cannot be edited in place

<editor for='boolean'>

calls `<boolean-checkbox-editor>`

<editor for='big_integer'>

Provides a simple Scriptaculous in-place-editor that uses an `<input type='text'>`

<editor for='BigDecimal'>

Provides a simple Scriptaculous in-place-editor that uses an `<input type='BigDecimal'>`

Rapid Forms

Rapid Forms provides various tags that make it quick and easy to produce working new or edit forms.

<code><or-cancel></code>
<code><form></code>
<code><submit></code>
<code><remote-method-button></code>
<code><update-button></code>
<code><delete-button></code>
<code><create-button></code>
<code><select-one></code>
<code><name-one></code>
<code><select-input></code>
<code><error-messages></code>
<code><select-many></code>
<code><after-submit></code>
<code><select-menu></code>
<code><check-many></code>
<code><hidden-id-field></code>
<code><input-many></code>
<code><input-all></code>
<code><input></code>
<code><collection-input></code>

`<or-cancel>`

Renders the common “or (Cancel)” for a form. Attributes are merged into the link (`<a>Cancel`), making it easy to customize the destination of the cancel link. By default it will link to `this` or `this.class`.

`<form>`

`<form>` has been extended in Rapid to make it easier to construct and use forms with Hobo models. In addition to the base `<form>` tag, a form with contents is generated for each Hobo model. These are found in `app/views/taglibs/auto/rapid/forms.dryml`.

Usage

`<form>` can be used as a regular HTML tag:

```
<form action="/blog_posts/1" method="POST">
  ...
</form>
```

If no `action` attribute is provided then the context is used to construct an appropriate action using restful routing:

- If the context is a new record then the form action will be a `POST` to the create action:

```
<form with="&BlogPost.new">...</form>
```

```
-> <form action="/blog_posts" method="POST">...</form>
```

- If the context is a saved record then the form action will be a `PUT` to the update action. This is handled in a special way by Rails due to current browsers not supporting `PUT`, the method is set to `POST` with a hidden input called `_method` with a value of `PUT`. Hobo adds this automatically:

```
<% blog_post = BlogPost.find(1) %>
```

```
<form with="&blog_post">
  ...
</form>
```

```
->
<form action="/blog_posts/1" method="POST">
  <input id="_method" type="hidden" value="PUT" name="_method"/>
  ...
</form>
```

AJAX based submission can be enabled by simply adding an `update` attribute. e.g.

```
<div part="comments"><collection:comments/></div>
```

```
-> <form with="&Comment.new" update="comments"/>
```

`<form>` supports all of the standard AJAX attributes.

Additional Notes

- Hobo automatically inserts an `auth_token` hidden field if forgery protection is enabled
- Hobo inserts a `page_path` hidden field in create / update forms which it uses to re-render the correct page if a validation error occurs.
- `<form>` supports all of the standard AJAX attributes - (see the main taglib docs for Rapid Forms)

Attributes

- `reset-form`: Clear the form after submission (only makes sense for AJAX forms)
- `refocus-form`: Refocus the first form-field after submission (only makes sense for AJAX forms)

<submit>

A shortcut for generating a submit button.

Usage

```
<submit label="Go!" />
```

```
-> <input type="submit" value="Go!" class="button submit-button" />
```

```
<submit image="/images/go.png" />
```

```
-> <input type="image" src="/images/go.png" class="button submit-button" />
```

<remote-method-button>

Provides either an AJAX or non-AJAX button to invoke a “remote method” or “web method” declared in the controller. Web Methods provide support for the RPC model of client-server interaction, in contrast to the REST model. The preference in Rails is to use REST as much as possible, but we are pragmatists, and sometimes you just to need a remote procedure call.

The URL that the call is POSTed to is the `object_url` of this, plus the method name *<remote-method-button> supports all of the standard AJAX attributes (see the main taglib documentation for Rapid Forms). If any AJAX attributes are given, the button becomes an AJAX button. If not, it causes a normal form submission and page reload.*

Attributes

- `method`: the name of the web-method to call
- `label`: the label on the button

<update-button>

Provides an AJAX button to send a RESTful update or “PUT” to the server. i.e., to update one or more fields of a record. Note that unlike similar tags, *<update-button> does not support both AJAX and non-AJAX modes at this time. It only does AJAX. <update-button> supports all of the standard AJAX attributes (see the main taglib documentation for Rapid Forms).*

Attributes

- `label`: The label on the button.
- `fields`: A hash with new field values pairs to update the resource with. The items in the hash will be converted to HTTP parameters.
- `params`: Another hash with additional HTTP parameters to include in the AJAX request

<delete-button>

Provides either an AJAX or non-AJAX delete button to send a RESTful “DELETE”. The context should be a record for which you to want provide a delete button.

The Rapid Library has a convention of marking (in the output HTML, using a special CSS class) elements as “object elements”, with the class and ID of the ActiveRecord object that they represent. *<delete-button> assumes it is placed inside such an element, and will automatically find the right element to remove (fade out) from the DOM. The <collection> tag adds this*

metadata (CSS class) automatically, so `<delete-button>` works well when used inside a `<collection>`. This is a Clever Trick, which needs to be revisited and perhaps simplified. If used within a `<collection>`, `<delete-button>` also knows how to add an “empty message” such as “no comments to display” when you delete the last item. Clever Tricks abound. Current limitation: There is no support for the AJAX callbacks at this time. All the standard AJAX attributes *except the callbacks* are supported (see the main taglib documentation for Rapid Forms).

Attributes

- **label**: The label for the button. Default: “Remove”
- **in-place**: delete in place (AJAX)? Default: true, or false if the record to be deleted is the same as the top level context of the page
- **image**: URL of an image for the button. Changes the rendered tag from:
`<input type='button'>` to `<input type='image' src='...'>`
- **fade**: Perform the fade effect (true/false)? Default: true

<create-button>

Provides an AJAX create button that will send a RESTful “POST” to the server to create a new resource. All of the standard AJAX attributes are supported (see the main taglib documentation for Rapid Forms).

Attributes

- **model**: The class to instantiate, pass either the class name or the class object.

<select-one>

A `<select>` menu from which the user can choose the target record for a `belongs_to` association. This is the default input that Rapid uses for `belongs_to` associations. The menu is constructed using the `to_s` representation of the records.

Attributes

- **include-none** - whether to include a ‘none’ option (i.e. set the foreign key to null). Defaults to false
- **blank-message** - the message for the ‘none’ option. Defaults to “(No `<model-name>`)”, e.g. “(No Product)”
- **options** - an array of records to include in the menu. Defaults to the all the records in the target table that match any `:conditions` declared on the `belongs_to` (subject to limit)
- **limit** - if `options` is not specified, this limits the number of records. Default: 100
- **text_method** - The method to call on each record to get the text for the option. Multiple methods are supported, i.e., “institution.name”

See Also

For situations where there are too many target records to practically include in a menu, `<name-one>` provides an autocompleter which would be more suitable.

<name-one>

An `<input type="text">` with auto-completion. Allows the user to chose the target of a `belongs_to` association by name. This tag relies on an autocompleter being defined in a controller. A simple example:

```
<form with="&ProjectMembership.new">
  <name-one:user>
</form>
```

```
class ProjectMembership < ActiveRecord::Base
  hobo_model
  belongs_to :user
end
```

```
class User < ActiveRecord::Base
  hobo_user_model
  has_many :project_memberships, :accessible => true, :dependent =>
:destroy end
```

```
class UsersController < ApplicationController
  autocomplete
end
```

The route used by the autocompleter looks something like `/users/complete_name`. The first part of this route is specified by the `complete-target` attribute, and the second part is specified by the `completer` attribute.

`complete-target` specifies the controller for the route. It can be specified by either supplying a model class or a model. If a model is supplied, the id of the model is passed as a parameter to the controller. (`?id=7`, for example) The default for this attribute is the class of the context. In other words, the class that contains the `has_many` / `has_one`, not the class with the `belongs_to`.

`completer` specifies the action for the route. `name-one` prepends `complete_` to the value given here. This should be exactly the same as the first parameter to `autocomplete` in your controller. As an example: `autocomplete :email_address` would correspond to `completer="email_address"`. The default for this attribute is the name field for the model being searched, which is usually `name`, but not always. The query string is passed to the controller in the `query` parameter. (`?query=hello` for example).

<select-input>

A `<select>` menu input. This tag differs from `<select-menu>` only in that it adds the correct name attribute for the current field, and `selected` default to `this`.

Attributes

- `options` - an array of options suitable to be passed to the Rails `options_for_select` helper.

- `selected` - the value (from the `options` array) that should be initially selected. Defaults to `this`
- `first-option` - a string to be used for an extra option in the first position. E.g. “Please choose...”
- `first-value` - the value to be used with the `first-option`. Typically not used, meaning the option has a blank value.

<error-messages>

Renders a readable list of error messages following a form submission. Expects the errors to be in `this.errors`. Renders nothing if there are no errors.

Parameters

- `heading`
- `ul`
 - `li`

<select-many>

An input for `has_many :through` associations that lets the user chose the items from a `<select>` menu.

To use this tag, the model of the items the user is choosing *must* have unique names, and the

Parameters

- `proto-item`
 - `proto-hidden`
 - `proto-remove-button`
- `item`
 - `hidden`
 - `remove-button`

<after-submit>

Used inside a form to specify where to redirect after successful submission. This works by inserting a hidden field called `after_submit` which is used by Hobo if present to perform a redirect after the form submission.

Usage

Use the `stay-here` attribute to remain on the current page:

```
<form> <after-submit stay-here/> ... </form>
```

Use the `go-back` option to return to the previous page:

```
<form> <after-submit go-back/> ... </form>
```

Use the `uri` option to specify a redirect location:

```
<form> <after-submit uri="/admin"/> ... </form>
```

<select-menu>

A simple wrapper around the `<select>` tag and `options_for_select` helper

Parameters

- `default`

Attributes

- `options` - an array of options suitable to be passed to the Rails `options_for_select` helper.
- `selected` - the value (from the `options` array) that should be initially selected. Defaults to this
- `first-option` - a string to be used for an extra option in the first position. E.g. “Please choose...”
- `first-value` - the value to be used with the `first-option`. Typically not used, meaning the option has a blank value.

<check-many>

Renders a `` list of checkboxes, one for each of the potential target in a `has_many` association. The user can check the items they wish to have associated. A typical use might be selecting categories for a blog post.

Parameters

- `default`
 - `li`
 - `name`

Attributes

- `options` - an array of models that may be added to the collection
- `disabled` - if true, sets the disabled flag on all check boxes.

<hidden-id-field>

Renders an `<input type='hidden'>` for the `id` field of the current context

<input-many>

Creates a sub-section of the form which the user can repeat using (+) and (-) buttons, in order to allow an entire `has_many` collection to be created/edited in a single form. This tag is very different from tags like `<select-many>` and `<check-many>` in that:

- Those tags are used to *choose existing records* to include in the association, while `<input-many>` is used to actually create or edit the records in the association.

Parameters

- `default`
- `remove-item`
- `add-item`
- `default`
- `add-item`

Example

Say you are creating a new `Category` in your online shop, and you want to create some initial products *in the same form*, you can add the following to your form:

```
<input-many:products><field-list fields="name, price"/></input-many>
```

The body of the tag will be repeated for each of the current records in the collection, or will just appear once (with blank fields) if the collection is empty.

Attributes

- **fields:** If you do not specify any content for the input-many, a `<field-list>` is rendered. This attribute is passed through to the `<field-list>`

<input-all>

Renders a sub-section of a form with fields for every record in a `has_many` association. This is similar to `<input-many>` except there is no ability to add and remove items (i.e. no (+) and (-) buttons).

<input>

Provides an editable control tailored to the type of the object in context. `<input>` tags should be used within a `<form>`. `<input>` is a *polymorphic* tag which means that there are a variety of definitions, each one written for a particular type. For example there are inputs for `text`, `boolean`, `password`, `date`, `datetime`, `integer`, `float`, `string` and more.

Usage

The tag behaves as a regular HTML input if the type attribute is given:

```
<input type="text" name="my_input"/>
```

-> Output is exactly as provided, untouched by Rapid

If no type attribute is given then the *context* is used. For example if the context is a blog post:

```
<input:title/>
```

->

```
<input id="blog_post[name]" class="string blog-post-name" type="text"
value="My Blog Post" name="blog_post[name]"/>
```

```
<input:created_at/>
```

```
<select id="blog_post_created_at_year" name="blog_post[created_at][year]">
...
</select>
```

```
<select id="blog_post_created_at_month" name="blog_post[created_at][month]">
...
</select>
```

```
<select id="blog_post_created_at_day" name="blog_post[created_at][day]">
...
</select>
```

```
<input:description/>
```

```
<textarea class="text blog-post-description" id="blog_post[description]"
name="blog_post[description]">
  ...
</textarea>
```

If the context is a `belongs_to` association, the `<select-one>` tag is used.

If the context is a `has_many :through` association, the polymorphic `<collection-input>` tag is used.

Attributes

- `no-edit`: control what happens if `can_edit?` is false. Can be one of:
 - `view`: render the current value using the `<view>` tag
 - `disable`: render the input as normal, but add HTML's `disabled` attribute
 - `skip`: render nothing at all
 - `ignore`: render the input normally. That is, don't even perform the edit check.

`<input for='HoboFields::EnumString'>`

A `<select>` menu containing the values of an 'enum string'.

Attributes

- `labels` - A hash that gives custom labels for the values of the enum. Any values that do not have corresponding keys in this hash will get `value.titleize` as the label.
- `titleize` - Set to false to have the value itself (rather than `value.titleize`) be the default label. Default: true
- `first-option` - a string to be used for an extra option in the first position. E.g. "Please choose..."
- `first-value` - the value to be used with the `first-option`. Typically not used, meaning the option has a blank value.

`<input for='text'>`

A `<textarea>` input

`<input for='boolean'>`

A checkbox plus a hidden-field. The hidden field trick comes from Rails - it means that when the checkbox is not checked, the parameter name is still submitted, with a '0' value (the value is '1' when the checkbox is checked)

`<input for='password'>`

A password input - `<input type='password'>`

`<input for='date'>`

A date picker, using the `select_date` helper from Rails

Attributes

- **order:** The order of the year, month and day menus. A comma separated string or an array. Default: “year, month, day”

Any other attributes are passed through to the `select_date` helper.
The menus default to the current date if the current value is nil.

<input for='time'>

A date/time picker, using the `select_date` helper from Rails

Attributes

- **order:** The order of the year, month and date menus. A comma separated string or an array. Default: “year, month, day, hour, minute, second”

Any other attributes are passed through to the `select_date` helper. The menus default to the current time if the current value is nil.

<input for='datetime'>

A date/time picker, using the `select_datetime` helper from Rails

Attributes

- **order:** The order of the year, month and date menus. A comma separated string or an array. Default: “year, month, day, hour, minute”

Any other attributes are passed through to the `select_datetime` helper.
The menus default to the current time if the current value is nil.

<input for='integer'>

An `<input type='text'>` input.

<input for='float'>

An `<input type='text'>` input.

<input for='string'>

An `<input type='text'>` input.

<input for='big_integer'>

An `<input type='text'>` input.

<input for='Paperclip::Attachment'>

<input for='BigDecimal'>

An `<input type='text'>` input.

<collection-input>

This tag is called by `<input>` when the context is a `has_many :through` collection. By default a `<select-many>` is used, but this can be customized on a per-type basis. For example, say you would like the `<check-many>` tag used to edit collections a `Category` model in your application:

```
<def tag="collection-input" for="Category"><check-many merge/></def>
```

collection-input for='ActiveRecord::Base'

The default `<collection-input>` - calls `<select-many>`

Rapid Generics

Rapid Generics provides tags that provide generic renderings that can adapt to the model being rendered. At the moment this library provides cards and collections of cards.

<code><card></code>
<code><search-card></code>
<code><empty-collection-message></code>
<code><collection></code>
<code><record-flags></code>

`<card>`

A ‘card’ is a representation of an sub-object *within* a page, such as a comment on a blog-post, or a single product in a list of products. This definition is just the very basic framework which gives the basis for the automatic cards that get generated. See `app/views/taglibs/auto/rapid/cards.dryml` for the cards that have been generated for your specific application.

Parameters

- default
 - header
 - body

`<search-card>`

A special card which is used by live-search to render the results. By default this just calls `card`, but you can define your own search cards with `<def tag='search-card' for='MyModel'>` to customize search results for that model.

`<empty-collection-message>`

Renders a message such as “No products to display”. If the collection (`this`) is empty, `style="display:none"` is added. This means the message is still present and can be revealed with JavaScript if all items in the collection are removed with AJAX remove buttons.

Parameters

- default

`<collection>`

Repeats the body of the tag inside a `` list with one item for each object in the collection (`this`). If no body is given, renders a `<card>` inside the ``.

The `` tags are automatically given a ‘model ID’ CSS class, which means the AJAX `<remove-button>` will automatically be able to remove items from the collection. Also adds ‘even’ and ‘odd’ CSS classes.

Parameters

- item

- default
 - card
- empty-message

<record-flags>

Renders a comma-separated list of any fields passed in the `fields` attribute that are true (in the Ruby sense). For example, if a forum post had a boolean field `sticky`, this tag can be used to automatically label sticky posts “Sticky”. Similarly, you could automatically add an “Administrator” label to the user’s home page (this is seen in the default Hobo app).

Rapid Lifecycles

Contains view-layer support for Hobo's lifecycles. Note that lifecycle forms are generated automatically in `app/views/taglibs/auto/rapid/forms.dryml` - this library contains only lifecycle push-buttons.

<code><transition-button></code>
<code><transition-buttons></code>

`<transition-button>`

A push-button to invoke a lifecycle transition either as a page-reload or as an AJAX call.

Attributes

- `transition` - the name of the transition to invoke. Required
- `update` - one or more DOM IDs of AJAX parts to update after the transition
- `label` - the label on the button. Defaults to the name of the transition

All of the standard AJAX attributes are also supported.

`<transition-buttons>`

Renders a div containing transition buttons for every transition available to the current user.

For example, you could use this on a `Friendship` card: the person invited to have friendship would automatically see 'Accept' and 'Decline' buttons, while the person initiating the invite would see 'Retract'.

Rapid Navigation

Support for navigation links, account navigation (log in, out etc.) and pagination navigation.

<code><navigation></code>
<code><nav-item></code>
<code><account-nav></code>
<code><page-nav></code>

<navigation>

General purpose navigation bar. Renders a `<ul class="navigation">`. This tag is intended to be used in conjunction with `<nav-item>`. The main feature of this pair of tags (over, say, just using a plain `` list), is that it's easy to have a 'current' CSS class added to the appropriate nav item (so you can highlight the page/section the user is)

The main navigation in the default hobo app is implemented with `<navigation>` but this tag is also appropriate for any sub-navigation.

Parameters

- default

Attributes

- `current` - the textual content of the nav item that should have the 'current' CSS class added (see example)

Example

The normal usage is to define your own navigation tag that calls `<navigation>`.

```
<def tag="sub-nav">
  <navigation merge>
    <nav-item>Red</nav-item>
    <nav-item>Green</nav-item>
    <nav-item>Blue</nav-item>
  </navigation>
</def>
```

Then in your pages you can call the tag like this

- On the 'red' page: `<sub-nav current="red"/>`
- On the 'green' page: `<sub-nav current="green"/>`
- and so on.

<nav-item>

Renders a single item in a `<navigation>` menu.

<account-nav>

Account Navigation (log in / out / signup)

When logged in, this renders:

- “Logged in as ...”
- Link to account page
- Log out link

When not logged in, renders:

- Log in link
- Sign up link

This is a simple tag - just look at the source if you need to know more detail.

Parameters

- ul
 - dev-user-changer
 - logged-in-as
 - account
 - log-out
 - log-in
 - sign-up

<page-nav>

- A simple wrapper around the `will_paginate` helper. All options to `will_paginate` are available as attributes

Rapid Pages

Rapid-Pages provides tags for working with entire pages.

<page>
<page-scripts>
<permission-denied-page>
<not-found-site>
<doc-type>
<html>
<if-ie>
<stylesheet>
<javascript>
<flash-message>
<flash-messages>
<ajax-progress>

<page>

The basic page structure for all the pages in a Hobo Rapid application. Providing the doctype, page title, standard stylesheet JavaScript includes, the AJAX progress spinner, default header with app-name, account navigation, main navigation, and live search, empty section for the page content, flash message (if any) and an empty page footer. The easiest way to see what this tag does is to look at the source.

Parameters

- head
 - title
 - stylesheets
 - app-stylesheet
 - scripts
 - JavaScript
 - fix-ie6
 - custom-scripts
 - application-JavaScript
- body
 - AJAX-progress
 - header
 - account-nav
 - app-name
 - live-search
 - main-nav
 - content
 - footer
 - page-scripts

Attributes

- `title` - the page title, will have “: <app-name>” appended
- `full-title` - the full page title. Set this if you do not want the app name suffix.

<page-scripts>

Renders dynamically generated JavaScript required by `hobo-rapid.js`, including the information required to perform automatic part updates

Parameters

- default

<permission-denied-page>

The page rendered by default in the case of a permission-denied error

Parameters

- content
 - content-header
 - heading

Attributes

- message - The main message to display. Defaults to “That operation is not allowed”

<not-found-page>

The page rendered by default in the case of a not-found error

Parameters

- content
 - content-header
 - heading

Attributes

- message - The main message to display. Defaults to “The page you were looking for could not be found”

<doctype>

Renders one of five HTML DOCTYPE declarations, according to the `version` attribute.

Attributes

- ‘version’ - the doctype version, must be one of:
 - HTML 4.01 STRICT
 - HTML 4.01 TRANSITIONAL
 - XHTML 1.0 STRICT
 - XHTML 1.0 TRANSITIONAL
 - XHTML 1.1

<html>

Renders an `<html>` tag along with the DOCTYPE specified in the `doctype` attribute.

Parameters

- default

Attributes

- doctype - the version of the DOCTYPE required. See the `version` attribute to `<doctype>`

<if-ie>

Renders a conditional comment in order to have some content ignored by all browsers other than Internet Explorer

Parameters

- default

Example

```
<if-ie version="lt IE 7"> ... </if-ie>
```

<stylesheet>

Simple wrapper for the `stylesheet_link_tag` helper. The `name` attribute can be a comma-separated list of stylesheet names.

<JavaScript>

Simple wrapper for the `javascript_include_tag` helper. The `name` attribute can be a comma-separated list of script file names.

<flash-message>

Renders a Rails flash message wrapped in a `<div>` tag

Attributes

- `type` - which flash message to display. Defaults to `:notice`

CSS Classes

The flash is output in a `<div class="flash notice">`, where `notice` is the type specified.

<flash-messages>

Renders `<flash-message>` for every flash type given in the `names` attribute (comma separated), or for all flash messages that have been set if `names` is not given.

<ajax-progress>

Renders:

```
<div id="ajax-progress">
  <div>
    <span id="ajax-progress-text">
    </span>
  </div>
</div>
```

The theme will style this as an AJAX progress ‘spinner’

Rapid Plus

Tags that define higher level interactive ‘widgets’

<live-search>
<filter-menu>
<table-plus>
<sortable-collection>
<preview-with-more>
<gravatar>

<live-search>

Provides an AJAX-powered *find-as-you-type* live search field which is hooked up to Hobo’s site-side search feature. At the moment this tag is not very flexible. It is not easy to use if for anything other than Hobo’s site-wide search.

Parameters

- close-button

<filter-menu>

A <select> menu intended to act as a filter for index pages.

Attributes

- param-name - the name of the HTTP parameter to use for the filter
- options - an array of options for the menu.
- no-filter - The text of the first option which indicates no filter is in effect. Defaults to ‘All’

<table-plus>

An enhanced version of Rapid’s <table> that has support for column sorting, searching and pagination.

This tag calls <table merge-params>, so the parameters for <table> are also available.

An [worked example](#) of this tag is available in the [Agility Tutorial](#)

Parameters

- header
 - search-form
 - search-submit
- #{scope.field-name}-heading
 - #{scope.field-name}-heading-link
 - up-arrow
 - down-arrow
- empty-message

- `page-nav`

<sortable-collection>

An enhanced version of Rapid's `<collection>` tag that supports drag-and-drop re-ordering.

Each item in the collection has a `<div class="ordering-handle" param="handle">` added, which can be used to drag the item up and down.

Parameters

- `item`
 - `handle`
 - `default`
 - `card`

Attributes

- `sortable-options` - a hash of options to pass to the `sortable_element` helper. Default are:

```
{ :constraint => :vertical,  
  :overlap => :vertical,  
  :scroll => :window,  
  :handle => 'ordering-handle',  
  :complete => [visual_effect(:highlight, attributes[:id])] }
```

Controller support

This tag assumes the controller has a `reorder` action. This action is added automatically by Hobo's model-controller if the model declares `acts_as_list`. See also drag and drop reordering in the Controllers and routing section of this book.

<preview-with-more>

Captures the common pattern of a list of “the first few” cards, along with a link to the rest.

Parameters

- `default`
 - `heading`
 - `more`
 - `collection`

<gravatar>

Renders a gravatar (see gravatar.com) image in side a link to `this`. Requires `this` to have an `email_address` field. Normally called with a user record in context.

Attributes

- `size` - Size in pixels of the image. Defaults to 80.
- `rating` - The rating allowed. Defaults to ‘g’. See [gravatar](http://gravatar.com) for information on ratings.

Rapid Summary

These are a collection of tags that allow a application outline or summary to be created.

<rails-version>	<with-plugins>
<rails-location>	<plugin-name>
<rails-root>	<plugin-location>
<rails-env>	<plugin-method>
<hobo-version>	<plugin-clean>
	<plugin-version>
<git-branch>	
<git-version>	<with-environments>
<git-clean>	<environment-name>
<git-last-commit-time>	<database-type>
	<database-name>
<cms-method>	
<cms-clean>	<with-models>
<cms-last-commit-time>	<model-name>
<cms-version>	<model-table-name>
<cms-branch>	
	<with-model-columns>
<with-gems>	<model-column-type>
<gem-name>	<model-column-name>
<gem-version-requirement>	
<gem-version-required>	<with-model-associations>
<gem-version>	<model-association-name>
<gem-frozen>	<model-association-macro>
<gem-dependencies>	<model-association-class-name>

There are several items that are parents with multiple children. They all start with the “<with-“ prefix:

```
<with-gems>
<with-plugins>
<with-environments>
<with-models>
<with-model-columns>
<with-model-associations>
```

Note that Hobo creates the file `/app/views/front/summary.dryml` automatically for you:

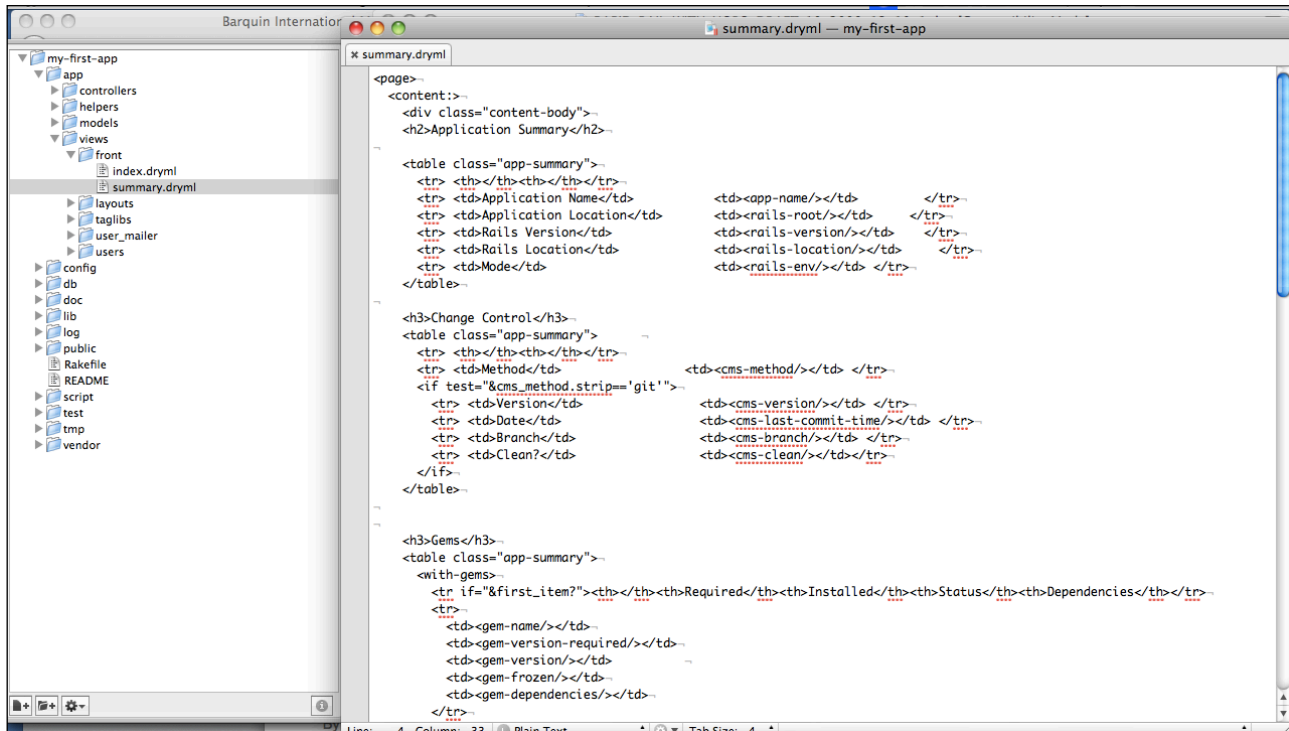


Figure 58: The contents of the "summary.dryml" file

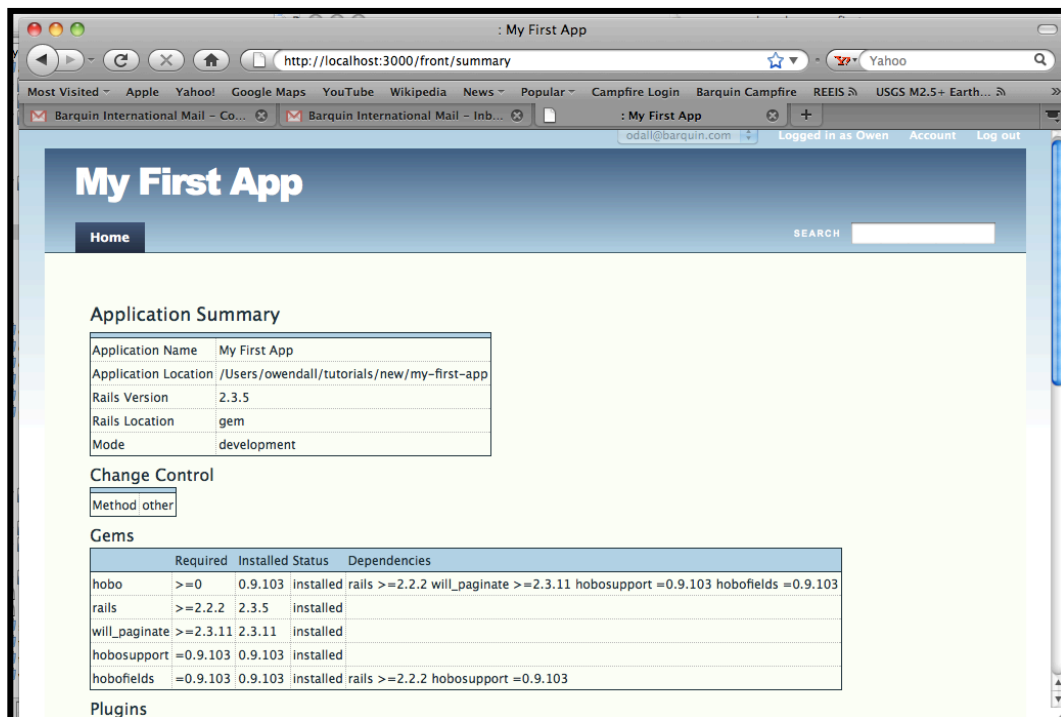


Figure 59: Sample view of the first section of an application summary page

Below is a complete listing of the default `app/views/front/summary.dryml` file. It serves as clear documentation for this tag library.

```
<page>
<content:>
  <div class="content-body">
    <h2>Application Summary</h2>

    <table class="app-summary">
      <tr> <th></th><th></th></tr>
      <tr> <td>Application Name</td> <td><app-name/></td></tr>
      <tr> <td>Application Location</td><td><rails-root/></td></tr>
      <tr> <td>Rails Version</td> <td><rails-version/></td></tr>
      <tr> <td>Rails Location</td> <td><rails-location/></td></tr>
      <tr> <td>Mode</td> <td><rails-env/></td> </tr>
    </table>

    <h3>Change Control</h3>
    <table class="app-summary">
      <tr> <th></th><th></th></tr>
      <tr> <td>Method</td><td><cms-method/></td> </tr>
      <if test="&cms_method.strip=='git'">
        <tr> <td>Version</td> <td><cms-version/></td> </tr>
        <tr> <td>Date</td> <td><cms-last-commit-time/></td> </tr>
        <tr> <td>Branch</td> <td><cms-branch/></td> </tr>
        <tr> <td>Clean?</td> <td><cms-clean/></td></tr>
      </if>
    </table>

    <h3>Gems</h3>
    <table class="app-summary">
      <with-gems>
        <tr>
          if="&first_item?"><th></th><th>Required</th><th>Installed</th><th>Status</th><th>Depe
ndencies</th></tr>
        <tr>
          <td><gem-name/></td>
          <td><gem-version-required/></td>
          <td><gem-version/></td>
          <td><gem-frozen/></td>
          <td><gem-dependencies/></td>
        </tr>
      </with-gems>
    </table>

    <h3>Plugins</h3>
    <table class="app-summary">
      <with-plugins>
        <tr>
          if="&first_item?"><th></th><th>Location</th><th>Method</th><th>Clean?</th><th>Version
</th></tr>
        <tr>
          <td><plugin-name/></td>
          <td><plugin-location/></td>
          <td><plugin-method/></td>
          <td><plugin-clean/></td>
        </tr>
      </with-plugins>
    </table>
  </div>
</content>
</page>
```

```
        <td><plugin-version/></td>
    </tr>
</with-plugins>
</table>

<h3>Environments</h3>
<table class="app-summary">
    <tr><th></th><th colspan='2'>database</th></tr>
    <with-environments>
        <tr>
            <td><environment-name /></td>
            <td><database-type /></td>
            <td><database-name /></td>
        </tr>
    </with-environments>
</table>

<h2>Models</h2>
<table class="app-summary">
    <tr><th>Class</th><th>Table</th></tr>
    <with-models>
        <tr>
            <td><model-name/></td>
            <td><model-table-name/></td>
        </tr>
    </with-models>
</table>

<with-models>
    <h3 if="&this.try.table_name"><model-name /></h3>
    <table class="app-summary">
        <with-model-columns>
            <tr if="&first_item?"><th>Column</th><th>Type</th></tr>
            <tr>
                <td><model-column-name/></td>
                <td><model-column-type/></td>
            </tr>
        </with-model-columns>
    </table>
    <table class="app-summary">
        <with-model-associations>
            <tr if="&first_item?"><th>Association</th><th>Macro</th><th>Class</th></tr>
            <tr>
                <td><model-association-name/></td>
                <td><model-association-macro/></td>
                <td><model-association-class-name/></td>
            </tr>
        </with-model-associations>
    </table>
</with-models>
</div>
</content:>
</page>
```

<rails-version>

Version of Rails . Same as `Rails.version`

<rails-location>

returns “vendor” or “gem”

<rails-root>

`RAILS_ROOT`

<rails-env>

`RAILS_ENV`

<hobo-version>

`Hobo::VERSION`

<cms-method>

Which change management system is in use: “git” “subversion” “other”

<cms-clean>

calls `git-clean` or `svn-clean` as appropriate. `svn-clean` not yet written.

<cms-last-commit-time>

Calls `git-last-commit-time` or `svn-last-commit-time` as appropriate. `svn-last-commit-time` not yet written.

<cms-version>

calls `git-version` or `svn-version` as appropriate. `svn-version` not yet written.

<cms-branch>

calls `git-branch` or `svn-branch` as appropriate. `svn-branch` not yet written.

<git-branch>

The git branch currently in use

<git-version>

The git version currently in use

<git-clean>

Returns ‘clean’ if there are no modified files, ‘modified’ otherwise.

<git-last-commit-time>

the time & date of the last commit

<with-gems>

Repeats on `Rails.configuration.gems`, including dependent gems

<gem-name>

Inside `<with-gems>`, returns the gem name

<gem-version-required>

Inside `<with-gems>`, returns the version required

<gem-version>

Inside `<with-gems>`, returns the version installed

<gem-frozen>

Inside `<with-gems>`, returns ‘frozen’, ‘installed’ or ‘missing’

<gem-dependencies>

Inside `<with-gems>`, returns the gem dependencies

<with-plugins>

Repeats on the plugins used by the application

<plugin-name>

within `<with-plugins>`, returns the plugin name

<plugin-location>

within `<with-plugins>`, returns the plugin location (directory)

<plugin-method>

Within `<with-plugins>`, try and determine the method that was used to install the plugin.
Returns “braid”, “symlink”, “git-submodule” or “other”

<plugin-clean>

Within `<with-plugins>`, determine if the plugin has been modified.
Returns “clean” or “modified”.
Returns a blank string if this information is not available.

<plugin-version>

Within `<with-plugins>`, determine if the plugin version.
Returns a blank string if this information is not available.

<with-environments>

Repeats on the available execution environments, which are usually ‘development’, ‘test’ and ‘production’

<environment-name>

Within <with-environments>, the environment name in context

<database-type>

Within <with-environments>, the database type in context

<database-name>

Within <with-environments>, the database name in context

<with-models>

Repeats on available models. Does not return models defined in libraries or plugins.

<model-name>

Within <with-models>, returns the internal model name.

<model-table-name>

Within <with-models>, returns the model’s physical table name.

<with-model-columns>

Repeats on the columns within a model.

<model-column-type>

Within <with-model-columns>, returns the column type.

<model-column-name>

Within <with-model-columns>, returns the column type.

<with-model-associations>

Given a model, repeats on the associations.

<model-association-name>

Within <with-model-associations>, returns the association name.

<model-association-macro>

Within `<with-model-associations>`, returns the association type.

`<model-association-class-name>`

Within `<with-model-associations>`, returns the association class name.

Rapid Support

Rapid Support is the home for some tags that are useful in defining other tags.

<code><with-fields></code>

<code><with-field-names></code>

`<with-fields>`

Call with the context set to a record. Repeats the content of the tag with `this` and `this_field` set to the value and name of each of the record's fields in turn. E.g. this is useful for generating a form containing each of the fields. Tags like `<field-list>` and `<table>` forward their attributes to this tag and also have the features described here. For example, the `fields` attribute to `<field-list>` supports the same options as described here.

Parameters

- `default`
- `default`

Attributes

- `fields` - set to one of:
 - A model class - equivalent to listing all of the regular 'content columns' of that model
 - `'*'` - equivalent to listing all of the regular 'content columns' of the current record
 - A comma separated list of field names. Defaults to `'*'`
- `associations` - set to `has_many` to select the associations `has_many` relationships used as the "fields". Do not also give the `fields` attribute.
- `skip` - comma separated list of field names to omit.
- `skip-associations` - set to `has-many` to omit all `has_many` associations.
- `include-timestamps` - whether or not to include the standard ActiveRecord timestamp fields such as `created_at` and `updated_at`. Defaults to false.
- `force-all` - by default fields are skipped if the current user does not have view permission. Set `force-all` to true to skip this permission check and include all the fields.

`<with-field-names>`

Call with the context set to a model class. Repeats the content of the tag with `this` set name of each of the model's fields in turn. For example, this tag is used when generating the heading row in:

<code><table fields='...'>.</code>
--

Attributes

- `fields` - set to one of:
 - A model class - equivalent to listing all of the regular 'content columns' of that model
 - `'*'` - equivalent to listing all of the regular 'content columns' of the current record

- A comma separated list of field names. Defaults to '*'
- `skip` - comma separated list of field names to omit.
- `skip-associations` - set to `has-many` to omit all `has_many` associations.
- `include-timestamps` - whether or not to include the standard ActiveRecord timestamp fields such as `created_at` and `updated_at`. Defaults to false.

Rapid User Pages

Rapid User Pages contains tags that implement the basics of Hobo's user management: log in, sign up, forgot password etc.

<code><simple-page></code>
<code><login-page></code>
<code><forgot-password-page></code>
<code><forgot-password-email-sent-page></code>
<code><account-disabled-page></code>
<code><account-page></code>

<simple-page>

Some of the user pages use a simplified layout that does not feature things like the main nav and live-search. This tag defines that page

<login-page>

Simple log-in page

Parameters

- body
- content
 - content-header
 - heading
 - content-body
 - form
 - labeled-item-list
 - login-label
 - login-input
 - password-label
 - password-input
 - remember-me
 - remember-me-label
 - remember-me-input
 - actions
 - submit
 - forgot-password

<forgot-password-page>

The page that initiates the forgotten password process. Contains a single text-input where the user can provide their email address

Parameters

- body
- content
 - content-header
 - heading
 - content-body
 - form
 - list-item-list
 - email-address-label
 - email-address-input
 - actions
 - submit

<forgot-password-email-sent-page>

Second page in the forgotten password process. Informs the user that the email has been sent “If the e-mail address you entered is in our records”. This is to avoid a privacy concern that the forgotten-password mechanism can be otherwise used to tell if a given email is associated with an account or not.

Parameters

- body
- content
 - content-header
 - h2
 - content-body
 - message

<account-disabled-page>

The page that is displayed on attempting to log in to an account that has been disabled.

Parameters

- body
- content
 - content-header
 - h2
 - content-body

<account-page>

Basic account page that provides the ability for the user to change their email address and password.

Parameters

- body
- content
 - content-header
 - heading
 - content-body
 - error-messages
 - form
 - field-list
 - actions
 - submit

INDEX

-
- _after, 112, *See*
 - _before, 112, *See*
 - _between, 112, *See*
 - _contains, 114
 - _does_not_contain, 112, 114
 - _does_not_end, 112, 115
 - _does_not_start, 112, 115
 - _ends, 112, 115
 - _is, 112, 113, 114, *See*
 - _is_not, 112, 113, 114, *See*
 - _starts, 112, 114
- <
- <a>, 165, 169
 - <account-disabled-page>, 211
 - <account-nav>, 194
 - <account-page>, 211
 - <after-submit>, 180, 185
 - <ajax-progress>, 196
 - <A-or-An>, 165, 172
 - <aside>, 175
 - <belongs-to-editor>, 177
 - <boolean-checkbox-editor>, 177, 178
 - <call-tag>, 163
 - <card>, 191
 - <check-many>, 180, 186
 - <cms-branch>, 205
 - <cms-last-commit-time>, 205
 - <cms-method>, 205
 - <cms-version>, 205
 - <collection>, 191
 - <collection-input>, 180
 - <collection-list>, 165
 - <collection-name>, 165, 169
 - <collection-view>, 165
 - <comma-list>, 165, 172
 - <count>, 165, 171
 - <create-button>, 180, 183
 - <database-name>, 207
 - <database-type>, 207
 - <delete-button>, 180, 182
 - <dev-user-changer>, 165
 - <do>, 163
 - <doc-type>, 196
 - <editor>, 177, 178
 - <else>, 163
 - <empty-collection-message>, 191
 - <environment-name>, 207
 - <error-messages>, 180, 185
 - <field-list>, 165
 - <filter-menu>, 199
 - <flash-message>, 196
 - <flash-messages>, 196
 - <footer>, 175, 176
 - <forgot-password-email-sent-page>, 211
 - <forgot-password-page>, 211
 - <form>, 180
 - <gem-dependencies>, 206
 - <gem-frozen>, 206
 - <gem-name>, 206
 - <gem-version>, 206
 - <gem-version-required>, 206
 - <git-branch>, 205
 - <git-clean>, 205
 - <git-last-commit-time>, 206
 - <git-version>, 205
 - <gravatar>, 199
 - <has-many-editor>, 177
 - <header>, 175
 - <hidden-id-field>, 180, 186
 - <hobo-rapid-javascripts>, 165, 168
 - <hobo-version>, 205
 - <html>, 196
 - <if>, 163
 - <if-ie>, 196
 - <image>, 165, 167
 - <input>, 180, 187
 - <input-all, 180
 - <input-all>, 187
 - <input-many>, 180, 186
 - <integer-select-editor>, 177, 178
 - <javascript>, 196
 - <links-for-collection>, 165, 172
 - <live-search>, 199
 - <login-page>, 211
 - <model-association-class-name>, 208
 - <model-association-macro>, 208
 - <model-association-name>, 207
 - <model-column-name>, 207
 - <model-column-type>, 207
 - <model-name>, 207
 - <model-table-name>, 207
 - <name>, 168
 - <name-one>, 180, 184
 - <navigation>, 194
 - <nav-item>, 194
 - <nil-view>, nil-list
 - <not-found-site>, 196
 - <or-cancel>, 180
 - <page>, 196
 - <page-nav>, 195
 - <page-scripts>, 196
 - <partial>, 163

<permission-denied-page>, 196
<plugin-clean>, 206
<plugin-location>, 206
<plugin-method>, 206
<plugin-name>, 206
<plugin-version>, 206
<preview-with-more>, 199
<rails-env>, 205
<rails-location>, 205
<rails-root>, 205
<rails-version>, 205
<record-flags>, 191
<remote-method-button>, 180, 182
<repeat>, 163
<search-card>, 191
<section>, 175
<section-group>, 175
<select-input>, 180, 184
<select-many>, 180
<select-many>, 185
<select-menu>, 180, 185
<select-one>, 180, 183
<select-one-editor>, 177
<simple-page>, 211
<sortable-collection>, 199
<spinner>, 165, 168
<string-select-editor>, 177
<stylesheet>, 196
<submit>, 180, 181
<table>, 165
<table-plus>, 199
<theme-stylesheet>, 165, 172
<transition-button>, 193
<transition-buttons>, 193
<type-name>, 165, 168, 169
<unless>, 163
<update-button>, 180, 182
<view>, 165, 172
<with>, 163
<with-environments>, 207
<with-field-names>, 209
<with-fields>, 209
<with-gems>, 206
<with-model-associations>, 207
<with-model-columns>, 207
<with-models>, 207
<with-plugins>, 206
<wrap>, 163
<You>, 165, 172
<Your>, 172

A

account, 73, 104, 162, 194, 195, 196, 212, 213
Account Navigation, 194
acting_user, 57, 58, 59, 60, 63, 64, 67, 68, 69, 70, 91, 95, 96, 97

acting_user.signed_up?, 57, 58
action, 34, 52, 53, 55, 57, 73, 74, 75, 76, 77, 78, 79, 82, 83, 84, 94, 97, 100, 101, 102, 103, 105, 129, 170, 171, 180, 181, 184, 200
actions, 39, 53, 54, 56, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 83, 84, 87, 91, 92, 99, 100, 101, 102, 103, 211, 212, 213
--add-gem, 37, 38, 41, 45
--add-routes, 37, 38, 41, 42, 45, 46, 49
AJAX-progress, 168, 178, 196
alias-of, 151
all_attributes, 133, 134, 147
all_changed?, 59, 65
any_changed?, 59, 65
application.css, 37, 45
application.dryml, 37, 41, 45, 52, 53, 54, 55, 150
Association Scopes, 113, 117
associations, 33, 51, 57, 58, 59, 64, 177, 183, 185, 209, 210
attr_accessible, 61
attr_protected, 61, 67, 92
attr_readonly, 61
attribute, 56, 57, 58, 59, 60, 61, 62, 63, 65, 67, 84, 94, 95, 96, 121, 123, 125, 126, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 141, 147, 151, 152, 156, 158, 159, 163, 164, 166, 167, 168, 169, 171, 172, 173, 174, 181, 184, 185, 187, 188, 192, 197, 198, 209
attributes, 54, 58, 59, 60, 61, 62, 63, 66, 80, 89, 90, 91, 94, 96, 121, 122, 130, 131, 132, 133, 134, 135, 137, 138, 139, 141, 143, 146, 147, 155, 171, 173, 177, 181, 182, 183, 189, 193, 195, 209
attrs, 132, 133, 134, 135, 157
auto_actions, 72, 73, 75, 76, 84, 99
autocomplete, 83, 84, 184
auto-generated, 52, 53, 54

B

before_filter, 42, 46
belongs_to, 58, 59, 64, 65, 68, 73, 74, 91, 95, 96, 97, 113, 117, 129, 177, 183, 184, 188
body, 107, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 140, 141, 144, 145, 146, 147, 148, 151, 152, 156, 157, 163, 164, 167, 173, 187, 191, 196, 211, 212, 213
Boolean Scopes, 112, 115
by_most_recent, 112, 116

C

can_call?, 71
can_create?, 71
can_delete?, 71
can_edit?, 71, 188
can_update?, 71
card, 54, 110, 143, 144, 145, 146, 147, 149, 150, 151, 152, 153, 154, 155, 191, 192, 193, 200
card merge-params=, 149

cards.dryml, 52, 54, 191
Chaining, 113, *See*
changed, 33, 58, 59, 60, 62, 63, 64, 65, 177, 178
changed?, 58, 59, 60, 62, 63, 64, 65
children, 110
close-button, 199
collection, 27, 53, 54, 74, 97, 110, 131, 136, 137, 138,
144, 145, 151, 163, 164, 166, 167, 169, 171, 172, 177,
181, 182, 183, 186, 187, 190, 191, 200
Colons in tag names, 159
Constraints, 25
content, 53, 87, 104, 110, 123, 124, 125, 126, 127, 129,
130, 134, 135, 136, 139, 140, 141, 142, 146, 152, 154,
155, 157, 158, 159, 163, 164, 167, 170, 175, 176, 187,
194, 196, 197, 209, 211, 212, 213
content-body, 211, 212, 213
content-header, 197, 211, 212, 213
context, 74, 76, 95, 96, 97, 111, 128, 129, 130, 131, 136,
137, 138, 140, 151, 163, 164, 166, 168, 169, 170, 172,
181, 182, 183, 184, 186, 187, 188, 190, 200, 209
controls, 37, 52, 166, 167
Core, 162, 163, 165
Create, 15, 56, 66, 67, 100, 105
create a new tag from an existing tag, 149
create an alias of a tag, 151
create_permitted?, 57, 59, 68, 69
CRUD, 66
CSS, 125, 134, 156, 175, 182, 191, 194, 198

D

dasherize, 169
data flow, 29
database schema, 33
database.yml, 15, 17, 18, 36
Date Scopes, 112, *See*
def index, 76, 78, 81
--default-name, 34
Delete, 66
delete-button, 53, 54, 166, 167, 182, 183
--delete-index, 38, 42, 46, 49
destroy_permitted?, 57, 60
dev-user-changer, 165, 195
do_transition_action, 102
down-arrow, 199
DRY, 27, 51
DRYML, 25, 27, 28, 31, 52, 54, 56, 74, 76, 84, 108, 111,
120, 121, 122, 123, 125, 126, 127, 128, 129, 130, 131,
132, 133, 134, 135, 136, 137, 138, 142, 143, 146, 149,
151, 154, 155, 156, 158, 159, 160, 161, 162, 163, 164
DRYML Guide, 120, 128, 151

E

edit_permitted?, 61, 62, 63, 67
edit-link, 166, 167
email-address-input, 212

email-address-label, 212
empty-message, 192, 199
environment.rb, 36, 39, 40
erb, 37, 41, 43, 46, 54, 120, 121, 122, 156
error-messages, 213
Extending a tag, 149

F

field_names, 107, 109
field-heading-row, 166
field-list, 53, 54, 165, 186, 187, 209, 213
flexibility, 27, 28, 108, 120, 143
force, 171, 173
force-all, 166, 209
forgot-password, 211
form, 53, 54, 61, 62, 69, 77, 80, 83, 100, 102, 104, 109,
114, 117, 122, 127, 131, 158, 177, 180, 181, 182, 184,
185, 186, 187, 209, 211, 212, 213
format, 78, 79, 167, 171, 173, 174

G

gem env, 4
gem list, 4
gem update --system, 3
gems, 4, 16, 26, 31, 34
GitHub, 32
gravatar, 200
gravatar.com, 200

H

has_many, 64, 79, 98, 113, 118, 172, 184, 185, 186, 187,
188, 190, 209, 210
header, 54, 110, 154, 155, 167, 175, 191, 196, 199
heading, 110, 138, 140, 141, 142, 144, 145, 146, 147,
149, 151, 152, 154, 155, 167, 185, 197, 200, 209, 211,
212, 213
Hobo, 8, 1, 15, 17, 18, 22, 25, 26, 27, 28, 29, 30, 31, 51,
52, 53, 54, 55, 56, 57, 58, 60, 61, 62, 63, 64, 65, 66,
68, 69, 70, 72, 73, 74, 75, 76, 77, 78, 79, 80, 82, 83,
84, 85, 86, 92, 94, 99, 100, 102, 104, 105, 107, 108,
109, 110, 111, 114, 115, 120, 121, 128, 133, 140, 141,
143, 148, 161, 162, 164, 169, 175, 177, 180, 181, 185,
192, 193, 199, 200, 211
Hobo Controllers, 72
Hobo Lifecycles, 31, 86
Hobo Migration, 34
Hobo Model Controller, 31
Hobo Scopes, 112
hobo_create, 79, 80, 81
hobo_create_for, 80, 81
hobo_index_for, 80, 81
hobo_migration, 32, 34, 51
hobo_model, 41, 45, 51, 70, 72, 75, 76, 87, 108, 113, 184
hobo_model_controller, 41, 45, 72, 75, 76

hobo_model_resource, 51
hobo_new, 77, 79, 80
hobo_new_for, 80
hobo_rapid, 37, 38, 41, 45
hobo_reorder, 84
hobo_show, 76, 77, 79
hobo_update, 77, 78, 80, 81, 82
hobo_user_model, 37, 38, 41, 43, 45, 70, 184
hobofields, 31
hobosupport, 31
href, name, 134, 171
HTML, 53, 78, 82, 127, 128, 134, 139, 143, 146, 158,
159, 160, 166, 169, 174, 175, 176, 180, 182, 187, 188,
197

I

if-present, 168
implicit, 128, 136, 137, 138
implicit context, 128, 136, 137, 138
--import-tags, 37, 38, 41, 45
include, 8, 42, 46, 59, 70, 72, 79, 97, 98, 99, 105, 109,
113, 117, 139, 141, 158, 168, 172, 177, 178, 182, 183,
186, 198, 209, 210
include-timestamps, 209, 210
index_action, 75, 78
index.dryml, 38, 41, 42, 45, 46, 49
Indexes, 24, 51
Inline Booleans, 110
instance variables, 77
--invite-only, 35, 41, 42, 45, 46, 47, 48
Invite-only website, 42, 46

J

join, 64, 164, 172

K

Key Scopes, 112, *See*

L

label, 107, 109, 165, 171, 177, 182, 183, 188, 192, 193
labelled-item-list, 211, 212
lifecycle actions, 99
Lifecycle scopes. *See*
Lifecycles, 31, 44, 85, 86, 92, 94, 100, 102, 162, 193
Lifecyle Scopes, 112
limit, 8, 83, 92, 112, 116, 183
Link to account page, 195
Logged in as, 164, 195
logged-in-as, 195
log-in, 195, 211
login-input, 211
login-label, 211
log-out, 195

lowercase, 169, 171

M

MagicMailer, 87, 94
markdown-help, 133, 134
Matz, 25, 136
merge-attrs, 122, 133, 134, 135, 144, 146, 147, 149, 151,
157, 158
method, 27, 57, 58, 60, 61, 62, 65, 67, 68, 70, 71, 74, 75,
77, 78, 79, 80, 82, 83, 84, 89, 90, 91, 92, 94, 95, 96,
97, 98, 99, 100, 102, 111, 112, 117, 123, 128, 132,
134, 137, 138, 144, 163, 169, 170, 171, 180, 181, 182,
183
method_callable_by?(user, method_name), 67
--migrate, 34
migration, 18, 27, 32, 33, 34, 51, 89, 94, 113
migrations, 33, 51
model_name, 108, 111
Model-View-Controller, 25, 27, 29
MVC, 25, 27, 29, 86
MySQL, 10, 11, 12, 14, 15, 16

N

name, 18, 22, 34, 43, 51, 53, 54, 55, 58, 60, 62, 63, 67,
68, 71, 72, 73, 74, 80, 83, 89, 91, 92, 94, 95, 96, 97,
98, 99, 105, 107, 108, 109, 111, 113, 114, 115, 116,
117, 118, 119, 121, 122, 125, 127, 128, 129, 130, 131,
134, 138, 140, 147, 149, 151, 156, 157, 159, 163, 165,
166, 167, 168, 169, 170, 171, 177, 178, 181, 182, 183,
184, 186, 187, 188, 193, 196, 198, 199, 209
name_changed?, 62
name_was, 62
name:, 63, 151
named_scope, 112
nested parameters, 143, 144
never_show, 60, 92
New tags from old, 146
no-filter, 199
none_changed?, 59, 65
not_, 84, 112, 114, 115, 118

O

only_changed?, 59, 60, 65
options, 16, 25, 34, 35, 41, 78, 79, 83, 92, 94, 95, 96,
103, 177, 178, 183, 184, 185, 186, 195, 199, 200, 209
Oracle, 17, 18, 19, 20, 23, 24
Oracle Object Browser, 23
Oracle XE, 20
order, 27, 31, 35, 67, 68, 74, 80, 84, 110, 112, 117, 128,
129, 131, 150, 156, 157, 161, 179, 186, 189, 197
order_by, 112, 117
owner, 57, 58, 59, 60, 64, 68, 73, 74, 80, 82, 97, 98

P

page-nav, 194, 200
pages.dryml, 52, 53, 54, 55, 161
param, 83, 123, 125, 126, 127, 129, 132, 133, 134, 140, 141, 144, 145, 146, 147, 149, 150, 151, 152, 157, 158, 200
parameter tag, 124, 127, 129, 141
params, 53, 76, 79, 80, 84, 90, 91, 94, 96, 102, 103, 105, 129, 146, 171, 182
password-input, 211
password-label, 211
Permissions, 31, 64, 69
plural, 54, 117, 169
polymorphic, 54, 128, 151, 152, 153, 172, 187, 188
polymorphic tag, 151, 152, 153, 172, 187

Q

query-params, 171

R

rake, 35, 40
rake db
 create
 all, 35
Rapid Core, 162
Rapid Document Tags, 162, 175
Rapid Editing, 162, 177
Rapid Forms, 162, 180, 181, 182, 183
Rapid Generics, 162, 191
Rapid Lifecycles, 162, 193
Rapid Navigation, 162, 194
Rapid Pages, 196
Rapid Support, 209
Rapid Tag Library, 28, 52, 161, 162
Rapid User Pages, 162, 211
rapid_core.dryml, 161
rapid_document_tags.dryml, 161
rapid_editng.dryml, 161
rapid_forms.dryml, 161
rapid_generics.dryml, 161
rapid_lifecycles.dryml, 161
rapid_navigation.dryml, 161
rapid_pages.dryml, 161
rapid_plus.dryml, 161
rapid_support.dryml, 161
rating, 200
Read, 66, 77
READ_ONLY_ATTRS, 59
recent, 112, 116
record.createable_by?(user), 67
record.destroyable_by?(user), 67
record.editable_by?(user, attribute=nil), 67
record.updatable_by?(user), 67
record.viewable_by?(user, attribute=nil), 67

redirect_to, 77, 82, 100, 101
remember-me, 211
remember-me-input, 211
remember-me-label, 211
repeat tag, 137
Roles, 98
ruby script/generate, 34, 37, 38, 41, 42, 43, 45, 46, 49, 87

S

scoped variables, 156, 157
Scopes, 112, 113, 117
Scoping Associations. *See*
search, 113, 115, 117, 152, 191, 196, 199
search-form, 199
search-submit, 199
Self-closing tags, 158, 159
show_action, 75, 78
show.dryml, 55
sign-up, 195
Simple log-in page, 211
Simple Scopes, 114
size, 78, 143, 200
skip, 1, 34, 43, 122, 136, 166, 188, 209, 210
skip-associations, 209, 210
--skip-timestamps, 34
sortable-options, 200
sqlite3.dll, 9
SQLite3-ruby gem, 9
state, 59, 62, 69, 86, 88, 89, 90, 91, 92, 94, 95, 96, 97, 105, 113, 114, 116
Static Scopes, 112, *See*
status_changed?, 58
status_was, 58
stylesheets, 36, 37, 41, 45, 196
submit, 53, 54, 82, 99, 105, 106, 181, 182, 185, 211, 212, 213

T

taglibs, 34, 37, 41, 45, 52, 158, 180, 191, 193
tbody, 166
test_generators, 36
tfoot, 166
thead, 166
theme, 37, 38, 41, 45, 168, 172, 198
this_field, 111, 130, 131, 165, 166, 209
this_field_help, 111
this_field_name, 111
this_parent, 130, 131
timestamps, 34, 113, 166
tnsnames.ora, 22

tr, 166
transition, 90, 91, 95, 96, 97, 98, 99, 102, 104, 105, 193

transitions, 90, 91, 92, 95, 97, 99, 102, 103, 104

U

UI, 69, 107, 108
ul, 137, 138, 144, 157, 185, 186, 191, 194, 195
up-arrow, 199
update, 3, 18, 53, 56, 57, 58, 59, 60, 61, 62, 64, 66, 67,
70, 71, 72, 73, 76, 77, 78, 80, 81, 82, 86, 91, 100, 102,
147, 177, 178, 181, 182, 193
update_permitted?, 57, 58, 59, 60, 61, 62, 64
user_find, 66, 79
Users Controller, 43, 47

V

validates_inclusion_of, 69

View Hints, 107
view_hints, 107, 111
view_permitted?, 57, 60
view-hints, 107, 108, 111
ViewHints, 107, 108, 109, 110

W

with_, 67, 113, 117
without_, 113, 117
Wrapping inside a parameter, 154
Wrapping outside a parameter, 154

X

XML, 52, 79, 127, 129, 158, 159, 160