



RAPID RAILS 3 WITH **HOB**O

A step-by-step introduction to rapid development of data-rich web applications using the Hobo extensions for Ruby on Rails

3

Owen Dall - Jeff Lapides - Tom Locke - Bryan Larsen

with contributions by

Venka Ashtakala - Domizio Demichellis - Tiago Franco

Marcelo Giorigi - Matt Jones

Contents

Contents	i
List of Figures	ix
Preface	xxi
AUTHORS	xxi
CONTRIBUTORS	xxii
PREFACE TO RAPDID RAILS 3 WITH HOBO	xxiii
PREFACE TO THE HOBO 1.0 VERSION FOR RAILS 2	xxiv
I INTRODUCTION AND INSTALLATION	1
Chapter 1	
INTRODUCTION	3
What is Hobo?	3
Fundamentals	10
Rails and Hobo	11
Hobo Enhancement Summary	13
Chaper 2	
INSTALLATION	17
Introductory Concepts and Comments	17
Installing Ruby, Rails and Hobo	18
Using MySQL with Hobo	26
Using Oracle with Hobo	39

II TUTORIALS 49

Chapter 3		
INTRODUCTORY TUTORIALS		51
Tutorial 4 – Permissions		84
Tutorial 6 – Navigation Tabs		100
Chapter 4		
INTERMEDIATE TUTORIALS		125
Introductory Concepts and Comments		125
Tutorial 9 Editing Auto-Generated Tags		128
Tutorial 10 DRYML I: A First Look at DRYML		143
Tutorial 11 DRYML II: Creating Tags from Tags		151
Tutorial 12 Rapid, DRYML and Collections		158
Tutorial 13 Listing Data in Table Form		173
Tutorial 14 Working with the Show Page Tag		179
Tutorial 15 New and Edit Pages		189
Tutorial 16 The <a> Hyperlink Tag		198
Chapter 5		
ADVANCED TUTORIALS		203
Introductory Concepts and Comments		203
Tutorial 17 The Agile Project Manager		204
Tutorial 18 Using CKEditor (Rich Text) with Hobo		247
Tutorial 19 Using FusionCharts with Hobo		251

Tutorial 20	
Adding User Comments to Models	262
Tutorial 21	
Replicating the Look and Feel of a Site	269
Tutorial 22	
Creating a “Look and Feel” Plugin	291
Tutorial 23	
Using Hobo Lifecycles for Workflow	295
Tutorial 24	
Creating an Administration Sub-Site	302
Tutorial 25	
Using Hobo Database Index Generation	305
Chapter 6	
DEPLOYING YOUR APPLICATIONS	309
Introductory Concepts and Comments	309
Tutorial	
Installing and Using Git	309
Tutorial	
Rapid Deployment with Heroku	319
III HOBO UNDER THE HOOD	333
Chapter 7	
HOBO GENERATORS	335
Changes from Hobo 1.0 to 1.3	335
Hobo Generators	335
Automatically Generated DRYML Tags	346
application.dryml file.	347
Chapter 8	
THE HOBO PERMISSION SYSTEM	349
Introduction	349
Defining permissions	350
Change tracking	351
Permissions and associations	357

The Permission API	358
Permissions vs. validations	362
View helpers	363
 Chapter 9	
HOBO CONTROLLERS AND ROUTING	365
Introduction	365
Owner actions	366
Adding extra actions	368
Changing action behavior	369
Writing an action from scratch	370
The default actions	372
Owner actions	374
Autocompleters	376
Further Customization	377
Drag and drop reordering	377
 Chapter 10	
HOBO LIFECYCLES	379
Introduction	379
Key concepts	384
Defining a lifecycle	385
Defining states	386
Defining creators	386
Defining transitions	387
Repeated transition names	389
Validations	390
Controller actions and routes	390
Transitions	392
Keys and secure links	394

Chapter 11**HOBO VIEW HINTS AND LOCALES** **397**

Introduction	397
Internationalization (I18n)	397
Child relationships	400
Inline Booleans	401
Locale (I18n) Friendly Tags	402

Chapter 12**HOBO SCOPES** **419**

Preparation	420
Simple Scopes	421
Boolean scopes	423
Date scopes	423
Lifecycle scopes	424
Key scopes	424
Static scopes	424
Association Scopes	425
Scoping Associations	426
Chaining	428

Chapter 13**THE HOBO DRYML GUIDE** **429**

What is DRYML?	429
Simple page templates and ERB	430
Where are the layouts?	431
Defining simple tags	431
Parameters	432
Changing Parameter Names	433
Multiple Parameters	434
Default Parameter Content	434
The Default Parameter	436
The Implicit Context	437

Field chains	440
Tag attributes	441
Flag attributes	442
Merging Attributes	443
Merging selected attributes	444
Repeated and optional content	446
Even/odd classes	447
Using the implicit context	448
Pseudo parameters - before, after, append, prepend, and replace	449
Nested parameters	452
Customizing and extending tags	455
Aliasing tags	459
Polymorphic tags	460
Wrapping content	462
Local and scoped variables.	463
Taglibs	465
Divergences from XML and HTML	466

Chapter 14

THE HOBO RAPID TAG LIBRARY	469
Core	470
Rapid Core	473
Rapid Document Tags	484
Rapid Editing	486
Rapid Forms	490
Rapid Generics	503
Rapid Lifecycles	504
Rapid Navigation	505
Rapid Pages	507
Rapid Plus	511
Rapid Summary	513
Rapid Support	524
Rapid User Pages	526

CONTENTS

INDEX **529**

Index **530**

CONTENTS

CONTENTS

List of Figures

1	Data flow for a typical Application using a MVC framework	12
2	Data flow for a Rails application	12
3	Data flow for a Hobo application	13
4	Download Site for the Rails Installer	18
5	Rails Installer Setup Wizard	19
6	Choose the Installation Directory	20
7	Completed Setup	20
8	Menu Path	21
9	Command Prompt	21
10	Rails Installer Sample Application Folder	21
11	IRB	22
12	RubyGems Documentation	22
13	Command Prompt	23
14	List of Ruby Gems installed by the Rails Installer	23
15	Testing SQLite3	24
16	The Sample Rails 3 app included by Rails Installer	24
17	Command Prompt with Ruby and Rails	24
18	Sample console output from the "gem list" command after installing Hobo	25
19	TextMate Webpage	26
20	Download site for MySQL	27
21	Using the downloaded .msi file to install MySQL on Windows	28

22	Choose the “Custom” setup type	28
23	Specify the destination folder "C:\MySQL" for the server software	29
24	Specify the destination folder "C:\MySQL\data" to hold MySQL data	30
25	MySQL Enterprise Monitor	30
26	The MySQL Instance Configuration Wizard	31
27	MySQL Instance Configuration Wizard	31
28	MySQL Instance Configuration Wizard	32
29	MySQL Instance Configuration Wizard	33
30	MySQL Instance Configuration Wizard	33
31	MySQL Instance Configuration Wizard	34
32	MySQL Command Line Client	34
33	MySQL Command Line	35
34	MySQL Command Line	35
35	Ruby.exe Error	36
36	MySQL lib Directory	36
37	Ruby bin Directory	37
38	The automatically created database.yml file	38
39	MySQL Command Line	38
40	Console output after installing Oracle gems for Ruby and Rails	39
41	Console output after installing Oracle gems for Ruby and Rails	39
42	The generated database.yml file for Oracle	40
43	Oracle database install download site	41
44	Running the Oracle XE installation	42
45	Specifying the database passwords	42
46	Launch the Database home page	43
47	Log is as SYS to configure your database	43
48	Creating a schema/user to use with Hobo	44
49	The tnsnames.ora file created during installation	44
50	Log into Oracle to view the created table	45
51	Access the Oracle Object Browser	46
52	Review the User table from within Oracle	46

LIST OF FIGURES

53	Review the Indexes view for Users	47
54	Review the Constraints view for User	47
55	Command Prompt with Ruby and Rails	54
56	Command Prompt with Ruby on Rails	55
57	Completion Message	56
58	Register Administrator Screen	57
59	Register Administrator Screen	58
60	Completed Registration	58
61	The default User model created by Hobo	59
62	Contents of the first Hobo migration file	60
63	Contents of the "schema.rb" file after the first migration	60
64	Drop down selector for the active user	61
65	Location of the Rapid templates	62
66	Folder location for Models and Views	64
67	Migration file changes	65
68	Contacts tab on "My First App"	66
69	New Contact page for "My First App"	67
70	Remove field from contact model	68
71	Default config/locales/app.en.yml File	72
72	app.en.yml File with Fields Renamed	73
73	View of fields relabeled using the Hobo i18n module	74
74	Adding help text using the Hobo i18n "attribute_help" method	74
75	Contact entry page with ViewHints enabled	75
76	CSS definitions for the input text fields	76
77	Modified entry in "application.css" to shorten text prompts	76
78	Page view of validation presence of name	77
79	Page view of double validation error	78
80	Adding "validate_numericality_of" validation	79
81	Page view of triggering the "validates_numericality_of" error	79
82	Page view of uniqueness validation error	80
83	Page view of triggering a range validation error	81

LIST OF FIGURES

84	Page view of validation of text length error	81
85	Page view of “validates_acceptance_of” error	82
86	Welcome to One Table in the Permissions tutorial	85
87	Recipes tab	86
88	Page view of created recipes	87
89	Page view of a Recipe	90
90	Making the Recipes tab disappear	93
91	Error message “The page you were looking for could not be found” . .	94
92	How Hobo finds the default “name” attribute for a model	95
93	Creating your own custom “name” attribute	95
94	Page view of the custom name attribute	96
95	Viewing the edit URL	97
96	“Unknown action” error page	98
97	Customizing the name of a tab	102
98	Removing the default Home tab	103
99	Changing the Application Name	104
100	Using “enum_string” to create a drop-down list of Countries	106
101	Index page for Countries	111
102	Selecting a Country for a Recipe	112
103	Active link on Country name in the Recipe show page	113
104	The Country show page access from the Recipe show page	113
105	Only an Administrator is provided the Country Edit link	114
106	The Categories tab on the Four Table app	118
107	The Index page for Categories	118
108	“Category Assignments” on the Recipe show page	119
109	Assignment multiple Categories to a Recipe	119
110	Edit page view of a Recipe with multiple Categories assigned	120
111	Using the Hobo “children” declaration to enhance the view of related records	121
112	Show page for a Category before using ViewHints	122
113	Category page view after adding the ViewHints “children :recipes” declaration	122

114 Model Relationships Structure	123
115 Folder view of \taglibs\auto\rapid	126
116 Front page view of the Four Table application	129
117 Folder view of the rapid DRYML files	130
118 The Hobo Rapid <index-page> tag definition in the pages.dryml file . .	133
119 The Recipes Index page	133
120 View of the taglibs/auto/rapid folder	134
121 Adding the definition of index-page into the application.dryml file . .	135
122 Page view of "My Recipes" after modifying the <index-page> tag . .	137
123 Adding the <index-page/> tag to index.dryml	138
124 How a change to the <index-page> tag affects a collection	140
125 Changing the tab order for the main navigation menus	141
126 Changing the application name with the app-name tag	142
127 The \views\front\index.dryml file after the first modification	144
128 The Home page with the first set of custom messages	145
129 How the passed parameter displays on the page	146
130 Passing three parameters to your <messages> tag	147
131 Calling <span:> explicitly within to your <bd-it> tag	148
132 Adding the custom <more-messages> tag to front\index.dryml	152
133 Page rendering with <more-messages>	152
134 Extending the tag <messagex> in application.dryml	154
135 Using the extended <messagex> tag	154
136 Page view of the next additions to <messagex>	155
137 Page view of the <more-messages> tag usage	156
138 Page view of overriding the default message 0.	156
139 More parameter magic	157
140 The Four Tables application as we left it	159
141 Creating the /views/recipes/index.dryml file	160
142 page view of using a blank "<collection:></collection:>" tag	163
143 How the <collection> tag iterates	163
144 Using the <a> hyperlink tag within a collection	164

145 Specifying what <collection> tag will display	166
146 Changing the display style within <collection>	167
147 Changing the implicit context within <collection>	168
148 Creating comma-delimited multi-valued lists in a <collection>	169
149 Adding the count of values in the <card> tag	171
150 Using “if–else” within a tag to display a custom message	172
151 Using <table-plus> to display a columnar list	174
152 Adding a “Categories Count” to <table-plus>	175
153 Adding a comma-delimited list within a <table-plus> column	176
154 Adding a search facility to <table-plus> using Hobo’s apply_scopes method	177
155 Found Recipes searching for “French”	178
156 The Recipe show page before modification	181
157 Recipe show page after removing three critical lines of code	182
158 Using the <field=list> tag to choose which fields to display	182
159 Using the <collection-heading:> tag	183
160 Using the <body-label:> parameter tag	184
161 Using the <country-label:> parameter to change the label on the page	186
162 A new show page for Recipes	187
163 Page view of using the replace attribute in the <content-body:> parameter tag	188
164 Default Hobo form rendering	192
165 Modifying the <field-list> tag to remove fields on a page	193
166 First step using the <Input> tag	195
167 Adding the label for the field “Title”	196
168 Generating an active link to a list of Countries	199
169 The Countries index page activated by your custom link	199
170 Constructing a custom link to the “New County” page	200
171 Page view of custom <show-page> tag	201
172 Adding "has_many :requirements" to the Project class	206
173 Adding "belongs_to :project" and "has_many :tasks" to the Requirement model	207

174	Adding the “belongs_to” and “has_many” declarations to the Task model	207
175	Adding the two "belongs_to" definitions to the TaskAssignment model	208
176	Adding the "has_many" declarations to the User model	208
177	First Hobo migration for Projects	209
178	View of indexes created by the migration	209
179	The default Home page for the Projects application	211
180	The Projects index page	212
181	New Requirement page	212
182	Index view for Requirements	213
183	New Task page	213
184	Index view for Tasks	214
185	Part 1 of the Application Summary page	215
186	Part 2 and 3 of the Application Summary page	216
187	Part 4 of the Application Summary page	217
188	Effect of removing the "index" action from the Tasks controller	218
189	View of "No Requirements to display" message	218
190	The "New Requirement" link now appears	219
191	View of the "New Requirement" page	219
192	View of the in-line "Add a Task" form	220
193	Requirement page after modifying controller definitions	221
194	Defining available roles using “enum_string”	222
195	Modifying the "create_permitted" method to the User model	223
196	Users Controller with "auto actions :all:	223
197	The Users tab is now active	223
198	The Edit User page with the new Role field	224
199	Adding the use of Role in Permissions	225
200	Modifying the “update_permitted?” method in the Requirement model	227
201	Assigning multiple Users to a Task in the Edit Task page	228
202	The New Project page using “ProjectHints”	229
203	The default application name and welcome message	230
204	Changing the application name in "config/application.rb"	230

205	Modifying "\front\index.dryml"	231
206	Home page modified by changing "/front/index.dryml"	232
207	Extending the card tag for Task in "application.dryml"	233
208	Viewing assigned users on a the Task card	233
209	Listing the contents for the "app\views>taglibs\auto\rapid" folder	234
210	contents of the pages.dryml file	235
211	The auto-generated "show-page" tag for User in "pages.dryml"	235
212	View of the enhanced User "show-page"	236
213	The Users tab showing all assignments	237
214	Using the Hobo "<table-plus>" feature to enhance the Requirements listing	239
215	Enhancing the <table-plus> listing	240
216	Using a search within the Requirements listing	240
217	The Edit Requirement form with selectable status codes	241
218	Creating an AJAX status update for Requirements	242
219	Requirement Status view	243
220	Task model with "due_date" and a validation for the date	245
221	Error message from trying to enter a date earlier than today	246
222	CKEditor source folder listing	247
223	Using the ":html" field option to trigger rich-text editing	249
224	Sample Hobo form using CKEditor	250
225	Registration form to request FusionCharts	252
226	Download page for FusionCharts	252
227	Target location for the FusionCharts SWF files	253
228	Adding the required <extend tag='page'> definition in application.dryml	254
229	Screen shot of sample recipe data for the tutorial	255
230	Enhancements to RecipesController to provide data to FusionCharts	256
231	Content of recipes/index.dryml used to render FusionCharts	258
232	Screen shot of rendered FusionCharts bar chart	259
233	The recipe/index.dryml file to render a FusionCharts pie chart and bar chart	261
234	Screen shot of the rendered FusionCharts bar and pie charts	261

235	The comment form	266
236	Comment posted	266
237	Updated comment card	267
238	User page comments	267
239	Screen shot of the nifa.usda.gov home page	269
240	The NIFA banner image	270
241	The NIFA photo image	271
242	The NIFA main navigation bar	271
243	NIFA navigation panels	271
244	NIFA footer navigation	272
245	The NIFA Demo default home page	273
246	Using the "app-name" tag to change the default application name	273
247	Using Firebug to locate the background color	274
248	Using Firebug to find the images used by Hobo for the default background	274
249	Adding the new background color to "application.css"	275
250	First pass at modifying "application.dryml"	276
251	The two images used in NIFA's top banner	276
252	How to reference the banner gif in "application.css"	277
253	View of the NIFA Demo login page	278
254	The Navigation Panel before refactoring	278
255	View of our first pass at the main navigation menu	279
256	Still need more to fix the top navigation menu...	280
257	The fixed NIFA man navigation bar	282
258	View of the default three-column formatting	283
259	View of the left panel contact without styling	284
260	View of the left panel content with correct styling	285
261	View of the right panel content with styling	288
262	View of the main content panel	289
263	NIFA Demo with final footer styling	290
264	Batch file with commands to create the plugin folders and content	291
265	Guest view Recipes - All recipes are in state "Not Published"	298

266	Recipes ready to Publish	299
267	Omelet recipe after being placed in the "Published" state	299
268	Recipe index with buttons for "Publish" and "Not Publish"	300
269	Guest user can only see the published Recipe	300
270	Generator console output for creating an admin sub-site	302
271	Requirement Status Permissions	303
272	View of the Admin folder contents	304
273	View of the Admin Sub-Site	304
274	Hobo source code on github.com	310
275	Hobo gems are also available on github.com	310
276	Installing Git for Mac OSX	311
277	Download the mysysgit installer for Windows	311
278	Running the Git Setup Wizard	312
279	Git setup options	312
280	Select the OpenSSH option	313
281	Select to option to run Git from the Windows command prompt	313
282	Select Windows style line endings	314
283	Running the PuTTY Key Generator install	315
284	Generate SSH key pairs for use with Git	316
285	The default file names generated by PuTTYGen	317
286	Locating your USERPROFILE setting	318
287	View of "no ssh public key found" error	318
288	Naming your SSH key pairs	319
289	The original Heroku beta invitation	319
290	Using the free "Blossom" database hosting option on Heroku.com	320
291	Sign Up for a Heroku account	321
292	Heroku notification that "Confirmation email sent"	321
293	Locating your "Invitation to Heroku" email	322
294	The "Welcome to Heroku" signup page	322
295	The "Account Created" message at Heroku.com	323
296	Installing the Heroku Ruby gem	323

297	Console output from the "heroku create" command	325
298	Using heroku git push	325
299	Telling Heroku where to find your application's gems	326
300	Adding your ".gems" config file to your git repository	327
301	Migrating your database schema to Heroku.com	328
302	Testing your Heroku app	328
303	Running the "Four Table" app on Heroku.com	329
304	Installing the Taps gem to upload data to Heroku.com	329
305	Using "heroku db:push" to push data to your app on Heroku.com	330
306	The "Four Table" app on Heroku.com with data	331
307	Add a recipe on Heroku.com	331
308	Pull changed data from Heroku.com to your local app	332
309	Hobo precedence logic for action tags	348
310	Defining the Friendship model	381
311	Lifecycle diagram	382
312	config/locale/app.en.yml	398
313	Name and input help	399
314	hobo.en.yml	400
315	rapid.dryml	469
316	The contents of the "summary.dryml" file	515
317	Sample view of the first section of an application summary page	516

LIST OF FIGURES

LIST OF FIGURES

Preface

AUTHORS

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.

Jeff Lapidés

Jeff Lapidés was educated as a physicist and has worked as a CIO and senior operating executive in a large public corporation. For most of the past decade, he consulted with private industry in information technology, business management, and science. He is currently engaged at a nationally ranked research university where he develops relationships between research scientists in engineering, information technology, physical, and life sciences, foundations and corporations.

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 1996.

Bryan Larsen

Bryan sold his first video game in 1987 and has never stopped. Becoming a father 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.

Domizio Demichelis

Domizio is a free-lance application designer and developer who played the lead role in migrating from Hobo 1.0 for Rails 2 to Hobo 1.3 for Rails 3. He also created the new Hobo Generation Wizard.

Venka Ashtakala

Venka is a Software Engineering Consultant with over 10 years experience in the Information Technology industry. His expertise lies in the fields of rapid development of data driven web solutions, as well as search, reporting, data warehousing solutions for both structured and non structured data and implementing the latest in open source technologies is another expertise. He has consulted on a variety of projects in both the Private and Public sectors most recently with the National Institute of Food and Agriculture.

CONTRIBUTORS

Tola Awofolu

Tola is a software engineer with over seven years of experience with standalone Java and Java web development frameworks. She's been working with Ruby on Rails and Hobo for over a year as part of the Barquin International team at USDA on two major projects. She is a protégé of Tom Locke and Bryan Larsen and has become Barquin International's leading Ruby developer.

Tiago Franco

Tiago Franco is a Project and Technical Manager working in Software development for more than ten years, currently working for the Aerospace & Defense market. He's been working with Ruby on Rails since 2006, and adopted Hobo in 2008 to re-design Cavortify.com.

Marcelo Giorgi

Marcelo is a software engineer with over seven years of experience in standalone Java and Java web development frameworks. He's been working with Ruby on Rails for more than two years and had the opportunity to work with (and make some contributions to) Hobo during the last year, 2011.

Matt Jones

Matt is a software engineer who can remember when Real Web Programmers wrote NPH CGI scripts to be loaded up in Mosaic. When he's not building Hobo applications, he's often found hunting Rails bugs or helping new users on rails-talk and hobo-users. He also has the dubious honor of being the unofficial maintainer of the Rails 2.x-era "config.gem" mechanism, earned after fixing the borked 2.1 series version to work better with Hobo.

PREFACE TO RAPID RAILS 3 WITH HOBO

It has been a little over a year since we published the first PDF "Rapid Rails with Hobo" and "Hobo at Work". We are pleased that there have been over 20,000 downloads since that time.

The current "Rapid Rails 3 with Hobo" combines both "Rapid Rails with Hobo" and "Hobo at Work". We hope that having all of the material in one indexed volume would prove more valuable.

There has been such a dramatic change in the structure of Hobo since version 1.0 we really should have called it Hobo 3.0! Each major feature of Hobo can now be used independently and has been re-factored to work seamlessly with Rails 3.

Domizio Demichelis did a fantastic job with the new Hobo Setup Wizard as well the heavy lifting on almost every new feature in Hobo 1.3. It has been gratifying to see another extremely talented and seasoned contributor following in the footsteps of Tom Locke, James Garlick, Bryan Larsen and Matt Jones.

It is also very gratifying to see the self-sustaining Hobo User community and those talented Hoboists who volunteer their time to answer questions and provide insight. If you look at the member statistics you will see the top five all time posters (at least since we switched to using Google Groups in June of 2008) include Kevin Porter, Tom Locke, Matt Jones, Bryan Larsen, and Tiago Franco.

At Barquin International, the last year has been a fruitful one for Hobo. We have four major mission-critical applications that have Hobo as a critical component. We have been fortunate to have the skills of Tom Locke and his crew at Artisan to support us, including Bryan Larsen, Gustav Paul, and Angus Miller, and our internal Barquin Hoboists Venka Ashtakala and Jack Compton. They have been critical to our

success with the National Institute of Food and Agriculture (NIFA) Applications Portal (<http://portal.nifa.usda.gov>) and the Leadership Management Dashboard (LMD), as well as other initiatives that will be revealed later.

Please stop by Hobo Central (<http://hobocentral.net>) and join the growing community of developers who are having a great deal of fun while providing their clients with the huge benefits of a state-of-the art agile development framework!

*Owen Dall
Annapolis, Maryland
March 2011*

PREFACE TO THE HOBO 1.0 VERSION FOR RAILS 2

What was our goal?

I starting writing this preface almost exactly a year ago, but put it aside while Jeff and I toiled over iterations of the book outline. While building and rebuilding the outline of what we thought were the book's requirements, we soon realized that would take much more focus and energy than we anticipated completing the project. Our goal seemed simple enough:

"Create a full set of rock-solid instructions and tutorials so that even a novice developer can create, revise, and deploy non-trivial data-rich Web 2.0 applications. The user must have fun while learning, and develop the confidence to take the next step of diving in to learn more about Hobo, Rails and the elegant and powerful object-oriented language behind these frameworks - Ruby."

Right. Well, you know how these things go. So We bit off more than we could chew, at least in the timeframe we envisioned. Instead of three months it took a year...at least it comes out synchronized with the release of Hobo 1.0!

So—we hope we have been at least partially successful. We have had a few “beta” testers of early versions that have made it through without serious injury. More recently it has been reports of minor typos and suggested phrasing enhancements. Letting this simmer for a while has been a good thing.

I hope you are grateful that we parsed off the last 200+ pages into a more advanced companion book with the title “Hobo at Work”.

A brief history

The search for a new web development framework began with the frustrating learning curve and the lack of agility I experienced with the current open source frameworks at

the time. A major client had stipulated that we were to use a totally open source technology stack. In the early 2000's that meant to us Linux, JBoss, Hibernate, MySQL, and Java web frameworks such as Struts. We eventually moved "up" to using Java Server Faces (JSF). The learning curve was steep for our new programmers who were learning on the job.

I was frustrated since I had experience with the "agile" tools of the 1980's and 1990's, which included Revelation and PowerBuilder. These client-server technologies didn't manage to survive into the Internet age. With Revelation an application prototype that could be built in front of a client. We didn't call it Agile Development. We just did it. We built dozens of mission-critical applications and many shrink-wrapped tools. Things were good. Then they weren't. The dinosaurs didn't survive the meteor that hit with the World Wide Web.

So, as the development team lead at one of our major sites as well as the chief systems architect of our small company, I thought it was my duty to start looking for another solution in earnest.

It was in the middle of 2006 that I had a long discussion with Venka Ashtakala about this new quest. (Venka and I had survived two unsuccessful framework searches together starting in 1998. The first was as Alpha testers of the PowerBuilder web converter. Our goal was to migrate a very successful client-server budgeting system used by a large number state and local governments to the web. That experiment was a disaster at the time, so we dropped it.)

A few days after our initial discussion he emailed me about a relatively new framework called "Ruby on Rails" that had gotten some good press. He heard of a few guys who vouched for it, but couldn't find any "mission critical" apps we could use as references. I was intrigued. I did a search and found the first edition of "Agile Development with Rails", I tried it out.

My first simple application worked, but it looked very plain and uninspiring to me. I was a designer and architect and didn't want to code HTML and JavaScript. I didn't want to go backward. "I am too old for this!" was my mantra at the time. I couldn't understand why the framework didn't take care of the basics I had been using for over 20 years. I was also looking for a data-driven navigation system, user authentication, and a decent user interface baked in.

I dropped the search for almost a year. I stumbled on a link in January 2007 about interesting add-ons to Rails, which led to a post by the renowned Ruby evangelist, Peter Cooper. Here are two short quotes.

"You may have thought Ruby on Rails was enough to get Web applications developed quickly, but enter Hobo. Hobo makes the process of creating Web applications and prototypes even quicker. For example, out of the box, with no lines of code written, you get a dummy app with a user signup, login, and authentication system.

... There's quite a lot to Hobo, so you'll want to go through its comprehensive official site and watch the Hobo screen cast to get a real feel for it where a classified ads app is created within minutes."

I watched the screen cast three times. I was blown away. I had finally found someone who *found a solution to web 2.0 apps*. It was Tom Locke.

Following an open source project was totally new to me. I owned my own revision and deployment software business for ver a decade. Proprietary tools had hefty license fees for each installation. The source code wasn't available. Oracle and Microsoft didn't give me the code to their database servers, applications servers, or WYSIWYG design tools. I paid support and expected THEM to fix the problems that were discovered while building our vertical applications in the 1980's and 1990's.

The closest I came to the open source world was being a senior member of the Revelation Roundtable, a board of key developers and integrators for the Revelation and Advanced Revelation development tools. A few products were shrink-wrapped add-ons for other developers. There was a foundation to recommend new development and the ability for a client to speak with the company's president when a serious issue arose.

Posting to a forum and waiting for an answer to my (probably stupid) question didn't come easy for me. This was the thing (I thought) for generation X, not an aging survivor of decades of software wars.

What a welcome and pleasant surprise to find supportive, generous, and incredibly talented people who were willing to help. Even Tom Locke, the creator would patiently answer my questions. Later I was lucky enough to spend time with Tom increasing my respect for his vision and capabilities.

In Early 2008 an opportunity arose at one of our major clients, The National Institute for Food and Agriculture (Formerly CSREES), to migrate a legacy app to the web. I invited the CIO, Michel Desbois, (a forward-looking open source advocate) to experience a demo of building an application using Hobo. My position at NIFA is Chief Systems Architect of the Barquin team, not one of our senior developers. Michel was intrigued that I was going to sit with him without a coder coaching.

That demo led to a small "proof of concept" task to build a "Topic Classification" system for agriculture research projects using Hobo and Oracle as a back end. Michel took a risk and started the ball rolling for us with Hobo.

As this project moved forward, with the addition of more intersted Barquin team members,we urgently needed a solid resource for training not only developers, and requirements analysts and designers. We were building wireframes using software (e.g., Axure) that built great documentation. It even generated HTML pages so you could simulate the page flow of an application.

Unfortunately these became throwaway artifacts, as there was no way of generating a database driven application. *What we needed was a prototyping tool designers could use and then pass on to developers*. Hobo appeared to be the best solution for both prototyping and mission-critical web development. Here is what I reported in May of 2008 about Barquin International's decision to provide some seed money to Hobo:

"This is the first time in over a decade I have been excited about the potential in a new development framework," explains Owen Dall, Chief

Systems Architect for Barquin International, "Although Hobo is already a brilliant and significant enhancement to Rails, we are looking forward to the great leap forward we know is coming..."

More recently we have two significant Hobo development efforts underway that we will put in production this year. The first one is Leadership and Management Dashboard (LMD) led by Joe Barbano, the NIFA Reporting Portal and The REEport (Research, Education and Economics Reporting) project lifecycle reporting system under the direction of John Minge. Dennis Unglesbee, Director of the NIFA Applications Division, has had overall lead responsibility for all of these endeavors.

Anyone who thinks government cannot be agile should come on by and have coffee with the NIFA application development project managers. NIFA has become an innovative "skunk works" that, IMHO, should become a model for public/private collaboration.

A Challenge

How fast could you build an application with the following set of requirements using your current development tool, and have it running, *without touching the database engine*?

- Books have been disappearing from your team's bookshelves. You have been asked to quickly develop a web application that will maintain this library and always know who has what copy of which book.
- Each book title may have any number of copies. Only the administrator, who will be the first one to log in, can enter or edit book titles and details about each copy.
- There will be an automatic signup and login capability accessible from the home page that allows each member of your team to join in, check a book out, or find out who has it so you can track him or her down in the lunch room.
- There is a built-in text search facility that will allow you to search by book name or description.
- Basic Application documentation is generated for you automatically so you can show your team leader what is behind the curtain.

(Now write your estimates down before reading the rest of this page)

OK. Time's up. By the time you consolidated your estimates you would already be up and running with this application using Hobo.

*Owen Dall
Annapolis, Maryland
February, 2010*

Part I

INTRODUCTION AND INSTALLATION

Chapter 1

INTRODUCTION

What is Hobo?

By Tom Locke

Hobo is a software framework that radically reduces the required effort 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 motivation for the Hobo project can be summed up 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. We add features to our products that would otherwise have been far beyond our reach.

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. 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 having 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 almost 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?”, 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!

Pre-Hobo

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 to 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.

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 has 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, The Python language has Django, a web framework with some similar features.

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 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 desired user experience.

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.

Let's get back to the question: "How can the ideas of declarative programming be taken to higher and higher levels?". We mentioned before that one answer to this question is crucial: 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 optimist 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 counter-productive. 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

Bigger library

Obviously the whole point in discovering the secrets of how to build high-level components, is to build 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 1.3 will ship with a decent library of useful “building blocks” to get your app up and running quickly, but there's so much more 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. 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, like prudent use of caching.

This 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.

From another perspective—it was 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. Since Rails was fast enough in 2005, 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 usually comes from very experienced developers who like to know exactly how everything works. Because Hobo gives you so much out-of-the-box, initially you'll be scratching your head about where it all comes from. Fortunately, this is 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 where to look when you need to customize.

As Hobo has developed, we've definitely learned how important it is to make things clear and transparent. The changes from Hobo 0.7 to 0.8 removed a great deal of hard-to-understand “magical” code, 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 “hobo users” discussion group to start “I am new to web programming” or “This is my first attempt to create a web app”. With Hobo people can see a finished result is within their reach, a powerful motivator!

Now that we've seen the appeal to beginners, it's really interesting to find out how far we can push. We've already seen simple Hobo applications created by people that don't really know computer programming. 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 the essence of Hobo. 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 your 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 develop stuff. Enjoy!

Fundamentals

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 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 Ruby metaprogramming power.

Hobo takes this paradigm further and 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 othertimes 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.

Hobo views writing this kind of code as a ubiquitous repetitive pattern. This, Hobo allows you to 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-ins, 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:

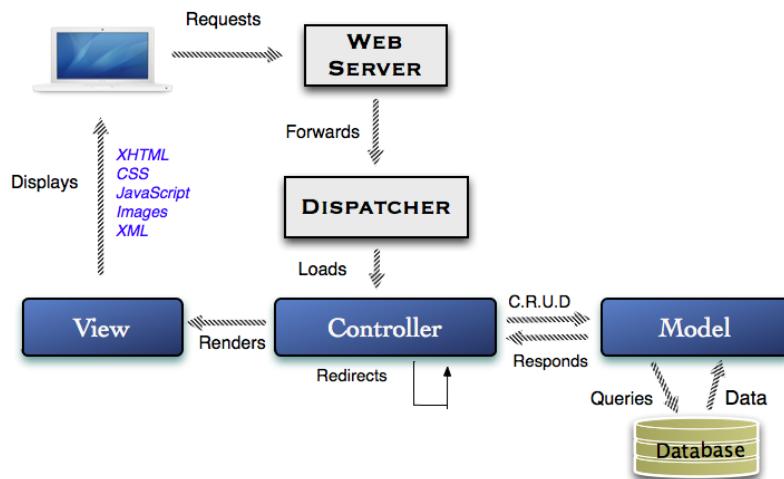


Figure 1: Data flow for a typical Application using a MVC framework

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

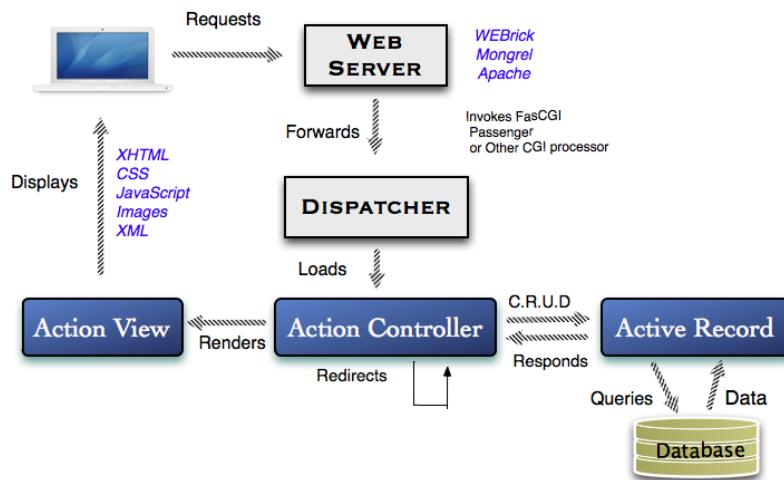


Figure 2: Data flow for a Rails application

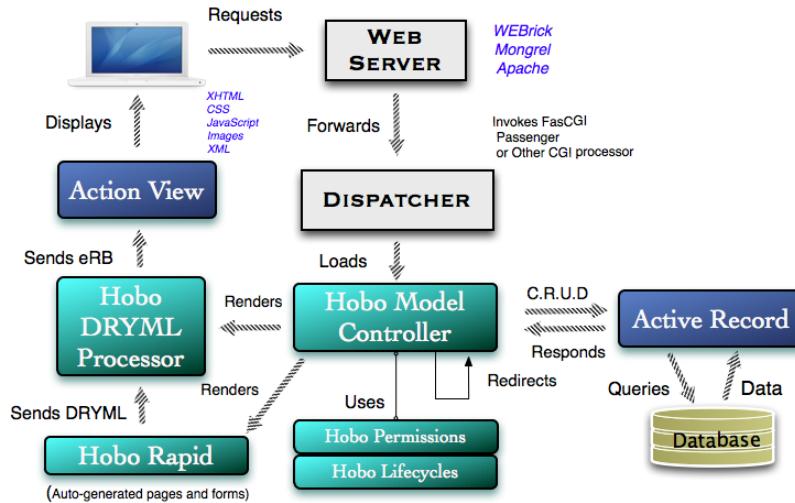


Figure 3: Data flow for a Hobo application

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.

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 editing 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 “resource” generator creates models, controllers, and views:

```
> hobo generate resource [parameters]
```

Any changes to field definitions or associations in the model can be propagated throughout the application with the Hobo “migration” generator:

```
> hobo generate migration [parameters]
```

There is no need to edit the migration file. The migration generator handles this for you.

If you only need to create a model without other resources, use the Hobo model generator:

```
> hobo generate model [parameters]
```

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.

Chaper 2

INSTALLATION

Note: This Book is for Hobo version 1.3 which is only compatible with Rails version 3. If you are using Rails 2, please use the previously published book, “Rapid Rails with Hobo.”

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 give included 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. It 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 for 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 and Hobo

- If you already have Ruby and Rails version 3 installed, you can skip to step #3.
- If you have a Mac with OS X Snow Leopard, Ruby 1.8.7 and Rails 2.3.5 are pre-installed. You can also skip to step #3
- If you are using a PC with Linux, First see this link for installing Ruby and Rails on Ubuntu and Debian Linux, and then skip to step #3:
<http://wiki.rubyonrails.org/getting-started/installation/linux-ubuntu>
- If you are using a PC with Windows XP or Windows 7, and are new to Ruby and Rails, Start with Step #1 below:

Step 1. Download the “Rails Installer” Kit from <http://railsinstaller.org>:

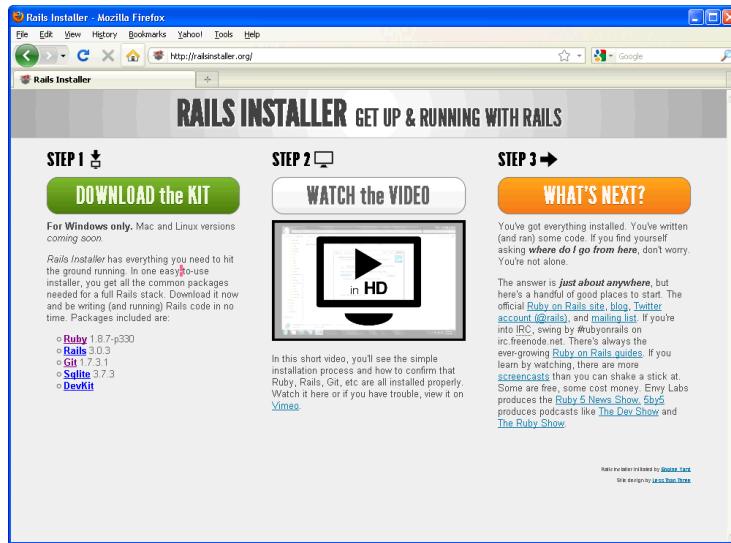


Figure 4: Download Site for the Rails Installer

Step 2. Double-click on the file `railsinstaller-1.0.4.exe` (as of February 14, 2011) to run the installer.

The following will be installed for you through the visual wizard:

- Ruby 1.8.7-p330
- Rails 3.0.3
- Git 1.7.3.1

- SQLite 3.7.3

- DevKit

(Again, these versions will change over time.)



Figure 5: Rails Installer Setup Wizard

Pressing the “Next” button will start the installation. You will be prompted at each step to provide configuration information.

The first configuration you will be prompted for is an installation directory. A default one is provided to you, but in this tutorial we chose the option to install all in the folder:

C:\Rails3

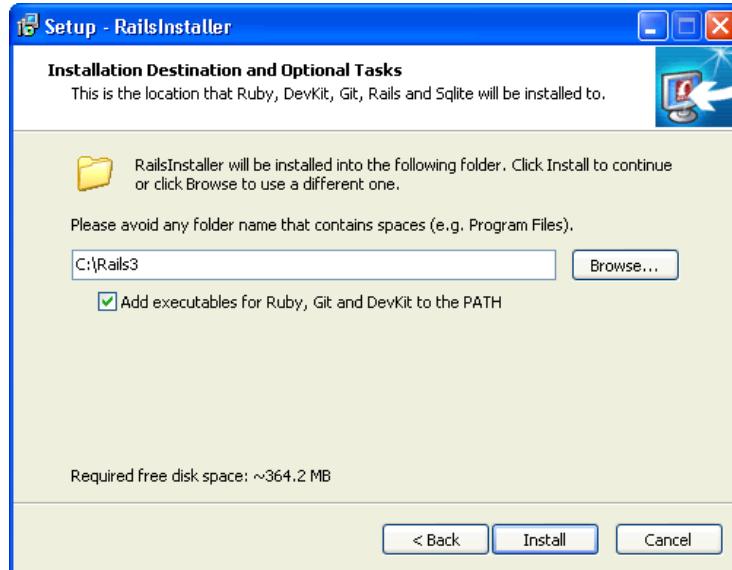


Figure 6: Choose the Installation Directory



Figure 7: Completed Setup

After the installation is complete, you will see a new menu option called “RailsInstaller” and four sub-menus similar to the following:

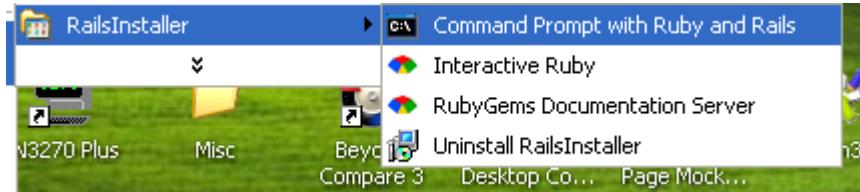


Figure 8: Menu Path

If you select the sub-menu option “Command Prompt with Ruby and Rails” you will see a command window that appears as follows:

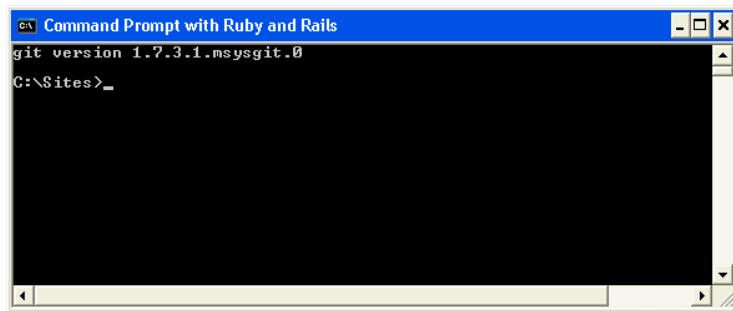


Figure 9: Command Prompt

Note that a default “Sites” folder was created for you, and that there is even a sample Rails 3 application included:

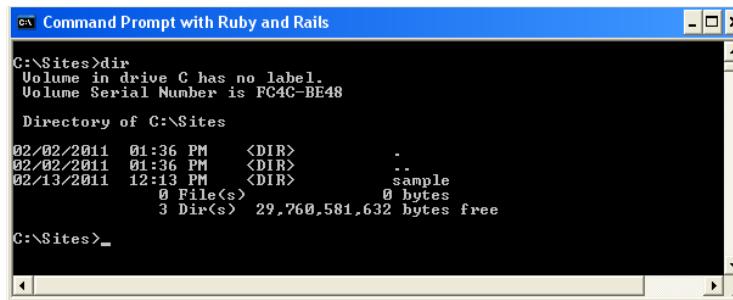


Figure 10: Rails Installer Sample Application Folder

Selecting the Interactive Ruby sub-menu brings up a console similar to the following:

INSTALLING RUBY, RAILS AND HOBO

CHAPTER 2 INSTALLATION

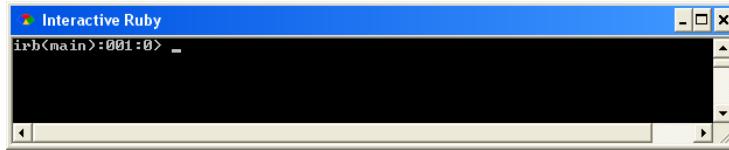


Figure 11: IRB

For more information, please check out the following link:

<http://www.ruby-lang.org/en/documentation/quickstart/>

Selecting the RubyGems Documentation Server sub-menu option will start a web server on port 8808 that provides hyperlinked information about each gem you have installed:

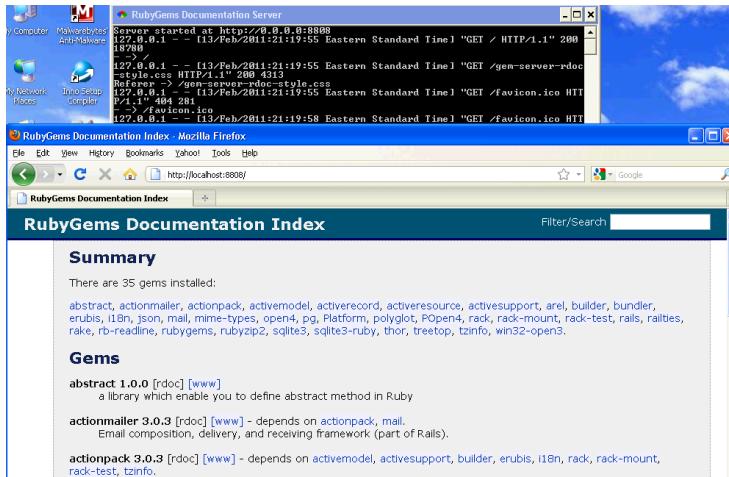


Figure 12: RubyGems Documentation

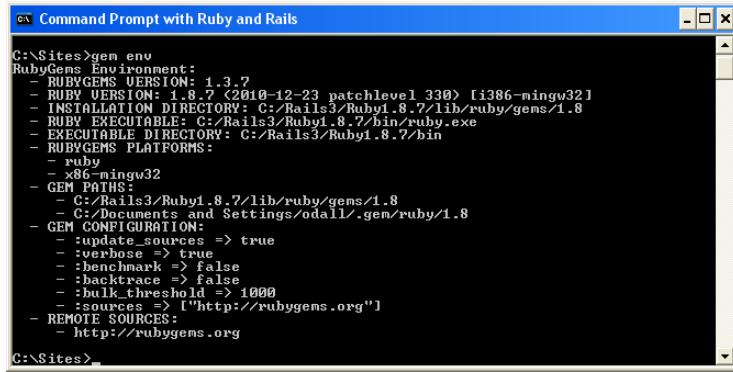
For more information, please check out the following link:

<http://docs.rubygems.org/>

Click on the “Command Prompt with Ruby and Rails” and type in the following command:

```
C:\Sites> gem env
```

The “gem env” (gem environment) provides information that will later be useful for debugging.



```
C:\Sites>gem env
RubyGems Environment:
  - RUBYGEMS VERSION: 1.3.7
  - RUBY VERSION: 1.8.7 (2009-12-23 patchlevel 330) [i386-mingw32]
  - INSTALLATION DIRECTORY: C:/Rails3/Ruby1.8.7/lib/ruby/gems/1.8
  - RUBY EXECUTABLE: C:/Rails3/Ruby1.8.7/bin/ruby.exe
  - EXECUTABLE DIRECTORY: C:/Rails3/Ruby1.8.7/bin
  - RUBYGEMS PLATFORMS:
    - ruby
    - x86-mingw32
  - GEM PATHS:
    - C:/Rails3/Ruby1.8.7/lib/ruby/gems/1.8
    - C:/Documents and Settings/odall/.gem/ruby/1.8
  - GEM CONFIGURATION:
    - :update_sources => true
    - :verbose => true
    - :benchmark => false
    - :backtrace => false
    - :bulk_threshold => 1000
    - :sources => ["http://rubygems.org"]
  - REMOTE SOURCES:
    - http://rubygems.org

C:\Sites>
```

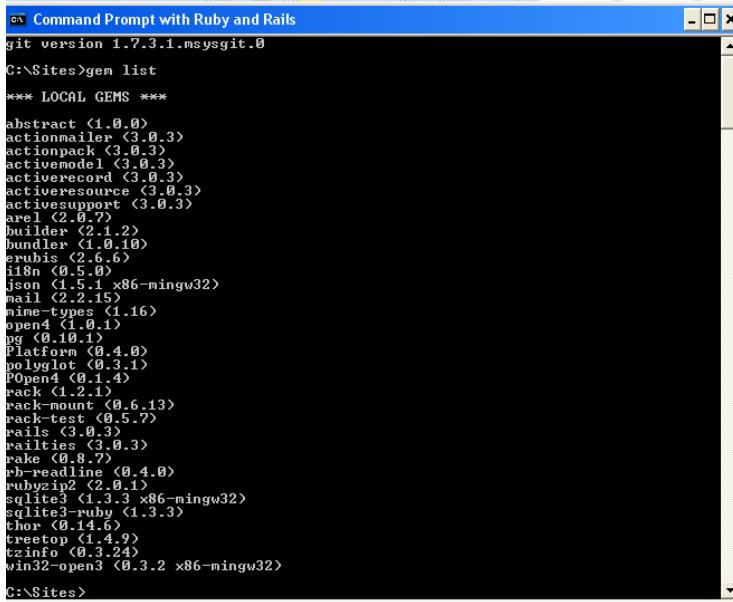
Figure 13: Command Prompt

Now type the `gem list` command:



```
C:\Sites> gem list
```

The `gem list` command provides information about which ruby gems (modules) are installed and what versions are available.



```
git version 1.7.3.1.msysgit.0
C:\Sites>gem list
*** LOCAL GEMS ***
abstract <1.0.0>
actionmailer <3.0.3>
actionpack <3.0.3>
activemodel <3.0.3>
activerecord <3.0.3>
activereource <3.0.3>
activesupport <3.0.3>
arel <2.0.7>
builder <2.1.2>
bundler <1.0.10>
erubis <2.6.6>
i18n <0.5.0>
json <1.5.1-x86-mingw32>
mail <2.2.15>
minitest <1.16>
open4 <1.0.1>
pg <0.10.1>
Platform <0.4.0>
polyglot <0.3.1>
POpen4 <0.1.4>
rack <1.2.1>
rack-mount <0.6.13>
rack-test <0.5.7>
rails <3.0.3>
railties <3.0.3>
rake <0.8.7>
rb-readline <0.4.0>
rubyzip2 <2.0.1>
sqlite3 <1.3.3 x86-mingw32>
sqlite3-ruby <1.3.3>
thor <0.14.6>
treetop <1.4.9>
tzinfo <0.3.24>
win32-open3 <0.3.2 x86-mingw32>

C:\Sites>
```

Figure 14: List of Ruby Gems installed by the Rails Installer

Now test to see the SQLite3 is available. From the command line type “`sqlite3`”:

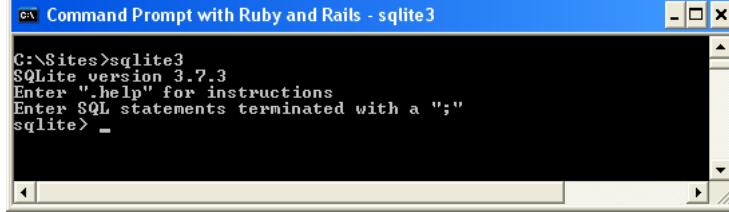


Figure 15: Testing SQLite3

For a nice introduction to the use of SQLite3, access the following link:

<http://www.sqlite.org/quickstart.html>

Note that the Rails Installer comes with a sample application in the `Sites/sample` directory:

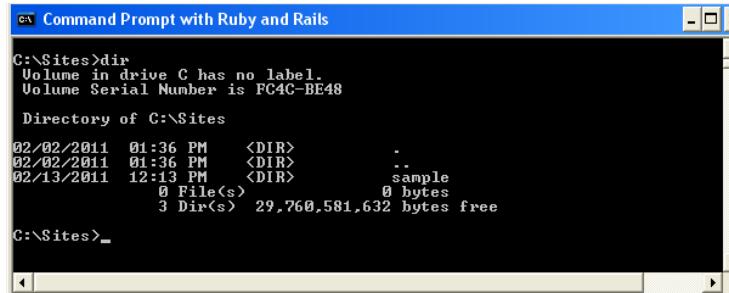


Figure 16: The Sample Rails 3 app included by Rails Installer

Step 3. Install Hobo.

Type the following command at the command prompt:

```
C:\Sites> gem install hobo -v 1.3.0.pre26 --pre
```

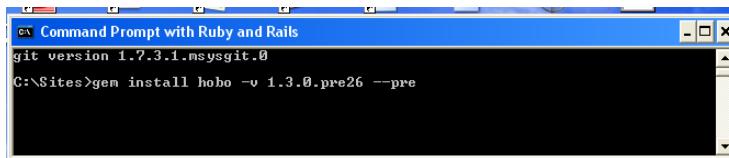


Figure 17: Command Prompt with Ruby and Rails

Note: In the screen shots captured here we used a recent pre-release version of Hobo. Notice we were required to use the “—pre” command option to install it.

When Hobo 1.3.0 is officially released, the installation command will be:

```
C:\Sites> gem install hobo -v 1.3.0
```

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

```
C:\Sites> gem list
```

```
git version 1.7.3.1.msysgit.0
C:\Sites>gem list
*** LOCAL GEMS ***
abstract (1.0.0)
actionmailer (<3.0.4, 3.0.3>
actionpack (<3.0.4, 3.0.3>
activemodel (<3.0.4, 3.0.3>
activerecord (<3.0.4, 3.0.3>
activeresource (<3.0.4, 3.0.3>
activesupport (<3.0.4, 3.0.3>
arel (<2.0.7>
builder (<2.1.2>
bundler (<1.0.10>
dryml (<1.3.0.pre26>
erubis (<1.3.0>
hobon (<1.3.0.pre26>
hobo-fields (<1.3.0.pre26>
hobo-support (<1.3.0.pre26>
i18n (<0.5.0>
json (<1.5.1.x86-mingw32>
mail (<2.2.15>
mime-types (<1.16>
open4 (<1.0.1>
pg (<0.10.1>
Platform (<0.4.0>
polyglot (<0.3.1>
POpen4 (<0.1.4>
rack (<1.2.1>
rack-mount (<0.6.13>
rack-test (<0.5.7>
rails (<3.1.3>
railties (<3.0.4, 3.0.3>
rake (<0.8.7>
rb-readline (<0.4.0>
rubyzip2 (<2.0.1>
sqlite3 (<1.3.3.x86-mingw32>
sqlite3-ruby (<1.3.3>
thor (<0.14.6>
treefort (<1.4.9>
tzinfo (<0.3.24>
will_paginate (<3.0.pre2>
win32-open3 (<0.3.2.x86-mingw32>
C:\Sites>
```

Figure 18: Sample console output from the "gem list" command after installing Hobo

Note: In the example above, 3.0.4 was installed by Hobo 1.3.0, as Rails 3.0.4 was defined as a dependency. This may differ over time.

If you find the need to start completely fresh, select the Uninstall RaisInstaller sub-menu option.

CHAPTER 2 USING MYSQL WITH HOBO

Step 4. Choose and configure a text editor (optional) You can work through the tutorials in this book using Textpad or your favorite editor. However, for years TextMate (<http://macromates.com/>) for the Mac has been the most popular light-weight editor for Ruby and Rails, and offers many productivity features. There is an inexpensive “clone” for Windows called “E” that is very good facsimile of TextMate. You can download a 30-day evaluation version from <http://www.e-texteditor.com/>:

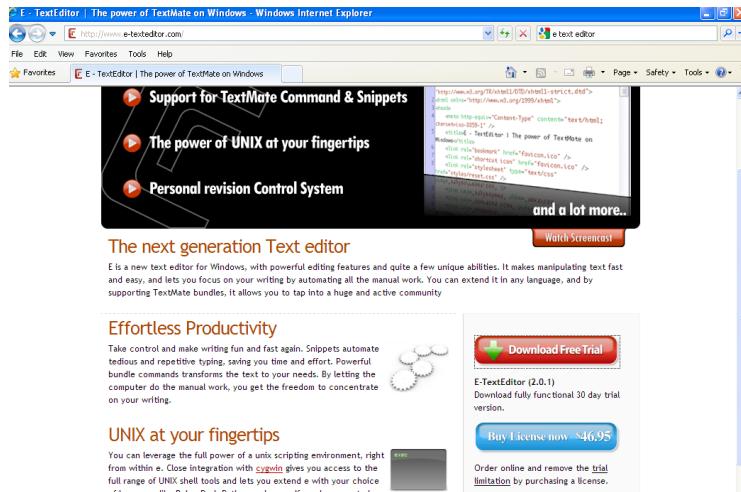


Figure 19: TextMate Webpage

Also take a look at two popular full Integrated Development Environments you might find useful:

<http://www.aptana.com/products/radrails>

<http://netbeans.org/features/ruby/index.html>

Now you are ready to start using Hobo with the default database engine for Rails and Hobo—SQLite.

Skip to Chapter 3 (Introductory Tutorials) unless you prefer to use MySQL or Oracle.

Instructions for installation of these database engines are next.

Using MySQL with Hobo

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:

<http://dev.mysql.com/downloads/mysql/#downloads>

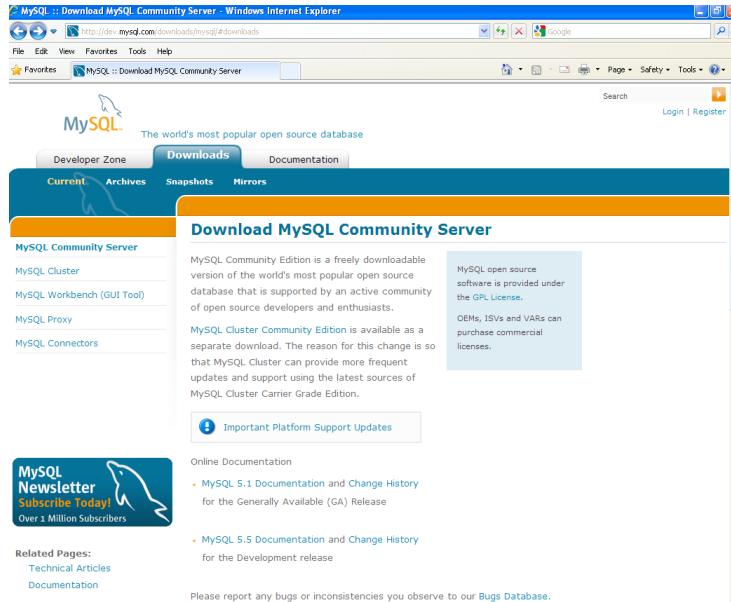


Figure 20: Download site for MySQL

Although the Community Server is free, you will need to create an account before you download. After creating an account, you will be directed to the download page:

Click on one of the mirror sites to begin the download. Then click the “Run” button when prompted to begin the installation:



Figure 21: Using the downloaded .msi file to install MySQL on Windows

Choose the “Custom” option when prompted:

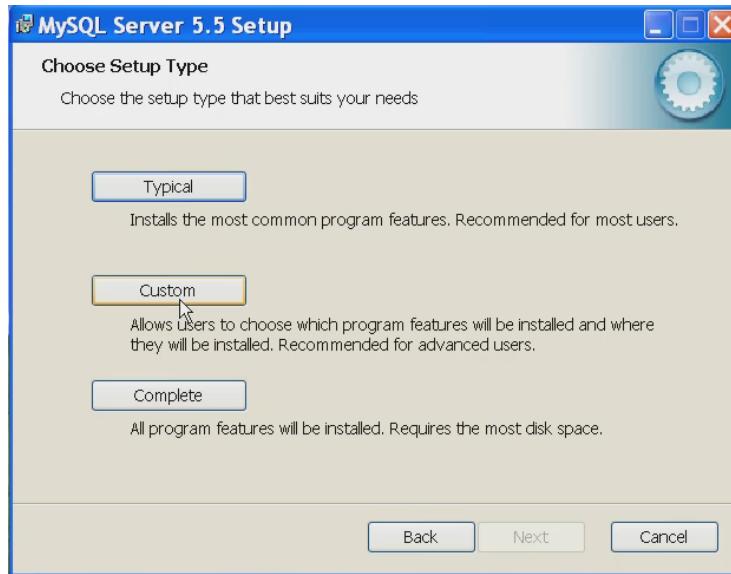


Figure 22: Choose the “Custom” setup type

Next choose the locations for the server code and data files. Note the default location

in Windows is:

C:\Program Files\MySQL\MySQL Server 5.5\

We suggest a directory path that is more succinct:

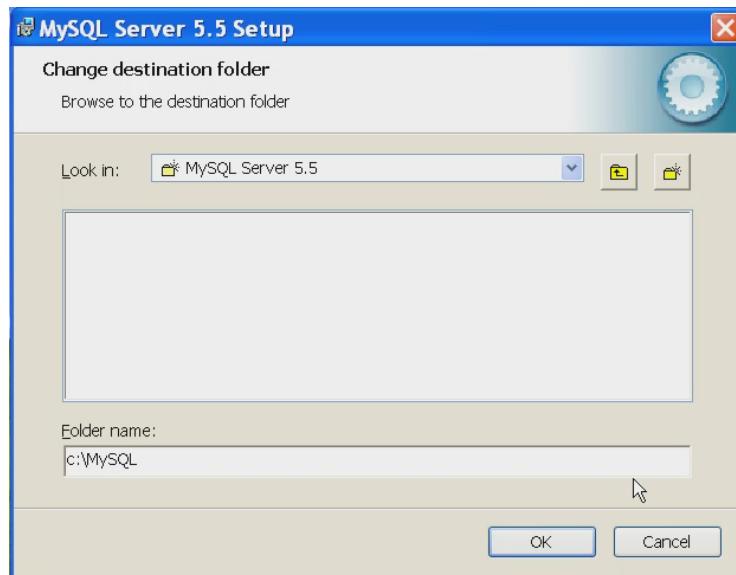


Figure 23: Specify the destination folder "C:\MySQL" for the server software

Specify the location of the data files used by MySQL databases:

CHAPTER 2
USING MYSQL WITH HOBO

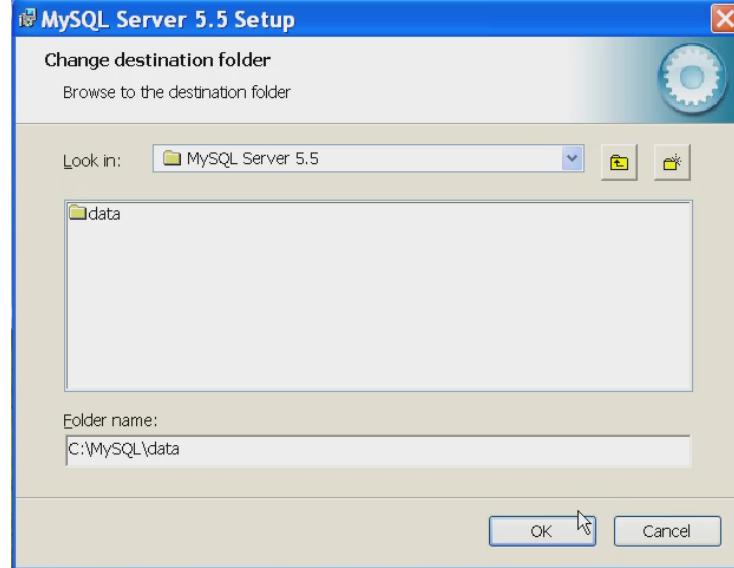


Figure 24: Specify the destination folder "C:\MySQL\data" to hold MySQL data

Click “OK” to continue:

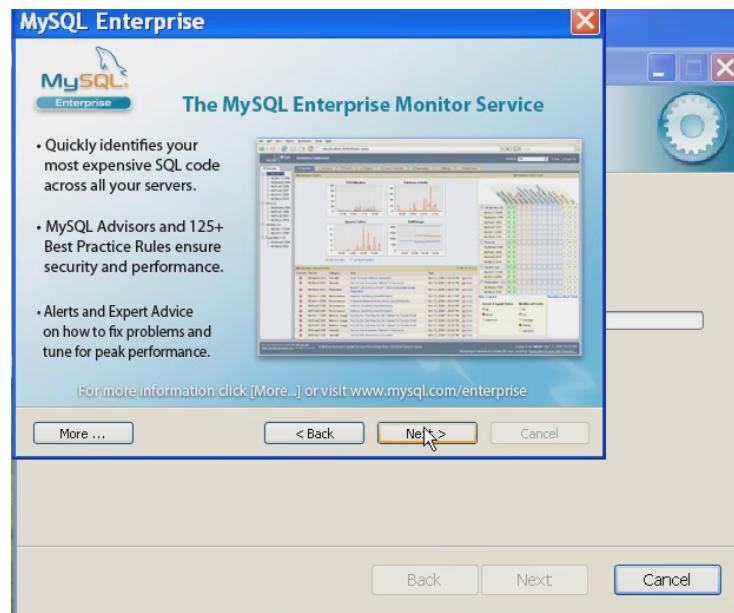


Figure 25: MySQL Enterprise Monitor

Then Click “Next” to launch the MySQL Instance Configuration Wizard:



Figure 26: The MySQL Instance Configuration Wizard

Now click “Finish”:



Figure 27: MySQL Instance Configuration Wizard

The next step is to choose instance configuration option. We recommend choosing the “Standard Configuration” option.



Figure 28: MySQL Instance Configuration Wizard

Click “Next” to obtain further configuration options. Select both “Install As Windows Service” and “Include Bin Directory in Windows PATH”:



Figure 29: MySQL Instance Configuration Wizard

To make the application creation process for MySQL similar to using SQLite, un-click the default checkbox for “Modify Security Settings.” (This will remove the need to provide the MySQL “root” user password when using the Hobo setup wizard:



Figure 30: MySQL Instance Configuration Wizard

CHAPTER 2
USING MYSQL WITH HOBO

Click “Next” to continue. A status window similar to the following will be displayed:

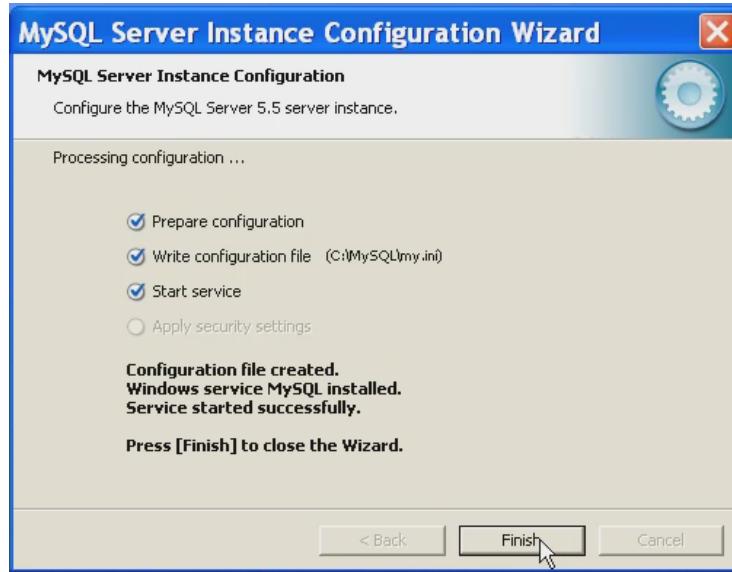


Figure 31: MySQL Instance Configuration Wizard

Click “Finish” to complete the installation.

You should now see a new “MySQL” menu item in, from which you can launch the MySQL Command Line Client:



Figure 32: MySQL Command Line Client

The Command Line Client will appear similar to the following:

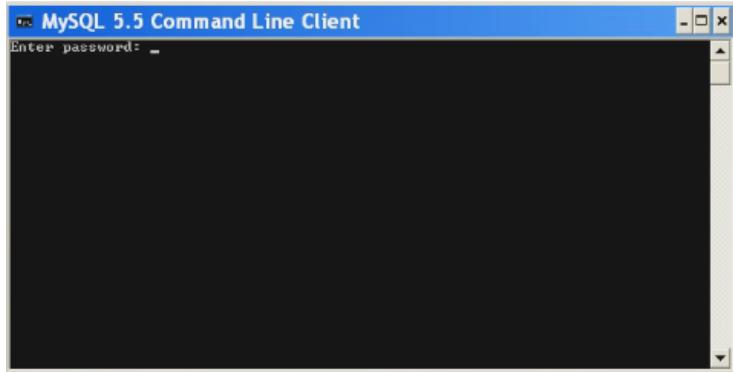


Figure 33: MySQL Command Line

Because we opted to use the default MySQL security settings in our installation (no password requirement for the “root” user), simply press [Enter] when prompted for a password to access the MySQL command prompt:

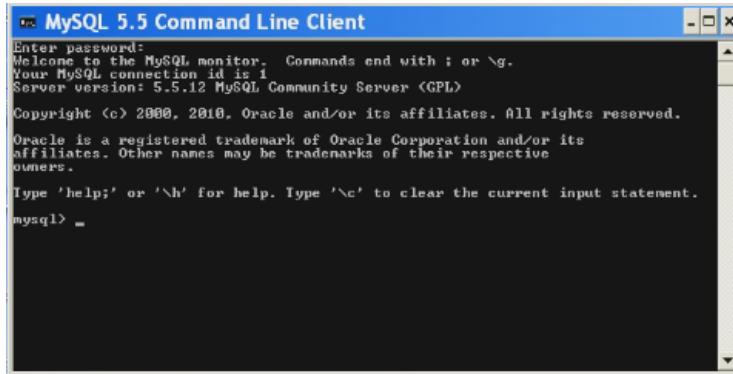


Figure 34: MySQL Command Line

Step 2: Install the Ruby Gem for MySQL 5.5

The next step is to install the following Ruby gem for connecting to MySQL 5.5:

```
C:\Sites\tutorials> gem install mysql2 -v 0.2.7 --  
'--with-mysql-lib="C:\MySQL\include"'
```

If you installed MySQL in a different location you may have to adjust the previous command to reflect the appropriate location in your system.

CHAPTER 2

USING MYSQL WITH HOBO

You may run into the issue in Windows systems with having the “libmysql.dll” file in a path that the mysql2 Ruby gem can access:

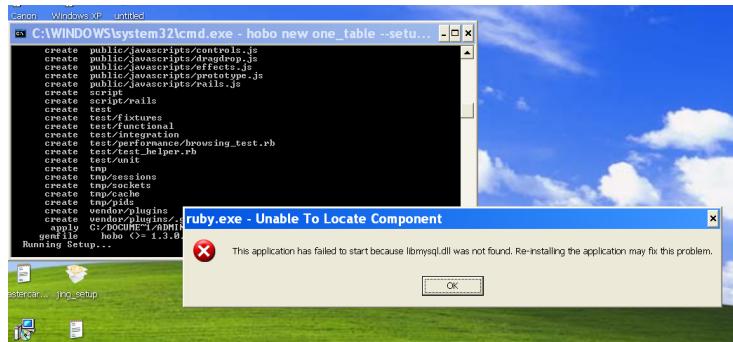


Figure 35: Ruby.exe Error

If you have this problem, you will find this DLL in the following MySQL directory:

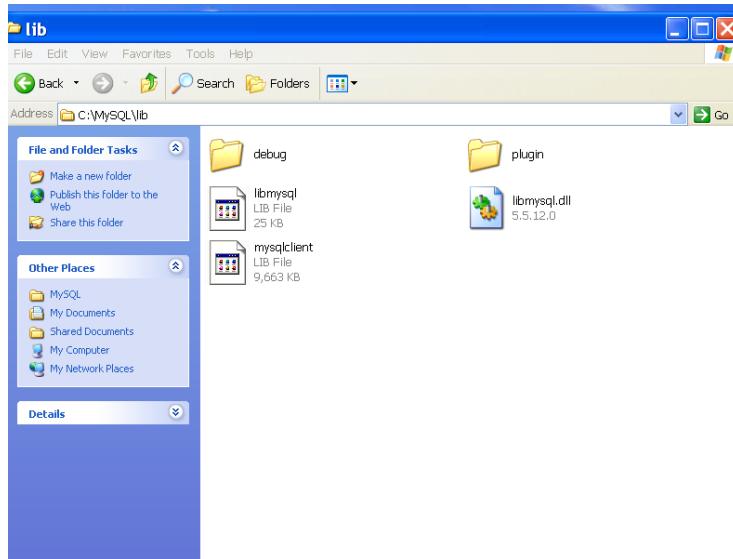


Figure 36: MySQL lib Directory

Copy this file and place it in the Ruby “bin” directory:

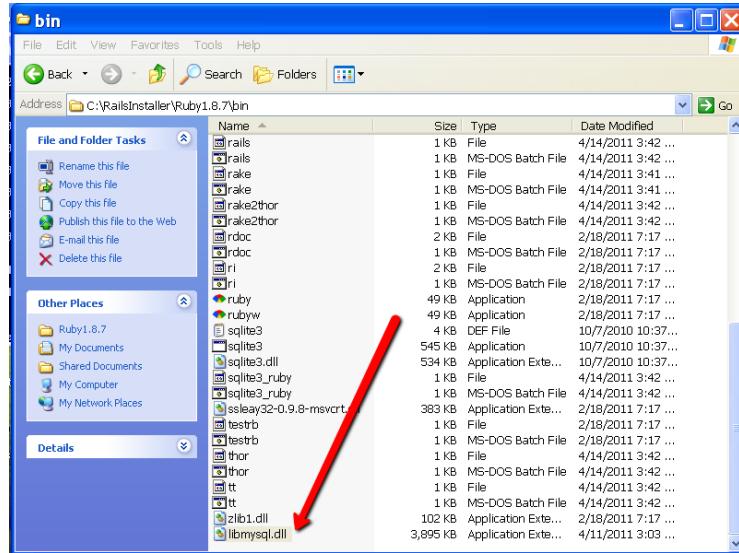


Figure 37: Ruby bin Directory

Step 3: Generate a Hobo MySQL Application from the Command Line

Now you can generate a Hobo MySQL app using the following command:

```
C:\Sites\tutorials> hobo new one_table -setup -d mysql
```

This will create the “one_table” application and run the migrations necessary for the default “user” Hobo user model.

Now edit the database.yml file to see what was created automatically:

CHAPTER 2 USING MYSQL WITH HOBO

```

C:\hobo1_3\Tutorials\one_table\config\database.yml - e [UNREGISTERED - 30 DAYS LEFT OF TRIAL]
File Edit View Text Navigation Bundles Help
Project database.yml
1 # MySQL. Versions 4.1 and 5.0 are recommended.
2 # Install the MySQL driver:
3 #   gem install mysql2
4 #
5 # And be sure to use new-style password hashing:
6 #   http://dev.mysql.com/doc/refman/5.0/en/old-client.html
7 #
8 development:
9   adapter: mysql2
10   encoding: utf8
11   reconnect: false
12   database: one_table_development
13   pool: 5
14   username: root
15   password:
16   host: localhost
17
18 # Warning: The database defined as "test" will be erased and
19 # re-generated from your development database when you run "rake".
20 # Do not set this db to the same as development or production.
21 test:
22   adapter: mysql2
23   encoding: utf8
24   reconnect: false
25   database: one_table_test
26   pool: 5
27   username: root
28   password:
29   host: localhost
30
31 production:
32   adapter: mysql2
33   encoding: utf8
34   reconnect: false
35   database: one_table_production
36   pool: 5
37   username: root
38   password:
39   host: localhost

```

Figure 38: The automatically created database.yml file

Note: It is pre-filled with the proper parameter structure for MySQL.

Fire up your MySQL command line again and use the “show databases” command to see what was created:

```

MySQL 5.5 Command Line Client
Your MySQL connection id is 1
Server version: 5.5.12 MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| one_table_development |
| one_table_test     |
| performance_schema |
| test               |
+--------------------+
6 rows in set (0.38 sec)

mysql>

```

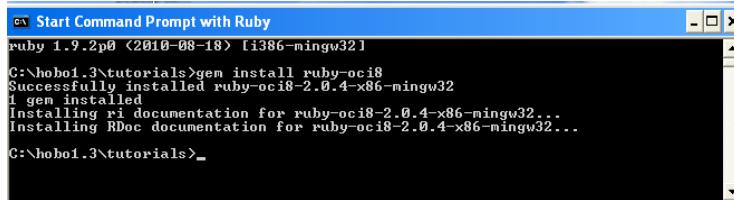
Figure 39: MySQL Command Line

Note: Hobo and Rails make some assumptions about the naming of the development and test database names. You can change these as needed, but it is useful to understand the default convention used by Rails, which is what Hobo uses.

Using Oracle with Hobo

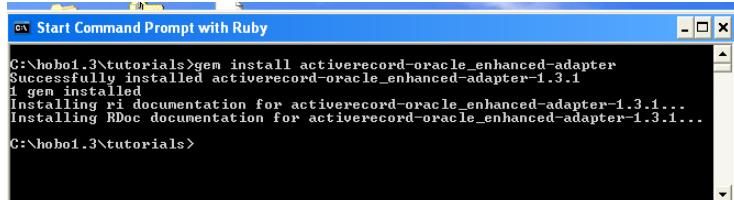
We will discuss the following two configuration options: 1. Use an existing Oracle database 2. Download and install a fresh Oracle database For either of these options you will first need to install the following two ruby gems:

```
C:\Sites\tutorials> gem install ruby-oci8
C:\Sites\tutorials> gem install
    activerecord-oracle_enhanced-adapter
```



```
Start Command Prompt with Ruby
ruby 1.9.2p0 <2010-08-18> [i386-mingw32]
C:\hobo1.3\tutorials>gem install ruby-oci8
Successfully installed ruby-oci8-2.0.4-x86-mingw32
1 gem installed
Installing ri documentation for ruby-oci8-2.0.4-x86-mingw32...
Installing RDoc documentation for ruby-oci8-2.0.4-x86-mingw32...
C:\hobo1.3\tutorials>-
```

Figure 40: Console output after installing Oracle gems for Ruby and Rails



```
Start Command Prompt with Ruby
C:\hobo1.3\tutorials>gem install activerecord-oracle_enhanced-adapter
Successfully installed activerecord-oracle_enhanced-adapter-1.3.1
1 gem installed
Installing ri documentation for activerecord-oracle_enhanced-adapter-1.3.1...
Installing RDoc documentation for activerecord-oracle_enhanced-adapter-1.3.1...
C:\hobo1.3\tutorials>-
```

Figure 41: Console output after installing Oracle gems for Ruby and Rails

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.

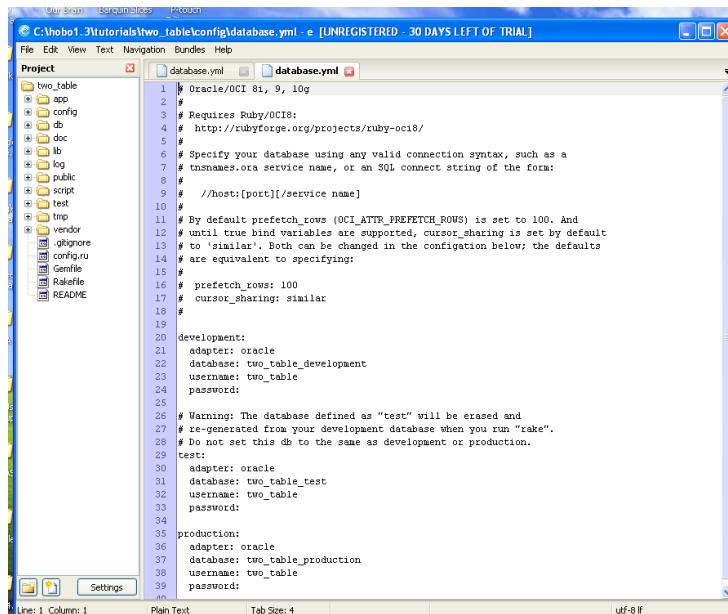
You probably have different database “instances” for development, test, and production systems. If you are lucky you might even have rights to create a new database user (i.e., schema) in your development environment. In most large shops you will probably need to request that the database administrator (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:\Sites\tutorials> hobo new two_table -d oracle
```

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



The screenshot shows a code editor window titled "C:\hobo1.3\Tutorials\two_table\config\database.yml - e [UNREGISTERED - 30 DAYS LEFT OF TRIAL]". The file content is a configuration for an Oracle database:

```

# Oracle/OCI 8i, 9, 10g
# Requires Ruby/OCI8:
#   http://rubyforge.org/projects/ruby-oci8/
#
# Specify your database using any valid connection syntax, such as a
# tnsnames.ora service name, or an SQL connect string of the form:
#   /host:[port]/[service name]
# By default prefetch_rows (OCI_ATTR_PREFETCH_ROWS) is set to 100. And
# until true bind variables are supported, cursor_sharing is set by default
# to 'similar'. Both can be changed in the configuration below; the defaults
# are equivalent to specifying:
#   prefetch_rows: 100
#   cursor_sharing: similar
#
development:
  adapter: oracle
  database: two_table_development
  username: two_table
  password:
  #
# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: oracle
  database: two_table_test
  username: two_table
  password:
  #
production:
  adapter: oracle
  database: two_table_production
  username: two_table
  password:

```

Figure 42: The generated database.yml file for Oracle

Since we recommended using the “oracle_enhanced” adapter, we will need to replace the plain “oracle” with “oracle_enhanced”

Also, 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 create an Oracle user (schema) called “two_table_development”.

The changed entries in the database.yml file will look like the following:

```
development:  
  adapter: oracle_enhanced  
  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 the database migrations (“`hobo g migration`”) as you will see in Tutorial 1.

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 accessible via a web browser.

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/xe/htdocs/102xewinsoft.html>

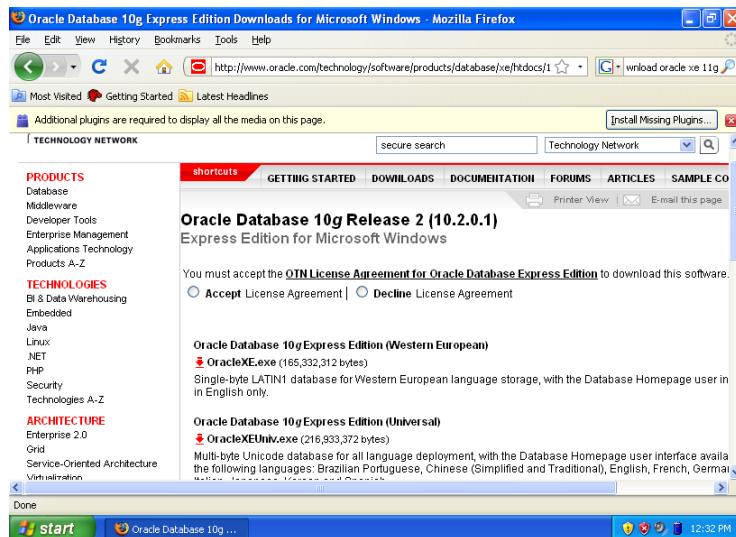


Figure 43: Oracle database install download site

CHAPTER 2 USING ORACLE WITH HOBO

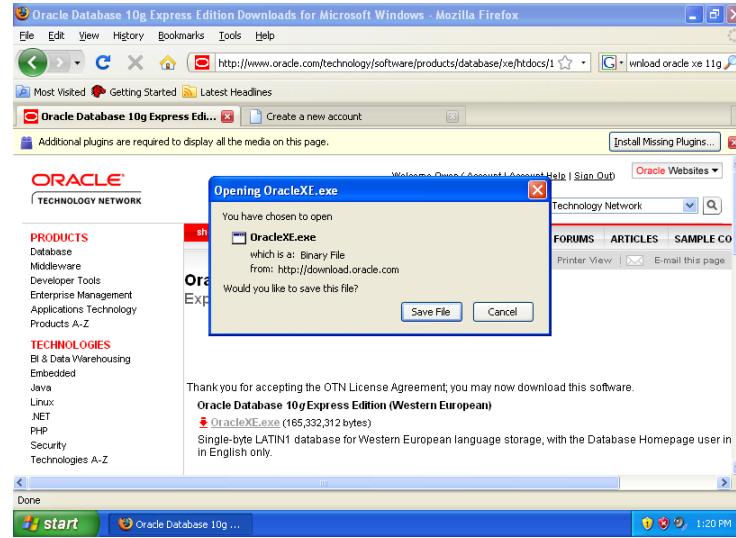


Figure 44: Running the Oracle XE installation

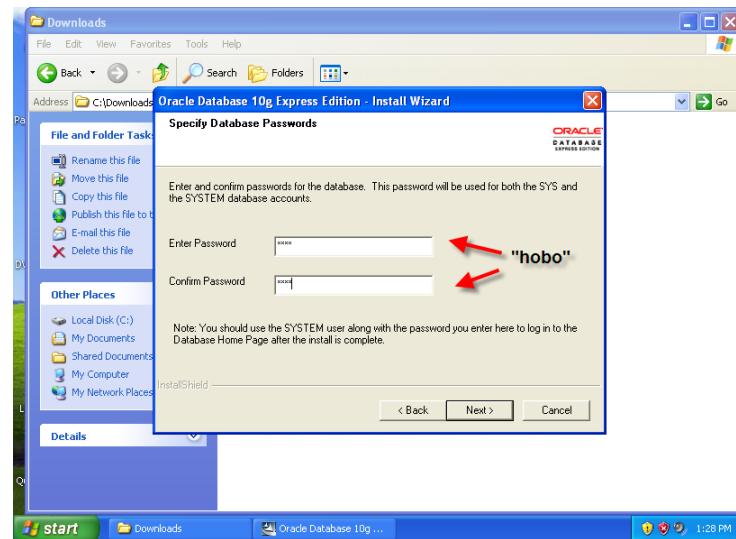


Figure 45: Specifying the database passwords

CHAPTER 2
INSTALLATION

USING ORACLE WITH HOBO

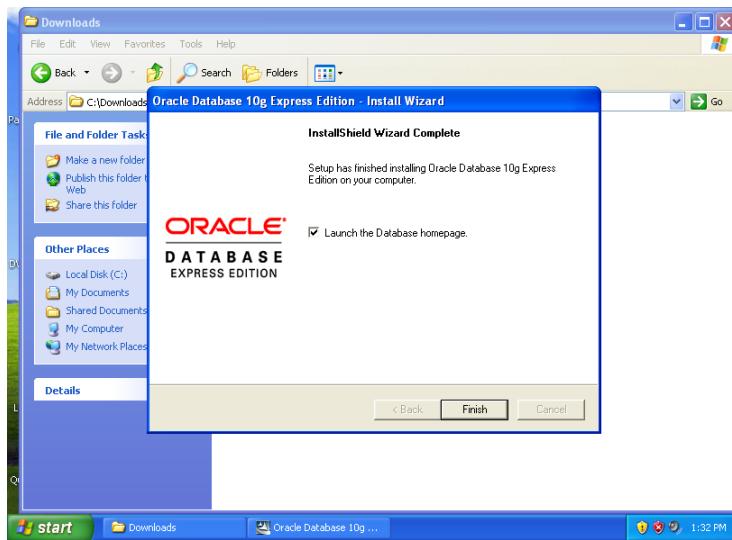


Figure 46: Launch the Database home page

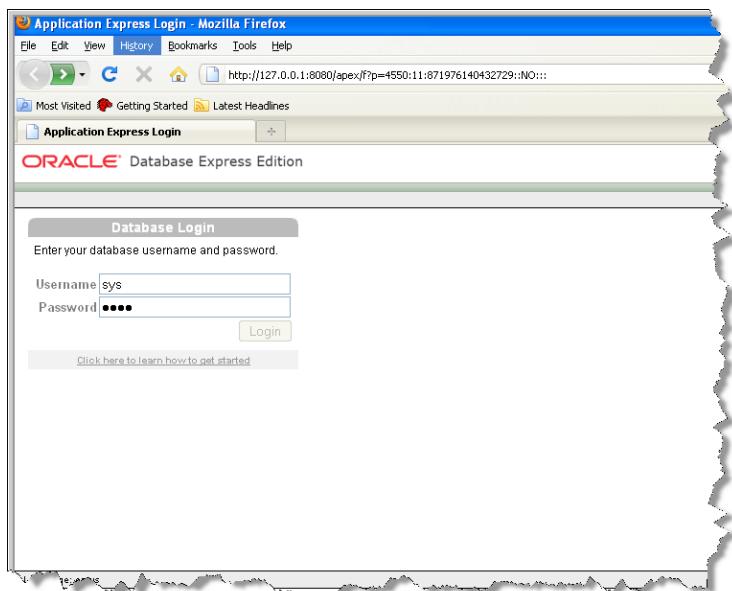


Figure 47: Log in as SYS to configure your database

CHAPTER 2 USING ORACLE WITH HOBO

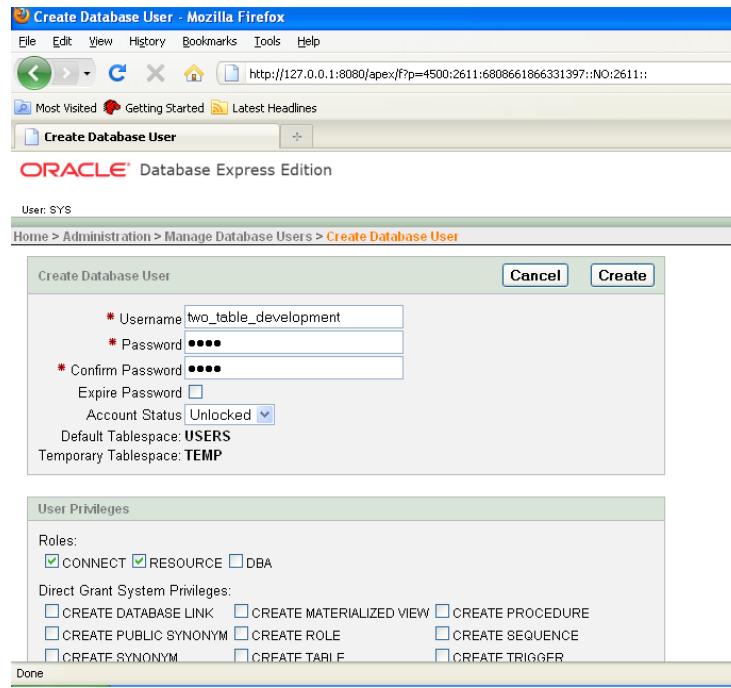


Figure 48: Creating a schema/user to use with Hobo

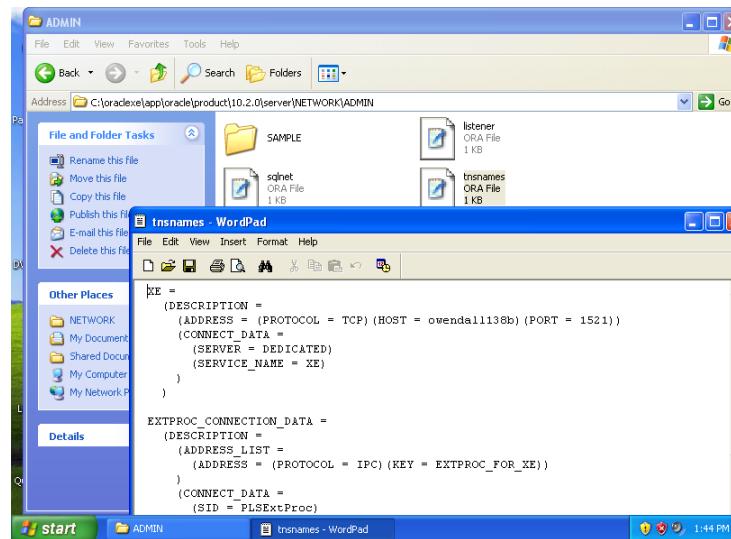


Figure 49: The tnsnames.ora file created during installation

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

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

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

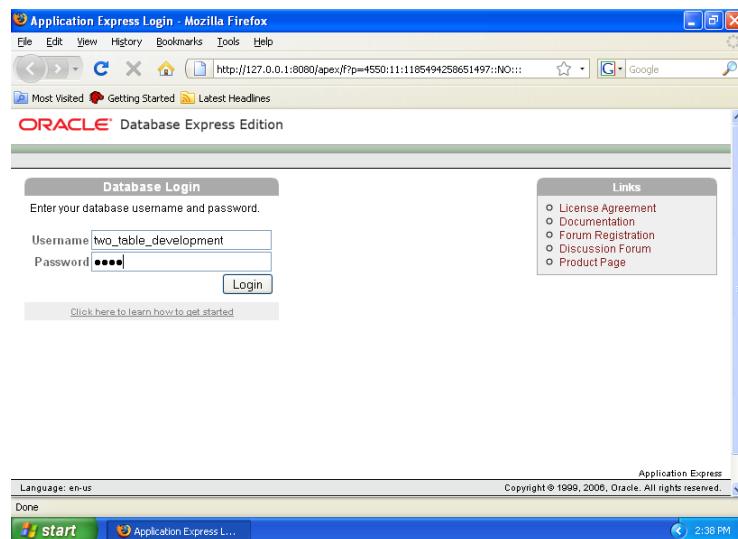


Figure 50: Log into Oracle to view the created table

CHAPTER 2 USING ORACLE WITH HOBO

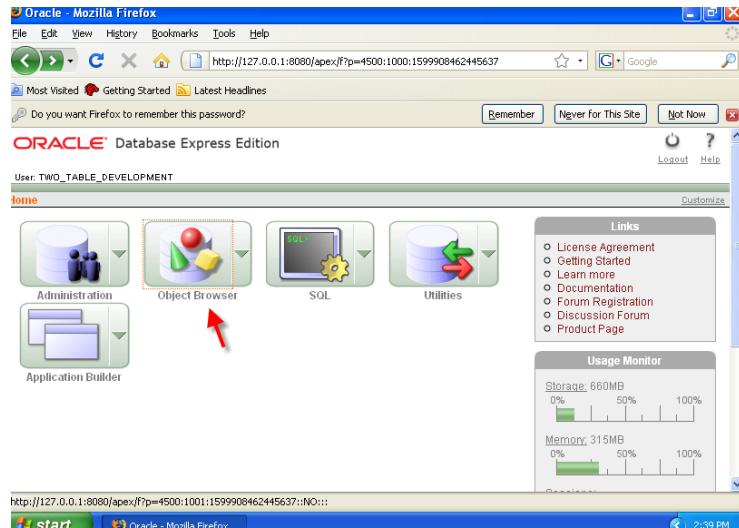


Figure 51: Access the Oracle Object Browser

The screenshot shows the Oracle Object Browser interface. The title bar says 'Object Browser - Mozilla Firefox'. The main area displays the 'USERS' table structure. The table has 12 columns: ID, CRYPTED_PASSWORD, SALT, REMEMBER_TOKEN, REMEMBER_TOKEN_EXPIRES_AT, NAME, EMAIL_ADDRESS, ADMINISTRATOR, CREATED_AT, UPDATED_AT, STATE, and KEY_TIMESTAMP. The 'ID' column is defined as NUMBER(38,0) and is the primary key. The 'NAME' column is VARCHAR2(255). The 'EMAIL_ADDRESS' column is VARCHAR2(255). The 'ADMINISTRATOR' column is NUMBER(1,0). The 'CREATED_AT' and 'UPDATED_AT' columns are DATE. The 'STATE' column is VARCHAR2(255) with a default value of 'active'. The 'KEY_TIMESTAMP' column is DATE. There are tabs for 'Table', 'Data', 'Indexes', 'Model', 'Constraints', 'Grants', 'Statistics', 'UDF Defaults', 'Triggers', 'Dependencies', and 'SQL'. A 'Create' button is visible at the top right. The URL in the address bar is [http://127.0.0.1:8080/apex/f?p=4500:1001:1599908462445637::NO:::.](http://127.0.0.1:8080/apex/f?p=4500:1001:1599908462445637::NO:::)

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(38,0)	No	-	1
CRYPTED_PASSWORD	VARCHAR2(40)	Yes	-	-
SALT	VARCHAR2(40)	Yes	-	-
REMEMBER_TOKEN	VARCHAR2(255)	Yes	-	-
REMEMBER_TOKEN_EXPIRES_AT	DATE	Yes	-	-
NAME	VARCHAR2(255)	Yes	-	-
EMAIL_ADDRESS	VARCHAR2(255)	Yes	-	-
ADMINISTRATOR	NUMBER(1,0)	Yes	0	-
CREATED_AT	DATE	Yes	-	-
UPDATED_AT	DATE	Yes	-	-
STATE	VARCHAR2(255)	Yes	'active'	-
KEY_TIMESTAMP	DATE	Yes	-	-

Figure 52: Review the User table from within Oracle

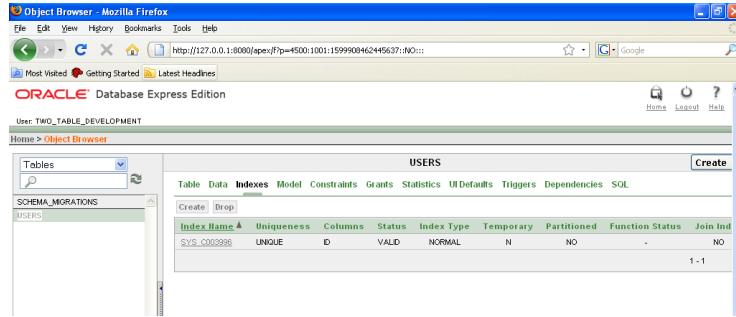


Figure 53: Review the Indexes view for Users

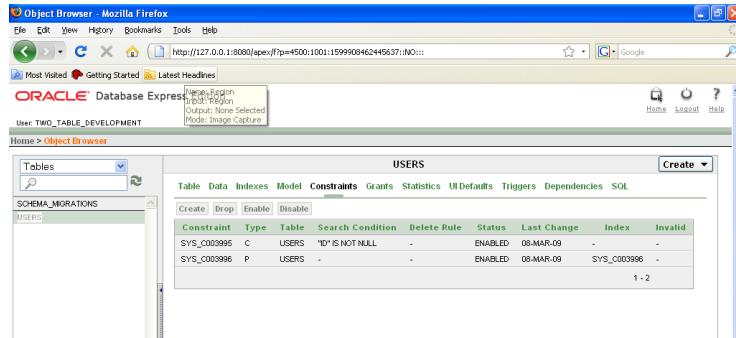


Figure 54: Review the Constraints view for User

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.rubyenterpriseedition.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 de-

development framework. Hobo is written in Ruby. One of Hobo's secret weapons is the powerful and succinct DRYML (**D**on't **R**epeat **Y**ourself **M**arkup **L**anguage).

- 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) and Thin. For production implementation there are 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 Tomcat, 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 for the tutorials!

Part II

TUTORIALS

Chapter 3

INTRODUCTORY TUTORIALS

CHAPTER 3
INTRODUCTORY TUTORIALS

- Introductory Concepts and Comments
- Tutorial 1 - Directories and Generators
- Tutorial 2 - Changing Field Names
- Tutorial 3 - Field Validation
- Tutorial 4 - Introduction to Permissions
- Tutorial 5 - Hobo
- Tutorial 6 - Editing the Navigation Tabs
- Tutorial 7 - Model Relationships
- Tutorial 8 - Model Relationships

Introductory Concepts and Comments

A magic trick is explained before it is performed, you risk spoiling the enjoyment. After you work through the tutorials there will be plenty of time after you work through a few of the tutorials to learn what is going on “behind the curtain.”

So, in the spirit of this adventure we will explain just enough, allowing you to dive in head first...

Tutorial 1 – Directories and Generators

You will create a single-table application that demonstrates how Hobo constructs a nice user interface that includes a built-in login system and basic search capability. Hobo 1.3 generators are compatible with the new Rails 3 generator API and operate quite differently from Hobo 1.0 and Rails 2.x. We will explain in more detail below.

Tutorial Application: my-first-app

Topics

- Creating a Hobo application
- Learning the Hobo Directory structure
- Generating Hobo models and controllers
- Generating Hobo models
- Generating Hobo controllers
- Creating Migrations and Databases
- Editing Models and propagating the changes

CHAPTER 3
INTRODUCTORY TUTORIALS

Tutorial Application: my-first-app

Steps

1. Description of development tools. You will use three tools to do the work in these tutorials. They include:

- A shell command prompt to run scripts
- A text editor for you to edit your application files
- A browser to run and test your application

Ordinarily you will have two shell windows or tabs open: one from which to run Hobo scripts or operating system commands and one from which to run a web server (Mon-grel in these tutorials). These tutorials are not done with an integrated development environment (IDE).

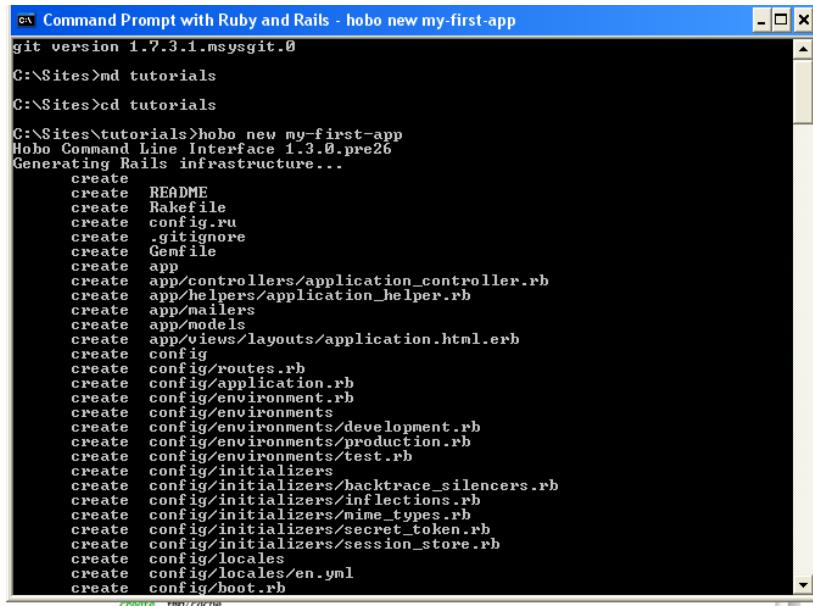
2. Create a Hobo application directory. Before you create your first Hobo application, create a directory called tutorials. This will be the directory where you keep all of your Hobo tutorials. Navigate to the tutorials directory using your shell application. You should now see the following prompt:

```
\tutorials>
```

3. Create a Hobo application. All you have to do to create a Hobo application is to issue the Hobo command:

```
\tutorials> hobo new my-first-app
```

CHAPTER 3
INTRODUCTORY TUTORIALS



The screenshot shows a Windows Command Prompt window titled "Command Prompt with Ruby and Rails - hobo new my-first-app". The window displays the output of the "hobo new my-first-app" command. The output shows the creation of various files and directories, including README, Rakefile, config.ru, gitignore, Gemfile, app, app/controllers/application_controller.rb, app/helpers/application_helper.rb, app/mailers, app/models, app/views/layouts/application.html.erb, config, config/routes.rb, config/application.rb, config/environment.rb, config/environments, config/environments/development.rb, config/environments/production.rb, config/environments/test.rb, config/initializers, config/initializers/backtrace_silencers.rb, config/initializers/inflections.rb, config/initializers/mime_types.rb, config/initializers/secret_token.rb, config/initializers/session_store.rb, config/locales, config/locales/en.yml, and config/boot.rb.

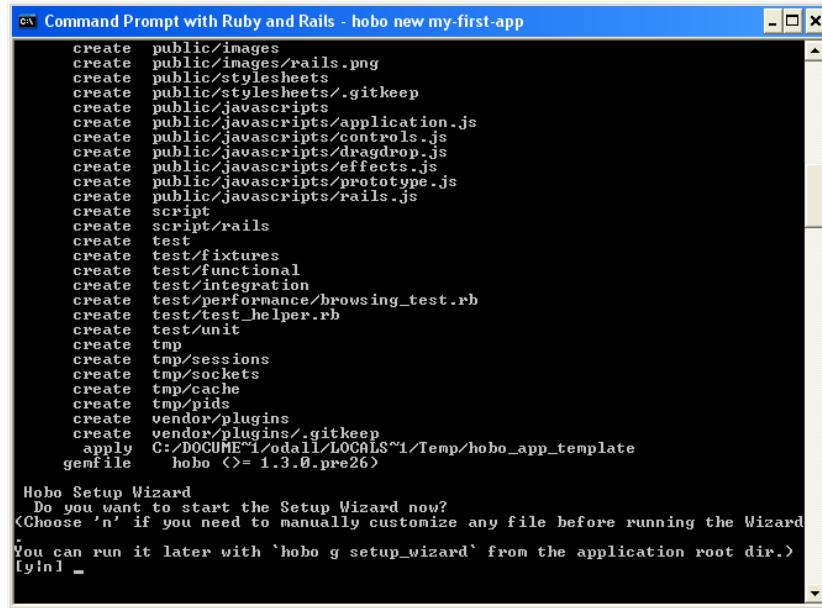
```
git version 1.7.3.1.msysgit.0
C:\Sites>md tutorials
C:\Sites>cd tutorials
C:\Sites\tutorials>hobo new my-first-app
Hobo Command Line Interface 1.3.0.pre26
Generating Rails infrastructure...
  create  README
  create  Rakefile
  create  config.ru
  create  gitignore
  create  Gemfile
  create  app
  create  app/controllers/application_controller.rb
  create  app/helpers/application_helper.rb
  create  app/mailers
  create  app/models
  create  app/views/layouts/application.html.erb
  create  config
  create  config/routes.rb
  create  config/application.rb
  create  config/environment.rb
  create  config/environments
  create  config/environments/development.rb
  create  config/environments/production.rb
  create  config/environments/test.rb
  create  config/initializers
  create  config/initializers/backtrace_silencers.rb
  create  config/initializers/inflections.rb
  create  config/initializers/mime_types.rb
  create  config/initializers/secret_token.rb
  create  config/initializers/session_store.rb
  create  config/locales
  create  config/locales/en.yml
  create  config/boot.rb
```

Figure 55: Command Prompt with Ruby and Rails

Note: The screen captures were taken with Hobo 1.3 pre-release # 26 as of 2/15/2011.

You will see a log of what Hobo is creating go by within the shell window that you will better understand as you learn Hobo's directory structure. The first prompt from the Hobo Setup Wizard will appear as follows:

CHAPTER 3
INTRODUCTORY TUTORIALS



```
create  public/images
create  public/images/rails.png
create  public/stylesheets
create  public/stylesheets/.gitkeep
create  public/javascripts
create  public/javascripts/application.js
create  public/javascripts/controls.js
create  public/javascripts/dragdrop.js
create  public/javascripts/effects.js
create  public/javascripts/prototype.js
create  public/javascripts/rails.js
create  script
create  test
create  test/fixtures
create  test/functional
create  test/integration
create  test/performance/browsing_test.rb
create  test/test_helper.rb
create  test/unit
create  tmp
create  tmp/sessions
create  tmp/sockets
create  tmp/cache
create  tmp/pids
create  vendor/plugins
apply  C:/DOCUMENTS/odall/LOCALS^1/Temp/hobo_app_template
gemfile  hobo (>= 1.3.0.pre2b)

Hobo Setup Wizard
Do you want to start the Setup Wizard now?
(Choose 'n' if you need to manually customize any file before running the Wizard)
You can run it later with 'hobo g setup_wizard' from the application root dir.
[y\!n] =
```

Figure 56: Command Prompt with Ruby on Rails

The Hobo Setup Wizard will present you with the following eight options, for this tutorial choose the following “default” options:

```
Question Defaults
Do you want to customize the test_framework? [y|n] n
a. Choose a name for the user resource
    [<enter>=>user|<custom_name>] <enter>
b. Do you want to send an activation
    email to activate the user? [y|n] n
c. Do you want to add the features
    for an invite only website? [y|n] n
d. Will your application use only hobo/dryml
    web page templates? (Choose 'n' only if you
    also plan to use plain rails/erb web page
    templates) [y|n] y
e. Choose a name for the front controller
    [<enter>=>front|<custom_name>]: <enter>
f. Initial Migration: [s]kip, [g]enerate
    migration file only, generate and
    [m]igrate [s|g|m]: m
g. Type the locales (space separated) you
    want to add to your application or
    <enter> for 'en': <enter>
h. Do you want to initialize a git repository
    now? [y|n] n
```

You will see the following message upon completion:

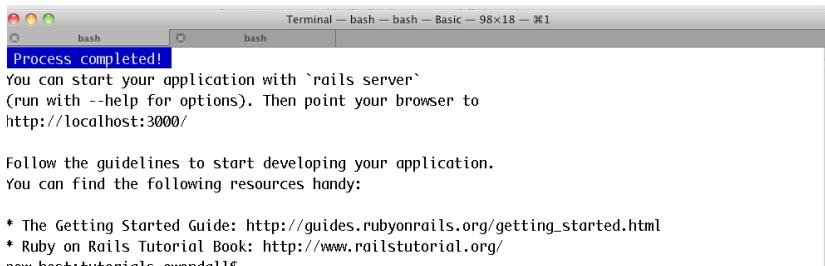


Figure 57: Completion Message

4. Start the web server. Create a second shell window (or tab). You are now going to start a local web server on your computer. This will enable you to run the Hobo application and see what a deployed application looks like in your browser.

Navigate to your application directory and fire up the local web server by issuing the following command at your command prompt.

```
\my-first-app> rails server
```

CHAPTER 3

INTRODUCTORY TUTORIALS

While your server is executing, it does **not** return you to your command prompt. As you run your application, it logs what it is doing to this shell. You can terminate the web server by typing “control-c” and restart it the same way you started it above. Do not terminate the server.

5. Open your application in a web browser. Type the following URL into your browser:

```
http://localhost:3000/
```

The following “Register Administrator” page will appear:

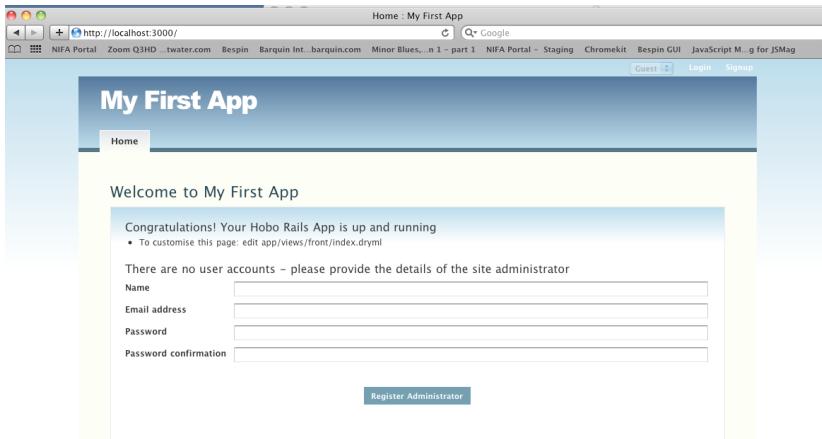


Figure 58: Register Administrator Screen

Hobo, by default, assumes the first person that launches the application will be an administrator. Go ahead and enter the information required and click on the “Register Administrator” button:

Note: The first person to register is assigned the administrator privileges by Hobo. Notice that in the upper right-hand corner of your web page there is a drop-down list of created users that allows you to sign in automatically to any of the user accounts without going through the login page if you are in development mode. This is turned off in production mode.

CHAPTER 3

INTRODUCTORY TUTORIALS

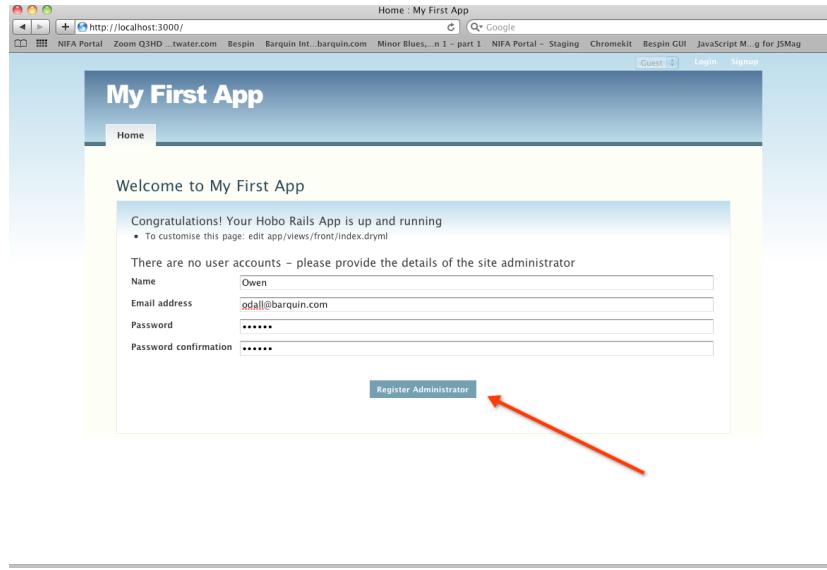


Figure 59: Register Administrator Screen

This is how your app looks after registering:

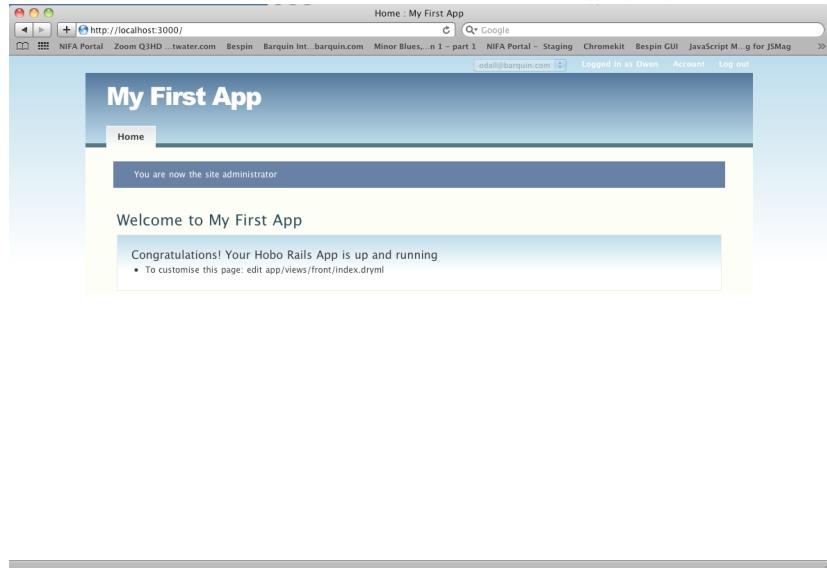


Figure 60: Completed Registration

Now let's take a look at what Hobo has generated so far. Use your text editor to locate the `user.rb` file under the `app/models` directory:

CHAPTER 3

INTRODUCTORY TUTORIALS

```

my-first-app
  app
    controllers
    helpers
    models
      user.rb
  config
  db
  doc
  Gemfile
  lib
  log
  public
  README
  test
  tmp
  vendor

x user.rb
class User < ActiveRecord::Base
  hobo_user_model # Don't put anything above this.

  fields do
    name :string, :required, :unique
    email_address :email_address, :login => true
    administrator ?boolean, :default => false
    timestamps
  end

  # This gives admin rights and an :active state to the first sign-up.
  # Just remove it if you don't want that.
  before_create do |user|
    if Rails.env.test? && user.class.count == 0
      user.administrator = true
      user.state = "active"
    end
  end

  # --- Signup lifecycle ---

  lifecycle do
    state :active, :default => true
    create :sign_up, :available_to => "Guest",
      :params => [ :name, :email_address, :password, :password_confirmation ],
      :become => :active
    transition :request_password_reset, { :active => :active }, :new_key => true do
      UserMailer.forgot_password(self, lifecycle.key).deliver
    end
    transition :reset_password, { :active => :active }, :available_to => :key_holder,
      :params => [ :password, :password_confirmation ]
  end

  # --- Permissions ---

  def create_permitted?
    Only the initial admin user can be created
  end
end

```

Figure 61: The default User model created by Hobo

Hobo took care of building the User model and generating the database table needed because we selected the “m” (generate and migrate) option for step 6 of the Hobo Setup Wizard:

Initial Migration: [s]kip, [g]enerate migration file only, generate and [m]igrate [s|g|m]:

6. Examine what Hobo created during the first “migration”. In the following figure, you can see that the db directory is populated.

- The file, `development.sqlite3`, is the database file.
- The `<timestampl>_initial_migration.rb` file defines the database table that will be created when the migration is executed.
- The `schema.rb` file shows the current database schema after all migration executions to date. In this case we have only created the user table.

CHAPTER 3 INTRODUCTORY TUTORIALS

```

x 20101209121350_initial_migration.rb
class InitialMigration < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.string :crypted_password, :limit => 40
      t.string :salt, :limit => 40
      t.string :remember_token
      t.datetime :remember_token_expires_at
      t.string :name
      t.string :email_address
      t.boolean :administrator, :default => false
      t.datetime :created_at
      t.datetime :updated_at
      t.string :state, :default => "active"
      t.datetime :key_timestamp
    end
    add_index :users, [:state]
  end

  def self.down
    drop_table :users
  end
end

```

Figure 62: Contents of the first Hobo migration file

Take a look at the schema and you will see that it corresponds to the migration file:

```

# This file is auto-generated from the current state of the database. Instead
# of editing this file, please use the migrations feature of Active Record
# to incrementally modify your database, and then regenerate this schema definition.
#
# Note that this schema.rb definition is the authoritative source for your
# database schema. If you need to create the application database on another
# system, you should be using db:schema:load, not running all the migrations
# from scratch. The latter is a flawed and unsustainable approach (the more migrations
# you'll mass, the slower it'll run and the greater likelihood for issues).
#
# It's strongly recommended to check this file into your version control system.

ActiveRecord::Schema.define(:version => 20101209121350) do

  create_table "users", :force => true do |t|
    t.string "crypted_password", :limit => 40
    t.string "salt", :limit => 40
    t.string "remember_token"
    t.datetime "remember_token_expires_at"
    t.string "name"
    t.string "email_address"
    t.boolean "administrator", :default => false
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string "state", :default => "active"
    t.datetime "key_timestamp"
  end

  add_index "users", [:state], :name => "index_users_on_state"

end

```

Figure 63: Contents of the "schema.rb" file after the first migration

Note: You can see that the User model does not display all the fields that are implemented in the database. Hobo does not expose all of the User fields but reserves them for its own use. All of the fields in other models will be reflected in the schema file. <def tag= new-page> </end>

CHAPTER 3

INTRODUCTORY TUTORIALS

Click on the “Logout” link and then click on the “Signup” link to create another account. In the example below we are creating another account for “John Smith”. We will call this and all other accounts you create user accounts, because by default they will not have administrative privileges.

Log out of the user (e.g., John Smith) account you just created and login using the account you created as administrator (e.g., Owen Dall) for now.

Note: You will use the user’s email address and password to log in, not the user’s name. Also, notice that in the upper right corner of your web page, there is a drop down box that lets you automatically login to any of your accounts without using the normal login page. This speeds up testing permission customizations in development mode. In production mode this option disappears. More on switching modes later.



Figure 64: Drop down selector for the active user

7. Check the changes in the views/taglibs directory. Notice that since you fired up your web server, there is now a change in the taglibs directory. There is a new branch called `views/taglibs/auto/rapid` and three files in that directory: `cards.dryml`, `forms.dryml` and `pages.dryml`. We are going to show you a few things to pique your curiosity but we will not cover how Hobo handles views in any detail until the intermediate tutorials. We will just make a few high level comments here in case you know something about Ruby on Rails and so you know what is coming next.

Familiarize yourself with the contents of these files. You will see many lines that look similar to:

```
<def tag= new-page>
.....
</end>
```

You will see mark-up in between the “def” and “end” tags. The contents are what we have mentioned before as “tag definitions.” Hobo uses them to construct view templates

on the fly.

These three files contain the libraries of tags that Hobo uses to construct view templates.

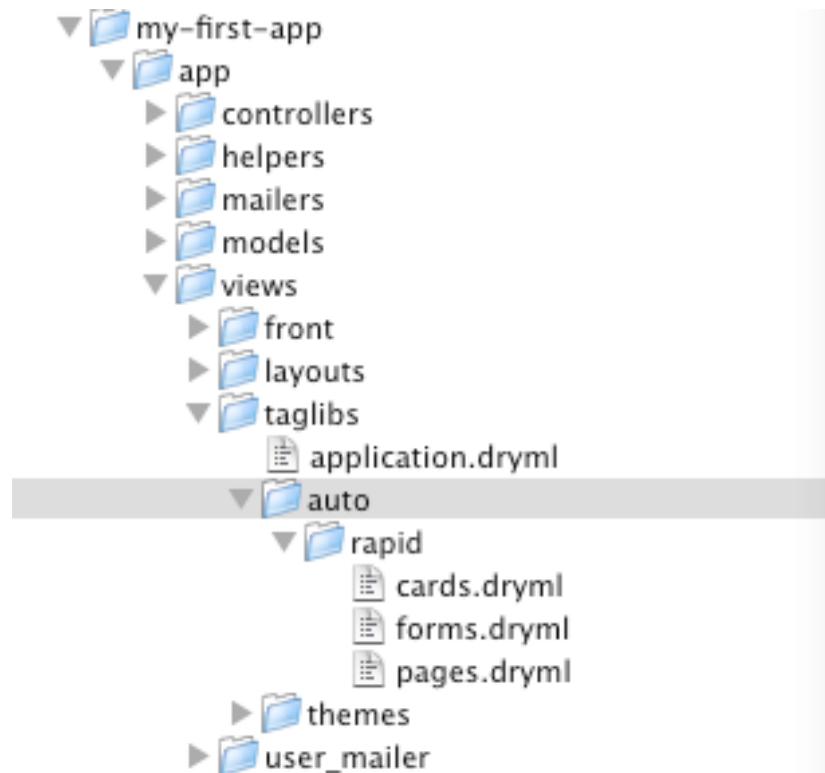


Figure 65: Location of the Rapid templates

Note: When Hobo makes a web page, it takes tags from the `pages.dryml` file. When it wants to construct a data entry form, tags in the `pages.dryml` file call tags in the `forms.dryml` file. When Hobo wants to list the records from a table, tags in the `pages.dryml` file call tags in the `cards.dryml` file. Card tags define how individual database table records are rendered.

(Actually, these files are a copy of what Hobo is doing on the fly behind the scenes but it is easier to think of it in this way.)

You will learn that you can edit and redefine the tags from the `/rapid` directory. When you want your changes to be available to the application, you can either put the new tags in the `application.dryml` file or create a `taglibs/application` folder and save your tags to a `taglibs/application/<filename>.dryml`. Any `dryml` file located in the `taglibs/application` directory will automatically load and its tags will be available application wide. When you want them to be available only in a particular view you can put them in a `dryml` file under the `app/views/<model>` directory named for the model.

So far, we only have the front (home page) and the users template directories. You will see after creating a new model (running `hobo g resource` or `hobo g model`) and running `hobo g migration`, that directories will be created and named for your new models.

8. Create a new model and controller. Let's create a simple contacts model and see what Hobo does for us.

```
\my-first-app> hobo g resource contact \
               name:string company:string
```

This generator will create both a model and controller. Execute it and then take a look at what has changed in your application directories.

You will see the new `contacts_controller.rb` file in the `/controllers` directory and the new `contact.rb` file in the `/models` directory.

Note: Unlike Hobo 1.0, a view template file is not created in the `views/contacts` folder. We will discuss later how the rapid taglibs in the `/auto/rapid` folder take care of the default views for you).

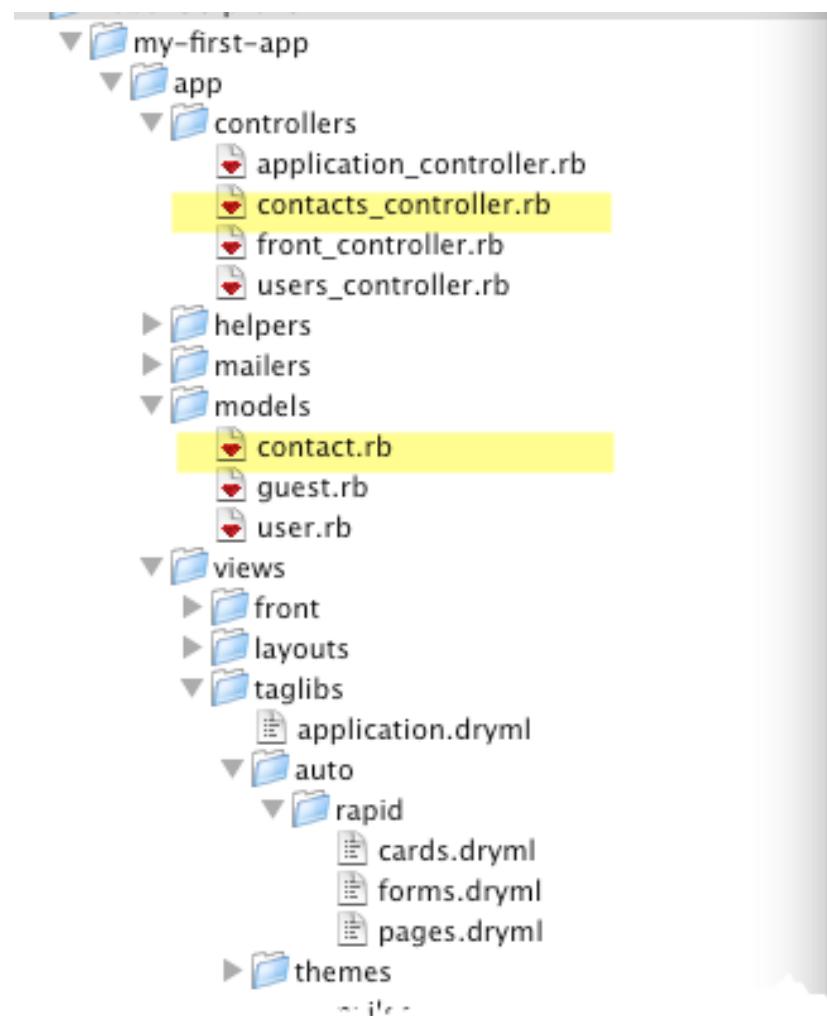


Figure 66: Folder location for Models and Views

9. Run a Hobo migration. Before you run the migration, take a look at the `contact.rb` model file. We just want to review the relevant part for now. The permissions part will be explained in a later tutorial.

Here is the code that declares the fields that you want in your database table that will be called *contacts*. When you ran `hobo g resource`, it generated this code.

CHAPTER 3
INTRODUCTORY TUTORIALS

```
class Contact < ActiveRecord::Base

    hobo_model # Don't put anything above this
    fields do
        name :string
        company :string timestamps
    end
```

When you run `hobo g migration`, Hobo will take this declaration and create a migration file. It will then in turn use the migration file to create the database table. These two steps will be executed within a single Hobo migration. You could do them separately but we will not do that here.

Now run `hobo g migration` and observe what happens.

```
\my-first-app> hobo g migration
```

Remember to select the ‘m’ option to both create and execute the migration file. Then hit return to accept the proposed name of the migration file.

You will notice some changes now in the `my-first-app/db` directory of your app.

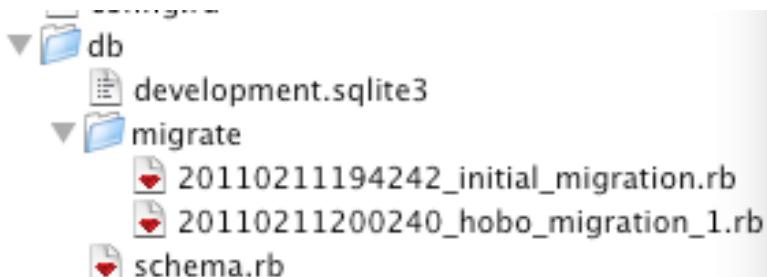


Figure 67: Migration file changes

There is a new migration file and changes in your schema file as well. The new migration file contains the following code:

```
def self.up
    create_table :contacts do |t|
        t.string :name
        t.string :company
        t.datetime :created_at
        t.datetime :updated_at
    end
```

The schema file (`schema.rb`), reflecting this code, shows the current state of the database in the `db/schema` file:

CHAPTER 3
INTRODUCTORY TUTORIALS

```
create_table "contacts", :force => true do |t|
  t.string "name"
  t.string "company"
  t.datetime "created_at"
  t.datetime "updated_at"
end
```

Now check out the application in your browser with the following URL after running the command rails server

<http://localhost:3000/>



Figure 68: Contacts tab on “My First App”

Now you have a new tab called “Contacts.”

10. Create some contacts. Now you should be able to create a new contact by clicking the ‘New Contact’ link in the Contacts tab. Go ahead and create a couple of new contacts to convince yourself that the database entry actually works. While you are at it also try editing a contact.

So far, Hobo is doing a pretty decent job. You have a usable UI, I/O capability for your contact model and a login system and you have written no code.

11. Try out the search facility. Type the name of one your contacts to exercise the search facility. The default search searches “name” fields. You need at lease three characters for a partial word search.

12. Add columns to the database. Now we are going to add a couple more fields to the model and have hobo add columns to the database. In this and the following steps, you will get a sense for the power of the `hobo g migration` generator. Since we have already generated our model using `hobo g resource`, we do not have to do that again. Go into the model and add some new fields. Your code should now look like this:

CHAPTER 3
INTRODUCTORY TUTORIALS

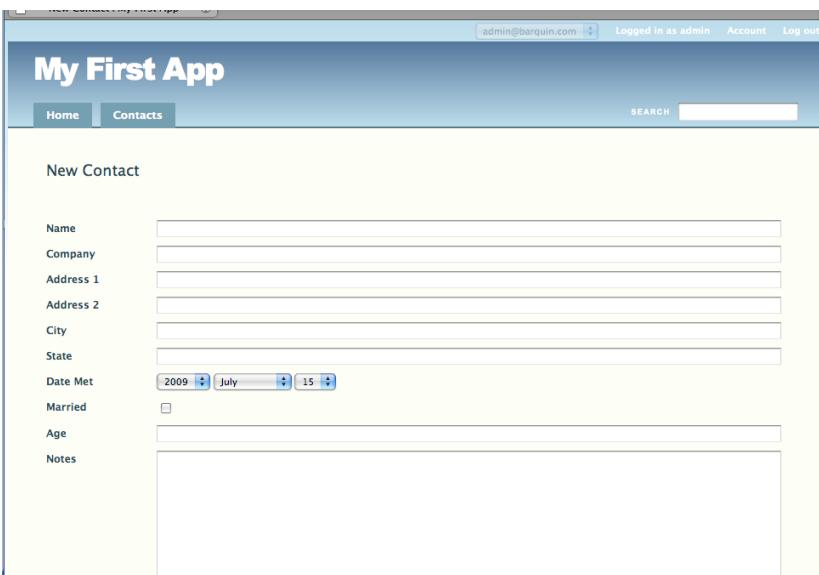
```
class Contact < ActiveRecord::Base

  hobo_model # Don't put anything above this
  fields do
    name :string
    company :string
    address_1 :string
    address_2 :string
    city :string
    state :string
    date_met :date
    married :boolean
    age :integer
    notes :text
    timestamps
  end
```

Make sure you save your changes and run `hobo g migration`. Select the ‘m’ option and accept the default filename for the migration.

```
\my-first-app> hobo g migration
```

Now refresh your browser. Go to the contacts tab and click ‘New Contact’



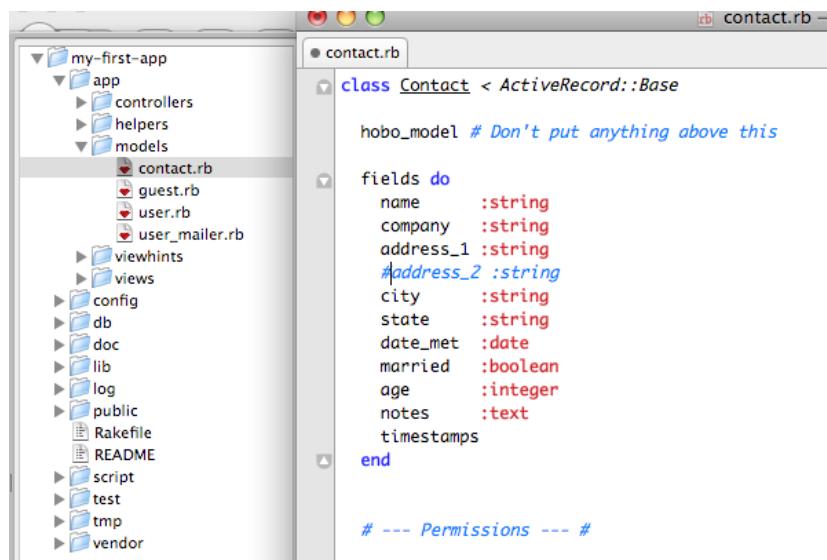
The screenshot shows a web browser window titled "My First App". The URL bar shows "http://localhost:3000/contact/new". The top navigation bar includes links for "Home", "Contacts", "SEARCH", and user information "Logged in as admin". The main content area is titled "New Contact". It contains form fields for "Name", "Company", "Address 1", "Address 2", "City", "State", "Date Met" (set to July 15, 2009), "Married" (unchecked checkbox), "Age", and "Notes".

Figure 69: New Contact page for “My First App”

Note what Hobo has done for you. It determines which entry controls you need based on the type of field you defined in your model. It has one-line fields for strings, a set of three combo boxes for dates, a one-line field for integers, a check box for boolean field, and a multi-line box for text fields. Later you will see that Hobo can provide the controls you need for multi-model situations.

Hobo has also provided reasonable names and styles from the field names. It removed the underscore characters and appropriately capitalized words to give the presentation a nice look and feel.

13. Remove columns from the database. Now suppose you decide that you need only one address field and you decide to remove the second one. Go back to the Contact model and delete it (we just commented it out with the # sign so you can see things clearer.)

A screenshot of a Mac OS X desktop environment. On the left is a file browser window titled "my-first-app" showing the directory structure. On the right is a code editor window titled "contact.rb" containing Ruby code for a ActiveRecord model.

```
class Contact < ActiveRecord::Base
  hobo_model # Don't put anything above this

  fields do
    name      :string
    company   :string
    address_1 :string
    #address_2 :string
    city      :string
    state     :string
    date_met  :date
    married   :boolean
    age       :integer
    notes     :text
    timestamps
  end

  # --- Permissions --- #
end
```

Figure 70: Remove field from contact model

CHAPTER 3
INTRODUCTORY TUTORIALS

```
class Contact < ActiveRecord::Base

    hobo_model # Don't put anything above this
    fields do
        name :string
        company :string
        address_1 :string
        #address_2:string
        city :string
        state :string
        date_met :date
        married :boolean
        age :integer
        notes :text
        timestamps
    end
```

Run `hobo g migration` again.

```
\my-first-app> hobo g migration
```

Hobo notices that you have deleted a model field and responds in this way.

```
CONFIRM DROP! column contacts.address_2
Enter 'drop address_2' to confirm:
```

You respond by typing what it asks (without the quotes).

```
CONFIRM DROP! column contacts.address_2
Enter 'drop address_2' to confirm: drop address_2
```

Complete the migration as you have learned above. Then go check the db directory. You will see another migration, `*_hobo_migration_4.rb` with the following code. (The asterisk (*) here stands for the time/date stamp that precedes the rest of the migration file name.)

```
class HoboMigration4 < ActiveRecord::Migration

    def self.up
        remove_column :contacts, :address_2
    end
    def self.down
        add_column :contacts, :address_2, :string
    end
end
```

Check out the schema.rb file now.

```
ActiveRecord::Schema.define(:version => 20090220154125) do
  create_table "contacts", :force => true do |t|
    t.string "name"
    t.string "company"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string "address_1"
    t.string "city"
    t.string "state"
    t.date "date_met"
    t.boolean "married"
    t.integer "age"
    t.text "notes"
  end
```

You can see that address_2 is gone.

14. Adding and removing database tables. You can also use `hobo g migration` to remove (drop) a table. Delete the model file and the associated helper and controller files. Then run `hobo g migration`. You will be prompted to confirm that you wish to drop the table. (If you neglect to delete the helper and controller file for this model you will get an error.)

15. Going back to earlier migrations. Hobo does not provide this facility within `hobo g migration`. You will need to use the `rake db:migrate VERSION = XXX` procedure. You can roll back your tables but the rest of your changes will not be synchronized so you will have to perform manual edits.

Tutorial 2 – Changing Field Names

We are going to continue from the previous tutorial and show you how to do rename fields in a couple of different ways and improve your UI with hints about what to enter in a particular field.

Topics

- Two ways of changing field names displayed
- Displaying data entry hints
- Changing field sizes: Hobo does not provide this facility now.

CHAPTER 3
INTRODUCTORY TUTORIALS

Tutorial Application: my-first-app

Steps

- 1. Rename a database column.** In Tutorial 1, we showed you how to make changes to your database by editing the model file. You can rename a field and database column in the same way. We will try this with the *married* field. Go to your `contacts.rb` file and rename `married` to `married_now` and run the `hobo g migration`.

```
class Contact < ActiveRecord::Base

  hobo_model # Don't put anything above this
  fields do
    name :string
    company :string
    address_1 :string
    #address_2 :string
    city :string
    state :string
    date_met :date
    #married :boolean
    married_now :Boolean
    age :integer
    notes :text
    timestamps
  end
```

```
\my-first-app> hobo g migration
```

Hobo should now respond:

```
DROP, RENAME or KEEP?: column contacts.married
Rename choices: married_now
Enter either 'drop married' or one of the rename
choices or press enter to keep:
```

Hobo is trying to confirm that what you really want to do is rename the column and not drop it. Enter `married_now` to rename. Check your `schema.db` file and you will see that the column has been renamed. The `KEEP` option is a safety option in case you mistakenly renamed the column,

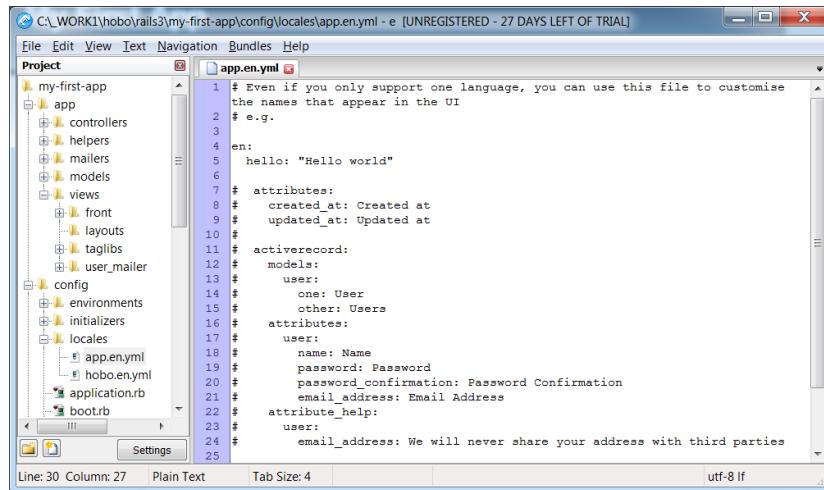
Note: Do not use question marks (?) in field names. You will get an error.

Refresh your browser and you will now see the field labeled ‘Married Now.’

2. Changing field names. There is no need to change the name of a field or column if all you wish to do is to change the name of a label in the user interface. Hobo provides this as part of its new Internationalization (i18n) module. This i18n module is very versatile and can be used for providing field/model renaming capabilities up to full multi-lingual support for your application. The i18n module is comprised of .yml files that reside in the config/locales directory; if you look in the directory you will see these files:

- app.en.yml (or app.<locale>.yml)
- hobo.en.yml (or hobo.<locale>.yml)

For this tutorial we will assume that english (en) was specified as the default locale. The hobo.en.yml file contains all the strings that the hobo framework uses, while application specific strings are stored in the app.en.yml file. We will modify the app.en.yml file to rename a couple of fields in the Contact model. When you open the config/locales/app.en.yml file, you will see:



```

C:\WORK1\hobo\rails3\my-first-app\config\locales\app.en.yml - e [UNREGISTERED - 27 DAYS LEFT OF TRIAL]
File Edit View Text Navigation Bundles Help
Project app.en.yml
1 # Even if you only support one language, you can use this file to customise
2 # the names that appear in the UI
3 #
4 en:
5   hello: "Hello world"
6 #
7   attributes:
8     created_at: Created at
9     updated_at: Updated at
10 #
11   activerecord:
12     models:
13       user:
14         one: User
15         other: Users
16     attributes:
17       user:
18         name: Name
19         password: Password
20         password_confirmation: Password Confirmation
21         email_address: Email Address
22     attribute_help:
23       user:
24         email_address: We will never share your address with third parties
25

```

Figure 71: Default config/locales/app.en.yml File

The commented yaml code is very useful in understanding how to setup your locale file. Our goal is to rename the name and address_1 fields of the Contact model. To do this, add the illustrated code to the file:

CHAPTER 3

INTRODUCTORY TUTORIALS

```
# Even if you only support one language, you can use this file to customise the
# names that appear in the UI
#
# e.g.
#
en:
  hello: "Hello world"
#
# attributes:
#   created_at: Created at
#   updated_at: Updated at
#
# activerecord:
#   models:
#     user:
#       one: User
#       other: Users
#       attributes:
#         user:
#           name: Name
#           password: Password
#           password_confirmation: Password Confirmation
#           email_address: Email Address
#           attribute_help:
#             user:
#               email_address: We will never share your address with third parties
#
# activerecord:
#   attributes:
#     contact:
#       field name: Friend
#       rename
#       address_1: Address
model
rename
```

Figure 72: app.en.yml File with Fields Renamed

As shown above, to declare new names for model fields in the app.en.yml file, the following pattern must be followed:

```
activerecord:
  attributes:
    <model1>
      <field1>: Label/Rename
      <field2>: Label/Rename
      ...
    <model2>
      <field1>: Label/Rename
      <field2>: Label/Rename
      ...
```

Note: Indentation and spacing is very important when working with yml files; the activerecord: line must start with 2 spaces.

Refresh your browser and you should see the fields relabeled with your choices from above. Notice that a migration is not necessary for any changes made using the i18n module.

CHAPTER 3
INTRODUCTORY TUTORIALS

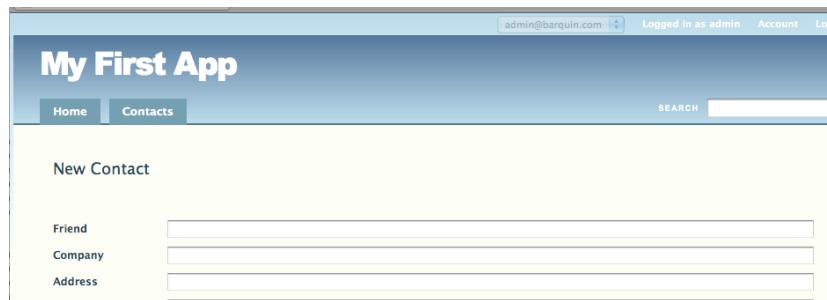


Figure 73: View of fields relabeled using the Hobo i18n module

3. **Using the i18n module to suggest field uses.** The application locale (`app.en.yml`) file also provides the facility to provide a suggestion below the field on what to enter into it. Edit your `app.en.yml` file to look like this.

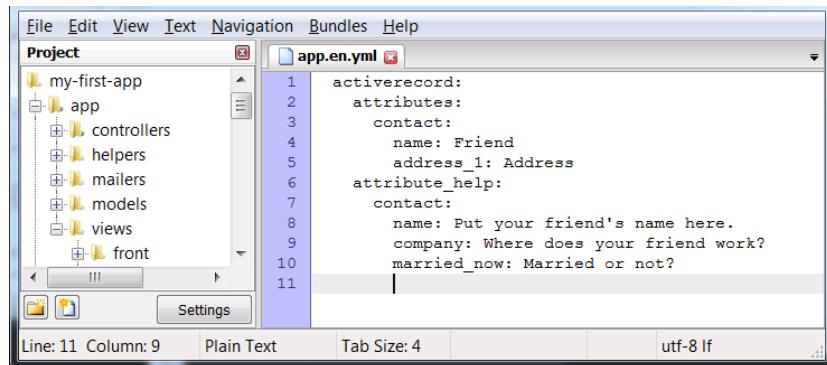
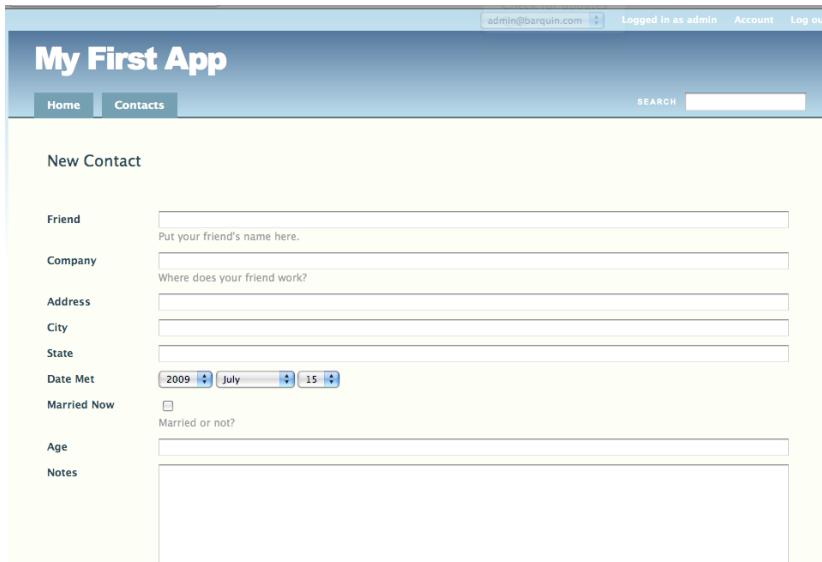


Figure 74: Adding help text using the Hobo i18n “attribute_help” method

Now refresh your browser and you will see hints on the field use in a small font below:

CHAPTER 3
INTRODUCTORY TUTORIALS



The screenshot shows a web application interface titled "My First App". At the top, there is a navigation bar with links for "admin@barquin.com", "Logged in as admin", "Account", and "Log out". Below the navigation bar, there is a search bar labeled "SEARCH". The main content area is titled "New Contact". It contains several input fields with labels and placeholder text:

- Friend:** A text input field with the placeholder "Put your friend's name here."
- Company:** A text input field with the placeholder "Where does your friend work?"
- Address:** A text input field.
- City:** A text input field.
- State:** A text input field.
- Date Met:** A date picker set to "2009 July 15".
- Married Now:** A checkbox labeled "Married or not?".
- Age:** A text input field.
- Notes:** A large text area for notes.

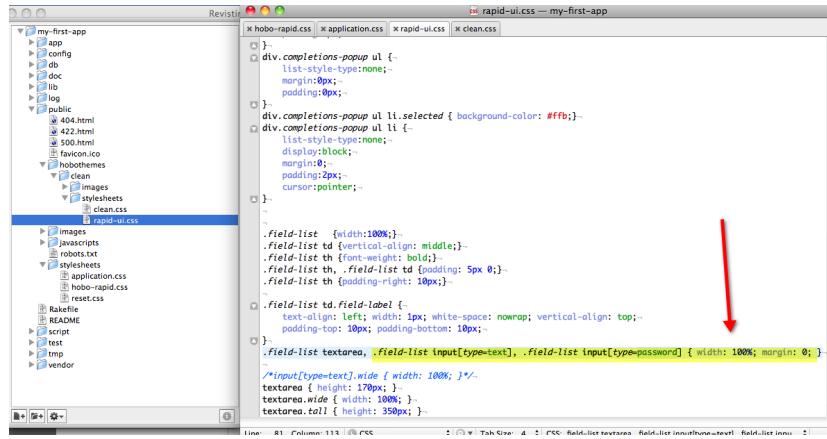
Figure 75: Contact entry page with ViewHints enabled

Note: In the Intermediate tutorials you will also learn how to use yet another way to manipulate the labels on a web page by using Hobo's view markup language called DRYML (Don't Repeat Yourself Markup Language). DRYML is used by the Rapid UI generator that creates much of Hobo's magic.

4. Changing field sizes. As of the latest version of Hobo, the way to change the field length on an input form is to add an entry to `application.css` that will override any other reference to the element you wish to modify.

Look for the relevant class definition used by Hobo's "Rapid" UI generator: `rapid-ui.css`, located at:

```
/public/hobothemes/clean/stylesheets/rapid-ui.css
```



```

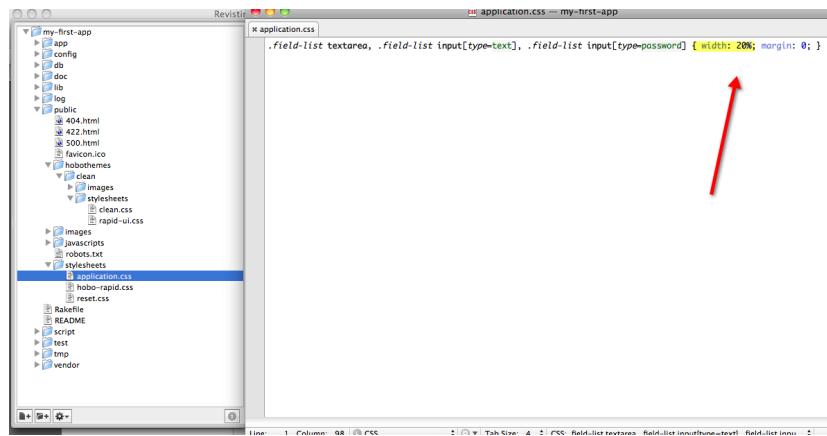
hobo-rapid.css application.css rapid-ui.css clear.css
}
div.completions-popup ul {
    list-style-type:none;
    margin:0px;
    padding:0px;
}
div.completions-popup ul li.selected {
    background-color: #ffb;
}
div.completions-popup ul li {
    display:block;
    margin:0px;
    padding:2px;
    cursor:pointer;
}
.field-list {width:100%;}
.field-list td {vertical-align: middle;}
.field-list th {font-weight: bold;}
.field-list th, .field-list td {padding: 5px 0; }
.field-list th {padding-right: 10px; }

.field-list td.field-label {
    text-align: left; width: 1px; white-space: nowrap; vertical-align: top;
    padding-top: 10px; padding-bottom: 10px;
}
.field-list textarea, .field-list input[type=text], .field-list input[type=password] { width: 100%; margin: 0; }

/*input[type=text].wide { width: 100%; }*/
textarea { height: 170px; }
textarea.wide { width: 100%; }
textarea.tall { height: 350px; }

```

Figure 76: CSS definitions for the input text fields



```

application.css
.width: 20%; margin: 0;

```

Figure 77: Modified entry in "application.css" to shorten text prompts

Tutorial 3 – Field Validation

You will be introduced to a couple of ways of validating data entry fields. This is a capability that is derived from what are called Rails helper methods. There are a couple of enhancements Hobo has made for the most common need.

Topics

- Field validation using Hobo's enhancements

CHAPTER 3

INTRODUCTORY TUTORIALS

- Field validation using Rails helper methods
- Validation on save, create and update processes

Tutorial Application: my-first-app

1. **Make sure data is entered.** Open up the model `contact.rb` file. Add the following code to the “name” field definition

```
name :string, :required
```

This is the simplified version that Hobo provides. To do this in the “normal” rails way, you would need to add this line after the “fields/do” block:

```
validates_presence_of :name
```

(The difference in the two is a matter of taste, but the former seems “DRYer” to us.)

By default Hobo will provide a message if a user fails to enter data. Try it out by trying to create a contact record with no data in it. Click the “Contacts” tab and then “*New Contact*”.

Without entering anything in the form, click “*Create Contact*”.

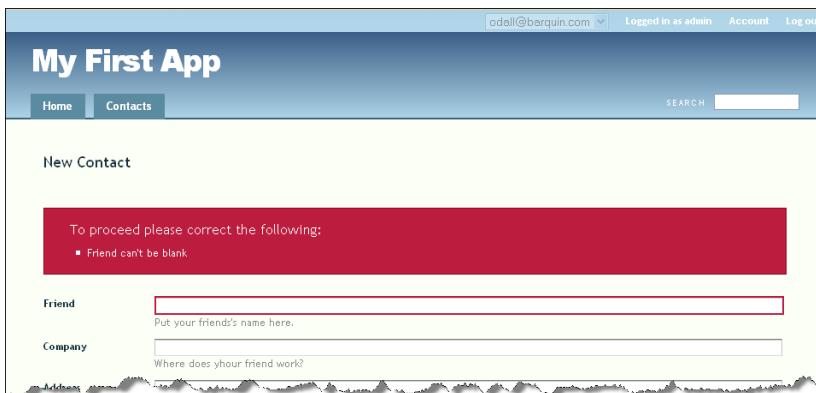


Figure 78: Page view of validation presence of name

2. **Validate multiple fields.** In order to validate multiple fields, add the “:required” label to another field:

```
address_1 :string, :required
```

Click the “Contacts” tab and then “*New Contact*”. Without entering anything in the form, click “*Create Contact*”.

CHAPTER 3
INTRODUCTORY TUTORIALS

The screenshot shows a web application interface for 'My First App'. At the top, there's a header bar with a user email 'odall@barquin.com', a 'Logged in as admin' link, and 'Account' and 'Log out' buttons. Below the header is a navigation bar with 'Home' and 'Contacts' tabs, and a search bar labeled 'SEARCH'. The main content area has a title 'New Contact'. A red error box contains the message 'To proceed please correct the following:' followed by two bullet points: '■ Friend can't be blank' and '■ Address can't be blank'. Below the error box are four input fields: 'Friend' (label 'Put your friend's name here.'), 'Company' (label 'Where does your friend work?'), 'Address' (label 'Address'), and 'City' (label 'City').

Figure 79: Page view of double validation error

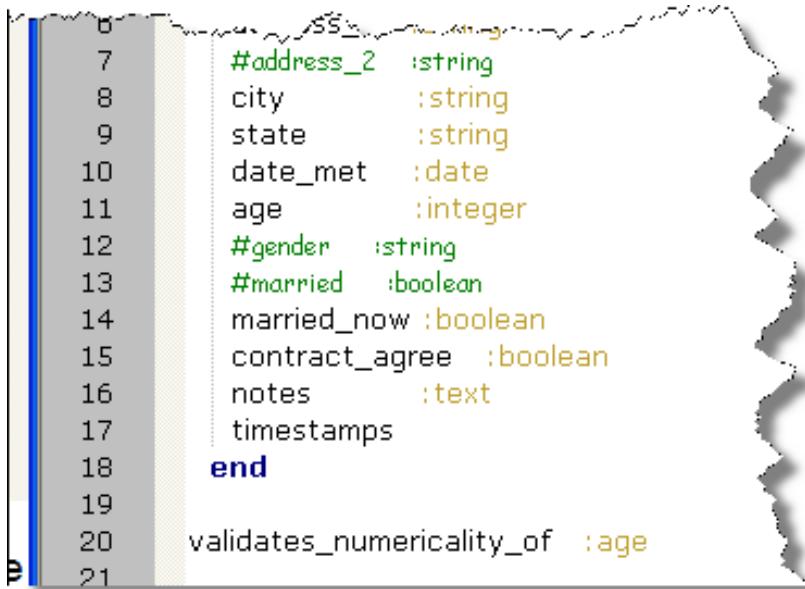
Notice the “declarative” nature of this validation. All you need to do is use the label “:required” for the name and address_1 fields and Hobo takes care of all of the logic associated with validation and delivering error messages.

Now let's try some other validations.

3. **Make sure the integer field contains a number.** Add this validation to the “age” field after the “fields do/end” block:

```
validates_numericality_of :age
```

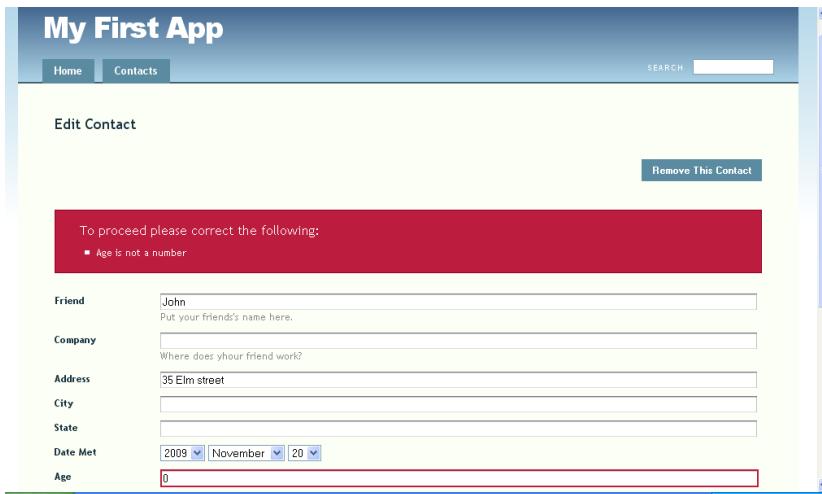
CHAPTER 3
INTRODUCTORY TUTORIALS



```
6
7     #address_2 :string
8     city      :string
9     state     :string
10    date_met  :date
11    age       :integer
12    #gender   :string
13    #married  :boolean
14    married_now :boolean
15    contract_agree :boolean
16    notes     :text
17    timestamps
18
19
20 validates_numericality_of :age
21
```

Figure 80: Adding “validate_numericality_of” validation

Now try this out by entering the text “old” in the age field. (Also put something in the name and address_1 fields so you won’t trip the validations we put into place earlier in the tutorial.)



The screenshot shows a web application titled "My First App" with a "Contacts" section. A user is editing a contact named "John". In the "Age" field, the user has entered the string "old". A red error box appears above the form with the message: "To proceed please correct the following:" and a single bullet point: "Age is not a number". The rest of the form fields are populated with valid data: "Friend" is "John", "Company" is blank, "Address" is "35 Elm street", "City" is blank, "State" is blank, "Date Met" is set to November 20, 2009.

Figure 81: Page view of triggering the "validates_numericality_of" error

Note: When you cause a validation error for integer, Hobo/Rails replaces what you entered with a zero (0). If the validation rule was not there, the text will be replaced by a zero, but the validation error will not be displayed.

4. **Prevent the entry of duplicates.** Use the following code to prevent a user from entering code that duplicates an existing record with a column value that is the same as the new record.

```
name :string, :required, :unique
```

The screenshot shows a web application titled "My First App". The top navigation bar includes links for "Home", "Contacts", and "Log out". A search bar is also present. The main content area is titled "New Contact". A red error message box contains the text: "To proceed please correct the following: Friend has already been taken". Below this, form fields are shown: "Friend" (input field containing "Jeff"), "Company" (input field), "Address" (input field containing "27 West street"), "City" (input field), "State" (input field), "Date Met" (dropdown menus for year, month, and day), and "Age" (input field containing "33").

Figure 82: Page view of uniqueness validation error

Note: This particular validation will only verify that there is no existing record with the same field value at the time of validation. In a multi-user application, there is still a chance that records could be entered nearly at the same time resulting in a duplicate entry. The most reliable way to enforce uniqueness is with a database-level constraint.

5. **Including and excluding values.** Now suppose we wish to exclude people who have an age between 0 and 17, and include people under 65 years of age. Try the following code after the “fields do/end” block:

```
validates_inclusion_of :age, :in => 18..65, :message => "Must be between 18 and 65"
```

CHAPTER 3
INTRODUCTORY TUTORIALS

The screenshot shows a web application titled "My First App" with a "Contacts" tab selected. A red error message box at the top states: "To proceed please correct the following: ▪ Age Must be between 18 and 65". Below this, there are input fields for "Friend" (Nancy), "Company" (empty), "Address" (530 Little John), "City" (empty), "State" (empty), "Date Met" (2009 July 19), "Age" (15, highlighted in red), and "Married Now" (checkbox). The "Age" field is the focus of the validation error.

Figure 83: Page view of triggering a range validation error

6. Validate length of entry. Suppose you wish to check the length of a string entry. You can specify a length range in the following way.

```
validates_length_of :name, :within => 2..20,
                     :too_long => "pick a shorter name",
                     :too_short => "pick a longer name"
```

Try to enter a one-character name. You will get the following response:

The screenshot shows the same "My First App" contact form. A red error message box at the top states: "To proceed please correct the following: ▪ Friend – enter a longer one!". Below this, the "Friend" field contains the letter "t", which is too short. The other fields are empty or have their correct values: "Company" (empty), "Address" (empty), "City" (empty), "State" (empty), "Date Met" (2009 November 22).

Figure 84: Page view of validation of text length error

CHAPTER 3
INTRODUCTORY TUTORIALS

7. Validate acceptance. If you wish to get the user to accept a contract, for example, you can use the following validation code. Assume you have a Boolean variable named `contract_agree`, which would show up in the UI as a checkbox.

```
validates_acceptance_of :contract_agree, :accept => true
```

Hobo will generate an error if the `contract_agree` check box is not checked setting the value to 1.

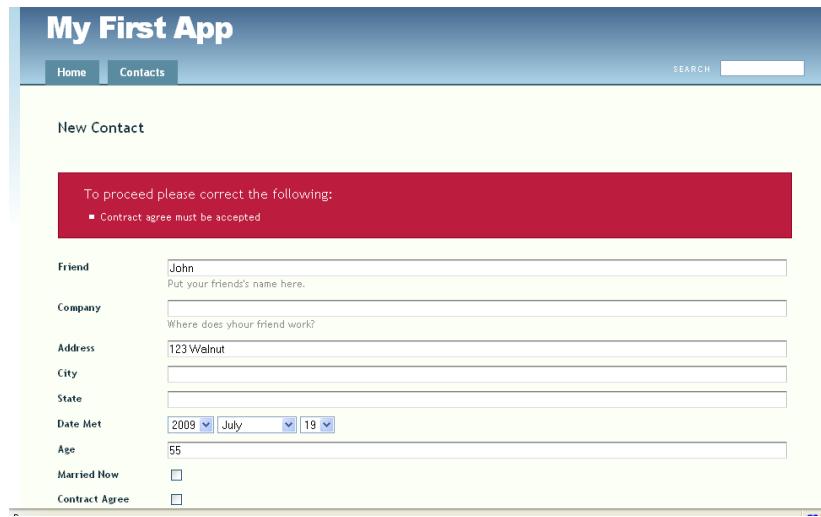


Figure 85: Page view of “`validates_acceptance_of`” error

8. Summary. Here is the list of validations we accumulated during this tutorial:

```
address_1 :string, :required  
name :string, :required, :unique
```

```
validates_numericality_of :age  
validates_acceptance_of :contract_agree,  
:accept => true  
validates_length_of :name, :within => 2..20,  
:too_long => "pick a shorter name",  
:too_short => "pick a longer name"  
validates_inclusion_of :age, :in => 18..65,  
:message => "Must be between 18 and 65"
```

There are several other very useful validation functions provided by Rails, and the ones that we have shown you above have many other options. These functions can provide very sophisticated business rule execution.

CHAPTER 3

INTRODUCTORY TUTORIALS

For example, the following is a sample of the list of options for the `validates_length_of` and `validates_size_of` (synonym) declarative expressions:

- `:minimum` - The minimum size of the attribute.
- `:maximum` - The maximum size of the attribute.
- `:is` - The exact size of the attribute.
- `:within` - A range specifying the minimum and maximum size of the attribute.
- `:in` - A synonym(or alias) for `:within`.
- `:allow_nil` - Attribute may be nil; skip validation.
- `:allow_blank` - Attribute may be blank; skip validation.
- `:too_long` - The error message if the attribute goes over the maximum (default is: "is too long (maximum is {{count}} characters)").
- `:too_short` - The error message if the attribute goes under the minimum (default is: "is too short (min is {{count}} characters)").
- `:wrong_length` - The error message if using the `:is` method and the attribute is the wrong size (default is: "is the wrong length (should be {{count}} characters)").
- `:message` - The error message to use for a `:minimum`, `:maximum`, or `:is` violation. An alias of the appropriate `too_long/too_short/wrong_length` message.
- `:on` - Specifies when this validation is active (default is `:save`, other options `:create`, `:update`).
- `:if` - Specifies a method, procedure, or string to call to determine if the validation should occur: `:if => :allow_validation`
The method, procedure, or string should return or evaluate to a true or false value.

- `:unless` - Specifies a method, procedure or string to call to determine if the validation should not occur:
`:unless => :skip_validation`
The method, procedure, or string should return or evaluate to a true or false value.

We encourage you to read about validation helpers (what Rails calls functions) in the many good Ruby on Rails references. The following is a useful on-line reference:

<http://api.rubyonrails.org/classes/ActiveRecord/Validations/ClassMethods.html>

Tutorial 4 – Permissions

In this tutorial you will learn some elementary aspects of Hobo’s permission system by changing what the admin user and users can do. Specifically, you will determine whether a user is permitted to view, create, edit, or delete records in the database.

Topics

- Experiment with altering user permissions.
- Naming conventions for database tables, models, controllers and views.

Tutorial Application: one_table

Steps

1. **Create the Hobo application.** Create the “one_table” Hobo application by issuing the following command at the command prompt. By adding the -setup command line parameter the application will bypass the setup wizard and create the application with default settings. Then change directory to the subdirectory “one_table:”

```
\tutorials> hobo new one_table --setup
\tutorials> cd one_table
\one_table>
```

Recall from Tutorial 1 that this sets up the Hobo directory tree and the user model and controller.

Note: Look at the file \one_table\app\models\user.rb and at the database schema file \one_table\db\schema.rb. There are more fields in the users table than in the user model. This is because Hobo creates several user model fields for you automatically. This will not be the case for models you create.

3. **Start the web server.** Open a new command prompt and navigate to the \tutorials\one_table directory. Fire up your web server by issuing the following command.

```
\one_table> rails server
```

4. **Initiate the web application.** Enter the local URL for the application in your browser’s URL window:

<http://localhost:3000/>

You should now see the following displayed on your browser.

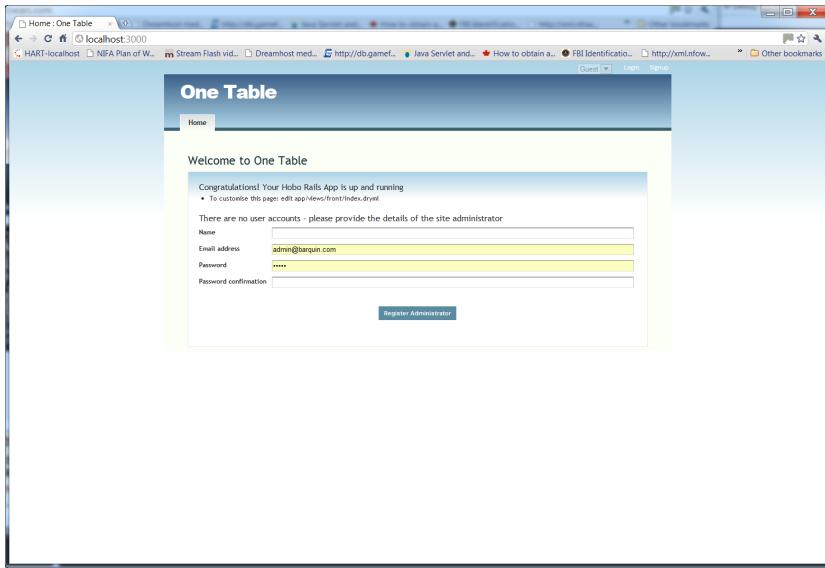


Figure 86: Welcome to One Table in the Permissions tutorial

5. Create user accounts. You will need a couple of accounts to exercise the functions of the One Table application. Let's do this now like you did in Tutorial 1.

Click “*Register Administrator*” to create the administrator account. We refer to this account as the “*admin*” account. Logout and create a second account. We will refer to this second account as the “*user*” account in the following tutorials. **By default, the “user” account does not have administrative privileges.**

Later in the tutorial, you will learn to customize the default permission features.

Log out of the user account and login to the admin account for now. Remember that you will use the admin email address and password to login, not the name.

6. Create the recipe model. Next create a model using the “*hobo resource*” generator, which will be called “*Recipe*”. It will contain three fields: title, body and country. We will complete this step by rerunning the Hobo migration from Step 3. This will take the model definitions and create a migration file and the database table “recipes”.

```
\one_table> hobo g resource recipe title:string
          body:text country:string
```

This generator created a *recipe.rb* model from which the *hobo migration* generator will create a migration file and a database table.

Note: When we talk about a model's name we are referring to its Ruby Class name that can be found at the top of the file.

It also created the `recipes_controller.rb` controller, the `recipes_helper.rb` helper file, and recipes view folder. Run the `hobo migration` generator:

```
\one_table> hobo g migration
```

IMPORTANT: Hobo is different from Rails in that the migration file and database table are both the result of the `hobo:migration` generator. In Rails, generators typically create both models AND migration files but NOT database tables.

Refresh your browser and you should see a Recipes tab added.

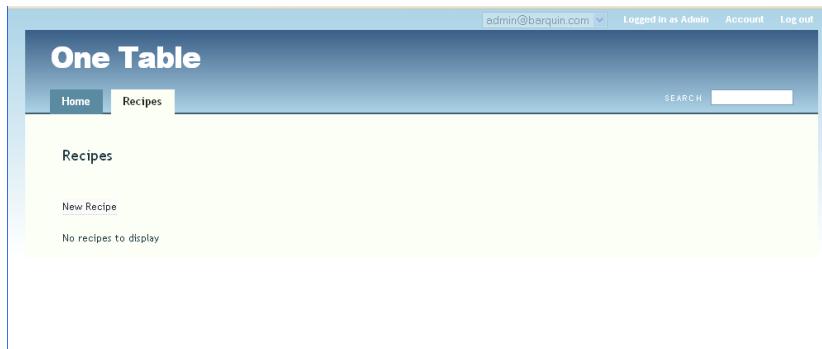


Figure 87: Recipes tab

7. Confirm your login info. Make sure you are logged in as the administrator. As long as you are logged in, you should see the “*New Recipe*” link on the left.

Create three recipes and take care to add info in all three fields. You can create them either from the *Home* or *Recipes* tab. The finished recipes should be displayed in both the *Home* tab and the *Recipes* tab automatically. You can click on any of the names of the recipes to edit them. Try it out.



Figure 88: Page view of created recipes

8. Login as a user. Sign out of the “*admin*” account and sign in as another. Note that you can still see the recipe title. Now, you can click on the recipe title and view the entire recipe record but you *cannot* create or edit a recipe. This is governed by the Hobo “Permissions” module. In the next step, you will change the user permissions and see how the user interface responds by automatically providing creation and editing capabilities in the user interface.

9. Edit permissions: Take a look at the `recipe.rb` model file.

```
# --- Permissions --- #
def create_permitted?
  acting_user.administrator?
end
def update_permitted?Permissions!
  acting_user.administrator?
end
def destroy_permitted?
  acting_user.administrator?
end
def view_permitted?(field)Permissions!
  true
end
```

There are four methods that define the basic permission system: `create_permitted?`, `update_permitted?`, `destroy_permitted?` and `view_permitted?`. In exercising the permission system, you are editing Ruby code. The permission methods are defined within Hobo. Each method evaluates a boolean-valued variable (actually a method on an object) that indicates whether the named action is allowed or not allowed.

Method	Refers to permission to:
<code>create_permitted?</code>	create a record
<code>update_permitted?</code>	edit a record
<code>destroy_permitted?</code>	delete a record
<code>view_permitted?(field)</code>	view a record or field

Table 1: Table of Hobo permission methods

For the code that is generated by the `hobo resource` generator, the method is checking whether the acting user, which is the user that is signed on, is or is not the administrator. In practice though, the boolean value may ask another question or a more complex question.

For example, one could write a line of Ruby code that determined if the signed on user was the admin AND the time was between 8:00 AM and 5:00 PM. In other words, there can be other logical determinations but you have to know a little Ruby.

Method	Meaning
<code>acting_user</code>	the user who is being tested for permission
<code>administrator?</code>	first user to sign up
<code>signed_up?</code>	any user who is signed up (including the administrator)
<code>guest?</code>	any user who is not signed up

Table 2: Table of Hobo "acting_user" options

For these tutorials, we will use the `acting_user` object and its methods: `administrator?`, `signed_up?`, and `guest?`. Hobo encodes information about the user of its applications in the `active_user` object that determines if the user is an administrator, other signed up user or a guest user.

For example, `acting_user.administrator?` is true if the user is the administrator and false if the user is not. If we place it within the `create_permitted?` method, Hobo only permits users who are administrators to create database records related to the model containing the method.

The meaning of the default permissions code can be summarized: Only the administrator is permitted to create, update or destroy records and anyone can view records. Using the `view_permitted?` method is a little more involved, so we will wait until the intermediate tutorials to tell you about it.

Before trying this out, it is useful to understand how Hobo implements these permissions within Hobo's UI. Yes, Hobo not only provides the facility to set permissions but it also takes care of providing the right links and controls within the UI.

Convention: The ‘?’ after signed up indicates the method is a Boolean method.

When there is no create permission, there is no “Create a New {model_name}” link.

- When there is no update permission, there is no edit link and no way to populate a form with an existing record.
- When there is no destroy permission, there is no “Remove this Record?” link.

This will make more sense when you learn about controller actions in the next tutorial. Hobo permissions essentially turn controller actions (what users do in the UI) on or off, depending on defined logical conditions.

Let's try something out.

As of now in your code, `users` who are not the admin can only view the records entered by the *administrator*. The *user* has no create, edit or delete permission; these options do not appear in the user interface.

Now let's make a minor change and see how the UI responds.

CHANGE:

```
def create_permitted?  
  acting_user.administrator?  
end
```

TO:

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

Update your browser and you will see the *New Recipe* link appear at the bottom of both the *Home* and *Recipes* tabs. Now do the following:

CHANGE:

```
def update_permitted?  
  acting_user.administrator?  
end  
def destroy_permitted?  
  acting_user.administrator?  
end
```

TO:

```
def update_permitted?
  acting_user.signed_up?
end
def destroy_permitted?
  acting_user.signed_up?
end
```

Click a recipe title. On the right hand side of the screen showing the record, you will see an *Edit Recipe* link now indicating editing permission. Click this *edit* link and you will now see a full editing page as well as a *Remove This Recipe* delete link in the upper right of the page.

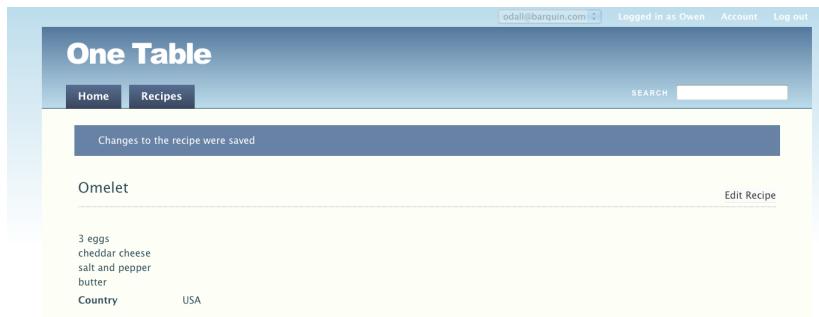


Figure 89: Page view of a Recipe

Try changing all of the `signed_up?` methods to `guest?` and you will observe that you have full permissions even if you are not signed in and none if you are!

Complete the tutorial by putting back all three methods to `signed_up?`.

Tutorial 5 – Controllers

Topics

- Introduce Hobo's controller/routing system.
- Hobo automatic actions
- Show examples of the permission system working with controllers

Tutorial Application: one_table

Steps

- Demonstrate controller actions.** Hobo has a set of built in actions for responding to user-initiated requests from browser actions (clicks). For example, when Hobo displayed the Recipes in Tutorial 3, it is the result of the *index action* found in the /app/controllers/recipes_controller.rb file. Open this file.

Note: Recall that controller and model files contain Ruby code whereas view templates contain HTML with embedded Ruby code.

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :all
end
```

There is not much you can see—but there is a lot going on behind the scenes.

The first line is similar to the first line of the *Recipe* model we told you about in Tutorial 1. It indicates that the *RecipesController* is part of the Rails *ApplicationController* and inherits general capabilities from this master controller.

The next line, *hobo_model_controller*, tells Rails to use Hobo's controller functionality to control the Recipe model and views. It is actually short for:

```
#Do not copy - although it won't change anything if you
do. hobo_model_controller Recipe
```

Hobo automatically infers the model name from the controller name in the first line above.

Note: The pound (or “hash”) character (#) is the symbol to indicate a Ruby comment. Everything on a line following # will be ignored by Ruby. Code starts again on the next line. To create view template comments, where you are not in a Ruby file you must surround comments like this <!-Comment-> which will be visible in the generated html source or <%# comment %> which will not.

The next line, *auto_actions :all*, makes all the standard actions available to the controller including: index (meaning “list”), show, new, create, edit, update, and destroy (meaning “delete”). If you are familiar with Rails, you will realize that Hobo has replaced quite a bit of Rails code in these two lines.

2. Edit the auto_actions. Clicking the “Recipes” tab in your app invokes the index action of the Recipes controller. The index action of the controller tells Hobo to list the records of the model. You probably noticed this as you created new records. Each time you created a new one, you probably clicked on the tab to see a list of all the records you created.

Now notice something else that you will learn to be important. When you click on the Recipes tab, the URL that is displayed in the URL window says:

<http://localhost:3000/recipes>

As you learn about the functions of the fundamental Hobo actions (listed in Step 1 above), you will learn that there is a unique URL entirely specified by the action and model name. Look at figure earlier in this book about “Actions and Routes”, and you will see the URL for an *index* action is the base URL, <http://localhost:3000/> concatenated with the plural of the model name, which in this case is “**recipes**”.

We are going to further demonstrate that attempting to route to this URL invokes the index action by turning off the action in Hobo and then putting turning it back on. First, go to your home page by clicking the Home tab. Then, in `recipes_controller.rb`,

CHANGE:

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :all
end
```

TO:

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :all, :except => :index
end
```

The `except` clause in this code tells the controller to turn off the index action. Refresh your browser and you should see this display:



Figure 90: Making the Recipes tab disappear

Your Recipes tab disappeared. You can also try invoking the index action by typing “<http://localhost:3000/recipes>” into your URL window. You will get a blank page.

Hobo will no longer invoke the index action because you told it not to in your code. Hobo decided to do more though; it changed the UI also.

In Tutorial 3, you learned that Hobo figures out how your UI should look depending on your model code. There it changed what links were available depending on permissions you specified in the code. In this case, Hobo figures out how to change the UI depending on the controller code. Here it has removed a tab, the “*Recipes*” tab, because you disallowed the action that it would invoke. Now, remove the “*except*” clause and you should get your “*Recipes*” tab back.

Note: If you are new to Ruby you are probably noticing all the colons(:) and arrows (=>). For now. Think of these two as a way of connecting a Ruby symbol (any text that begins with a colon) to a value (the entity after the expression “=>”). We recommend a companion book such as Peter Cooper’s “Beginning Ruby: From Novice to Professional” to learn more about Ruby symbols and their importance.

Now turn the index action back on by deleting the *:except* clause.

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :all
end
```

3. **Remove and restore the new and show actions.** Hobo allows you to edit this in two ways. You can either stipulate you want *all except certain actions* or that you want *only specific actions*. In other words, you can either indicate which actions you

wish to *include* or indicate which actions you wish to *exclude*. The former is what you did in step three. Let's try the latter where you declare which actions you want. The following code will do exactly what you did before but in a different way.

First, use the following code to *include* all seven actions, including the *index* action. This code is equivalent to the `auto_actions :all` statement above.

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :index, :show, :new, :create, :edit, :update,
                :destroy
end
```

Try removing the *index* action. When you save your code and refresh your browser, you will obtain the same result using the “`:except => index`” code. Now, put back the *index* action and try removing the “`:new`” option.

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :index, :show, :create, :edit, :update,
                :destroy
end
```

The result is that the New Recipe link to <http://localhost:3000/recipes/new>, the URL associated with the new action disappears. This is because you have disallowed the new action and Hobo takes care of cleaning up your UI for you. Even if you try to go to that URL by typing <http://localhost:3000/recipes/new> into the browser, Hobo tells you that you can no longer go there.

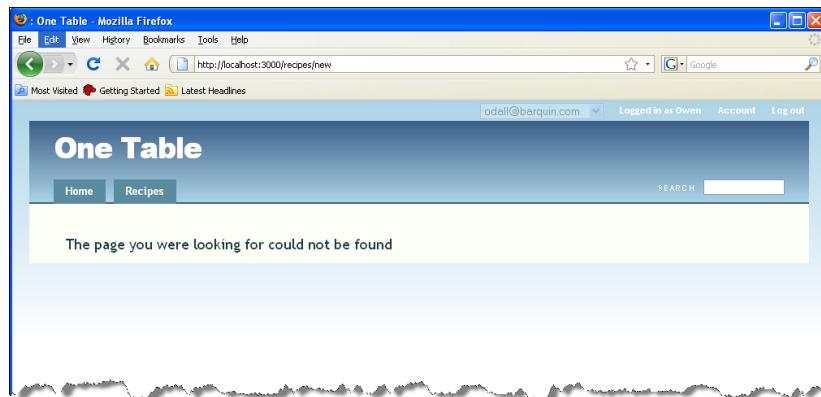
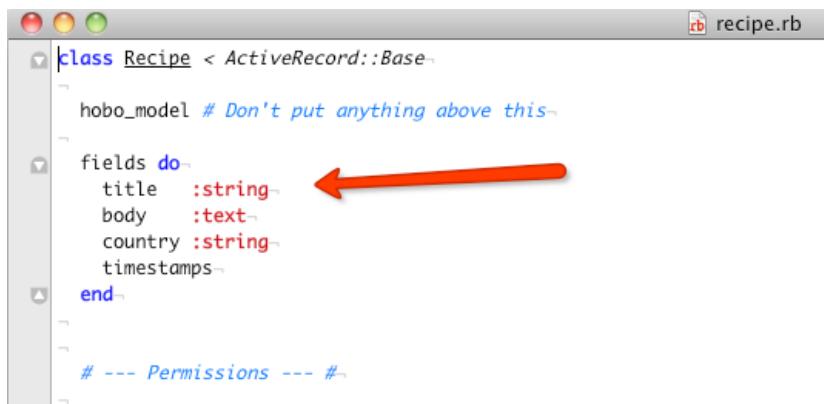


Figure 91: Error message “The page you were looking for could not be found”

Put the `:new` action back in and click the Recipes tab. Mouse over the Recipe links and note that the URL's look like, <http://localhost:3000/recipes/2-omelette> which are

of the form `http://localhost:3000/model(plural)/ID-model_name_variable` which is the form that we discussed earlier in this tutorial for the `show` action.

Note: Hobo picks a default `model_name` from the model with the value of the field it thinks is the most likely summary field. Hobo first looks for a field called `name`. Next it looks for the next most likely, which in this case it guesses is `title`. You can override the automatic name assignment by adding the option `:name => true` to the field you would like displayed as the “name”.



```
class Recipe < ActiveRecord::Base
  hobo_model # Don't put anything above this

  fields do
    title :string
    body :text
    country :string
    timestamps
  end

  # --- Permissions --- #

```

Figure 92: How Hobo finds the default "name" attribute for a model

You can also use a little “Hobo magic” to create your own version of `name` using a Ruby method as below:



```
class Recipe < ActiveRecord::Base
  hobo_model # Don't put anything above this

  fields do
    title :string
    body :text
    country :string
    timestamps
  end

  def name
    "My Custom Name = " + title
  end

```

Figure 93: Creating your own custom "name" attribute



Figure 94: Page view of the custom name attribute

Now back to our original train of thought... Remove the “:show” action.:

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :index, :new, :create, :edit, :update,
                :destroy
end
```

Now, when you refresh your browser you will note that you no longer have links to show(display) the details of a particular Recipe record. Even if you try to navigate your browser to <http://localhost:3000/recipes/2-omelette>, you will get an error.

Now, let's try one more but using the except version of auto_actions again but first make sure you are back to the all actions state. Use the code below.

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :all
end
```

Navigate to the Recipes link where you should now see a list of hyperlinks to each recipe. Click on a recipe.

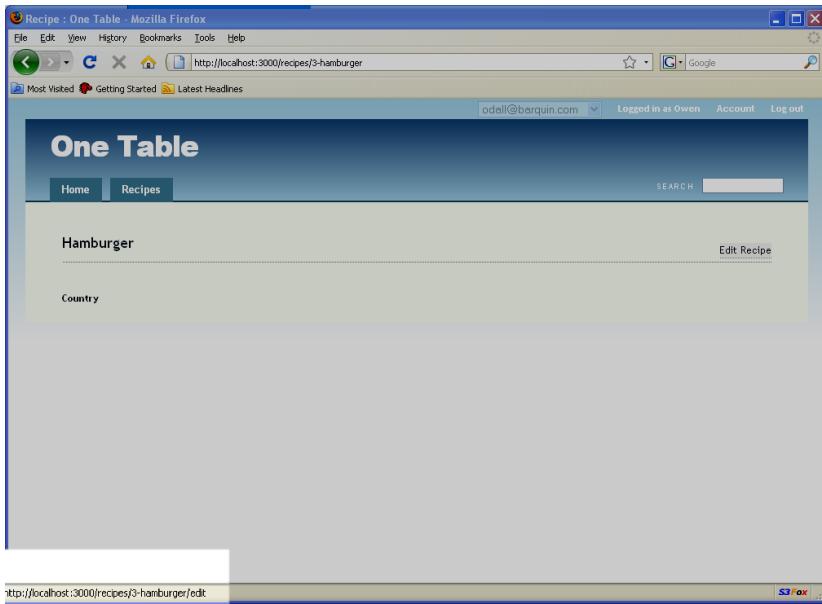


Figure 95: Viewing the edit URL

Observe the *Edit Recipe* link on the right hand side of the display. Click or mouse over it to convince yourself that the URL associated with this link is:

<http://localhost:3000/recipes/6-hamburger/edit>

This is just the result you would expect for the edit action of the form:

[http://localhost:3000/model\(plural\)/ID-model_name_variable/edit](http://localhost:3000/model(plural)/ID-model_name_variable/edit)

Now, make sure you are on the screen above, a particular *Recipe*. Edit your code to remove the `edit` action.

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :all, :except => :edit
end
```

Now you should see that Hobo removes the links to the edit action and even if you try to force Hobo to go to the above URL, it will not, giving you an error:



Figure 96: “Unknown action” error page

1. Remove multiple actions. So far we have showed you how to remove one action at a time. You can use the two methods we have showed you to remove two or more actions at a time. If you use the listing approach and you are starting with all the actions as in:

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :index, :show, :new, :create, :edit, :update,
                :destroy
end
```

If you want to remove both the “new” and the “create” actions, just delete them from your list so that you have:

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :index, :show, :edit, :update, :destroy
end
```

If you start be specifying all actions and use the `:except` clause, the equivalent code to the above will be:

```
class RecipesController < ApplicationController
  hobo_model_controller
  auto_actions :all, :except => [:new, :create]
end
```

Note: When removing the :new action, this actually adds a 'New' facility below the list of Recipes. When you remove the :show action, Hobo places an 'Edit' link against each listed item.

You may be wondering why the :except option encloses the list of actions in square brackets and the listing approach does not. The Ruby :except method takes a Ruby array as an input and Ruby arrays are enclosed in square brackets.

5. Using controller short cuts. There is one other way to add or remove controller actions and that is through the use of short cuts. The code:

```
auto_actions :read_only
```

is the same as:

```
auto_actions :index, :show
```

The code:

```
auto_actions :write_only
```

is the same as:

```
auto_actions :create, :update, :destroy
```

Note: You can append actions or use the except actions clause with either of these short cuts. The proviso is that you **must** use the shortcut first and [use only one] and use the except clause last.

6. Hobo Controller action summary. Below is a list of all controller actions

Action	Summary Meaning	URL Mapping	Example
index	display list of records	/base/model(plural)	/base/recipes
show	display a single record	/base/model(plural)/ID-name	/base/recipes/2-omlette
new	allocate memory for a new record and open a form to hold it.	/base/model(plural)/ID-name	/base/recipes/new
create	save the new record.	link without landing	/base/recipes
edit	retrieve a record from the database and display it in a form	/base/model(plural)/ID-name	/base/recipes
update	save the contents of an edited record	lands on show	/base/recipes
destroy	delete the record	lands on index	/base/recipes

Table 3: Hobo Controller action summary

Tutorial 6 – Navigation Tabs

This tutorial provides an introduction to Hobo’s automatically generated tags. We will start with the navigation tabs that are generated for each mode. We will show you where to find them and how to make a simple edit to change how navigation tabs are displayed. We will explore this more deeply in Chapter 4.

Topics

- Locate Rapid directories
- Edit the navigation tab

Tutorial Application: one_table

Steps

1. **Find Hobo’s auto-generated tags.** Open up the views directory and navigate to the rapid directory by following this tree: `views/taglibs/auto/rapid`. You will see three files called: `pages.dryml`, `forms.dryml`, and `cards.dryml`. It is here that Hobo keeps its default definition of the tags its uses to generate view templates.
2. **Open the `pages.dryml` file.** Take a quick look through this file and you will see tag definitions such as:

```
<def tag="main-nav"> . . .
<def tag="index-page" for="Recipe">
<def tag="new-page" for="Recipe">
<def tag="show-page" for="Recipe">
<def tag="edit-page" for="Recipe">
```

Notice how, except for the `<main-nav>` tag these correspond to the actions of Hobo Controller action summary above in Tutorial 5. You will further note that these are just the actions that require a view (remember index means list). The other actions, create, update, and destroy only needed a hyperlink. We are only mentioning this now to pique your curiosity for Chapter 4 where you will delve deeply into Hobo's way of creating and editing view templates.

3. **Edit the `<main-nav>` tag.** Copy the following code and paste it into your `views/taglibs/application.dryml` file. Hobo automatically uses code in this file instead of what it finds in `pages.dryml`. In other words, `application.dryml` overrides `pages.dryml` and further makes it available to the entire application.

```
<def tag="main-nav">
  <navigation class="main-nav" merge-attrs
    param="default">
    <nav-item href="#{base_url}"/>Home</nav-item>
    <nav-item with="&Recipe">
      <ht key="recipe.nav_item" count="100">
        <model-name-human count="100"/>
      </ht>
    </nav-item>
  </navigation>
</def>
```

5. **Rename a Navigation Tab.** By convention, Hobo names tabs, other than the Home tab with the plural of the model name. In this case, that is ‘Recipes’ Let’s try renaming this to ‘My Recipes’. There are a couple of ways to do this:

- Use Hobo’s i18n module to specify a new value for the `recipe.nav_item` key
- Override the `main-nav` tag in `application.dryml` and rewrite it to specify ‘My Recipes’ as the Recipe tab label

The preferred way of doing this is the first option and specify a new value for the `recipe.nav_item` key, the reason for this is that this maintains the application’s ability to handle multiple languages. If we overrode the `main-nav` tag in `application.dryml`, this would be a ‘hard-coded’ solution and would always show ‘My Recipes’ regardless of the current locale/language.

Just add the following to your `config/locales/app.en.yml` file:

```
en:
  recipe:
    nav_item: "My Recipes"
```

Refresh your browser and you will see the renamed tab:- “My Recipes”



Figure 97: Customizing the name of a tab

6. Remove the Home Tab. Instead of deleting the Home tab, just comment it out by surrounding it with <!-- ... -->.

Note: Since view files are essentially HTML and not Ruby code, you use the HTML commenting syntax instead of the Ruby comment syntax. However know that HTML comments will be visible in the generated page source.

```
<def tag="main-nav">
  <navigation class="main-nav" merge-attrs>
    <!--<nav-item href="#{base_url}"/>Home</nav-item>-->
    <nav-item with="&Recipe">
      <ht key="recipe.nav_item" count="100">
        <model-name-human count="100"/>
      </ht>
    </nav-item>
  </navigation>
</def>
```

Now refresh your browser and you will see the Home tab has been removed:



Figure 98: Removing the default Home tab

7. Reset the tabs. Since editing the application.dryml file will interfere with future tutorials, delete the code you copied above.

```
<def tag="main-nav">
  <navigation class="main-nav" merge-attrs>
    <!--<nav-item href="#{base_url}"/>Home</nav-item>-->
    <nav-item with="&Recipe">
      <ht key="recipe.nav_item" count="100">
        <model-name-human count="100"/>
      </ht>
    </nav-item>
  </navigation>
</def>
```

Tutorial 7 – Model Relationships: Part 1

You will learn how to create a new model that is related to another table. You will replace one of your table's original fields with a key that is linked to a foreign key in order to select values. You will see how Hobo automatically creates a drop-down control to select values that you have entered.

You will also make some controller action edits [and some permissions changes] to refine the user interface.

More specifically, you will add a new model to hold the names of countries that a user will select from the New Recipe page. The application will identify the foreign key for that country and place it in the recipes table.

Topics

- Model relationships

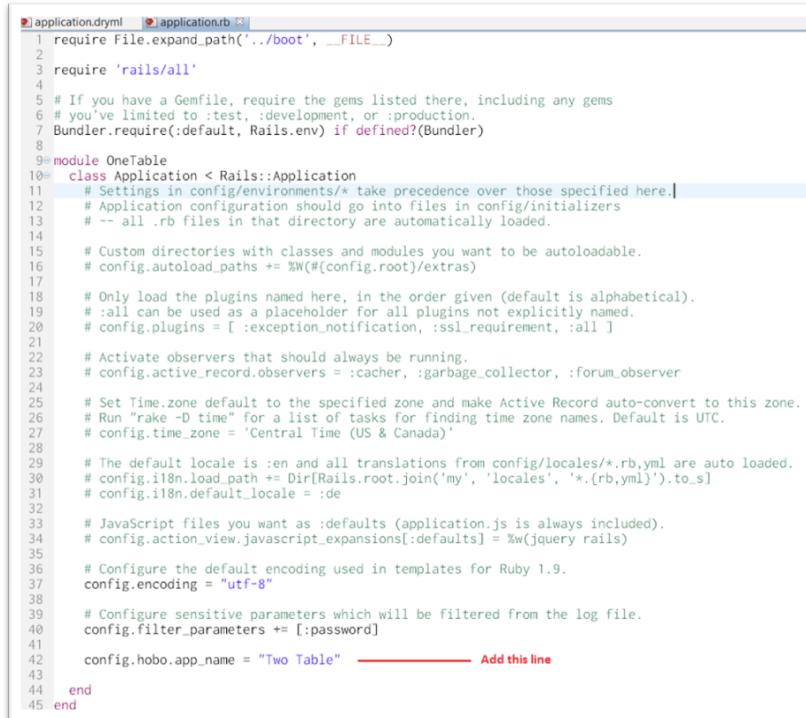
- Foreign keys
- Drop-down list boxes

Tutorial Application: one_table

Steps

1. **Copy the Application.** If you would like to preserve your application in its state as of the end of Tutorial 6, you may wish to copy the application and work on the new version. Copy the entire application directory and paste it into a folder called `two_table` in your `tutorials` directory.

To change the application name, make the following change to `config/application.rb`:



```

application.drb application.rb
1 require File.expand_path('../boot', __FILE__)
2
3 require 'rails/all'
4
5 # If you have a Gemfile, require the gems listed there, including any gems
6 # you've limited to :test, :development, or :production.
7 Bundler.require(:default, Rails.env) if defined?(Bundler)
8
9 module OneTable
10   class Application < Rails::Application
11     # Settings in config/environments/* take precedence over those specified here.
12     # Application configuration should go into files in config/initializers
13     # -- all .rb files in that directory are automatically loaded.
14
15     # Custom directories with classes and modules you want to be autoloadable.
16     # config.autoload_paths += %W[#{config.root}/extras]
17
18     # Only load the plugins named here, in the order given (default is alphabetical).
19     # :all can be used as a placeholder for all plugins not explicitly named.
20     # config.plugins = [ :exception_notification, :ssl_requirement, :all ]
21
22     # Activate observers that should always be running.
23     # config.active_record.observers = :cacher, :garbage_collector, :forum_observer
24
25     # Set Time.zone default to the specified zone and make Active Record auto-convert to this zone.
26     # Run "rake -D time" for a list of tasks for finding time zone names. Default is UTC.
27     # config.time_zone = 'Central Time (US & Canada)'
28
29     # The default locale is :en and all translations from config/locales/*.rb,yml are auto loaded.
30     # config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.rb,yml').to_s]
31     # config.i18n.default_locale = :de
32
33     # JavaScript files you want as :defaults (application.js is always included).
34     # config.action_view.javascript_expansions[:defaults] = %w(jquery rails)
35
36     # Configure the default encoding used in templates for Ruby 1.9.
37     config.encoding = "utf-8"
38
39     # Configure sensitive parameters which will be filtered from the log file.
40     config.filter_parameters += [:password]
41
42     config.hobo.app_name = "Two Table"  ----- Add this line
43
44   end
45 end

```

Figure 99: Changing the Application Name

Shut down the web server by issuing a `<control-c>` in the command window where you issued the rails server command.

Restart the web server and you are ready to go.

```
\two_table> rails server
```

2. Add drop down control for preset selections. This tutorial is about adding associations between tables. In subsequent steps, we are going to show you how to create a new Countries table to store the values of country names to associate with your recipes. Hobo will take care of the user interface rendering, as you will soon see.

Before we do that though, let's demonstrate the simpler approach. This is the easy way to go for applications when you know at design time all the possible values of a category. In this case, you would not need to add the additional complexity of creating a table to maintain all values for countries. All that is needed is to specify in the model the list of possible values using the `enum_string` attribute of a field. For this tutorial let's assume the only values for country will be: American, French & Chinese.

Your `recipe.rb` model code should now look like:

```
class Recipe < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    title :string
    body :text
    #country :string
    country enum_string(:American, :French, :Chinese)
    timestamps
  end
```

We have used the `enum_string` field method to declare the possible values for country. So we can easily see what we have done, we have commented out the old version of the `country` field declaration by preceding it with a '#' (hash). Now, refresh your browser and click 'New Recipe,' and you will see a drop-down control that lets you select values for country.



Figure 100: Using “enum_string” to create a drop-down list of Countries

This is fine as long as you don’t have to change the possible values. In the next steps, we will show you how to create a new table to store country values and be able to edit it on the fly and have it be reflected in your GUI. You will not have to write any queries. Hobo will take care of everything for you.

3. Remove drop down control. First let’s get back to where we started before adding a new table. Edit your code to look like this.

```
class Recipe < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    title :string
    body :text
    country :string
    #country enum_string(:American, :French, :Chinese)
    timestamps
  end
```

The drop-down control will now be gone when you refresh your browser.

Note: Remove the custom name attribute you created in the last tutorial before continuing.

4. Creating model associations. In the next several steps, we will add a Country model, set up a relationship between the Country model and the recipe model and then run a Hobo migration to create the Countries table. This last step will also set up

the foreign key in the Recipe model that will maintain the association to the index of the new Country model, `country_id`.

When you look in the `db/schema` file to review the fields in your tables, you will not see the ID's of any table listed but they are there. Every time you create a table using a migration in Hobo, it will also create the table index with a name defined by convention to be the model name with '`_ID`' appended.

5. Add a new model. We will use Hobo's resource generator to create a new model with one field to store a country's name. If you do not have a command prompt window open besides the window you used to start your web server, open a new one now and navigate to the root of the application.

```
\two_table>
```

Execute the following command from your command prompt:

```
\two_table> hobo g resource country name:string
```

Check the models directory and you should see a `country.rb` file with the following contents defining the Country name field:

```
class Country < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end
```

If you look in the `db/schema` file, however, you will not see a `countries` table because you have not run the migration yet. Now let's define our relationships.

The hobo resource generator also created some other files. An important one is the controller file called `countries_controller.rb`. Note that the class names (how Hobo refers to them) are `CountriesController` for the controller and `Country` for the model, which you can see, in the first line of code in the respective files.

Note: The controller has a file and class name that is the plural of the model name. The file names use underscores in the file names and removes them for class names.

6. Remove a field. In preparation for setting up a relationship between the Recipe and Country models, you must delete the `country` field in the Recipe model. It will not be needed any more since it is replaced by the `name` field in the Country model.

Open the `recipe.rb` model file and delete the `country` field from the `fields...do` block at the beginning of the file. So you can see what you have done, it would easiest to comment it out. Change this:

```
...
  fields do
    title :string
    body :text
    # country enum_string (:American, :French, :Chinese)
    timestamps
  end
...

```

7. **Add a belongs_to relationship.** The Recipe model will have what is called a `belongs_to` relationship with the new Country model. This relationship or association requires that every recipe have at the most one country associated with it. Add the `belongs_to` declaration just before the `#permissions` comment. Any `belongs_to` declarations must be added after the `fields...do` block.

```
class Recipe < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    title :string
    body :text
    # country enum_string (:American, :French, :Chinese)
    timestamps
  end
  belongs_to :country
  validates_presence_of :country
...

```

We have also added validation so that the country is always specified for all recipes.

Note: It is useful to read `belongs_to` as ‘refers to’ to remind yourself that when this relationship is declared, it causes the creation of a key field named `country_id` in the `recipes` table to “refer to” the `country` table, which contains the name field.

In the above `belongs_to` statement, `:country` is the name of a relationship. It is not the name of a field. Through its naming conventions, Hobo determines that the model to relate to is named `Country`. For the case when naming conventions fail, you can force the relationship as in the following code:

```
belongs_to :country, :class_name=>"Some_other_model"
```

8. **Add a has_many relationship.** The Country model needs the inverse relationship to the `belongs_to` in the Recipe model:

```
class Country < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end
  has_many :recipes
  ...

```

When you learn to do more sophisticated programming, this feature of naming relationships, which Hobo inherits from Rails, will become a powerful tool. Unlike standard relational database relationships, this capability essentially adds meaning to the relationship.

9. Run the Hobo migration. Now you have done everything needed for Hobo's intelligence to take over and create the new `countries` table and set up the proper foreign keys.

Go to your command prompt and run the Hobo migration. By doing this you will allow Hobo to accomplish several things. Hobo will:

- Create the migration file for the new table, `countries`
- Remove the `country` field from the `recipes` table
- Set up a foreign key to handle the relationship between `Recipe` and `Country`
- Execute the migration to create the new database table, `Countries`.

For every recipe record with a country entered, there will now be a `country_id` value written in the `recipe` stable that corresponds to a `country_id` in a `country` record.

```
\two_table> hobo g migration
```

You will get the following response: , showing a problem

```
DROP or RENAME?: column recipes.country
Rename choices: country_id
Enter either 'drop country' or one of the rename
choices:
```

Hobo has noticed that there is an ambiguity you have created that needs to be resolved. There is both a `country` field and a `Country` model. It knows you need a foreign key, `country_id`, to relate to the `Countries` table. So it gives you a choice to rename `country` to `country_id` or drop the `country` field and create a new `country_id` field. Since `country` has real country names in it, not foreign key integer values, it is best to drop it and let Hobo create a new field for the foreign key.

Enter 'drop country' (without quotation marks) in response.

Next the migration will respond as follows:

```
What now: [g]enerate migration, generate and [m]igrate  
now or [c]ancel?
```

You should type “m”.

Lastly it will prompt you to name the migration file:

```
Filename [hobo_migration_3]:
```

Hit the “enter” key and it will take the default name, “hobo_migration_3”.

10. Review the results of your migration. Let’s take a look at the database schema in db/schema.rb:

```
ActiveRecord::Schema.define(:version => 20100313165708) do  
  create_table "countries", :force => true do |t|  
    t.string    "name"  
    t.datetime "created_at"  
    t.datetime "updated_at"  
  end  
  create_table "recipes", :force => true do |t|  
    t.string    "title"  
    t.text     "body"  
    t.datetime "created_at"  
    t.datetime "updated_at"  
    t.integer   "country_id"  
  end  
  add_index "recipes", ["country_id"], :name => "index_recipes_on_country_id"  
  create_table "users", :force => true do |t|  
    t.string    "encrypted_password",           :limit => 40  
    t.string    "salt",                      :limit => 40  
    t.string    "remember_token"  
    t.datetime "remember_token_expires_at"  
    t.string    "name"  
    t.string    "email_address"  
    t.boolean   "administrator",           :default => false  
    t.datetime "created_at"  
    t.datetime "updated_at"  
    t.string    "state",                   :default => "active"  
    t.datetime "key_timestamp"  
  end  
  add_index "users", ["state"], :name => "index_users_on_state"  
end
```

Note: Hobo automatically creates appropriate indexes for table relationships with foreign keys. We will discuss how to enhance or disable this feature in a later tutorial.

11. **Double-check the tab code before refreshing your browser.** Back in Tutorial 6 #7, we asked you to delete the <navigation> tag. Go back there and make sure you completed that step before refreshing your browser. You should see a new tab for Countries.
12. **Review a few features of the UI.** Make sure you are signed in as the admin. Go to the “Countries” tab and click through to enter a few countries.



Figure 101: Index page for Countries

Then go to the Recipes tab and click through to edit one of your recipes. You should now see a drop down box just as you saw when you used the enum_string option for your attribute:

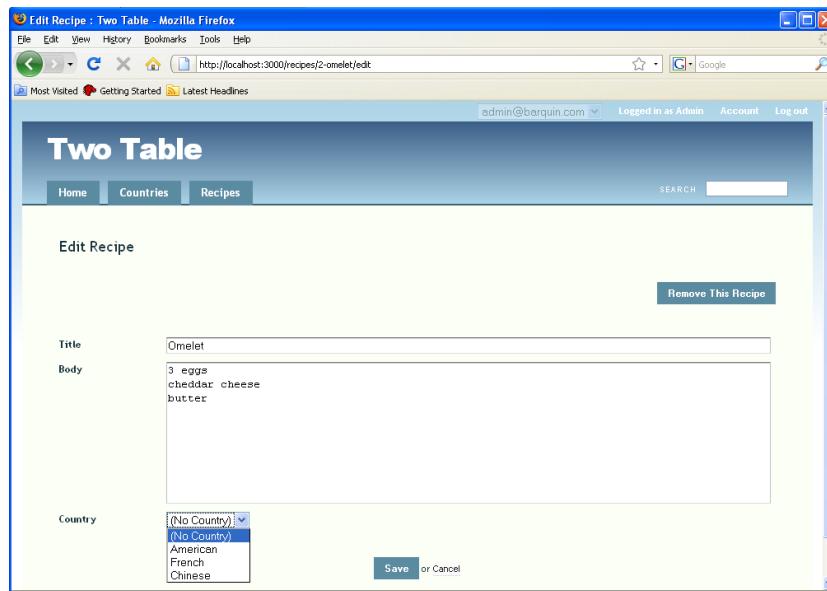


Figure 102: Selecting a Country for a Recipe

The difference is that you are actually selecting a `country_id` foreign key behind the scenes. Hobo takes care of querying the `countries` table (Country model) and displaying the actual country names. When you save this “Recipe” record, Hobo automatically maintains all of the necessary related keys.

After you do the “save”, note that the “Country” value in the page is an active hyperlink:

CHAPTER 3
INTRODUCTORY TUTORIALS

TUTORIAL 6 – NAVIGATION TABS

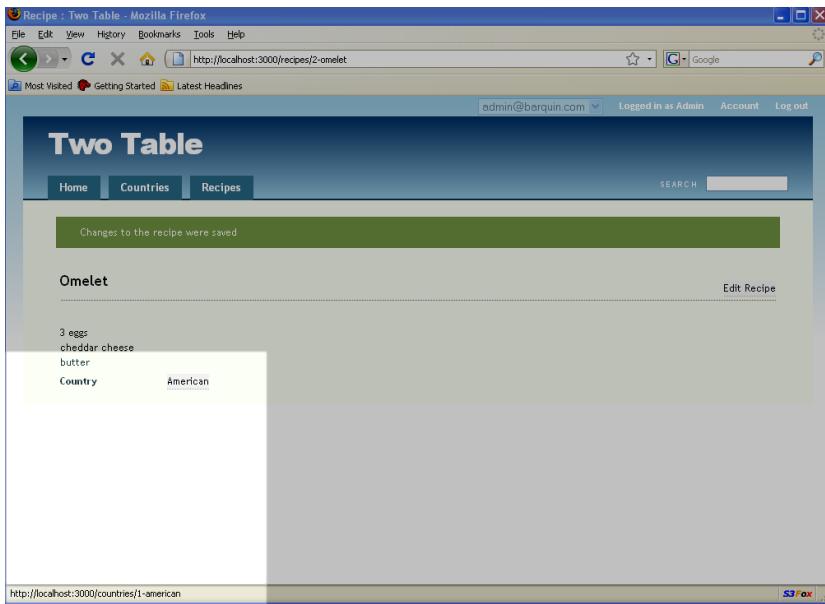


Figure 103: Active link on Country name in the Recipe show page

If you click it, you will see a screen that allows you to edit the “Country” record.



Figure 104: The Country show page access from the Recipe show page

You can edit a country record because you are logged in as the “administrator”. If you check the `countries.rb` file, you will see that the permission to edit the “Country” field is limited to the administrator. This means that if you log in as a regular user, Hobo should not allow the edit. Log out from the administrator account and login as regular user.

```
class Country < ActiveRecord::Base
  ...
  # --- Permissions --- #
  def create_permitted?
    acting_user.administrator?
  end
  def update_permitted?
    acting_user.administrator?
  end
  ...
  
```

Go to the Recipes tab, click on “recipe” link and edit the recipe. Next click on the country name on the page. The “Edit Country” link is no longer available.



Figure 105: Only an Administrator is provided the Country Edit link

12. One-to-many relationship discussion. The relationship or association that you have just implemented is known as a one-to-many relationship. In this particular situation, we have an individual country that is related to many recipes. More specifically, there is one record in the “Countries” table with the name ‘American,’ but, potentially, many American recipes.

Tutorial 8 – Model Relationships: Part II

In this tutorial you will learn to implement many-to-many relationships. These relationships are useful, for example, in categorizing a model’s records. You will implement the relationship using the “has_many”, “has_many => :through”, and “belongs_to” relationship declarations of Rails. You will learn how Hobo establishes a direct relationship between model relationships and the features of the UI.

In terms of our tutorial application, you will be adding recipe categories so that you can categorize recipes as, for example sweet, sour, or hot. You will implement an architecture where it is easy to invert the relationships so that you can display both which categories a recipe belongs to and which recipes are classified in a particular category.

PREREQUISITES: Tutorials 1-6.

Topics

- Many-to-many relationships
- Using the `has_many`, `has_many =>:through`, and `belongs_to` rails relationship declarations
- Fixing a UI assumption by Hobo when it is not the optimum.

Tutorial Application: `four_table`

Steps

1. **Copy the Application.** Just like you did in Tutorial 7, we suggest you copy your application from Tutorial 7 in order to easily go back to its state at the end of that tutorial. Shut down the web server by issuing a <Control-C> in the command window where you issued the rails server command.

Then, do a copy in whatever operating system you are using. We have called the new application directory “`four_table`”. Navigate to the new directory. Restart the web server and you are ready to go.

```
\four_table> rails server
```

You may wish to change the name of your application as displayed in the UI. Go to `config/application.rb`. Change the key `config.hobo.app_name` to read:

```
config.hobo.app_name = "Four Table"
```

Refresh your browser and you will see the new name.

2. **Create the models.** We are going to add two new models to our original application and keep the original `Recipe` and `Country` models. “`Category`” model and a “`CategoryAssignment`” model.

`CategoryAssignment` will have the two fields, `category_id` and `recipe_id` that correspond to keys of the same name in the `Category` and `Recipe` models.

Note: If you review the schema in the app/db directory, you will not see these fields listed in the Categories and Recipes table. They are the default keys for these tables. Rails does not list them.

As you will see shortly, you do not have to worry about creating or naming any of these fields, The Hobo generators will take care of it.

Go to your command prompt and issue the following two commands:

```
\four_table> hobo g resource category name:string  
\four_table> hobo g model category_assignment
```

The first command will create both a controller and model, Category being the name of the model. The second will create a CategoryAssignment model but no controller.

When you implement the relationships below, you will see that CategoryAssignment sits in between the Recipe and Category models. You do not need a CategoryAssignments controller because you will be accessing recipes and categories through these models directly and need no actions that pull data directly from the intermediary CategoryAssignment model.

3. Add relationships to your models. Edit the models as shown below to enter model relationships.

Note: Hobo migrations rely on both the field declarations in your models AND the relationship declarations. The relationship declarations allows Hobo to setup all the necessary keys to implement real model relationships.

recipe.rb

```
class Recipe < ActiveRecord::Base  
  hobo_model # Don't put anything above this  
  fields do  
    title :string  
    body :text  
    #country :string  
    timestamps  
  end  
  belongs_to :country  
  has_many :categories, :through => :category_assignments,  
           :accessible => true  
  has_many :category_assignments, :dependent => :destroy
```

category.rb

```
class Category < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end
  has_many :recipes, :through => :category_assignments
  has_many :category_assignments, :dependent => :destroy
```

category_assignment.rb

```
class CategoryAssignment < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    timestamps
  end
  belongs_to :category
  belongs_to :recipe
```

4. Discussion of model relationships. Above you used the `has_many` and the `belongs_to` relationships. You further used a `has_many` relationship with a `:through` option.

Let's start with the `belongs_to` relationship, which we used in Tutorial 7 and declared in the `CategoryAssignment` model above.

When you see `belongs_to`, think refers to, and you will understand that these declarations cause the `category_id` and `recipe_id` fields to be placed in the `category_assignments` table.

The `has_many :through` statements instructs Hobo/Rails to setup the necessary functions to access a category from a recipe or a recipe from a category. The vanilla `has_many` statements set up the one to many relationships between the `recipes` table and the `category_assignments` tables and between the `categories` and `category_assignments` tables.

The `:dependent => destroy` option makes sure that when either a recipe or category is deleted that the corresponding records in the `category_assignments` table are removed automatically too.

5. Run the hobo g migration. Go to your command prompt and run the following.

```
\four_table> hobo g migration
```

Remember to respond “m” when prompted for migration and just press “enter” when prompted with the migration file name.

Note: At this point, if your web server is still running from earlier tutorials, you need to terminate it and restart it. Rails and Hobo will not recognize a new database table without doing so.

```
\four_table> rails server
```

6. **Populate the new table.** Open up your browser to <http://localhost:3000/> and you should see the following (since you had copied over all the files from the Two Table application, your database came with you so you will not see the Register Administrator form):

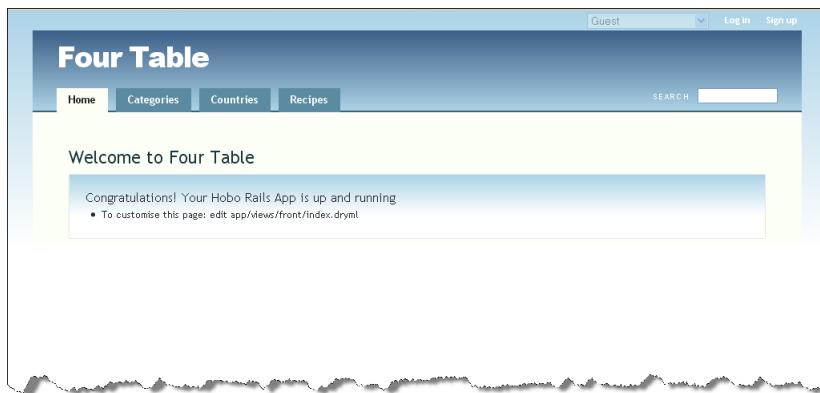


Figure 106: The Categories tab on the Four Table app

Now, go to the new Categories tab and enter in some food categories:

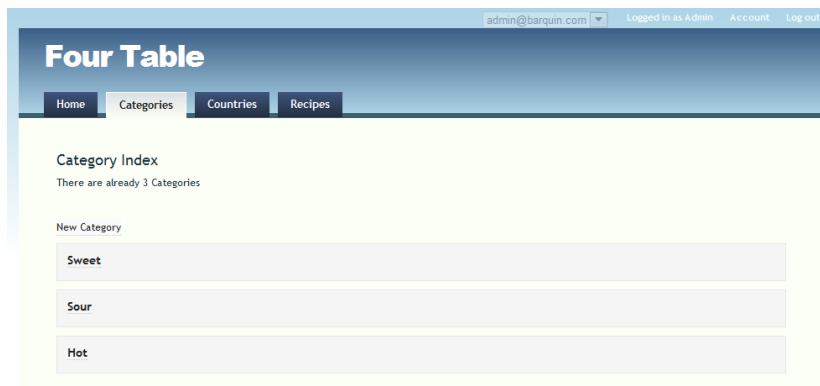


Figure 107: The Index page for Categories

7. Adding new records to the relationships. Go to the “Recipes” tab. Click on one of the recipes and you should get this.

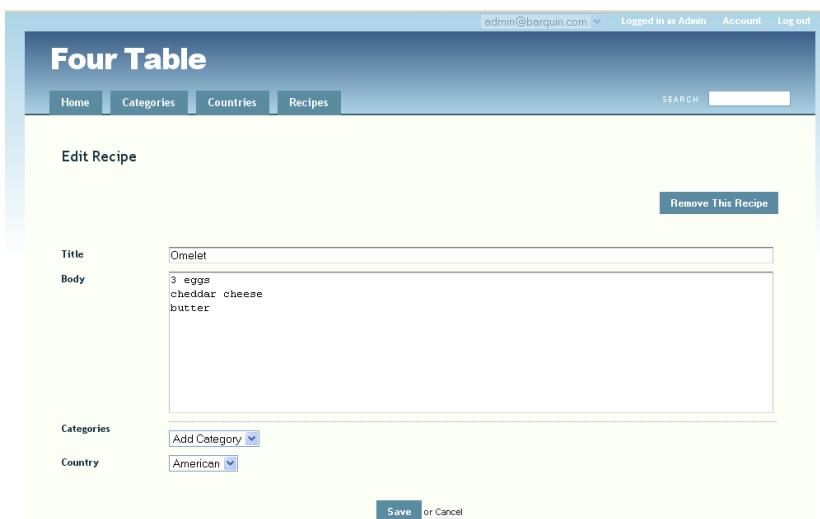


The screenshot shows the 'Four Table' application interface. At the top, there's a navigation bar with links for Home, Categories, Countries, and Recipes. The user is logged in as Admin. On the right, there are links for Account and Log out, and a search bar. The main content area is titled 'Four Table' and shows a recipe for 'Omelet'. The recipe details include '3 eggs', 'cheddar cheese', and 'butter'. Below this, under 'Country', it says 'American'. Under 'Category Assignments', it states 'No category assignments to display'. On the right side of the recipe card, there is a link labeled 'Edit Recipe'.

Figure 108: “Category Assignments” on the Recipe show page

Notice there is no category assignment.

Then click “*Edit Recipe*” on the right.



The screenshot shows the 'Edit Recipe' page for the 'Omelet' recipe. The title is 'Omelet' and the body contains '3 eggs', 'cheddar cheese', and 'butter'. In the 'Categories' section, there is a dropdown menu labeled 'Add Category'. The 'Country' section shows 'American'. At the bottom, there is a 'Save' button and a 'Cancel' link.

Figure 109: Assignment multiple Categories to a Recipe

Now you can see a new drop-down box that lets you add categories to your recipe. Hobo has taken care of this for you by inferring that you need it from your model relationship declarations.

Note: Here is a good example of the DRY (Don't Repeat Yourself) notion playing out. If the necessary UI controls can be directly inferred from model structure, there should be no need to directly code it yourself. You may wish to use a different control but Hobo picks a reasonable one for you so you do not have to bother unless you want to.

Take a look at the URL that activated the page. You will see that the URL is of the form for a “controller edit” action. If you need to remind yourself of the form look at the Hobo Controller Action Summary figure in Tutorial 5 step 6.

Try adding a couple of categories and save the changes.

The screenshot shows the 'Edit Recipe' page for a recipe titled 'Omelet'. The body of the recipe contains the ingredients: '3 eggs', 'cheddar cheese', and 'butter'. Under the 'Categories' section, 'hot' and 'sour' are listed as assigned categories, each with a 'Remove' button. There is also an 'Add Category' input field and a dropdown for 'Country' set to 'American'. The top navigation bar includes links for Home, Categories, Countries, and Recipes, along with user information and a search bar.

Figure 110: Edit page view of a Recipe with multiple Categories assigned

Here, on the *Edit Recipe* screen, you can see that Hobo is displaying the entries for the *Recipe categories* you have chosen to associate with the recipe, namely hot and sour. So far, Hobo is doing just what we would expect.

8. Add information to the model in order to display the associations. Hobo, before version 1.3, used “ViewHints” to indicate to the application how you would like to display related information on a model’s show page. To simplify things, this functionality has been moved into the model itself and ViewHints are no longer used as of Hobo 1.3. Now you modify the ActiveRecord model itself.

Edit the app/models/recipe.rb file. Enter the code (in *red italics* below) to tell Hobo explicitly to use categories as the child of recipes in its displays.

```
class Recipe < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    body :text
    timestamps
  end
  belongs_to :country
  has_many :categories, :through =>
    :category_assignments, :accessible => true
  has_many :category_assignments, :dependent => :destroy
  children :categories
  [ ... ]
```

Edit country.rb:

```
class Country < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end
  has_many :recipes
  children :recipes
  [ ... ]
```

Refresh your browser and choose a recipe to view:



Figure 111: Using the Hobo “children” declaration to enhance the view of related records

If you wish to see all the recipes, which are ‘hot’, you would click on ‘hot’ to check this out; or you could go to ‘Categories’ and then click on ‘hot’.



Figure 112: Show page for a Category before using ViewHints

Now let's enhance this view. Edit `app/models/category.rb`:

Enter the code (in italics and bold below) to tell Hobo explicitly to use `recipes` as the child of `categories` in its displays.

```
class Category < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end
  has_many :recipes, :through => :category_assignments
  has_many :category_assignments, :dependent => :destroy
  children :recipes
  [ ... ]
```

Refresh your browser.



Figure 113: Category page view after adding the ViewHints “children :recipes” declaration

Now you can see all of the “children” of the category “Hot” on the Category “show” page.

9. Comments on the many-to-many relationship. Let's review how you got this all to work. The end product is that you can see the categories associated with each recipe and the recipes associated with each category.

In each case you can click through to look at individual categories or recipes and edit them if you wish.

All of this is a result of having a recipe model related to a `category_assignment` model, which is, in turn, related to the category, model and vice versa. We will call the `category_assignment` model the intermediary model and the other two, “outer” models.

You have created a symmetrical set of model relationships where the two outer models have `has_many` relationships with the intermediary model and `has_many :through` relationships with each other. Conversely, the intermediary model has a `belongs_to` relationship with each of the outer models.

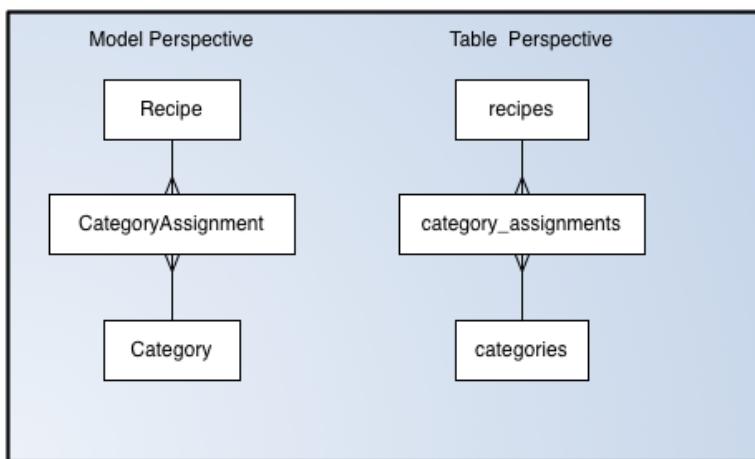


Figure 114: Model Relationships Structure

This structure will be used frequently in most data-rich applications. It is worth noting how you need only a few lines of code to implement this structure and how it lets you access each outer model from the other.

Chapter 4

INTERMEDIATE

TUTORIALS

Introductory Concepts and Comments

Tutorial 9 - Editing Auto-Generated Tag

Tutorial 10 - DRYML I: A First Look at DRYML

Tutorial 11 - DRYML

Tutorial 12 - Rapid, DRYML and Collections

Tutorial 13 - Listing Data in Table Form

Tutorial 14 - Working with the Show Page

Tutorial 15 - New and Edit Pages

Tutorial 16 - The <a> Tag Hyperlink

Introductory Concepts and Comments

In Chapter 3 we deliberately focused on helping you get something done without spending much time looking under the hood—or should we say—behind the “Magic Curtain.”

When Jeff and I first discovered Hobo, we were impressed by what seemed like little magic tricks that Tom had Hobo perform for us: dynamic AJAX without coding; automatic page flow; automatic checking and executing changes to the database when

declarations change; built-in permissions system and data lifecycles; high-level declarative markup language: you can do so much that looks and acts great.

Of course, there will ALWAYS be something you need to do that doesn't come ready-made out-of-the-box. So—just like learning magic tricks—you can learn how Hobo works and create some new magic tricks of your own to impress and help your clients in *Rapid* time.

No worthy magician will reveal all his tricks to an apprentice. A person cannot absorb it all at once. The trick to learning—as well as developing software—is to do it incrementally. Get grounded at each step. Most magic tricks use the same knowledge of human perception, habits, and expectations to create the illusions.

Learning one trick helps you learn another faster, then you learn the patterns. After that, you learn to make more patterns that others can repeatedly use.

So, in this chapter we will start revealing how “Rapid” (Hobo’s process of automatically rendering forms, views and routing) works in a way that is best be absorbed.

One of the ways is to examine the code that the author has written that runs the application itself. In the early versions of Hobo, the rendering of pages, forms, and navigation flow was done “auto-magically” by Rapid. You couldn’t see how it worked until version 0.8.0. It was in this release that Tom Locke made visible the DRYML code that was being executed in the background.

Now you can look, learn, and copy the DRYML that “Rapid” actually uses to generate Pages, Cards, Forms, and the Main Navigation Menu.

Take a close look at \apps\views\taglibs\auto\rapid folder of any of your Hobo apps:

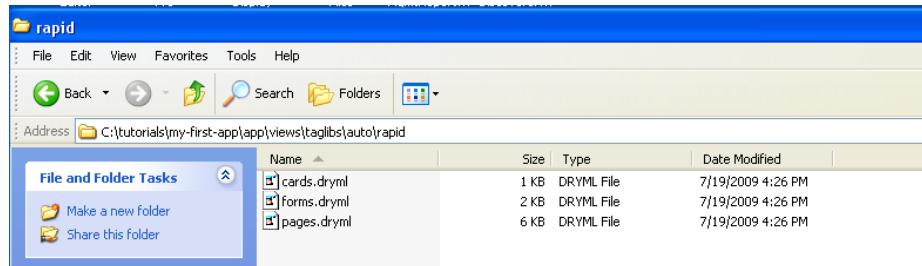


Figure 115: Folder view of \taglibs\auto\rapid

Notice that there are three DRYML files: `cards.dryml`, `forms.dryml`, and `pages.dryml`. These files include the DRYML XML-like formatted tags that are the declarative statements used as templates to render web page views and forms. They provide the logic to render a combination of HTML, JavaScript, and CSS code.

DRYML provides a high-level of abstraction for formatting web pages and dealing with all aspects of data-driven applications. Listing, displaying, creating, editing and

deleting records are simplified without specifying the granular level of detail that other frameworks require, such as Rails with its ERB (embedded Ruby) pages that are a hybrid of Ruby and HTML.

In this chapter we will explore:

1. The Hobo Rapid library of tags
2. The auto-generated DRYML files that expose the Rapid process
3. User-defined tags that you can use to extend Hobo

Hobo Rapid Library of Tags. Hobo comes with a pre-coded set of tags that you can use to build other tags. It provides tags to handle forms, display collections of records, and render a table of records. Hobo uses these to build the Rapid default web pages. You will learn to use some of the more common Rapid tags in this chapter.

Auto-generated DRYML. These DRYML files are saved replicas of Hobo's way of coding the view associated with all of the web site actions. For example, there is a <show-page> tag involved with displaying a single record, and <index-page> tag to display a list of records, and a <new-page> tag involved with generating the form to accept the data for a new record.

User-defined Tags. In order to create your own tags, Hobo provides tag definition language elements. You can build custom tags that include HTML, DRYML tags defined in Hobo's Rapid library and even imbedded custom Ruby code. There is great flexibility. The end result can be simple tag that you use in a Hobo view template to include in the definition of a web page.

Tutorial 9

Editing Auto-Generated Tags

In this tutorial, you will learn about Hobo's auto-generated tags that render views in response to controller actions. You will find your way around Hobo's Rapid directories and files where the auto-generated tags are stored. You will also learn how to make minor edits to the auto-generated tags to prepare you for making tags from tags and redefining tags in later tutorials.

Hobo's Rapid component handles the generation of an application's auto-generated tags. The auto-generated tags are built from both HTML and Hobo's internal library of XML tags called the "Rapid Library".

The most important lesson you will learn in this tutorial is how Hobo associates its fundamental auto-generated tags with the four fundamental controller actions:

- index for listing collections of records
- show for displaying a single record
- new for creating records
- edit for editing a single record

The other fundamental actions of saving new and edited records and deleting records are embedded within these fundamental tags as links because they do not need their own web pages. In addition to these four main tags, there is also a navigation tag that defines certain parts of the navigation interface.

Topics

- Edit an index page tag
- Edit a card tag
- Edit a form tag
- Edit the Navigation tags

Tutorial Application: `four_table`

Steps

1. **Start your web server.** We are going to continue on from Chapter 3 and use the `four_table` application. If you don't have it started, navigate to your `four_table` directory, in `tutorials/four_table`, and start the application.

```
\four_table> rails server
```

You should now have a UI that looks like this:

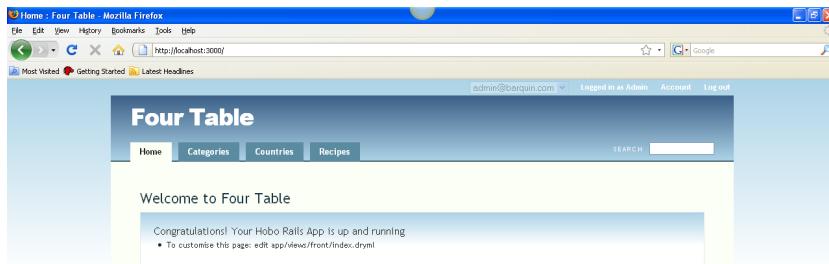


Figure 116: Front page view of the Four Table application

Open your editor and navigate to the `views/taglibs/auto/rapid` directory:

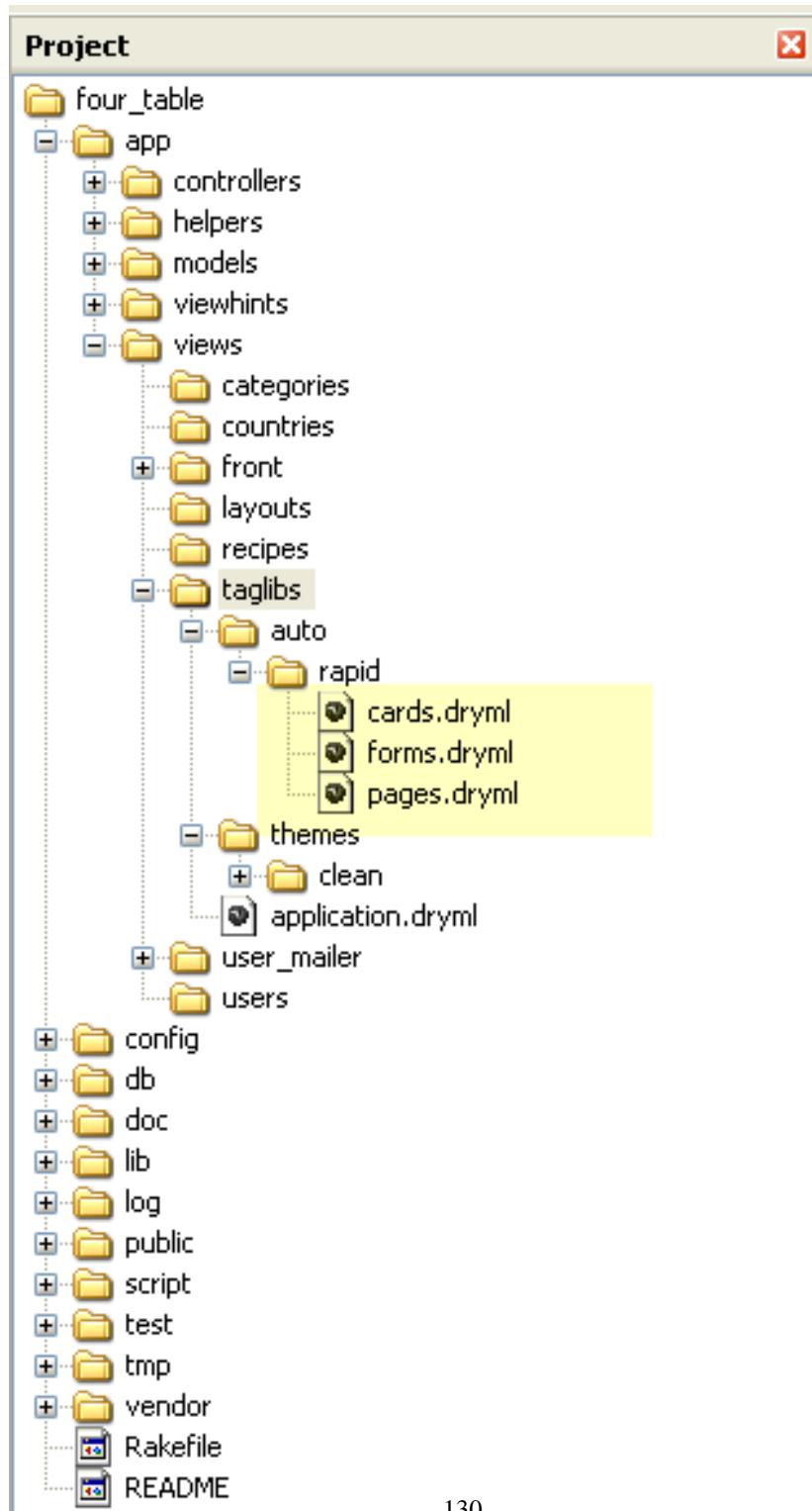


Figure 117: Folder view of the rapid DRYML files

Take a look at this directory structure. Focus on the files in the `views/taglibs/auto/rapid` directory. The Rapid auto-generated tags are stored in these files. Hobo updates the three Rapid directory files, `pages.dryml`, `forms.dryml` and `cards.dryml` every time you run a `hobo g migration`. *Don't edit these files* because Hobo will overwrite them. You can copy and paste pieces, and therefore override them, with code placed in either the `application.dryml` file or in a template file in a view directory named for a specific model, e.g., `views/recipes`. This will be explained below in this tutorial.

2. Familiarize yourself with the Rapid auto-generated files. Let's look at the `pages.dryml` file first. Open up the `views/taglibs/auto/rapid/pages.dryml` file. You will see a series of tag definitions. Look through the file. Notice that there is a *Main Navigation* section, a *Recipes* section and a *Users* section. There are also sections related to the app's other models.

We will be talking about the *Recipes* and *Navigation* section in this tutorial.

Open up the `forms.dryml` and `cards.dryml` files and page through them. You will see similar structures. You will see a section describing *Recipes* and the other models we have built so far.

Now that you have familiarized yourself with the three Rapid auto-generated tag files, go back to the `pages.dryml` file.

3. Understanding the `pages.dryml` file. We are not going to explain every detail about what you see in `pages.dryml` at this point. In subsequent tutorials in this chapter, you will learn most of the key points. The goal in this tutorial is to get some familiarity with the tag structures and how Hobo uses and overrides them.

Focus in on the *Recipes* section. You will see four tag definitions: `<index-page>`, `<show-page>`, `<new-page>` and `<edit-page>`:

```
<!-- ===== Recipe Pages ===== -->
<def tag="index-page" for="Recipe"> . . .
</def>
<def tag="new-page" for="Recipe"> . . .
</def>
<def tag="show-page" for="Recipe"> . . .
</def>
<def tag="edit-page" for="Recipe"> . . .
</def>
```

The following table explains what each of these does. Rapid automatically creates this set of four tags for each model in your application.

Tag	Meaning	Calls	Controller Action	Route(URL)
<index-page>	renders a list of model records	Cards	index	*/model_name(plural)
<new-page>	renders a data entry for a new record	Forms	new	*/model_name/new
<show-page>	renders a single record	None	show	*/model_name/ID-record_name
<edit-page>Tags!	renders a data entry for an existing record	Forms	edit	*/model_name/edit/ID-record_name

Table 4: Hobo Page Action Tag Definitions

You cannot see it explicitly in the `pages.dryml` file, but the `<index-page>` tag calls the Recipe `<card>` tag. We will demonstrate this by editing them shortly. The `<new-page>` and `<edit-page>` tags call the Recipe `<form>` tags.

These auto-generated tags, each of the four tags above as well as the `<form>` and `<card>` tags, are built from tags defined in the Rapid library of tags. The four *page* tags are built from the Rapid `<page>` tag, the form tag from the Rapid `<form>` tag and the card tag from the Rapid `<card>` tag.

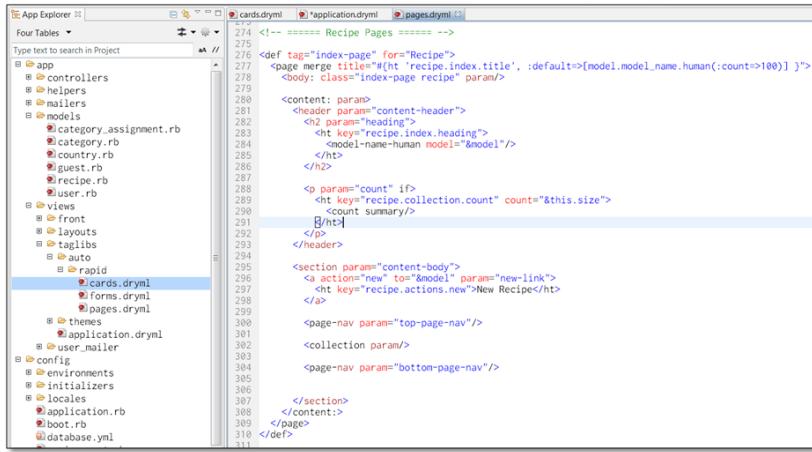
You might be confused at first because the auto-generated tags `<form>` and `<card>` have the same names as the Rapid auto-generated tags. What Hobo is really doing is redefining these tags and using the same tag name in the redefined tag.

There is a one-to-one association between these four tags and both controller actions and their associated routes. Routes are the URLs related to the web pages resulting from a particular controller action. Hobo automatically defines the routes, although they can be user-defined and customized too.

The controller action can be executed by navigating to the browser route URL listed in the figure at the top of this page.

Note: The asterisk (*) refers to the route URL for your app, which is usually `http://localhost:3000` for Ruby on Rails development setups.

4. **Edit the index page (method 1).** Open up the `pages.dryml` file and look at the `<index-page>` tag definition. Here is what it looks like:



```

<def tag="index-page" for="Recipe">
  <page merge title="#{ht 'recipe.index.title', :default=>[model.model_name.human(:count=>100)] }">
    <content param="param">
      <header param="content-header">
        <h2 param="key=recipe.index.heading">
          <model-name-human model="&model"/>
        </h2>
      </header>
      <param="count" if>
        <ht key="recipe.collection.count" count="#{&this.size}">
          <count summary>
            <ht>
              <param="count" if>
                <ht key="recipe.collection.count" count="#{&this.size}">
                  <count summary>
                    <ht>
                      <param="count" if>
                        <ht key="recipe.collection.count" count="#{&this.size}">
                          <count summary>
                            <ht>
                              <param="count" if>
                                <ht key="recipe.collection.count" count="#{&this.size}">
                                  <count summary>
                                    <ht>
                                      <param="count" if>
                                        <ht key="recipe.collection.count" count="#{&this.size}">
                                          <count summary>
                                            <ht>
                                              <param="count" if>
                                                <ht key="recipe.collection.count" count="#{&this.size}">
                                                  <count summary>
                                                    <ht>
                                                      <param="count" if>
                                                        <ht key="recipe.collection.count" count="#{&this.size}">
                                                          <count summary>
                                                            <ht>
                                                              <param="count" if>
                                                                <ht key="recipe.collection.count" count="#{&this.size}">
                                                                  <count summary>
                                                                    <ht>
                                                                      <param="count" if>
                                                                        <ht key="recipe.collection.count" count="#{&this.size}">
                                                                          <count summary>
                                                                            <ht>
                                                                              <param="count" if>
                                                                                <ht key="recipe.collection.count" count="#{&this.size}">
                                                                                  <count summary>
                                                                                    <ht>
                                                                                      <param="count" if>
                                                                                        <ht key="recipe.collection.count" count="#{&this.size}">
              </param>
            </param>
          </param>
        </param>
      </param>
    </content>
  </page>
</def>

```

Figure 118: The Hobo Rapid <index-page> tag definition in the pages.dryml file

You invoke the index action by clicking on a tab with a particular model name, which is *Recipes* in this example. Go ahead and click the Recipes tab to remind yourself where you left off in Tutorial 6 of Chapter 3:



Figure 119: The Recipes Index page

Note that the URL that generates the “Recipes Index” page, <http://localhost:3000/recipes>, has the form of an `index` action. (Refer to the Hobo Page Action Tag definitions figure earlier in this tutorial.) You can see three lines of text in the body of the tab beginning with the ‘Recipes’ title, then ‘There are 3 Recipes’, a ‘New Recipe’ hyperlink, and finally the list of recipes.

There are three levels of overriding. Hobo handles these by checking sequentially in

three directories for the tags or tag definitions it will use to render a view template.

The first place Hobo looks to find the information it needs to render a view template corresponding to a particular model is the /views directory corresponding to that model.

Note: Prior to Hobo 1.3, a view folder for each generated model was created. This is no longer the case. In the figure below, the categories, countries, and recipes folders were created manually.

In this case, note that /views/recipes is empty.

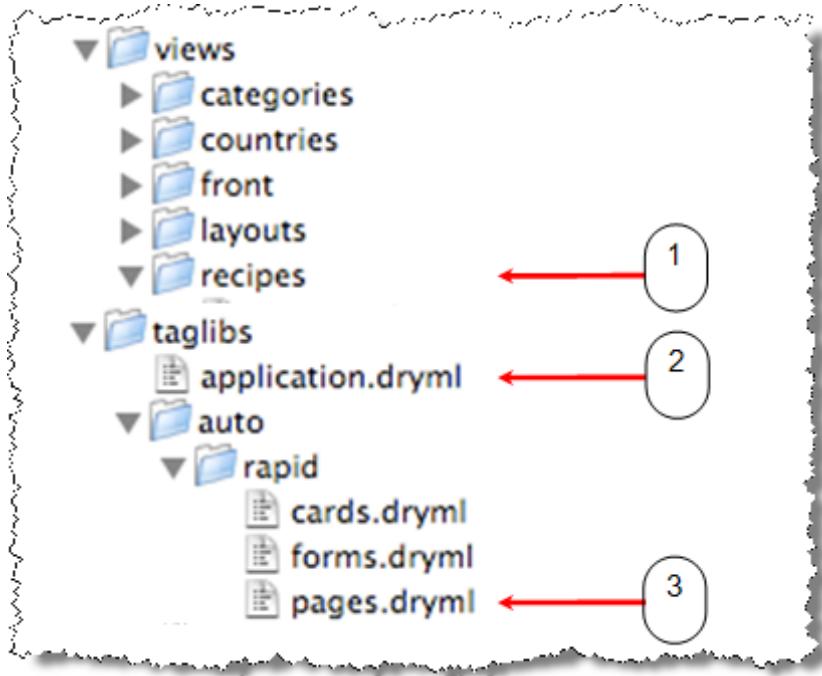


Figure 120: View of the taglibs/auto/rapid folder

The next place Hobo goes is the views/taglibs/application.dryml file. The last place Hobo goes is the views/taglibs/auto/rapid/pages.dryml file.

You are going to put the recipe index tag definition in application.dryml causing Hobo to use level 2. So take the code above from pages.dryml beginning with

```
<def tag="index-page" for="Recipe">
```

and paste it into /views/taglibs/application.dryml file. Paste it below the following code in views/taglibs/application.dryml file.

```

<include src="rapid" plugin="hobo"/>
<include src="taglibs/auto/rapid/cards"/>
<include src="taglibs/auto/rapid/pages"/>
<include src="taglibs/auto/rapid/forms"/>
<set-theme name="clean"/>

<def tag="index-page" for="Recipe">

    .
    .

```

The line in ***bold italics*** above is the first line from your copied code.

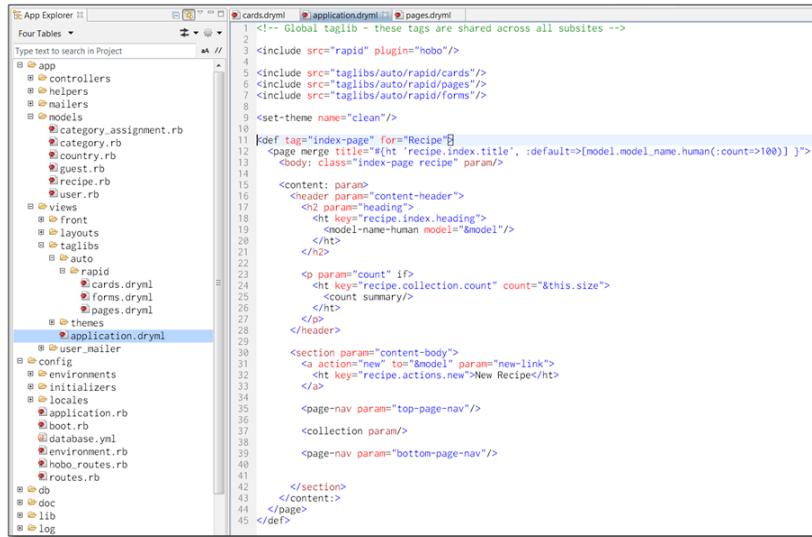


Figure 121: Adding the definition of index-page into the application.dryml file

Note: As you learn Hobo you might get confused between tag definitions and tags. This is often the case because Hobo does not need you to specifically invoke the tags that are defined in the Rapid files (pages.dryml, forms.dryml etc...) or in the application.dryml file. If the tags have the default names “index”, “new”, “show”, or “edit”, then Hobo creates the template on the fly. You do not have to put tag code in a template yourself unless you do not want to use Hobo’s default template.

First, refresh your browser to confirm that the UI has not changed. Simply copying a tag definition from pages.dryml to application.dryml with no changes to the tag definition should not change the page rendering. It is a good idea to double check in

case you copied something wrong so you won't confuse a copy mistake with a coding mistake.

Let's make a minor change to convince you that this is what is happening. Note that the line in *red italics* below is what has changed.

```
<def tag="index-page" for="Recipe">
    <page merge title="#{ht 'recipe.index.title',
:default=>[model.model_name.human(:count=>100)] }">
        <body: class="index-page recipe" param/>
        <content: param>
            <header param="content-header">
                <h2 param="heading">
                    My Recipes
                </h2>
                <p param="count" if>
                    <ht key="recipe.collection.count"
                        count="&this.size">
                        <count summary/>
                    </ht>
                </p>
            </header>
            <section param="content-body">
                <a action="new" to="#{model}" param="new-link">
                    <ht key="recipe.actions.new">New Recipe</ht>
                </a>
                <page-nav param="top-page-nav"/>
                <collection param/>
                <page-nav param="bottom-page-nav"/>
            </section>
        </content:>
    </page>
</def>
```

Refresh your browser and you will see that Hobo has changed the template it generated dynamically:



Figure 122: Page view of "My Recipes" after modifying the <index-page> tag

You should see that the first line of the page has changed from “Recipes” to “My Recipes”.

Here is what happened:

- Step 1: Hobo looked for a template in the `views/recipes/` directory called `index.dryml`.
- Step 2: Since `views/recipes/index.dryml` did not exist, Hobo next looked in `views/taglib/application.dryml` where it found the tag definition for the index page.
- Step 3: Hobo used this tag definition to generate the contents of the “index” page.

5. Change the index page(method 2). If you want to change the index page directly, you can create a new file in the `views/recipes` directory called `index.dryml`.

We haven’t given you enough information for you to build your own `index.dryml` template using Hobo’s tag library yet. We said above that Hobo will look there first for a page to render when the index action is invoked.

So if you place an empty file here, you get a blank page rendered. Go ahead and create a file called `index.dryml` in the `views/recipes` directory. Confirm for yourself that you get a blank page.

Now let’s do something a little more useful. Add the single line of code below to the `index.dryml` file:

```
<index-page/>
```

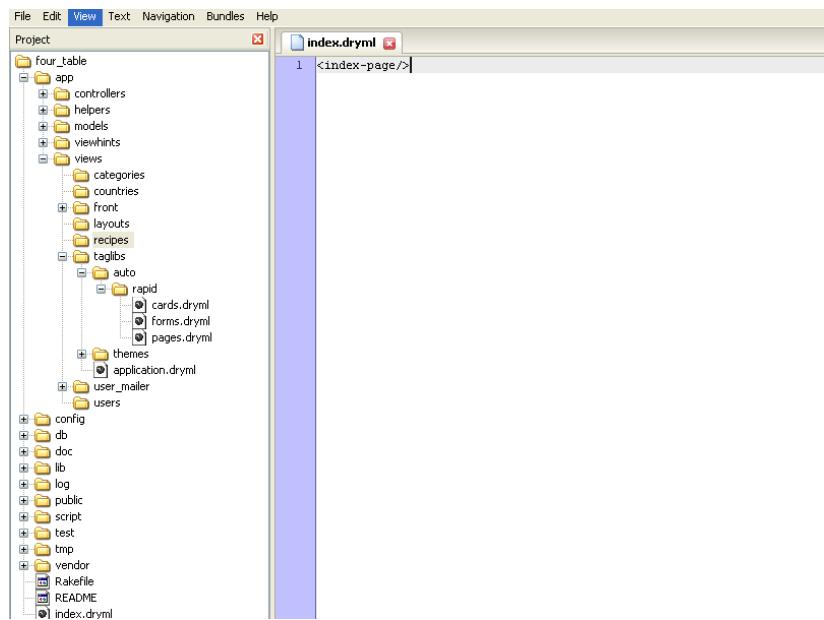


Figure 123: Adding the <index-page/> tag to index.dryml

Note: The Hobo tag syntax is just like you would expect from HTML or XML. The code <index-page/> is equivalent to <index-page></index-page>. Watch your placement of “/”. It was our most frequent error when we started with DRYML.

Now refresh your browser and you will see the same page rendered as in Step 4. What has happened is that Hobo has checked in the `views/recipes` directory for a file called `index.dryml`, found one and rendered it. When it encountered the `<index-page/>` tag, it first checked in `index.dryml` for a tag definition. Not finding one there, it checked in `application.dryml` where it found one to use in rendering the `<index-page/>` tag in `index.dryml`. If it had not found a tag definition in `application.dryml`, Hobo would have gone back to `pages.dryml` for the default `<index-page>` definition.

Note: You can put a tag definition in either a view template file or in `application.dryml` but Hobo will ignore tags in `application.dryml`. The `application.dryml` file is for tag definitions only.

6. **Edit an individual record's view in the index page.** By now, you should have entered a couple of recipes. Be sure to do that if you have not.

In Table 1 above, we indicated that the <index-page> tag calls <card> tags to render individual records. We can demonstrate this process by changing a <card> tag. Go to the cards.dryml file in the rapid directory and copy the <card> definition for recipe cards into the application.dryml file below the <index-page> definition. Hobo will now use this version of the <card> tag when it uses the <index-page>.

```
<def tag="card" for="Recipe">
  <card class="recipe" param="default" merge>
    <header: param>
      <h4 param="heading"><a><name/></a></h4>
    </header:>
    <body: param>
      <ht key="category.collection.count"
          count="&this.categories.size">
        <count:categories param/>
      </ht>
    </body:>
  </card>
</def>
```

Again, we will not explain the detailed syntax of this tag yet. Let's just make a simple change (in *red italics* below) to demonstrate how Hobo works:

```
<def tag="card" for="Recipe">
  <card class="recipe" param="default" merge>
    <header: param>
      <h4 param="heading"><a><name/></a>... test</h4>
    </header:>
    <body: param>
      <ht key="category.collection.count"
          count="&this.categories.size">
        <count:categories param/>
      </ht>
    </body:>
  </card>
</def>
```

Now refresh your browser. Click the ‘Recipes’ tab to invoke the index action using the <index-page> tag.



Figure 124: How a change to the <index-page> tag affects a collection

You see how each record displayed has been changed. You didn't need to iterate through a loop. Iterating through all records in a collection is built in to Hobo's tag processing. If you look back to Step 4 to see the <index-page> tag definition, you will see the following line:

```
<collection param/>
```

It is here that the <card> tag is called. The <collection> tag refers to a collection of records from a data model.

Now click on one of the recipe name hyperlinks, which will invoke the <show-page> tag in `pages.dryml`. Since you haven't changed this tag and since it does not use the <card> tag, you will NOT see '....test' appended to recipe names as you do when Hobo lists recipes using the <index-page> tag.

To finish up this step, remove the text '....test' to keep things looking nice.

6. Editing a form. To modify a form, you can do something similar to editing the <card> tag above. In this case, the relevant page tag is the <new-page> tag in `pages.dryml`. It calls the <form> tag. You can see that in the `forms.dryml` file.

7. Editing navigation tabs and their order. As you have seen, Hobo provides a predefined tab-based user interface. By default, it arranges the tabs alphabetically by model. This is probably not what you want. You more than likely want to set up an order that makes sense for your application.

This is readily done. Find the <main-nav> tag definition in the `pages.dryml` file and copy it into `application.dryml` right after the <app-name> tag definition.

```
<def tag="main-nav">
    <navigation class="main-nav" merge-attrs param="default">
        <nav-item href="#{base_url}"/>Home</nav-item>
        <nav-item with="&Category"><ht key="category.nav_item"
count="100"><model-name-human count="100"/></ht></nav-item>
        <nav-item with="&Country"><ht key="country.nav_item"
count="100"><model-name-human count="100"/></ht></nav-item>
        <nav-item with="&Recipe"><ht key="recipe.nav_item"
count="100"><model-name-human count="100"/></ht></nav-item>
    </navigation>
</def>
```

Now let's change the order of the tabs in your UI. Change the order of your tabs by moving the Recipes tab up to the position noted below in *red italics*.

```
<def tag="main-nav">
    <navigation class="main-nav" merge-attrs param="default">
        <nav-item href="#{base_url}"/>Home</nav-item>
        <nav-item with="&Recipe"><ht key="recipe.nav_item"
count="100"><model-name-human count="100"/></ht></nav-item>
        <nav-item with="&Category"><ht key="category.nav_item"
count="100"><model-name-human count="100"/></ht></nav-item>
        <nav-item with="&Country"><ht key="country.nav_item"
count="100"><model-name-human count="100"/></ht></nav-item>
    </navigation>
</def>
```

Now refresh your browser and you will see the new tab order:



Figure 125: Changing the tab order for the main navigation menus

8. Editing an application name. If you want to change the name of the application that

appears on all the UI web pages, you can do this easily also. The `config.hobo.app_name` key is found in the `config/application.rb` file and can be edited to change the application name. Make the following change, then restart the application, and then refresh your browser:

```
config.hobo.app_name = "Four Tables, No Waiting"
```



Figure 126: Changing the application name with the app-name tag

9. Summary. The Hobo Rapid generator creates tag definitions and places them in the files of the Rapid directory. The programmer overrides, redefines, and defines new tags in `application.dryml`. These definitions are available throughout the application. So far, you have just learned how to override tags.

There are no tag calls in `application.dryml` except within a tag definition because `application.dryml` is NOT a template file (see it as a library file). The programmer invokes—that is—calls tags in template files placed in the `view/<model_name>` directories.

The programmer may also override, redefine, or define a new tag within a template, but this modification is local (e.g., only available within that template).

Note: A new feature of Hobo 1.3 is that application tag definitions can be organized into multiple dryml files as long as they reside in the `app/views/application` directory.

So instead of having one large `application.dryml` file, you can organize your application specific tag definitions into multiple files and place them in the `app/views/application` directory.

Tutorial 10

DRYML I: A First Look at DRYML

You will be introduced to the concept of a user-defined tag, called a DRYML tag. The tutorial shows you how to make minor changes to the home page template by defining DRYML tags. You will also learn how to parameterize tags with the DRYML parameter attribute, `param`.

Note: Be sure not to confuse the DRYML `param` with the Rails `params` object.

Topics

- Define a DRYML tag in the front/index.dryml template
- Call the DRYML tag in the front/index.dryml template
- Add a parameter to the DRYML tag
- Add an attribute to the DRYML tag
- Tutorial Application: four_table

Steps

1. **Define a tag.** Open up the `views/front/index.dryml` file of the `four_table` application. This is Hobo's home page.

At the top of the file enter the following code. The `<def>` tag below is Hobo's DRYML tag for defining a custom tag. The code below defines a `<messages>` tag.

```
<def tag="messages">
<br/><br/>
<ul>
<li >Message 1</li>
<li >Message 2</li>
<li >Message 3</li>
</ul>
</def>
```

The entire markup between the `<def>` tags is standard HTML. When called, this `<messages>` tag will emit a three-line list.

2. **Call the tag.** Go to the line that reads:

```
<h3>Congratulations! Your Hobo Rails App is up and  

running</h3>
```

Add a line after this one so that it reads:

```
<h3>Congratulations! Your Hobo Rails App is up and  

running</h3>  

<messages/>
```

Note: The correct syntax is to place the forward slash after the tag name when you use the tag as a single tag rather than in the form of an opening and closing tag with no content in between.

```

<page title="Home">
  <body: class="front-page"/>
  <content:>
    <header class="content-header">
      <h1>Welcome to <app-name></h1>
      <section class="welcome-message">
        <h3>Congratulations! Your Hobo Rails App is up and running</h3>
        <messages/>
      </ul>
    </header>
    <section class="content-body">
      <li>To customise this page: edit app/views/front/index.dryml</li>
    </section>
  </content:>
</page>

```

Figure 127: The \views\front\index.dryml file after the first modification

Then refresh your browser: To display the home page.

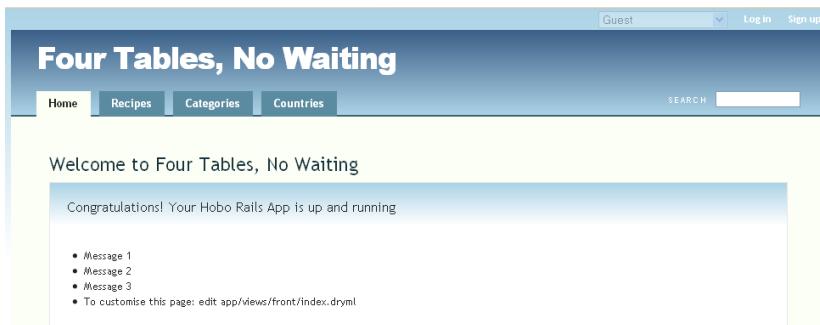


Figure 128: The Home page with the first set of custom messages

One of the things that is different from Tutorial 1, is that you are now working both with a DRYML tag definition and with a DRYML tag. In the previous tutorial, you edited the tag definitions, but you did not explicitly invoke a tag such as <index-page>

Hobo took care of invoking the tags for you on-the-fly. Since Hobo's Rapid component knows what the basic structure of a data driven web page is, it does not require you to explicitly code the template explicitly except when you want something different than the Hobo default.

In this tutorial you will be defining new tags unknown to Hobo, so you of course must explicitly invoke them.

3. Parameterize the tag. Change the following code in the <messages> tag definition from:

```
<li>Message 1</li>
<li>Message 2</li>
<li>Message 3</li>
```

to:

```
<li param="msg1">Message 1</li>
<li param="msg2">Message 2</li>
<li param="msg3">Message 3</li>
```

You have now created three parameters, which can be invoked in the following way:

```
<msg1:>message text</msg1:>
```

<msg1:> is called a *parameter tag*.

Note: The colon (:) suffix indicates that the tag is a defined parameter tag. Later you will learn that some parameter tags are defined for you in the Rapid library.

4. Use a parameter. Let's invoke the `<messages>` tag, but change the third message by addressing the `<msg3:>` parameter tag.

```
<h3>Congratulations! Your Hobo Rails App is up and  
running</h3>  
<messages>  
<msg3:>This is the third message passed as a  
parameter.</msg3>  
</messages>
```

The first two lines will remain the same while the third changes due to the use of the `<msg3:>` parameter tag. You have used a tag to pass data from the `<msg3:>` parameter tag into the `<messages>` tag.

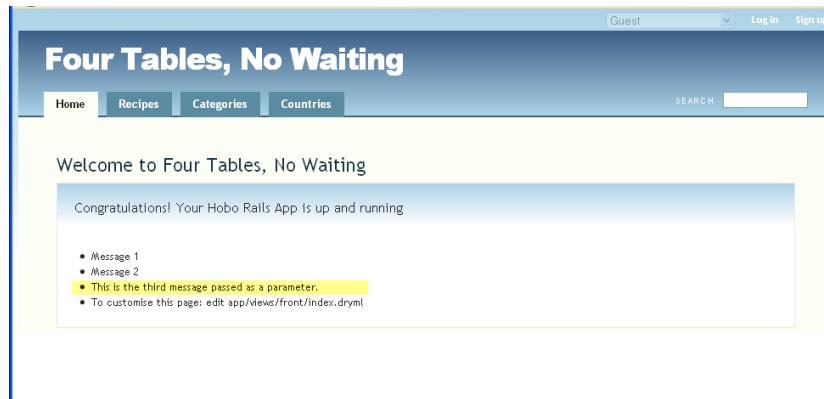


Figure 129: How the passed parameter displays on the page

5. Use some more parameters. Change the other two message lines likewise to:

```
<messages>  
<msg1:>This is the first message called as a  
parameter.</msg1>  
<msg2:>This is the second message called as a  
parameter.</msg2>  
<msg3:>This is the third message called as a  
parameter.</msg3>  
</messages>
```

and you should see:

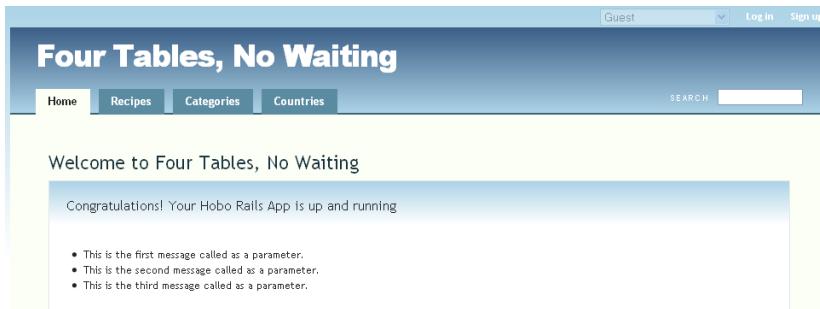


Figure 130: Passing three parameters to you <messages> tag

6. **Reverse the order of the parameter call.** Now, try the following code.

```
<messages>
<msg2:>This is the second message.</msg2:>
<msg1:>This is the first message.</msg1:>
<msg3:>This is the third message.</msg3:>
</messages>
```

You will see that this edit will not change the order of the list because the order is defined by the tag definition not by its call. The tag calls the messages in the order set in the tag definition, namely <msg1:>, then <msg2:> and then <msg3:>.

7. **Create an html-like tag using param = "default".** In the preceding steps, you learned how to reach into a tag with three parameter tags and change the default message text of the defined <messages> tag. In this step you will emulate a regular HTML formatting tag using the param="default" attribute.

Note: We have referred to an attribute above rather than a parameter because a change will be made by setting param to a value rather than by using a parameter tag.

Go back to the top of the views/front/index.dryml file and enter the following code after the first <def> . . . </def> tags.

```
<def tag="bd-it">
<br/>
<b><i><span param>stuff</span></i></b>
</def>
```

Here we have redefined the HTML `` tag to format the tag content with bold AND italic formatting. Since the `` tag is now parameterized, you can now replace the ‘stuff’ continent with something you might want to format.

Call the `<bd-it>` tag right after the closing `</messages>` tag without using the `<span:>` parameter. This will demonstrate that the tag will just emit the formatted default word **stuff**.

```
<messages>
<msg2:>This is the second message.</msg2:>
<msg1:>This is the first message.</msg1:>
<msg3:>This is the third message.</msg3:>
</messages>
<bd-it/>
```

If you use the `<span:>` parameter tag, you will format your content.

```
<bd-it/>
<bd-it><span:>More stuff</span:></bd-it>
```

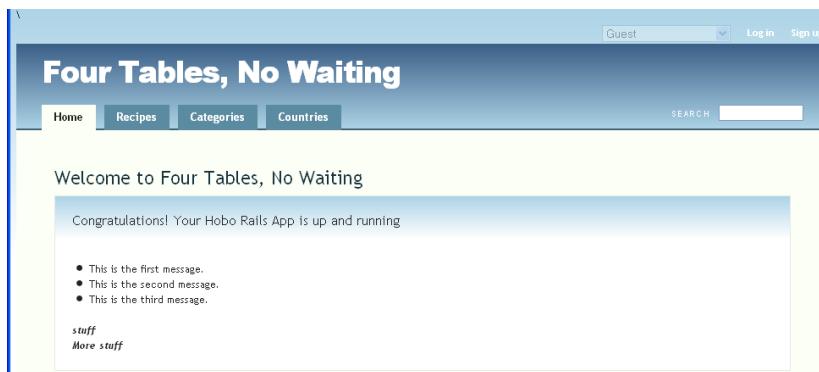


Figure 131: Calling `<span:>` explicitly within to your `<bd-it>` tag

However, the second line is a clumsy way to get: “**More stuff**”. Instead, change your `<def>` code to:

```
<def tag="bd-it">
<br/>
<b><i><span param="default">stuff</span></i></b>
</def>
```

What the `param="default"` text is saying is that the `<span:>` parameter is automatically assumed when you call the `<bd-it>` tag. You do not have to explicitly name it. Change your call to:

```
<bd-it/>
<bd-it>More Stuff</bd-it>
```

So now you have created a DRYML tag that looks just like an HTML tag.

Note: Once you change the `<span:>` parameter to the default parameter, Hobo will ignore explicit uses of it Hobo will only emit the default content if you name it explicitly. Once you use the default parameter attribute you are committed to the more compact notation. There can only be one “default” parameter in a tag definition.

The entire `/views/front/index.dryml` contents at the end of this tutorial is as follows:

```
<def tag="messages">
<br/><br/>
<ul>
<li param="msg1">Message 1</li>
<li param="msg2">Message 2</li>
<li param="msg3">Message 3</li>
</ul>
</def>
<def tag="bd-it">
<br/>
<b><i><span param="default">stuff</span></i></b>
</def>
<page title="Home">
<body: class="front-page"/>
<content:>
<header class="content-header">
<h1>Welcome to <app-name/></h1>
<section class="welcome-message">
<h3>Congratulations! Your Hobo Rails
App is up and running</h3>
<messages>
<msg2:>This is the seond message.</msg2>
<msg1:>This is the first messsage.</msg1>
<msg3:>This is the third message passed as a
parameter.</msg3>
</messages>
<bd-it/>
<bd-it>More stuff</bd-it>
</section>
</header>
<section class="content-body">
</section>
</content:>
</page>
```

Tutorial 11

DRYML II: Creating Tags from Tags

Going to the next step in your understanding of DRYML, you will learn how to define tags from other tags. Specifically, you will learn how to create new tags that inherit parameters from the tags they are based on.

Tutorial Application: `four_table`

Topics

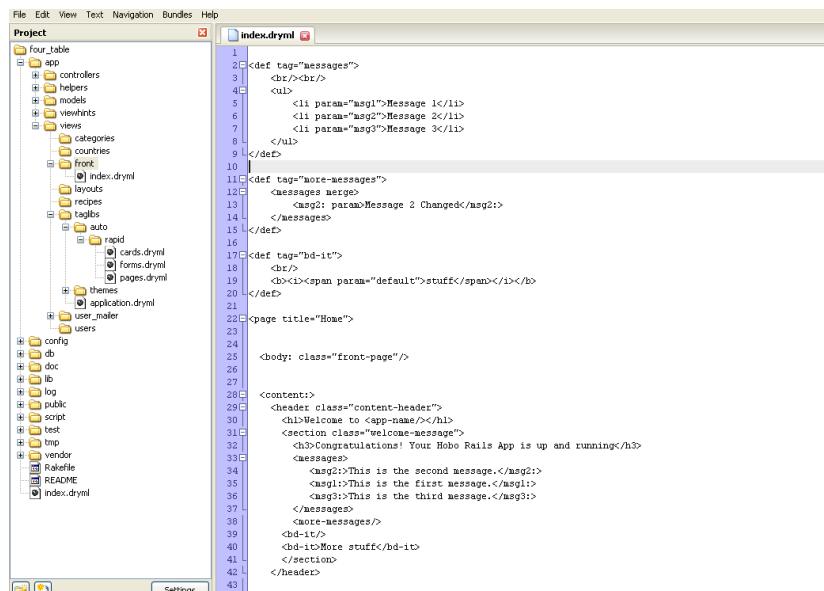
- Defining tags from tags using the `merge` attribute
- Defining tags from tags using the `extend` tag
- Replacing tag parameters (not tag content)

1. Define a tag based on another tag: In Tutorial 10, you learned how to define a tag called `<messages>` that output three lines of HTML. Now you will define a new tag based on `<messages>` called `<more-messages>`. Place the following code below the `<messages>` tag definition. (The order of tag definitions does not matter. This was just a recommendation for neatness.)

```
<def tag="more-messages">
<messages merge>
<msg2: param>Message 2 Changed</msg2:>
</messages>
</def>
```

What you have done here is to edit the `<msg2:>` parameter tag of the `<messages>` tag so that it has different default content. By using the `merge` attribute, you have told Hobo to use everything from the `<messages>` tag except for the change. Let's invoke this tag. Place the following code below your last code from the previous tutorial.

```
<more-messages/>
```



```

File Edit View Text Navigation Bundles Help
Project index.dryml
four_table
  app
    controllers
    helpers
    models
    viewhists
    views
      categories
        countries
      front
        index.dryml
      layouts
      recipes
      reports
      auto
        cards
        forms
        pages
      themes
      application.dryml
    user_mailer
    users
  config
  db
  doc
  lib
  log
  public
  script
  test
  tmp
  vendor
  .bundle
  README
  index.dryml

1<def tag="messages">
2  <br/><br/>
3  <ul>
4    <li param="msg1">Message 1</li>
5    <li param="msg2">Message 2</li>
6    <li param="msg3">Message 3</li>
7  </ul>
8</def>
9
10<def tag="more-messages">
11  <messages merge>
12    <msg2: param>Message 2 Changed</msg2:>
13    <messages>
14    </messages>
15</def>
16
17<def tag="bd-it">
18  <br/>
19  <b><i><span param="default">stuff</span></i></b>
20</def>
21
22<page title="Home">
23
24  <body class="front-page">
25
26
27
28<content>
29  <header class="content-header">
30    <h1>Welcome to <app-name/></h1>
31    <section class="welcome-message">
32      <h3>Congratulations! Your Hobo Rails App is up and running</h3>
33      <messages>
34        <msg1>This is the second message.</msg2:>
35        <msg1>This is the first message.</msg1:>
36        <msg3>This is the third message.</msg3:>
37      </messages>
38      <more-messages/>
39    </bd-it/>
40    <bd-it>More stuff</bd-it>
41  </section>
42</header>
43

```

Figure 132: Adding the custom <more-messages> tag to front\index.dryml

Refresh your browser to see the change the below.

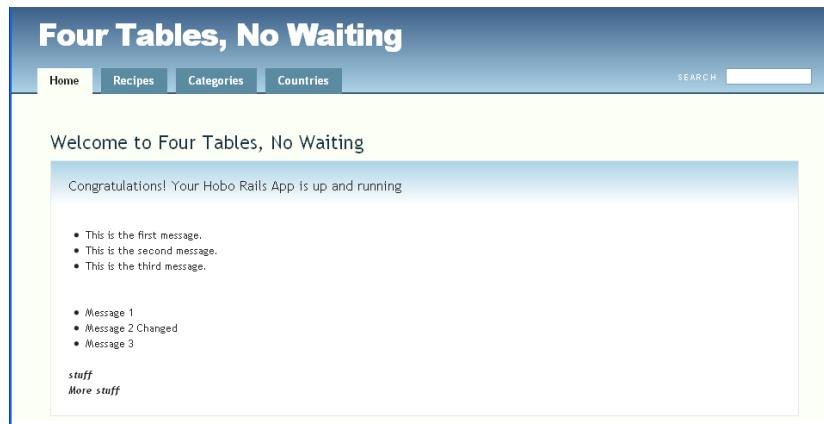


Figure 133: Page rendering with <more-messages>

Note: Later in this Chapter you will also learn how to add attributes to tags in addition to parameters. Merge means merge parameters AND attributes.

Remember that the text, ‘Message 1’ and ‘Message 3’ is the default text from the <messages> tag.

2. Define a tag based on another tag: “Extending”. In the last example, you learned how to define a new tag based on an old tag. The new tag is defined with a new name, <more-messages>. You cannot use the merge method to define a tag from a tag without changing the name.

Go ahead and change <more-messages> to <messages> to convince yourself that you will get an error.

However, Hobo does have a way of preserving tag names while creating tags from tags. It is called extending a tag. It works basically the same way as merging tags, except it uses the <extend> tag instead of the <def> tag to define the new tag.

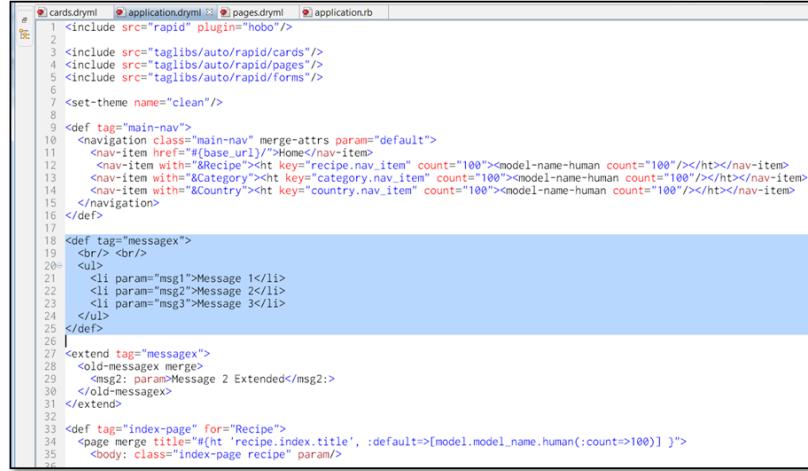
To create an extended tag we will begin by creating a new tag called <messagex> and then extend it using the same name.

```
<def tag="messagex">
<br/> <br/>
    <ul>
        <li param="msg1">Message 1</li>
        <li param="msg2">Message 2</li>
        <li param="msg3">Message 3</li>
    </ul>
</def>
<extend tag="messagex">
    <old-messagex merge>
        <msg2: param>Message 2 Extended</msg2:>
    </old-messagex>
</extend>
```

Instead of placing the code above in front\index.dryml, you need to put it in views/taglibs/application.dryml. Recall this will make the tag definition available throughout your application. There is another reason for putting it here. You cannot use the <extend> tag in a view template, you can only use it within application.dryml.

Note: To extend this tag and have the original one still available, you can use the Hobo “alias-of” parameter:

```
<def tag="new-messagex" alias-of="messagex"/>
And then extend “new-messagex” with the functionality you need.
```

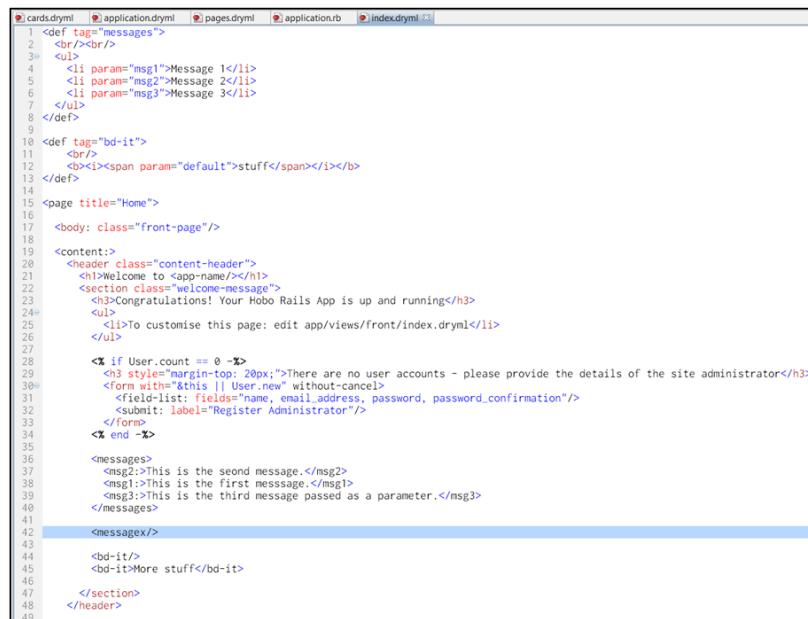


```

1 <include src="rapid" plugin="hobo"/>
2
3 <include src="taglibs/auto/rapid/cards"/>
4 <include src="taglibs/auto/rapid/pages"/>
5 <include src="taglibs/auto/rapid/forms"/>
6
7 <set-theme name="clean"/>
8
9 <def tag="main-nav">
10 <navigation class="main-nav" merge-attrs param="default">
11   <nav-item href="#{base_url}/">>Home</nav-item>
12   <nav-item with="&recipe"><ht key="recipe.nav_item" count="100"><model-name-human count="100"/></ht></nav-item>
13   <nav-item with="&Category"><ht key="category.nav_item" count="100"><model-name-human count="100"/></ht></nav-item>
14   <nav-item with="&Country"><ht key="country.nav_item" count="100"><model-name-human count="100"/></ht></nav-item>
15 </navigation>
16 </def>
17
18 <def tag="messagex">
19   <br/> <br/>
20   <ul>
21     <li param="msg1">Message 1</li>
22     <li param="msg2">Message 2</li>
23     <li param="msg3">Message 3</li>
24   </ul>
25 </def>
26
27 <extend tag="messagex">
28   <old-messagex merge>
29     <msg2: param>Message 2 Extended</msg2:>
30   </old-messagex>
31 </extend>
32
33 <def tag="index-page" for="Recipe">
34   <page merge title="#{ht.recipe.index.title}, :default=[model.model_name.human(:count=>100)] ">
35     <body: class="index-page recipe" param/>
36

```

Figure 134: Extending the tag <messagex> in application.dryml



```

1 <def tag="messages">
2   <br/><br/>
3   <ul>
4     <li param="msg1">Message 1</li>
5     <li param="msg2">Message 2</li>
6     <li param="msg3">Message 3</li>
7   </ul>
8 </def>
9
10 <def tag="bd-it">
11   <br/>
12   <b><i><span param="default">stuff</span></i></b>
13 </def>
14
15 <page title="Home">
16
17   <body: class="front-page"/>
18
19   <content:>
20     <header class="content-header">
21       <h1>Welcome to <app-name/></h1>
22       <section class="welcome-message">
23         <h3>Congratulations! Your Hobo Rails App is up and running</h3>
24         <ul>
25           <li>To customise this page: edit app/views/front/index.dryml</li>
26         </ul>
27
28         <% if User.count == 0 -%>
29           <h3 style="margin-top: 20px;">There are no user accounts - please provide the details of the site administrator</h3>
30           <form with="&this || User.new" without="cancel">
31             <field-list: fields="name, email_address, password, password_confirmation"/>
32             <submit: label="Register Administrator"/>
33           </form>
34         <% end -%>
35
36         <message>
37           <msg2>This is the second message.</msg2>
38           <msg1>This is the first message.</msg1>
39           <msg3>This is the third message passed as a parameter.</msg3>
40         </message>
41
42         <messagex/>
43
44         <bd-it/>
45         <bd-it>More stuff</bd-it>
46
47       </section>
48     </header>

```

Figure 135: Using the extended <messagex> tag

Before trying this out, you should delete (or comment out) the code for <more-messages>, so you will not get confused.

In the code example above, we created a new tag <messagex> just like the old <messages> tag. We then extended it so that it would look just like the <more-messages> tag from

Step 1.

Call the `<messagex>` tag in `front/index.dryml` to confirm that it yields output like the `<more-messages>` tag.

```
<messagex/>
```

You should see the following rendering:

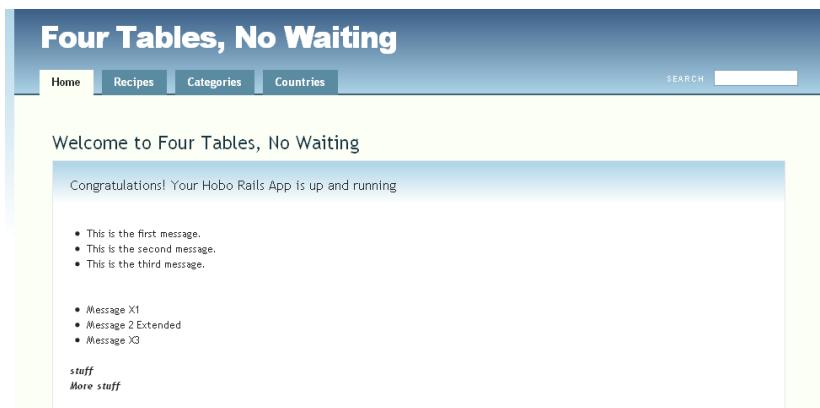


Figure 136: Page view of the next additions to `<messagex>`

3. **Edit the merged tag in more ways.** Let's modify our `<more-messages>` tag of Step 1, which is defined in `front/index.dryml`. Remove or comment out the `<messagex>` tag so you won't get confused.

We are going to show you now that DRYML can do lots of things within the same tag definition with ease. Add a new parameter tag before the `merge` line to demonstrate that you do not have to have the `merge` line right after your `<def>` line.

You can put both parameter tags and non-parameter HTML after `merge` markup. Let's do this in two steps.

Edit your `<more-messages>` tag to look like the following:

```
<def tag="more-messages">
  <br/><br/>
  <li param="msg0">Message 0</li>
  <messages merge>
    <msg2: param>Message 2 changed in merge.</msg2:>
  </messages>
</def>
```

Make sure you call your `<more-messages>` tag and refresh your browser.

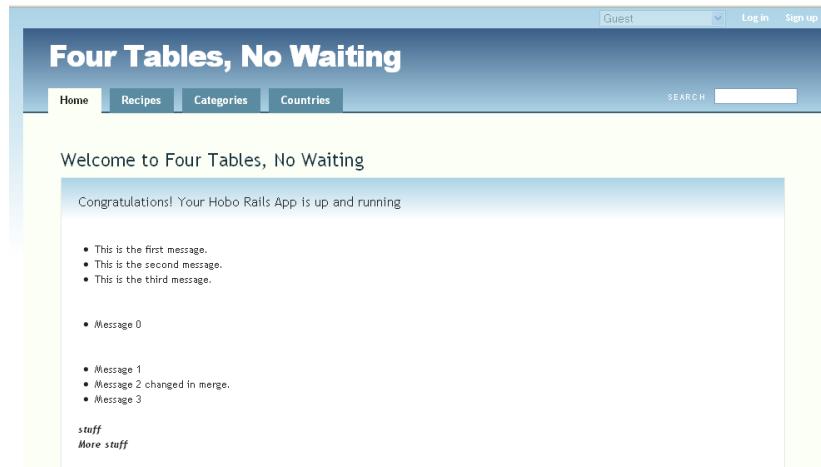


Figure 137: Page view of the <more-messages> tag usage

Let's demonstrate that <msg0:> is a real parameter tag with the following code where you call the <more-messages> tag.

```
<more-messages>
  <msg0:> Message 0 changed with parameter tag.</msg0:>
</more-messages>
```

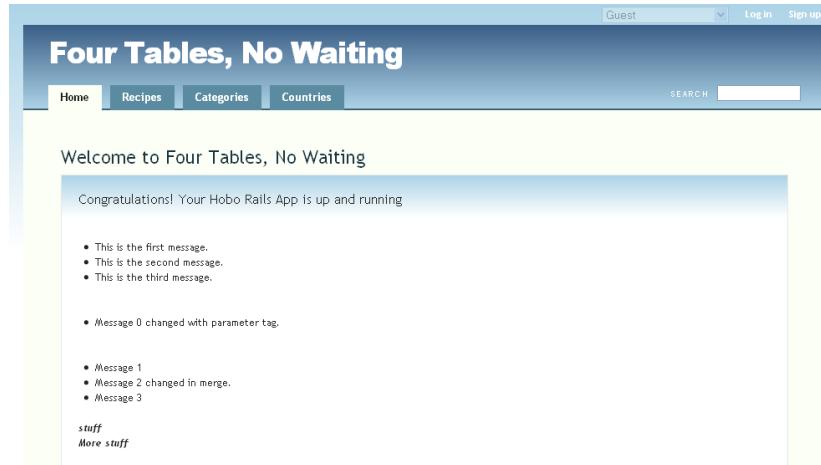


Figure 138: Page view of overriding the default message 0.

We have chosen this exercise to remind you that you have changed the text in two ways.

- You changed the third block of messages by changing the tag definition within a `merge`.
- You changed the second block (Message 0) by calling a parameter tag within a `tag`.

Edit the `<more-messages>` definition after the merge is closed with `</messages>`. We have added two lines of DRYML. The first is a parameter tag, `<msg4:>`. The second is pure HTML without any parameterization.

```
<def tag="more-messages">
    <li param="msg0">Message 0</li>
    <messages merge>
        <msg2: param>Message 2 changed in merge.</msg2:>
    </messages>
    <li param="msg4">Message 4</li>
    <li>No Parameter Here</li>
</def>
```

Invoke `<more-messages>` and change the default content of the `<msg4:>` parameter tag.

```
<more-messages>
    <msg0:> Message 0 changed with parameter tag.</msg0:>
    msg4:> Message 4 has changed with parameter tag
        too.</msg4:>
</more-messages>
```

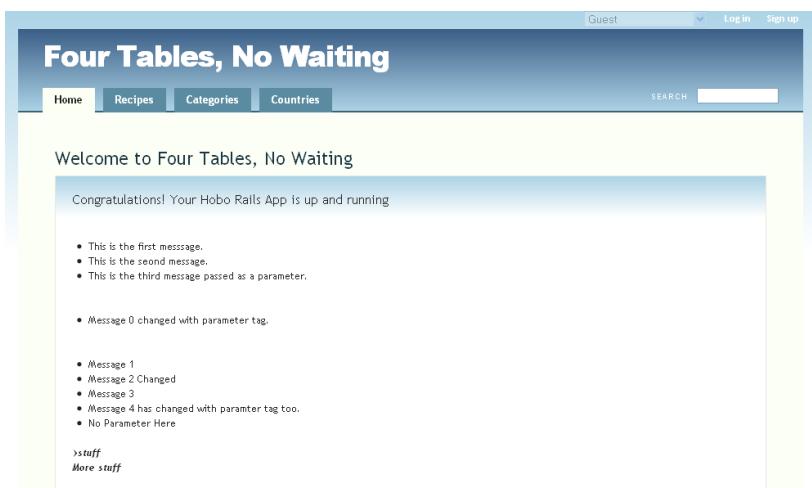


Figure 139: More parameter magic

Tutorial 12

Rapid, DRYML and Collections

Learn how to: create a new index page that will replace the default index page that Hobo generates on the fly, learn how to display data on this index page that is related through a many-to-many relationship.

Tutorial Application: `four_table`

Topics

- Learn how to create your own index template in a `view/model` directory.
- Work on using the `application.dryml` directory to override auto-generated tags.
- Learn about the Rapid `<collection>` tag.
- Get introduced to the Rapid `<a>` tag.
- Learn how to use the `<repeat>`, `<if>` and `<else>` tags.

Steps

1. **Click the model(*Recipes*) tab.** Load your browser again with the Four Table application we ended up with in Tutorial 11. Click the *Recipes* tag to remind yourself how Hobo automatically creates a list of your recipes. This is different than the *Home* tab you were working with in Tutorial 11. When you click the *Recipes* tab, Hobo goes through the three-step check you learned about in Tutorial 1 to locate a template or template definition.

Since we have already moved the `<index-page>` tag for recipes to `\taglibs\application.dryml`, Hobo will obtain its tag definition for generation of a view template here.

Note: You learned back in Tutorial 1 that each of Hobo's tabs, named with the plural of the model name by default, invoke the index action and list the records in the model.

Since there is not a file called `views\recipes\index.dryml`, Hobo will create its own template on the fly from the `<index-page>` tag definition in `\taglibs\application.dryml`. (We created a `views\recipes\index.dryml` in Step 1 but we asked you to remove it. If you did not do that, do it now so you do not have any conflicts as we proceed).



Figure 140: The Four Tables application as we left it

2. **Create a new template file.** Now, create the new file called `index.dryml` in the `views/recipes` folder. This is the folder automatically created when you did the `hobo g` resource generation in Tutorial 1. This file is called a DRYML template.

Note: We have used the word template quite frequently now but it is still worth reminding you not to be confused by it. It is a file used to render a specific web page, not a framework for creating one as the word may imply.

Now that this file exists, Hobo will use it when it finds it, so let's put a tag in it to make sure Hobo has a template to render.

```
<index-page/>
```

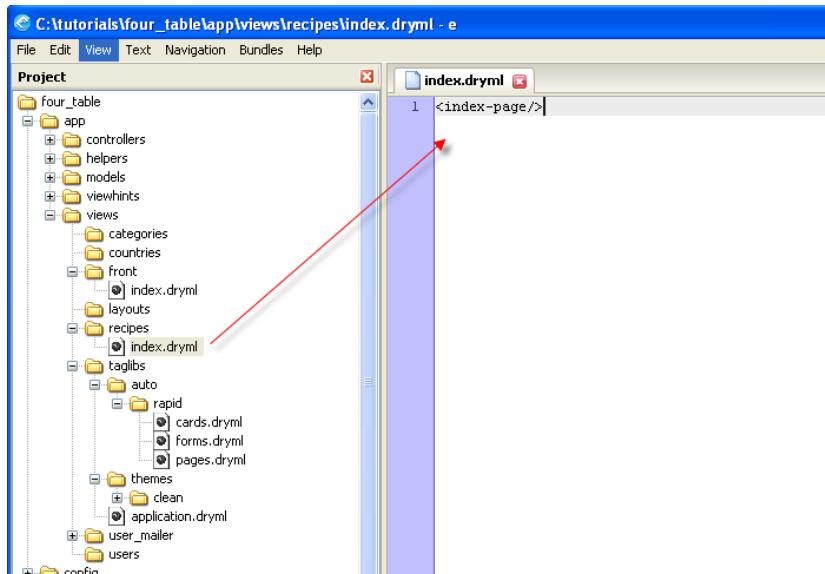


Figure 141: Creating the /views/recipes/index.dryml file

Refresh your browser. It should look just like it did in Step 1. This is because <index-page> is the exact tag that Hobo is calling to display this page. Instead of doing it automatically, you have added one step. Before, since there was no file in views\recipes, Hobo created its own version of the page using this tag. Now Hobo looks in the folder, finds the index.dryml file and does what it would have done anyways, namely use the <index-page> tag.

3. **Work with the <collection> tag.** From here on in this tutorial we will be moving back and forth between the template views/recipes/index.dryml and the <index-page> definition in views>taglibs\application.dryml. Keep this in mind so you do not get confused.

Go to the application.dryml and find the <index-page> tag definition for the *Recipe* model. Note the <collection> tag in italics and red below.

```
<def tag="index-page" for="Recipe">
    <page merge title="#{ht 'recipe.index.title',
:default=>[model.model_name.human(:count=>100)] }">
        <body: class="index-page recipe" param/>
        <content: param>
            <header param="content-header">
                <h2 param="heading">
                    My Recipes
                </h2>
                <p param="count" if>
                    <ht key="recipe.collection.count"
                        count="&this.size">
                        <count summary/>
                    </ht>
                </p>
            </header>
            <section param="content-body">
                <a action="new" to="&model" param="new-link">
                    <ht key="recipe.actions.new">New Recipe</ht>
                </a>
                <page-nav param="top-page-nav"/>
                <collection param/>
                <page-nav param="bottom-page-nav"/>
            </section>
        </content:>
    </page>
</def>
```

To remind yourself that this is the tag responsible for listing the recipe records, delete it and refresh your browser. You will still see a template rendered but without the list of recipes. OK, now let's put back the `<collection>` tag so that your file still reads like the above code.

Move back to the `views/recipes/index.dryml` template and explicitly call the collection tag. Change your code to read like this:

```
<index-page>
    <collection:/>
</index-page>
```

Your Recipes template should still look exactly like the one in Step 1.

You are now calling the `<collection>` tag. Notice the trailing colon (:). This colon is here because you are calling a parameter tag. You can see above that the `<collection>` tag was parameterized in `application.dryml` by adding the `param` attribute to the declaration. You might be wondering where the `<collection>` tag is defined.

Actually, it is a member of the Rapid library of tags that we have mentioned. As we go

through these tutorials, we will point out where tags, and in particular parameters tags come from. Here is a list of tag situations you will encounter:

- HTML tags which are often parameterized
- Rapid library tags which are often parameterized
- Rapid parameter tags, not defined in your app
- User-defined tags which are often parameterized
- Rapid auto-generated tags which are not usually parameterized

As we go forward, you will gradually learn how the auto-generated tags are built up out of Rapid library tags.

The <collection> tag does the following:

- Repeats the body (stuff between the tags) of the tag inside a list with one item for each object in the collection of records.
- If there is no content for the body, it renders a <card> inside the tag nested within the tags.

The following code corresponds to "no body":

```
<collection:/>
```

and this code corresponds to an empty or blank body:

```
<collection:></collection:>
```

You have already seen what the former will do, namely list your records in a bolded hyperlinked format, which it derives from the <card> tag. Now try the latter. You will get the blank repeated as many times as there are recipe records, which equals zero.

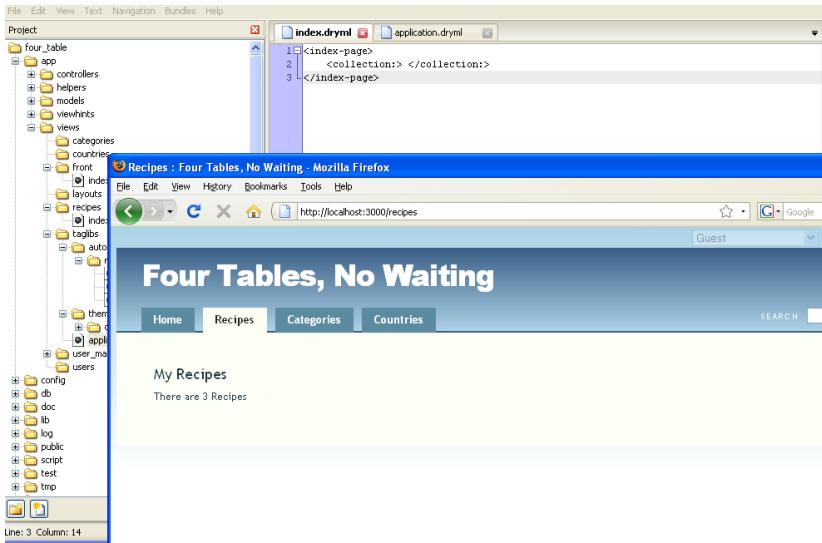


Figure 142: page view of using a blank "<collection:></collection:>" tag

Try the following code.

```
<collection:>Hello!</collection:>
```

Since there is a body, the 'Hello!' will be repeated and the <card> will no longer be called.

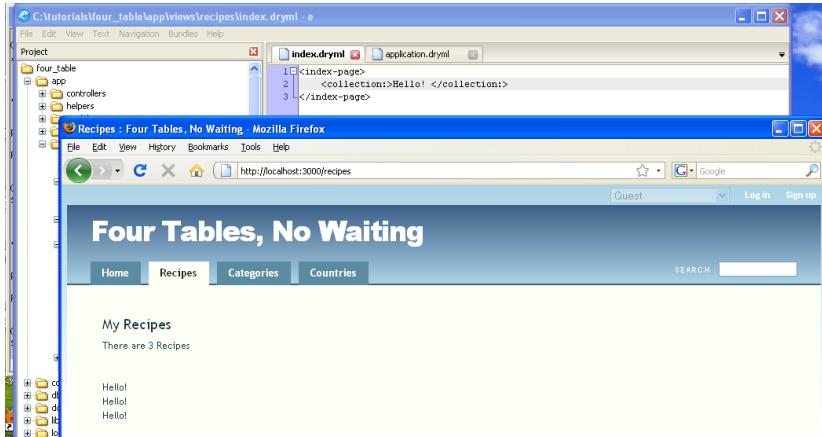


Figure 143: How the <collection> tag iterates

There are three records in our Recipes table so 'Hello!' is repeated three times. If you examine your page a little more in detail by hovering your mouse over the 'Hello's',

you will see that each is linked to different records and has a different route associated with it.

Let's get some content displayed. We are going to use Rapid's `<a>` tag, which is similar to the HTML `<a>` tag but has been redefined. The `<a>` tag is extended in Rapid to automatically provide a hyperlink to the route to show a particular record of the model. Let's try this out with the following code.

```
<collection:><a/></collection:>
```

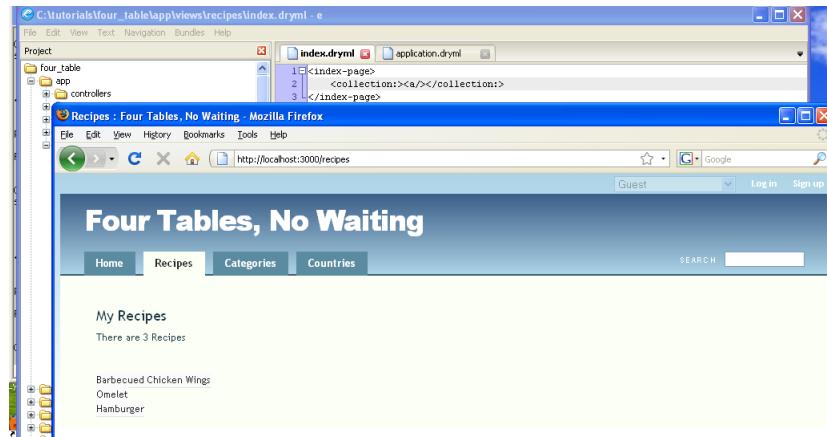


Figure 144: Using the `<a>` hyperlink tag within a collection

If you mouse over or click on one of the links you will discover a route like this

<http://localhost:3000/recipes/2-omelette>

The `<a>` link has created this route, which is the route for a show action.

Let's do a comparison with the `<card>` tag that Hobo would call if you were not overriding it. Here is the `<card>` tag definition.

```
<def tag="card" for="Recipe">
  <card class="recipe" param="default" merge>
    <header: param>
      <h4 param="heading"><a><name/></a></h4>
    </header:>
    <body: param>
      <ht key="category.collection.count"
          count="&this.categories.size">
        <count:categories param/>
      </ht>
    </body:>
  </card>
</def>
```

The `<card>` tag uses an `<h4>` heading tag which bolds and applies a larger font according to Hobo's CSS files. It also uses the `<a>` tag with a body provided by the `<name>` tag, which renders the field that Hobo figures out automatically to be the most likely field you want to display. The `<name>` tag will pick out field names such as `title`, for example, which is the name of the field in our `Recipe` model.

If you wish to explicitly display a different field other than the one that Hobo provides by default, you can use the Rapid `<view>` tag. The syntax for this tag is different than you have encountered so far. In this section we are just going to give you a simplified description of the syntax and postpone a more detailed discussion for a later chapter:

```
<index-page>
  <collection:><view:title/></collection:>
</index-page>
```

Note: You will observe the trailing colon (`:`) with the `<view>` tag. This is an entirely different use of colon (`:`) than you have seen with parameter tags. Here the colon (`:`) is telling Hobo to figure out what model you are referring to and display the field from that particular model. This called *implicit context*, Hobo's ability to know at all times what model you are working with in a particular view. In a later chapter you will learn how to change the implicit context.

If you refresh your browser, you will note that the recipes displayed are not clickable. That is because of the way that the `<collection>` tag works. Remember that when you add a body to the tag, it no longer uses the `<card>` tag so you are only asking Hobo to display the title field, not create a hyperlink. That is easily remedied by doing the following.

```
<index-page>
  <collection:><a><view:title/></a></collection:>
</index-page>
```

Refresh your browser and see what you've got now:

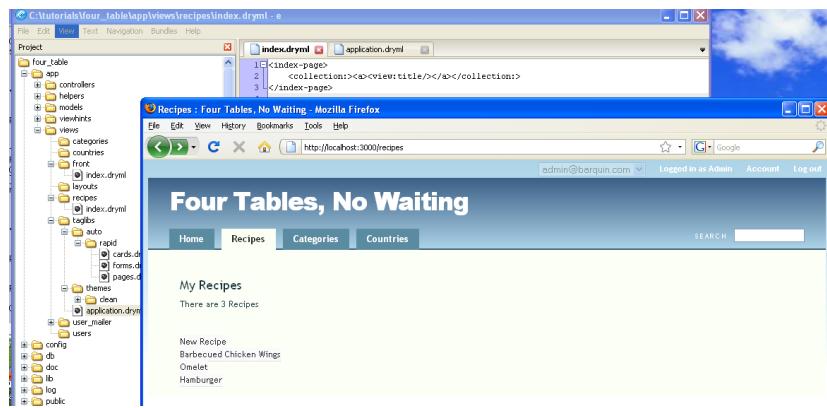


Figure 145: Specifying what <collection> tag will display

This looks pretty close to the default version of the <collection> tag. With the following use of the <h4> HTML tag, you can almost bring back the default appearance.

```
<index-page>
  <collection:><h4><a><view:title/></a></h4></collection:>
</index-page>
```

The only difference is the background provided to the record that you see above in Step 1 and the lack of the category count. The background is Hobo's default CSS formatting which in this case is associated with the <card> tag and since you are not using it, the formatting does not appear. Understanding how Hobo utilizes CSS files is covered in a later Chapter.



Figure 146: Changing the display style within <collection>

4. Display the associated record collection. Now that you see how to display collections of records, let's go a bit deeper. Our "Recipe" model has a many-to-many relationship with the "Category" Model, but it is possible to see this relationship without having to click through an individual recipe.

You can do this in several different ways. First we will do it in `views/recipes/index.dryml` template. Then we will try it in a `<card>` definition in `application.dryml`. Try out the following code.

```
<index-page>
  <collection:><h4><a><view:title/></a></h4>
    <view:categories/>
  </collection:>
</index-page>
```

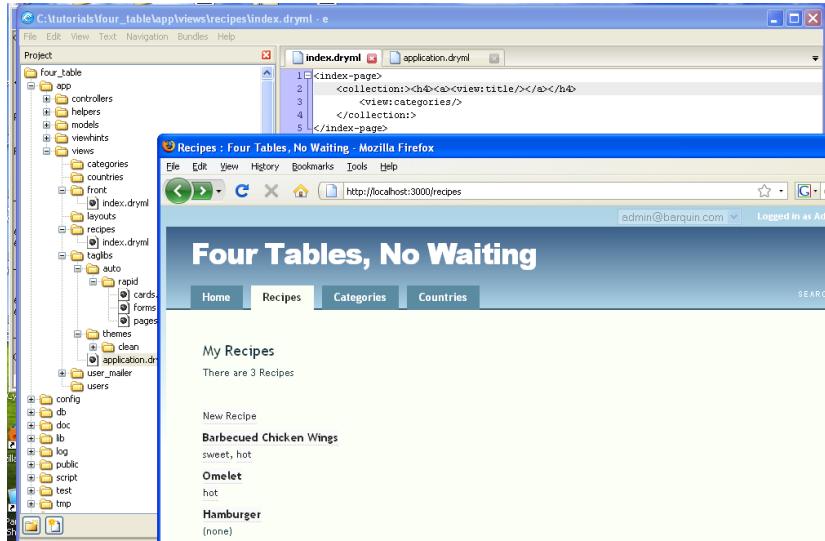


Figure 147: Changing the implicit context within <collection>

What we did here with the `<view>` tag was to tell Hobo to change its implicit context to the *Categories* model. The colon(:) is what did the trick and, of course, all the machinery inside Hobo which keeps it informed about the relationship between models that we set up.

Now we are going to do this slightly differently by using another Rapid library tag called `<repeat>`.

```
<index-page>
  <collection:><h4><a><view:title/></a></h4>
    <repeat:categories><a /></repeat>
  </collection:>
</index-page>
```

The repeat tag with the colon (:) tells Hobo to loop through the records in the implicit context and to display what is in the body of the tag, namely `<a>`. Try it and you will see the *categories* as hyperlinks, but they all run together. Fortunately, `<repeat>` has a join attribute to put in some additional character punctuation. Try this.

```
<index-page>
  <collection:><h4><a><view:title/></a></h4>
    <repeat:categories join="," "><a /></repeat>
  </collection:>
</index-page>
```

Now you get this:

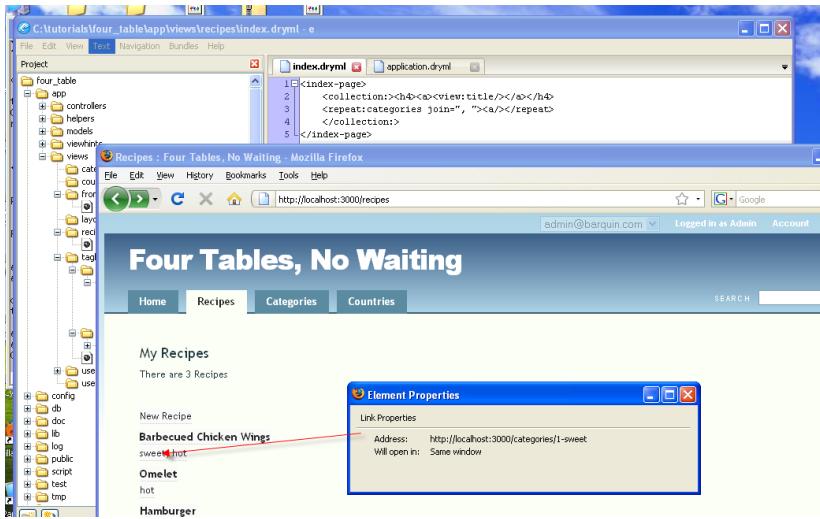


Figure 148: Creating comma-delimited multi-valued lists in a <collection>

If you don't want to have your categories linked you could do this:

```
<index-page>
  <collection:><h4><a><view:title/></a></h4>
    <repeat:categories join="," "><name/></repeat>
  </collection:>
</index-page>
```

or you could do this:

```
<index-page>
  <collection:><h4><a><view:title/></a></h4>
    <repeat:categories join="," "><view:name/></repeat>
  </collection:>
</index-page>
```

Note: The <name/> tag and the name attribute in <view:name/> are not the same. In the former, Hobo looks at the *Category* model to find a candidate field to output from the <name> tag. We made it easy for Hobo since there is a field called name which it picks, and displays. In the second example, we explicitly tell Hobo to display the name field of the categories model.

Now we are going to try the same thing within a tag definition so put your template, views/recipes/index.dryml back to the following:

```
<index-page/>
```

Go into application.dryml and find the recipe <card> definition. It should be there from Tutorial 1. If it is not there copy it from views\taglibs\auto\rapid\cards.dryml.

Edit it to look like the figure below. Not like the added code in italics and bold. We have added the same code we put in the template above. Since the code is now in the <card> tag definition, we should get all the formatting set up pre-defined in Hobo.

```
<def tag="card" for="Recipe">
  <card class="recipe" param="default" merge>
    <header: param>
      <h4 param="heading"><a><name/></a></h4>
    </header:>
    <body: param>
      <ht key="category.collection.count"
          count="&this.categories.size">
        <count:categories param/>
      </ht>
      <br/><view:categories/>
    </body:>
  </card>
</def>
```

Refresh your browser.

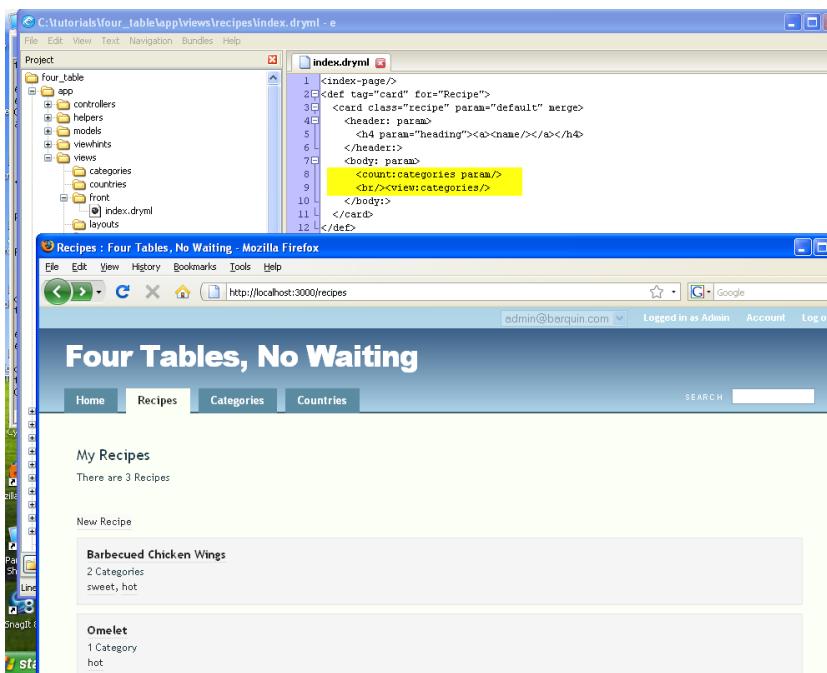


Figure 149: Adding the count of values in the <card> tag

Now you have succeeded in editing the recipe <card> tag to drill down to assigned categories for your recipes.

5. Use the <if> and <else> tags. We are going to show you one more version way of displaying the recipe records and the categories assigned to them. Notice that when there are no categories assigned, the <view> tag puts out the text, ‘none’. Let’s try to make this look a little nicer.

The <if> tag checks for null records in a record collection and outputs the body of the tag when the record exists. You use the <else> tag for the case when the record does not exist. Try this.

```

<def tag="card" for="Recipe">
    <card class="recipe" param="default" merge>
        <header: param>
            <h4 param="heading"><a><name/></a></h4>
        </header:>
        <body: param>

            <if:categories><view/></if> <else>There are no assigned
            categories yet.</else>
        </body:>
    </card>
</def>
```

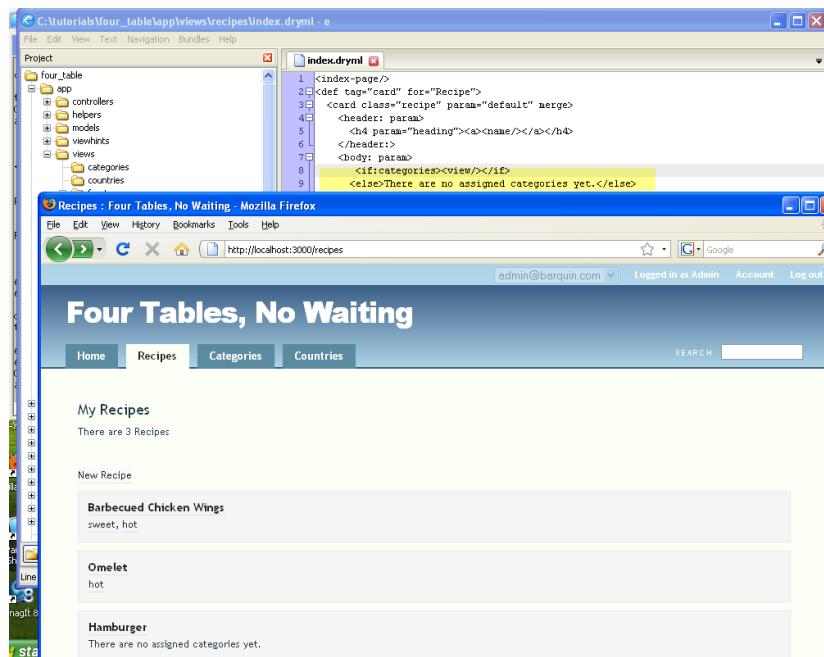


Figure 150: Using “if–else” within a tag to display a custom message

In the examples above, we used the trailing colon (:) syntax to tell Hobo what model context we wanted in the `<view>` or `<repeat>` tags. In this example, we take care of changing the context with the `<if>` tag so there is no need to do it again. In fact, if we introduced this redundancy, as in the code below, we would get an error:

```

<!--THIS CODE PRODUCES AN ERROR-->
<if:categories><view:categories/></if>
<else>There are no assigned categories yet.</else>
```

Tutorial 13

Listing Data in Table Form

You will learn how to display your data in a sortable, searchable table. The search will actually extend beyond the table entries to all the fields of each record. The sort and search code is an advanced topic that is provided here for completeness.

Tutorial Application: four_table

Topics

- Display model data in table form.
- Use the *replace* attribute to change the content of a parameter tag.
- Display associated record counts in the table
- Add search and sort to the table.

Steps

1. Display model in table form. In the following code, we use another built in feature of Hobo's parameter tags, the ability to replace what the parameter does with new tag code. The code below should be entered into your `views/recipes/index.dryml` file. Delete or comment out the `<index-page>` tag from Tutorial 12.

```
<index-page >
<collection: replace>
<div>
    <table-plus fields="title,country"/>
</div>
</collection:>
</index-page>
```

Refresh your browser to see your new table:

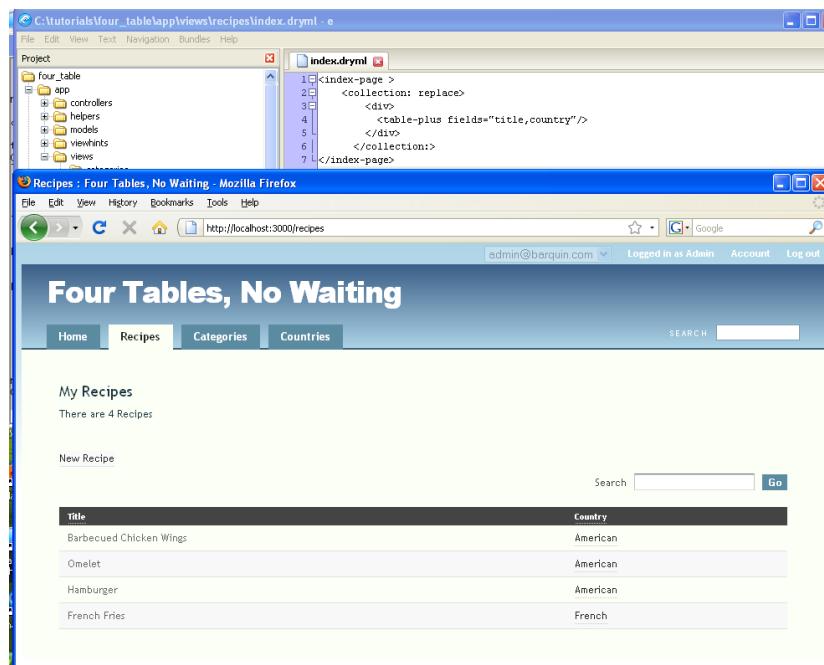


Figure 151: Using <table-plus> to display a columnar list

The `fields` attribute of the `<table-plus>` tag lets you specify a list of fields that will become the columns of a table.

So essentially one line of code sets up a pretty good table for you in Hobo.

2. Make your data hyperlinked. You might have noticed that the country names are clickable but the titles are not. Hobo provides a way to do this using the `this` keyword. `this` refers to the object currently in scope.

Note: The `this` keyword actually has a far deeper meaning that will be explored in more depth later. For now we will just outline how to use it.

Make the following change to your code and refresh your browser.

```
<index-page>
<collection: replace>
<div>
    <table-plus fields="this, country"/>
</div>
</collection:>
</index-page>
```

Now your *recipes* are hyperlinked to the show route that displays individual *recipe* records.

3. Show associated record counts. It would be nice to display how many associated category records there are. Again, since Hobo knows all about the relationships between records, you know it can figure this out.

However, if you are familiar with database programming, you know queries have to be done to compute this value. The Hobo framework does not require you to do this extra work. You already know what you want—so you should be able to declare it. Here is how you do it:

```
<index-page>
<collection: replace>
<div>
<table-plus fields="this, categories.count,
               country"/>
</div>
</collection:>
</index-page>
```

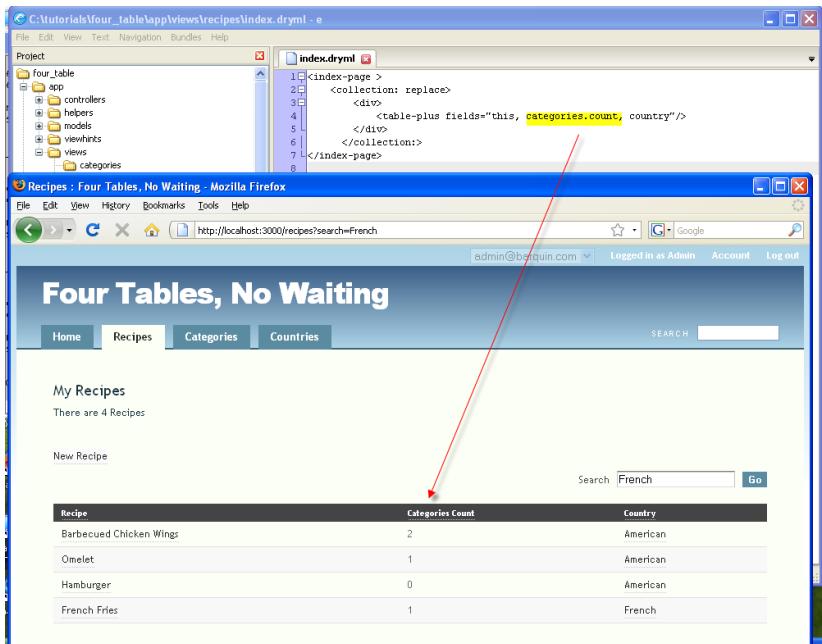


Figure 152: Adding a “Categories Count” to <table-plus>

That was straightforward. Before we refresh our browser again, let's display the actual *categories* in addition to the count.

Again, with other frameworks this would be a bit more complicated, but Hobo makes this easy. In the previous tutorial, you learned a few ways to display the categories associated with an individual recipe, the simplest of which was the <view> tag.

Here it is even easier—just add categories to the fields attribute:

```
<index-page>
<collection: replace>
<div>
<table-plus fields="this, categories.count,
               categories, country"/>
</div>
</collection:>
</index-page>
```

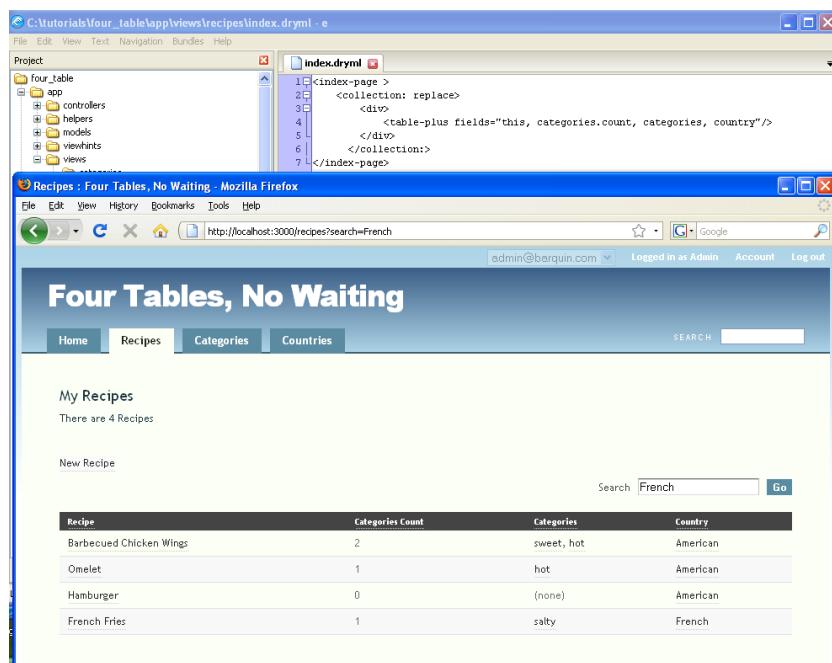


Figure 153: Adding a comma-delimited list within a <table-plus> column

4. Add search and sort capability to the table. Until now we have worked very little with controllers. You will quickly realize that to add search and sort, we will have to make a change in the *recipe* controller. Understand this by realizing that we want our application to respond to a click with two specific actions: one is a *sort* and the other is a *search*.

Go to your controllers/recipes_controller.rb file.

Note: This is actually an advanced topic since we are adding some Ruby code. You will learn more about the meaning of all the unfamiliar syntax in subsequent chapters. But for now, let's polish off this table functionality.

To get the search feature working, we need to update the controller side. Add an index method to `app/controllers/recipes_controller.rb` like this:

```
def index
  hobo_index Recipe.apply_scopes(:search => [params[:search],
    :title,:body], :joins=>:country, :order_by =>
  parse_sort_param(:title, :country, :count))
end
```

Note: The “apply scopes” for the search facility can only contain fields within the recipe model—not related models at this time, but the “order by” can.

Clicking on the “Country” label twice will trigger sorting in descending alphabetical order:

Recipe	Categories Count	Categories	Country
French Fries	1	salty	French
Barbecued Chicken Wings	2	sweet, hot	American
Omelet	1	hot	American
Hamburger	0	(none)	American

Figure 154: Adding a search facility to `<table-plus>` using Hobo’s `apply_scopes` method

Now search/filter by “French” in the title or body:

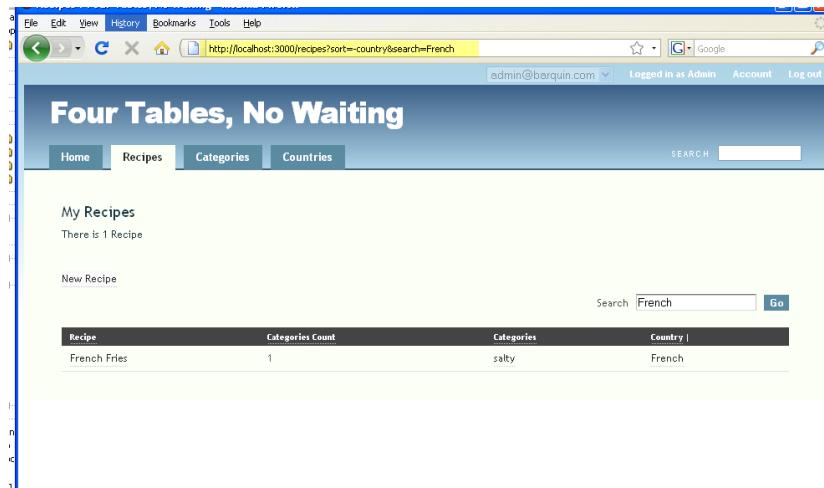


Figure 155: Found Recipes searching for “French”

Tutorial 14

Working with the Show Page Tag

In this tutorial you will learn the options for displaying details about single records. In the last two tutorials, we focused on displaying lists of records. Hobo has a specific auto-generated tag for handling the display of individual records and a route and view template associated with it.

Tutorial Application: four_table

Topics

- Edit the <show-page> tag.
- Create and work with the show.dryml template.
- Work with <field-list>, <fieldname-label> and <view> tags.

Steps

Step 1

Copy the <show-page> tag. Go to pages.dryml and copy the <show-page> tag for Recipes to application.dryml.

```
<def tag="show-page" for="Recipe">
    <page merge title="#{ht 'recipe.show.title',
        :default=>['Recipe'] }">
        <body: class="show-page recipe" param/>
        <content: param>
            <header param="content-header">
                <a:country param="parent-link">&laquo; <ht
                    key="recipe.actions.back_to_parent"
                    parent="Country"
                    name="&this">Back to
                    <name/></ht></a:country>
                <h2 param="heading">
                    <ht key="recipe.show.heading"
                        name="&this.respond_to?(:name)
                        ? this.name : "">
                        <name/>
                    </ht>
                </h2>
                <record-flags fields="" param/>
                <a action="edit" if="&can_edit?">
                    param="edit-link">
                    <ht key="recipe.actions.edit"
                        name="&this.respond_to?(:name)
                        ? this.name : "">
                        Edit Recipe
                    </ht>
                </a>
            </header>
            <section param="content-body">
                <view:body param="description"/>
                <field-list fields="country" param/>
                <section param="collection-section">
                    <h3 param="collection-heading">
                        <ht key="category.collection.heading"
                            count="&this.categories.count" >
                            <human-collection-name
                                collection="categories" your/>
                        </ht>
                    </h3>
                    <collection:categories param/>
                </section>
            </section>
        </content:>
    </page>
</def>
```

We are going to focus on three display components of this tag. The components are noted in bold italics above, to help you understand how to change the display of individual records. (Add the '`<field-list fields="country" param/&#gt;`' if it is not present.)

Click on the "Recipes" tab and then click on an individual recipe.

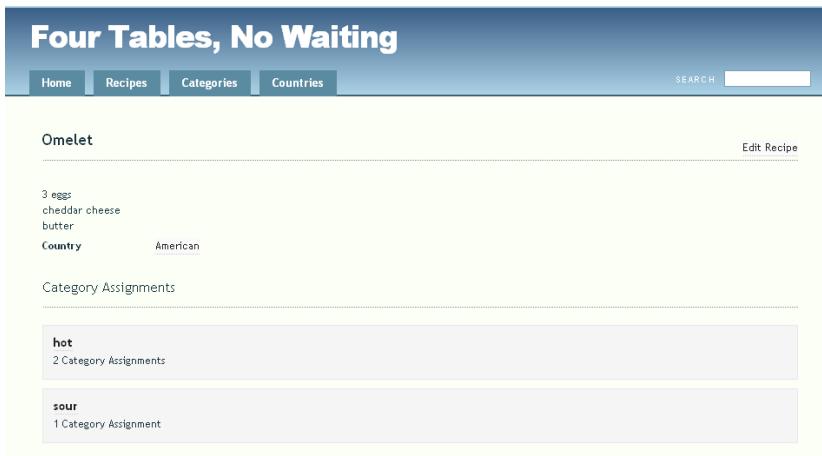


Figure 156: The Recipe show page before modification

Comment out the three lines above in bold italics using `<!-- ... -->`, and confirm that you have removed the display of the individual recipe record.

Step 2. **Create the show.dryml template.** Go to views/recipes and create a new template file called `show.dryml`. When a user invokes the "show" action by requesting the display of a single record, this is the first of the three places Hobo looks to determine how to display the record.

As with the index action, its next two stops are the `application.dryml` file to look for application wide tag definitions and, finally, in `pages.dryml` for the auto-generated tag definitions which are based on model and controller code.

Place the following code in `show.dryml` to invoke your show page.

```
<show-page/>
```

Refresh your browser and you should see the following:



Figure 157: Recipe show page after removing three critical lines of code

Step 3. Use the <field-list> tag. The <field-list> tag allows you to display rows of data in two columns. The first column contains the name of the field and the second column contains the contents of that field. The <field-list> tag has been parameterized in the <show-page> tag, so we need to invoke it with a trailing colon (:).

Remove the comments around the <field-list> tag in `application.dryml` and try the following in `show.dryml`.

```
<show-page>
  <field-list: fields = "body, country"/>
</show-page>
```

Here you are using the attribute `fields` to declare which fields in your model you wish to display.



Figure 158: Using the <field-list> tag to choose which fields to display

Hobo can even reach into the associated table and display the categories using <field-list>. Try this.

```
<show-page>
  <field-list: fields = "body, country, categories"/>
</show-page>
```

You can remove the collection heading since you no longer need it by observing that the `<show-page>` tag has a parameterized `<h3>` tag renamed as the `<collection-heading:>` parameter tag. You will see the following code in the `<show-page>` definition.

```
<h3 param="collection-heading">Categories</h3>
```

Now go into your `show.dryml` file and replace the default contents of the tag with nothing.

```
<show-page>
  <field-list: fields = "body, country, categories"/>
  <collection-heading:></collection-heading:>
</show-page>
```

Now you should have the following after refreshing your browser.

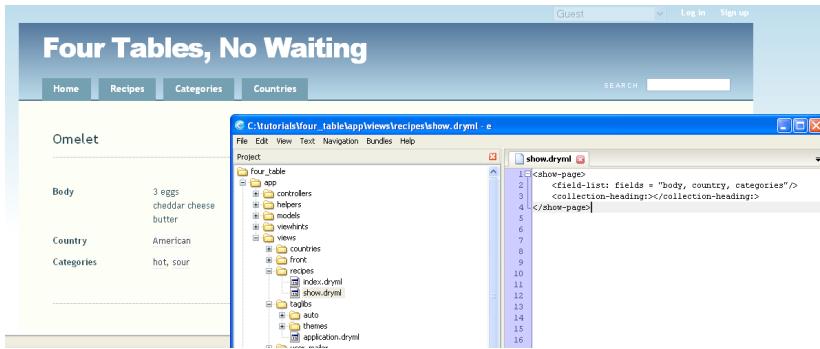


Figure 159: Using the `<collection-heading:>` tag

Step 4. Changing the `<field-list>` labels. We can now see that the `<field-list>` tag does a nice job of formatting the display of the fields of a record. The default display pictured in Step 1 uses a combination of the `<view>` and `<field-list>` tags. However the `<view>` tag does not automatically provide a label like the `<field-list>` tag. We will cover this further in Step 5. Now, let's learn how to change the labels.

Try the following code to change the *body* label to 'Recipe'.

```
<show-page>
<collection-heading:></collection-heading:>
<field-list: fields = "body, country, categories">
  <body-label:>Recipe</body-label:>
</field-list>
<show-page>
```

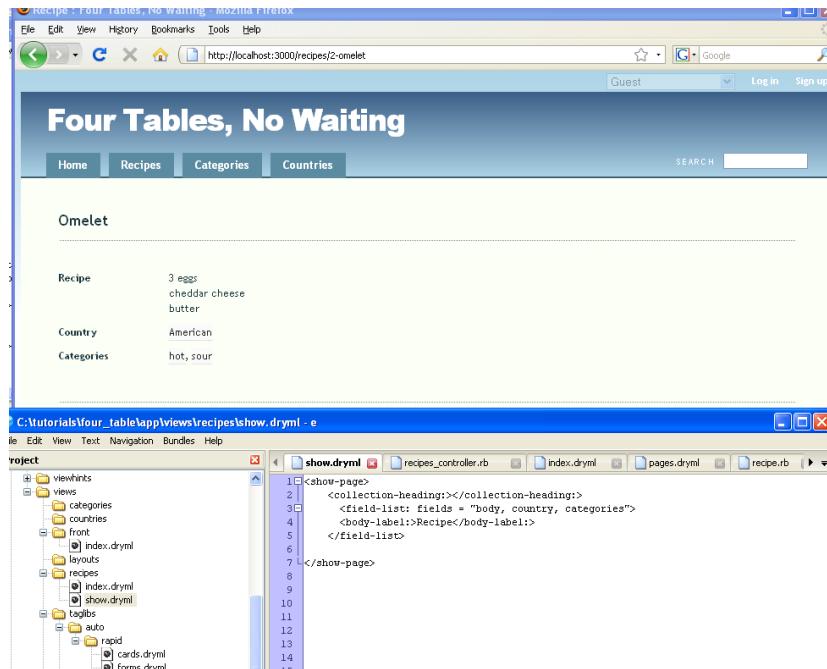


Figure 160: Using the `<body-label:>` parameter tag

There are a few new things going on here that you have not seen before.

- The `<body-label:>` tag is a parameter tag defined in the Rapid Library.
- The `<body-label:>` tag is a user customized Rapid library tag derived from the generic `<fieldname-label>` tag.
- The `<body-label:>` tag is nested within the `<field-list>` tag.

Let's go through these points one at a time.

Rapid Parameter Tag. This the tag is used with a trailing colon (:), meaning that `<body-label:>` is a parameter tag. However, it is not defined anywhere within either your code or the auto-generated code. (You will see user-customized tags again with pseudo tags in the next tutorial.)

If you have done any coding besides this tutorial, you have probably run into the error “You cannot mix parameter and non-parameter tags”.

If there were not a Rapid parameter tag to use here and you tried to use a regular Rapid tag, you would get an error. Try deleting the colon (:) from <body-label:> to confirm this.

User-customized tags. The tag name is dynamic depending on what field in the <field-list> is being addressed. For example, to change the label of the country field, you would use the <country-label:> tag.

Tag nesting. The feature that you see here is the ability to nest tags in order to pass data. Here you are passing the content of the tag to the label variable of the <field-list> tag.

Let’s go one step further and re-label the other two fields displayed on our page. You can just nest each <fieldname-label> tag after the other within <field-list> and Hobo will pass the content into the <field-list> tag.

You might be noticing that categories is not a field at all; it is a collection. That is not a problem for Hobo. Hobo can address the label using the <categories-label> just as if it was a field:

```
<show-page>
  <collection-heading:></collection-heading:>
    <field-list: fields = "body, country, categories">
      <body-label:>Recipe</body-label:>
      <country-label:>Origin</country-label:>
      <categories-label:>Flavors</categories-label:>
    </field-list>
  <show-page>
```

Refresh your browser and try this out.

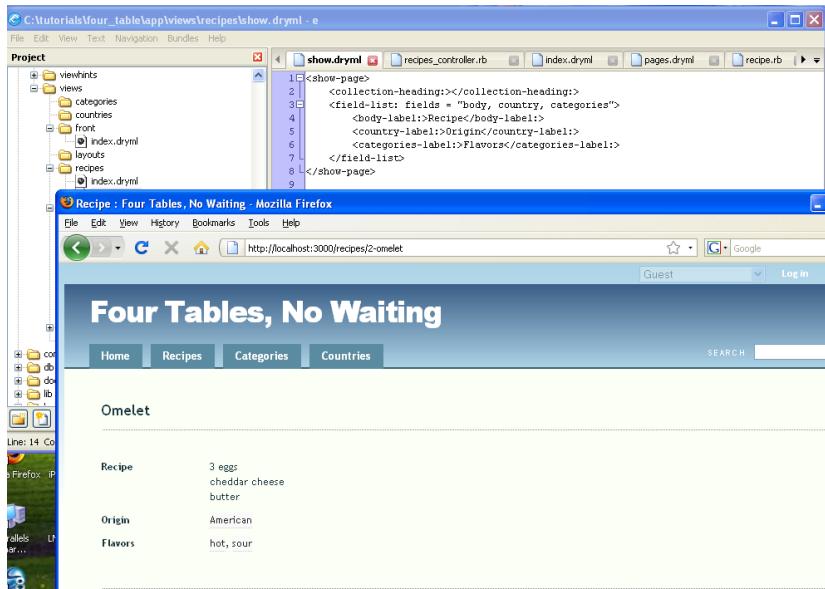


Figure 161: Using the <country-label:> parameter to change the label on the page

Step 5. Using the <view> tag to display a record. There is another way to work with the fields of an individual record and its associated records using the <view> tag.

Let's make a tag from the <show-page> tag within application.dryml. Recall that you can use the merge attribute within a template although you can't use the <extend> tag in a template, only in application.dryml.

Let's try out the following code in application.dryml.

```
<def tag="show-page-new">
  <show-page merge>
    <content-body:>
      <h2>Title:</h2>
      <view:title/><br/>
      <h2>Recipe:</h2>
      <view:body/>
      <h2>Categories:</h2>
      <view:categories/>
      <h2>Country:</h2>
      <view:country/>
    </content-body:>
  </show-page>
</def>
<show-page-new/>
```

In the above code, we are using the parameter tag `<content-body:>` defined from a parameterized `<section>` tag in the `<show-page>` tag:

```
<section param="content-body">
```

By placing new HTML and Rapid library tags within the `<content-body:>` tags, we are changing the default content defined in the `<show-page>` tag to the new content and preserving everything else in the `<show-page>` tag. We are not only preserving the content but also the formatting. Hobo has predefined CSS formatting that correspond to the Rapid tags.

If we had used the `replace` attribute in the `<content-body:>` tag like this...

```
<content-body: replace>
```

..we would have removed Hobo's built-in formatting.

Remove the last code in `show.dryml` and put `<show-page-new/>` at the top.

Refresh your browser without using the `replace` attribute and then try it with the attribute to confirm that the formatting will be removed.

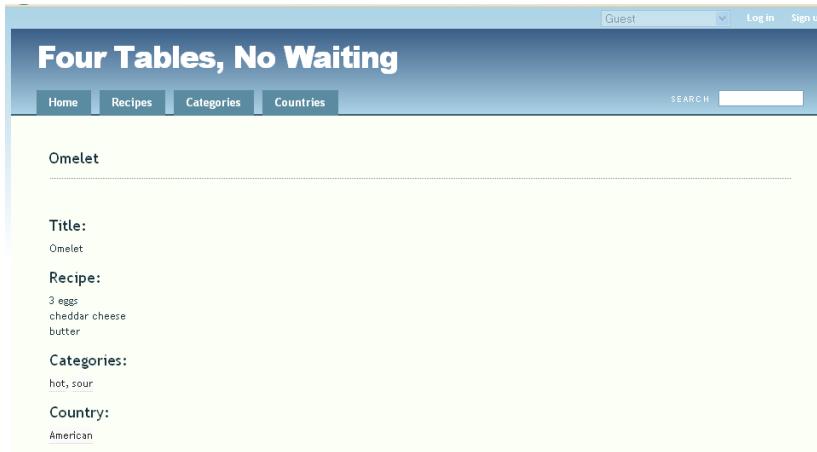


Figure 162: A new show page for Recipes

Here is what happens when you add the `replace` attribute.

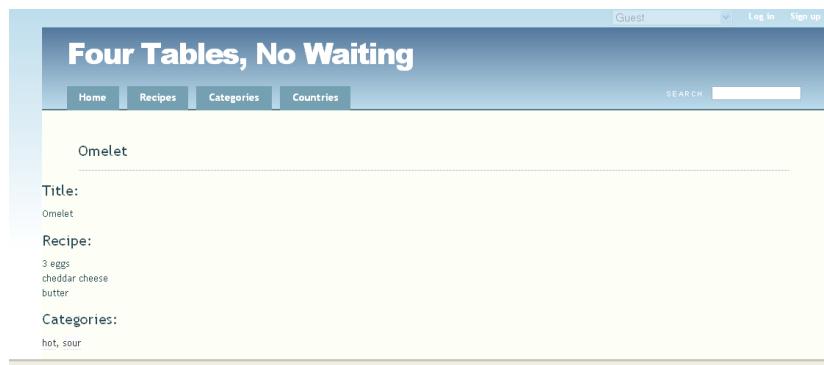


Figure 163: Page view of using the `replace` attribute in the `<content-body>` parameter tag

Now take out the `replace` attribute before proceeding.

Step 6. Summary. You have now learned how to create a new template called `show.dryml` in the `views/recipes` directory. It is used whenever there is an action to display an individual *recipe* record. Before you created this file, Hobo was constructing the template on the fly from the auto-generated `<show-page>` tag in `pages.dryml`.

Tutorial 15

New and Edit Pages

(with the `form` tag)

In this tutorial you will be introduced to the `<new-page>` and `<edit-page>` auto-generated tags. Both of these tags utilize the Rapid `<form>` tag as their basic building block. You will learn how the `<form>` tag utilizes both the `<field-list>` and `<input>` tags. You will also learn about the concept of a “polymorphic” tag, which renders form components based on field type and model structure.

Tutorial Application: `four_table`

Topics

- The `<new-page>` and `<edit-page>` tags
- The `<field-list>` tag
- The `<input-tag>`

Steps

Step 1. **Get introduced to the `<new-page>` and `<edit-page>` tags.** Go into `pages.dryml` and look at the code for both tags. Here is the `<new-page>` definition:

```
<def tag="new-page" for="Recipe">
    <page merge title="#{ht 'recipe.new.title',
        :default=>[' New Recipe'] }">
        <body: class="new-page recipe" param/>
        <content: param>
            <section param="content-header">
                <h2 param="heading">
                    <ht key="recipe.new.heading">
                        New Recipe
                    </ht>
                </h2>
            </section>
            <section param="content-body">
                <form param>
                    <submit: label="#{ht 'recipe.actions.create',
                        :default=>['Create Recipe']}" />
                </form>
            </section>
        </content:>
    </page>
</def>
```

And here is the `<edit-page>` definition:

```
<def tag="edit-page" for="Recipe">
    <page merge title="#{ht 'recipe.edit.title',
        :default=>['Edit Recipe'] }">
        <body: class="edit-page recipe" param/>
        <content:>
            <section param="content-header">
                <h2 param="heading">
                    <ht key="recipe.edit.heading"
                        name="#&this.respond_to?(:name) ? this.name : ''">
                        Edit Recipe
                    </ht>
                </h2>
                <delete-button label="#{ht
                    'recipe.actions.delete', :default=>['Remove This
                    Recipe']} " param/>
            </section>
            <section param="content-body">
                <form param/>
            </section>
        </content:>
    </page>
</def>
```

The components that we are going to focus on are shown in *red italics*. Also take a look at the <form> tag that both of these tags call.

```
<def tag="form" for="Recipe">
    <form merge param="default">
        <error-messages param/>
        <field-list fields="title, body, categories,
            category_assignments, country" param/>
        <div param="actions">
            <submit label="Save" param/><or-cancel
            param="cancel"/>
        </div>
    </form>
</def>
```

In a nutshell, you can see that each of these auto-generated tags call the auto-generated <form> tag which is defined by merging the Rapid <form> tag in addition to other tags. The specific fields that will be used in the form are declared within the *fields* attribute of the <field-list> tag that you learned about in Tutorial 14 on the <show-page> tag.

You will notice that the <field-list> tag is doing something different. Instead of displaying a two-column table consisting of field labels in the first column and field

data in the second, it is putting the appropriate data entry control in the second column. The data entry control choice depends on the type of field defined in the model.

Hobo puts a one-line data entry box for the *title* field which is a string field and a larger box for the *body* field which is a text field. Notice that Hobo also creates drop-down combo controls for the *country* field and for the *categories* collection.

Hobo does this by inspecting table relationships. The *recipe* model is related to both the *country* model and the *category* model. This is a powerful capability for one tag, especially given that the *Category* model is related to the *Recipe* model through a many-to-many relationship through the *CategoryAssignment* model.

The screenshot shows a Hobo application interface titled "Four Tables, No Waiting". At the top, there is a navigation bar with links for "Home", "Recipes", "Categories", and "Countries". On the right side of the header, there are links for "admin@barquin.com", "Logged in as Admin", "Account", and "Log out". Below the header, there is a search bar labeled "SEARCH". The main content area is titled "New Recipe". It contains four input fields: "Title" (a single-line text input), "Body" (a large multi-line text area), "Categories" (a dropdown menu with an option "Add Category"), and "Country" (a dropdown menu with an option "(No Country)"). At the bottom of the form, there is a blue button labeled "Create Recipe" and a link "or Cancel".

Figure 164: Default Hobo form rendering

All of this capability results from Hobo's implementation of tag polymorphism, an ability to do what is necessary from the *context* of the code. Polymorphism means 'many forms (not data entry form)' or 'many structures'. It is a hallmark feature of the Ruby language.

(There is even more going on in the `<field-list>` tag, but we will wait to discuss it until the next step.)

Before moving on, let's take care of a detail, using your knowledge of parameter tags. You will note that the `<new-page>` tag calls the `<submit:>` parameter tag and that the `<edit-page>` tag does not. However, there is a "submit" button on the "edit" page. The explanation can be found in the definition of the `<form>` tag. There you will see that the `<submit>` tag is declared as a parameter tag as is the `<or-cancel>` tag.

The `<new-page>` tag calls the `<submit:>` parameter tag and changes the label from its default value of 'Save' to a new value of 'Create Recipe'. There is no need to call the `<or-cancel>` tag with its parameterized name, `<cancel>`, because it is not changed.

On the other hand, the `<edit-page>` tag only relies on the default for both tags, so there are no calls to them in the `<edit-page>` tag definition.

Step2. Working with the `<field-list>` tag. You have already done some work with this tag in the last tutorial. Experiment with removing a field by editing the tag's fields attribute. First, copy the three tags above into `application.dryml`

(As we have mentioned, you probably want to be careful about editing tags this way in a real application, but this is the easiest way for us to acquaint you with how Hobo works.)

Let's remove the `categories` drop-down box as an experiment. Working in `application.dryml`, edit the `<form>` definition code. Change

```
<field-list fields="title, body, categories,
category_assignments, country" param/>
```

to:

```
<field-list fields="title, body,
category_assignments, country" param/>
```

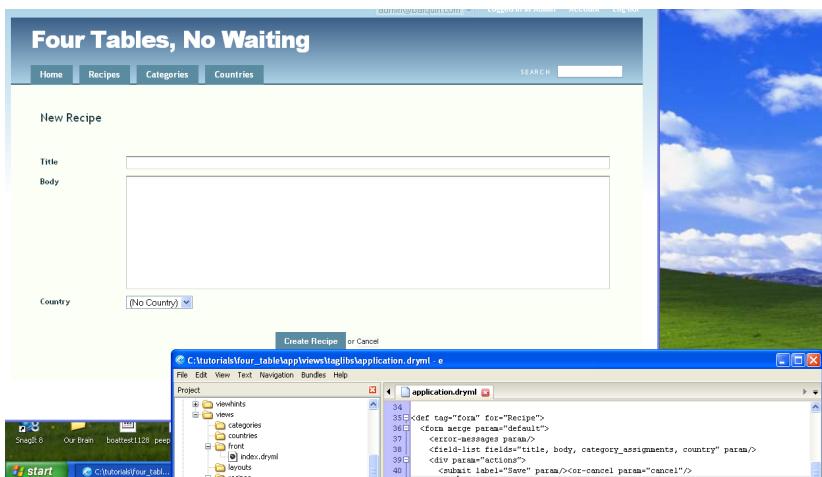


Figure 165: Modifying the `<field-list>` tag to remove fields on a page

Now, your `categories` drop-down box is gone.

You may be wondering why we did not also remove the `category_assignments` attribute, or why it is there at all. First, try removing `category_assignments` without removing `categories`. There is no effect. Try removing both. You get the same result as with removing `categories` alone. This is just how the `<field-list>` tag works. On the other hand, the model structure that connects the `Recipe` model to the `Category`

model through the *CategoryAssignments* model must, of course, be there for the drop-down box to be there at all. Put back the categories drop-down box to end this step of the tutorial.

Step 3. Working with the <field-list> and <input> tags. In the same way that *<field-list>* calls the *<view>* tag when it is showing a record's data, by default *<field-list>* calls the *<input>* tag when it is inside a form and the *<view>* tag when it is not. This is an illustration of tag polymorphism. That is, *<field-list>* does many different things depending on the context of its use.

The overall syntax of the *<input>* tag is the same as the *<view>* tag. When you wish to create an input control on a form, one at a time, you can invoke the control in the following way.

```
<input:title>
```

In the code above you are requesting that an input field be created for the title field of the *Recipe* model. Hobo knows to use the *Recipe* model as long as you are in the context of the *Recipe* model. In this case it is set by working within the *Recipe* form. As you've seen before, Hobo knows just what kind of control you are likely to need.

Below we are going to show you how to construct essentially the same form out of *<input>* tags that you created with the *<field-list>* tag in the previous step.

Let's be a bit more rigorous now in constructing tags from tags. First remove the form definition tag from *application.dryml*. You will now use the *<extend>* tag to redefine an auto-generated *<form>* tag with the same name.

Let's create the skeleton of an *extend* tag, so we can watch what happens one step at a time. The following code placed in *application.dryml* will cause no change because it substitutes this *<form>* tag for the original *<form>* tag.

```
<extend tag="form" for ="Recipe">
  <old-form merge/>
</extend>
```

The following code, which might seem to be identical, actually is not identical.

```
<extend tag="form" for ="Recipe">
  <old-form merge>
  </old-form>
</extend>
```

In the above case, Hobo replaced the default content of the parameterized *<form>* tag with blank content resulting in a blank form. Go to the 'Recipes' tab and pick a recipe. Then click 'New Recipe' to see the blank form.

Now, let's get some content into the parameter tag. Copy the following code into *application.dryml*:

```

<extend tag="form" for ="Recipe">
<old-form merge>
<error-messages param/>
<p><input:title/><p/>
    <div param="actions">
        <submit label="Save" param/><or-cancel
            param="cancel"/>
    </div>
</old-form>
</extend>

```

Refresh your browser.

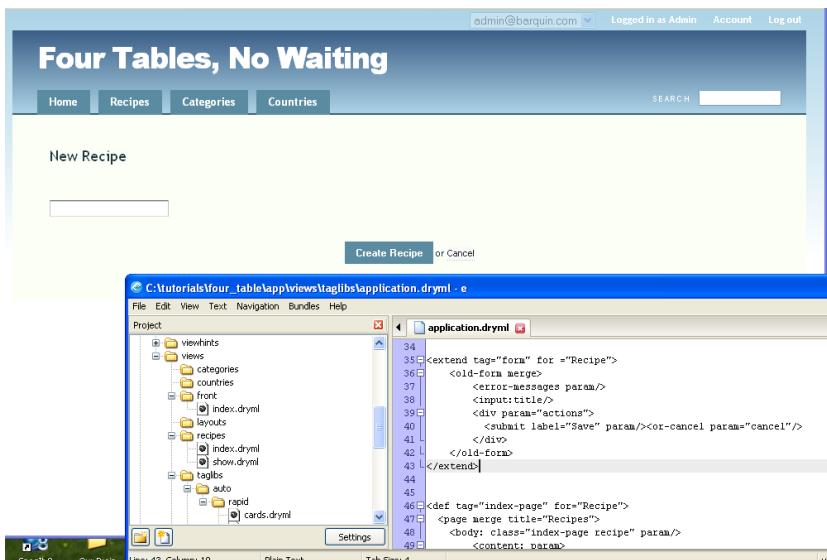


Figure 166: First step using the <Input> tag

We've got an entry control, but <input> has no built in labeling like <field-list>. We need to add it like we did with the <view> tag.

```

<extend tag="form" for ="Recipe">
<old-form merge>
<error-messages param/>
<p><b>Title</b></p>
<p><input:title/><p/><br/><br/>
<div param="actions">
    <submit label="Save" param/><or-cancel
        param="cancel"/>
</div>
</old-form>
</extend>

```

Refresh your browser:

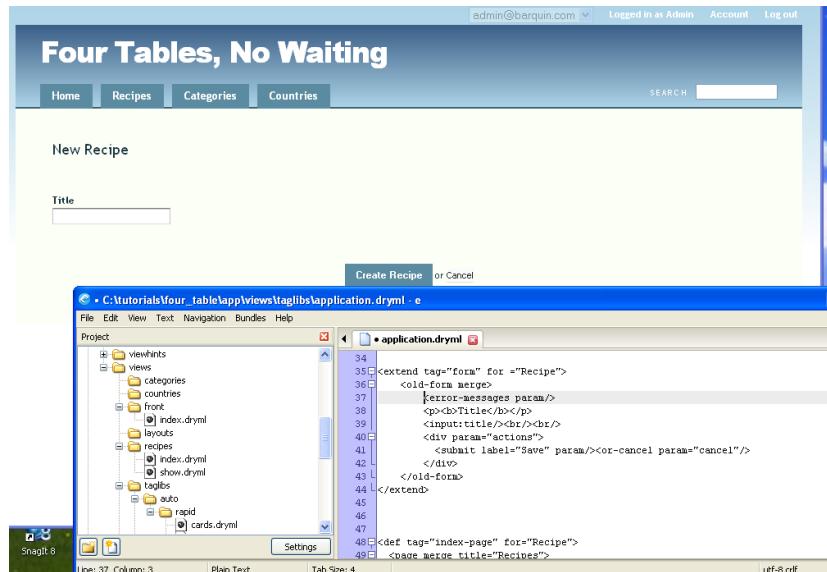


Figure 167: Adding the label for the field “Title”

Do the same thing for the rest of the fields. (Some of Hobo’s tags have differing built-in breaks which is why the number of breaks varies some below.)

```
<extend tag="form" for ="Recipe">
  <old-form merge>
    <error-messages param/>
    <p><b>Title</b></p>
    <p><input:title/></p>
    <p><b>Recipe</b></p>
    <p><input:body/></p>
    <p><b>Categories</b></p>
    <p><input:categories/></p>
    <p><b>Country</b></p>
    <p><input:country/></p>
    <div param="actions">
      <submit label="Save" param/><or-cancel
        param="cancel"/>
    </div>
  </old-form>
</extend>
```

Now, you have succeeded in reconstructing a form with the `<input>` tag and a little bit of additional HTML formatting.

Summary. Hobo provides great functionality for fine-tuning your application when the default rendering is not quite what you would like. You can experiment with them by going through the documentation on the Hobo web site or learn more about them in later chapters of this book.

Tutorial 16

The <a> Hyperlink Tag

In this tutorial you will learn to develop sophisticated data-driven hyperlinks in your Hobo pages.

Tutorial Application: four_table

Topics

- The <a> “hyperlink” tag for calling data-driven pages

Steps

Step 1. **Review the <a> tag usage within Hobo’s auto-generated tags.** Let’s take a look at the <a> tag usage in the auto-generated tags for the *Recipe* model.

```
<!--New Page Link from the Index Page Tag-->
<a action="new" to="&model" param="new-link"/>
```

This tag results in the ‘New Recipe’ hyperlink with the route ‘<http://localhost:3000/recipes/new>’.

```
<!--Edit Page Link from the Show Page Tag-->
<a action="edit" if="&can_edit?" param="edit-link">Edit
  Recipe</a>
```

This tag results in the ‘Edit Recipe’ hyperlink with a route like <http://localhost:3000/recipes/2-omelette/edit>.

Step 2. **Construct a link to an index (record listing) page.** Let’s work in the home page in the file views/front/index.html. We will place our test code after the “Congratulations . . . ” message.

```
<br/><h4>
<a to="&Country" action="index" >List My
  Countries</a><br/>
</h4>
```

This code will generate a link to a listing of countries in your database.

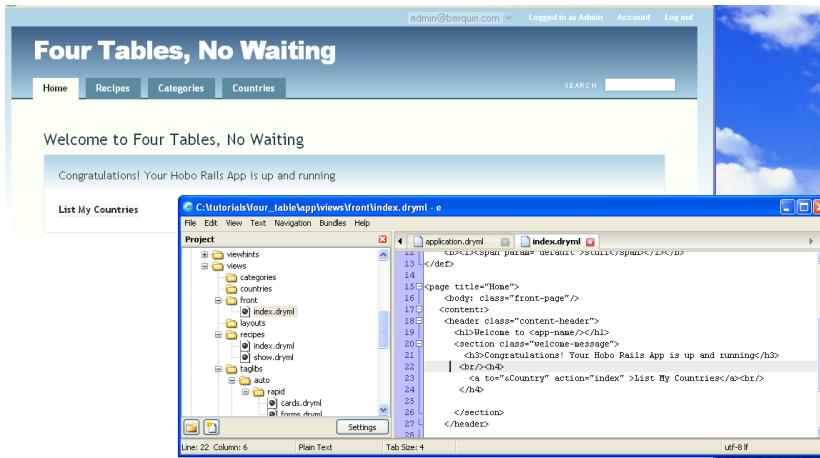


Figure 168: Generating an active link to a list of Countries

Note: The *to* attribute defines the model to be used in the listing. It is always prefixed by the “&” character. The *action* attribute defines the controller action which in the above case uses Hobo’s built-in *index* action. As you get more sophisticated, you will learn to define your own controller actions. These can also be referred to by the *action* attribute.

Of course, if you click on the ‘List My Countries’ link, you will now see a listing of countries.



Figure 169: The Countries index page activated by your custom link

Step 3. Construct a link to a new record page. We can construct a link to create new

countries in much the same way.

```
<a to="&Country" action="new" >New Country</a><br/>
```

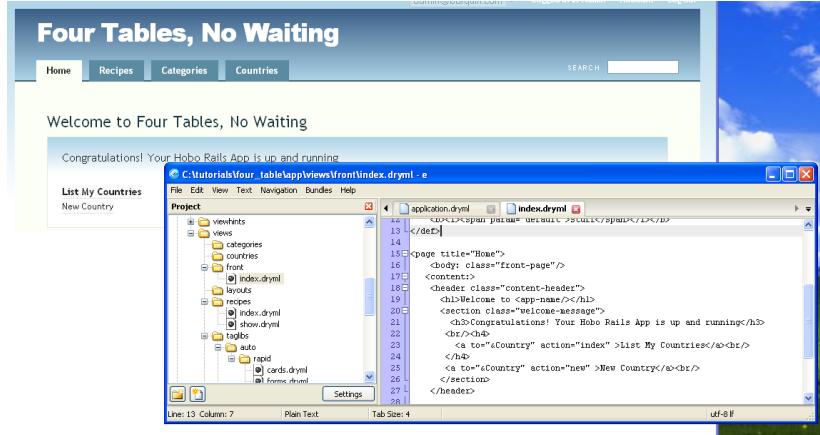


Figure 170: Constructing a custom link to the “New County” page

Now, you've got another link to try out.

Step4. Construct a link to an edit record page. If you want to create a custom link to an edit page, you have to be sure you are in the right context. Hobo can implicitly figure out which record you wish to edit, but only if you are displaying a particular record.

In the example from Step 1 above, the ‘edit page’ link occurs in a <show-page> tag definition so Hobo knows what record you want to edit.

Let's create our own link on the *Country* <show-page> tag by using the <content-body:> parameter tag that is defined in the auto-generated <show-page> tag for the *Country* model and create a new file called *show.dryml* in your *views/countries* directory.

You need to use the parameter tag or Hobo will ignore your code. This is just how the <show-page> tag was defined.

```
<show-page>
  <content-body:>
    <a action="edit" >Edit My Country</a><br/>
  </content-body:>
</show-page>
```

Go ahead and refresh your browser, click on the ‘Country’ tab and click on a country and you will see your new link to edit it on the bottom left.

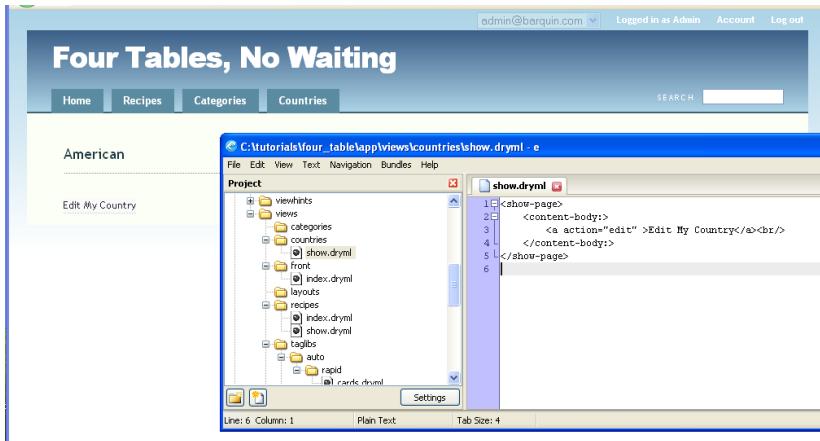


Figure 171: Page view of custom <show-page> tag

Step 5. Construct a link to a specific record. In general, Hobo takes care of linking to specific records for you by setting up the links implicitly in the <index-page>. If you need to link to a specific record, that will require a little Ruby to address a specific record in the database.

Chapter 5

ADVANCED TUTORIALS

Introductory Concepts and Comments

Tutorial 17 – The Agile Project Manager

Tutorial 18 – Using CKEditor (Rich Text) with Hobo

Tutorial 19 – Using FusionCharts with Hobo

Tutorial 20 – Adding User Comments to Models

Tutorial 21 – Replicating the Look and Feel of a Site

Tutorial 22 – Using Hobo Lifecycles for Workflow

Tutorial 23 – Using Hobo Lifecycles for Workflow

Tutorial 24 – Creating an Administration Sub-Site

Tutorial 25 – Using Hobo Database Index Generation

Introductory Concepts and Comments

This set of tutorials builds on the expertise you have developed so far with the Beginning Tutorials and Intermediate Tutorials.

You should be able to flex your muscles a bit, at rich text editing, charting, or even completely change the look and feel of a site.

The “Agile Project Manager” implements a large range of Hobo features into a fairly substantial and useful application. Try out enhancing and modifying it to fit your needs.

At the end of the Advanced Tutorials you will have the expertise to build, customize, and have your data-rich application ready to go into production. Enjoy!

Tutorial 17

The Agile Project Manager

Note: We have simplified this example somewhat by substituting the more traditional term “requirement” for what many agile development texts refer to “story”. One can extend this tutorial by linked one or many “requirements” for each user “story”.

Tutorial Application: projects

Overview

This tutorial is adapted from the classic “Agility” tutorial created by Tom Locke. It retains much of Tom’s text and style. We have also highlighted quotes from Tom at critical points in the tutorial.

Summary of the goals for this application:

1. The application “Projects” maintains a set of projects, requirements, and related tasks for a team of people.
2. Users access the application with a browser. The browser provides the capability to create, edit, delete and list projects, tasks, and task assignments.
3. All data entry fields have rollover hints to aid user data entry. Validation rules attached to the fields to prevent invalid entries.
4. Each project can have any number of associated tasks, and each task can have one or more team members assigned to it.
5. Each task has one status at any given time. A drop-down list of status codes will be displayed on a task creation page. Only one of these status codes can be selected and saved for this task.
6. There is a signup and login capability permitting each team member to create his/her own login name and password. The system administrator is determined by a simple rule—the first to log in to the application becomes the system administrator.
7. There will be a simple role facility that will allow an Administrator to assign roles to users. Both the Administrator and Coordinator roles can create and update projects, requirements, and tasks and assign team members to a task. Analysts, Developers, and Testers can change the status of a requirement.

8. The task assignment page will have a drop-down list of all existing team members. Only members of this list can have tasks assigned to them.
9. A project page will display a list of all tasks assigned to the project.
10. A task page will display a list of team members assigned to the task.

Getting Started

Step 1. Create the application like you have for the other tutorials:

```
> hobo new projects --setup
```

Look again at what we want this app to do:

- Track multiple projects
- Each project has a collection of requirements (“requirements”) which are described at a high-level requirements using the language of the user
- Each requirement is just a brief chunk of text
- A requirement can be assigned a current status and a set of outstanding tasks
- Tasks can be assigned to users
- Each user will have a simple view of the tasks they are assigned to

So:

- Project (with a name) has many requirements
- Requirement (with a title, description and status) belongs to a project AND has many tasks
- Task (with a description) belongs to a requirement AND has many users (through task-assignments)
- User has many tasks (through task-assignments)

Step 2. Now, we need to create the models outlined above using the Hobo generator:

```
> hobo g resource project name:string  
> hobo g resource requirement title:string \  
  body:text status:string  
> hobo g resource task name:string
```

Remember that the `hobo:resource` generator builds the entire MVC (Model/Controller/View) infrastructure needed for any model requiring a web-font end. The “task

assignments" model is simply the table required to support many-to-many relationships behind the scenes. Thus, a view or controller is not needed, so we only need the hobo model generator:

```
> hobo g model task_assignment
```

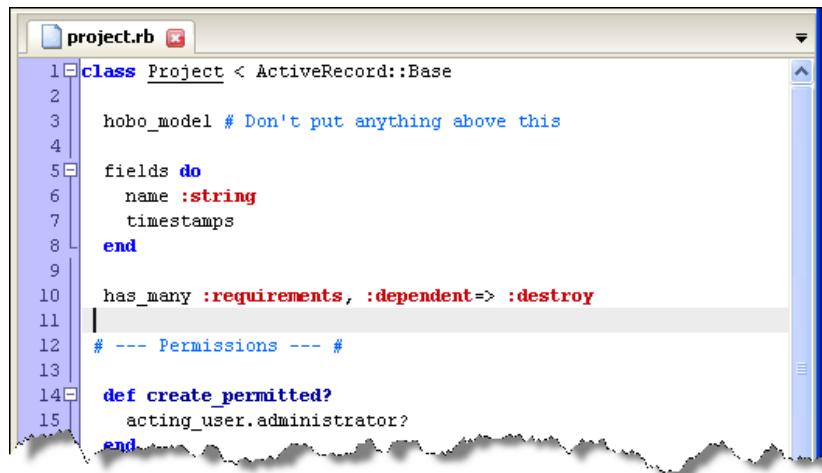
Note: We are using the convention of naming an association table with the combination of a model name with a descriptive intermediate name, with terms separated by an underscore:

task + assignment becomes: task_assignment

The field declarations have been created by the generators in each model file, but not the associations.

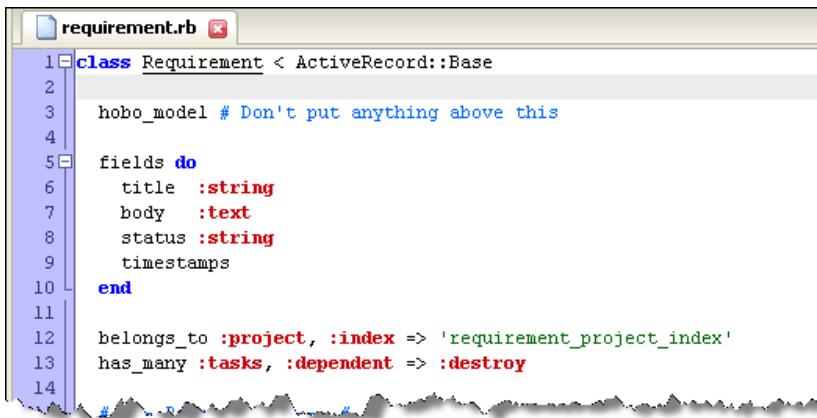
To create the associations, edit each model file as outlined below and declare the association just below the `fields do ... end` declaration in each model, as follows:

Adding "belongs_to :project" and "has_many :tasks" to the Requirement model



```
1 class Project < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   fields do
6     name :string
7     timestamps
8   end
9
10  has_many :requirements, :dependent=> :destroy
11
12  # --- Permissions ---
13
14  def create_permitted?
15    acting_user.administrator?
16  end
```

Figure 172: Adding "has_many :requirements" to the Project class



```
1 class Requirement < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   fields do
6     title :string
7     body :text
8     status :string
9     timestamps
10   end
11
12   belongs_to :project, :index => 'requirement_project_index'
13   has_many :tasks, :dependent => :destroy
14
```

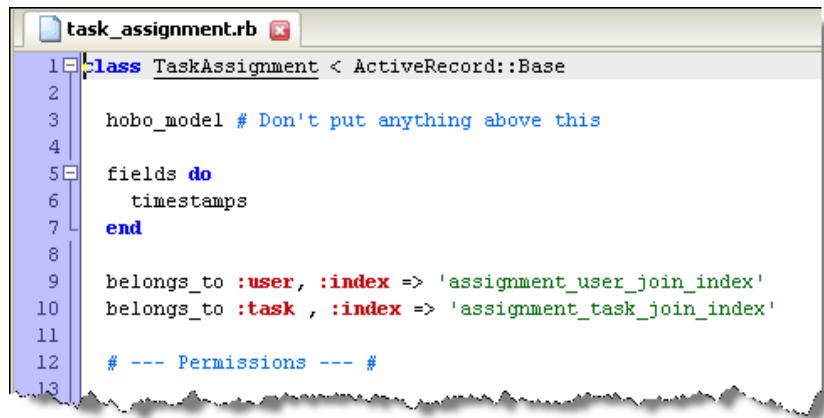
Figure 173: Adding "belongs_to :project" and "has_many :tasks" to the Requirement model

Note that we have chosen to specify the index name associated with the `belongs_to` declaration in the Requirement model. We did this in case we might want to port this app to Oracle at some point, but Oracle has this irritating limitation of 30 characters for table, column, and index names. If we had not specified the index name, Rails would have chosen a default name, which is often longer than 30 characters.



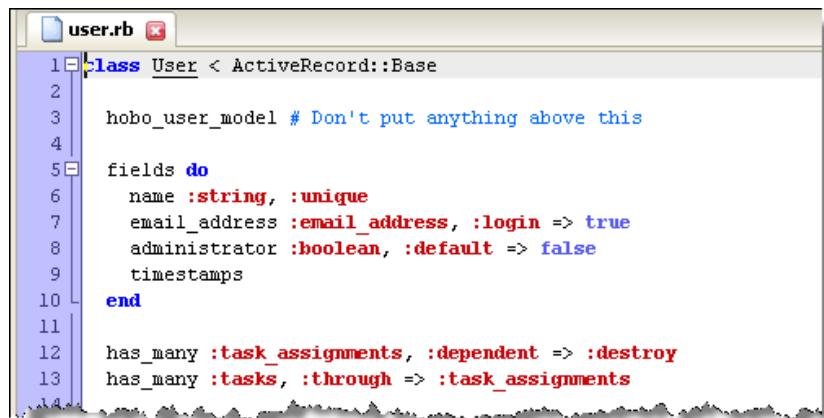
```
1 class Task < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   fields do
6     name :string
7     timestamps
8   end
9
10  belongs_to :requirement, :index => 'requirement_task_index'
11  has_many :task_assignments, :dependent => :destroy
12  has_many :users, :through => :task_assignments
13
14  # --- Permissions --- #
```

Figure 174: Adding the "belongs_to" and "has_many" declarations to the Task model



```
task_assignment.rb
1 class TaskAssignment < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   fields do
6     timestamps
7   end
8
9   belongs_to :user, :index => 'assignment_user_join_index'
10  belongs_to :task , :index => 'assignment_task_join_index'
11
12  # --- Permissions --- #
13
```

Figure 175: Adding the two "belongs_to" definitions to the TaskAssignment model



```
user.rb
1 class User < ActiveRecord::Base
2
3   hobo_user_model # Don't put anything above this
4
5   fields do
6     name :string, :unique
7     email_address :email_address, :login => true
8     administrator :boolean, :default => false
9     timestamps
10   end
11
12   has_many :task_assignments, :dependent => :destroy
13   has_many :tasks, :through => :task_assignments
14
```

Figure 176: Adding the "has_many" declarations to the User model

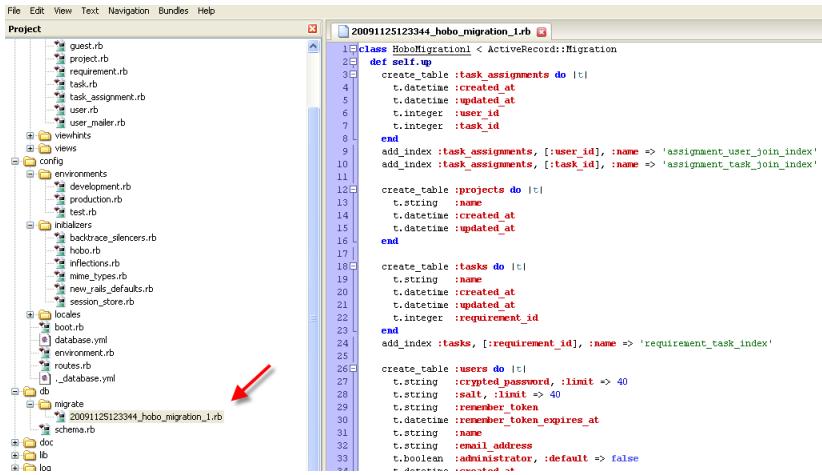
Hobo will create a single migration for all of these changes:

```
> hobo g migration
```

Load the migration file in your text editor to see what was generated:

CHAPTER 5
ADVANCED TUTORIALS

TUTORIAL 17
THE AGILE PROJECT MANAGER

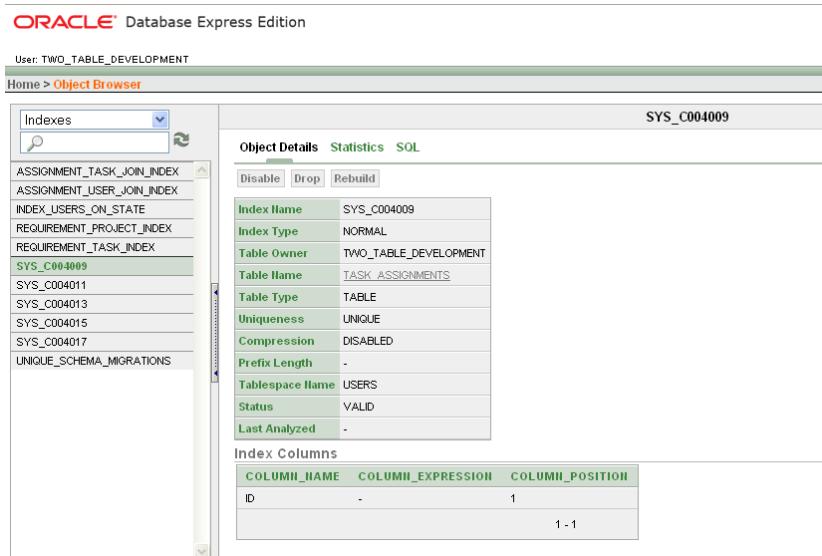


```

File Edit View Text Navigation Bundles Help
Project
  guest.rb
  project.rb
  requirement.rb
  task.rb
  task_assignment.rb
  user.rb
  user_mailer.rb
  viewhists
  views
  config
    environments
      development.rb
      test.rb
    migrations
      backtrace_splacers.rb
      hobo.rb
      inflections.rb
      mime_types.rb
      new_rals_defaults.rb
      session_store.rb
    locales
      boot.rb
      database.yml
      environment.rb
      routes.rb
    db
      migrate
        20091125123344_hobo_migration_1.rb
      schema.rb
    doc
    lib
    log
20091125123344_hobo_migration_1.rb
1: class HoboMigration1 < ActiveRecord::Migration
2:   def self.up
3:     create_table :task_assignments do |t|
4:       t.datetime :created_at
5:       t.datetime :updated_at
6:       t.integer :id
7:       t.integer :task_id
8:     end
9:     add_index :task_assignments, [:user_id], :name => 'assignment_user_join_index'
10:    add_index :task_assignments, [:task_id], :name => 'assignment_task_join_index'
11:    create_table :projects do |t|
12:      t.string :name
13:      t.datetime :created_at
14:      t.datetime :updated_at
15:      t.integer :requirement_id
16:    end
17:    create_table :tasks do |t|
18:      t.string :name
19:      t.datetime :created_at
20:      t.datetime :updated_at
21:      t.integer :requirement_id
22:    end
23:    add_index :tasks, [:requirement_id], :name => 'requirement_task_index'
24:    create_table :users do |t|
25:      t.string :encrypted_password, :limit => 40
26:      t.string :salt, :limit => 40
27:      t.string :remember_token
28:      t.datetime :remember_token_expires_at
29:      t.string :name
30:      t.string :email_address
31:      t.boolean :administrator, :default => false
32:    end
33:  end

```

Figure 177: First Hobo migration for Projects



Index Name	SYS_C004009
Index Type	NORMAL
Table Owner	TWO_TABLE_DEVELOPMENT
Table Name	TASK_ASSIGNMENTS
Table Type	TABLE
Uniqueness	UNIQUE
Compression	DISABLED
Prefix Length	-
Tablespace Name	USERS
Status	VALID
Last Analyzed	-

Index Columns	COLUMN_NAME	COLUMN_EXPRESSION	COLUMN_POSITION
ID	-	1	1 - 1

Figure 178: View of indexes created by the migration

In the figure below you can see the indexes that were created in an Oracle environment. Notice that in addition to our custom indexes, all of the tables have a unique identifier column called “ID” that is also indexed. All of these indexes start with the “SYS_” prefix.

Note: A change starting with Hobo 1.0 requires the developer to declare child models for the automatic Hobo Rapid display of child element counts as shown in the rest of this tutorial.

The code to add is *italicized* below:

```
class Project < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end

  has_many :requirements, :dependent=>:destroy

  children :requirements
  [...]
```

```
class Requirement < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    title :string
    body :text
    status :string
    timestamps
  end
  belongs_to :project, :index=>'requirement_project_index'
  has_many :tasks, :dependent=>:destroy

  children :tasks
  [...]
```

```
class Task < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end

  belongs_to :requirement, :index=>'requirement_task_index'
  has_many :task_assignments, :dependent=>:destroy
  has_many :users, :through=>:task_assignments
  children :task_assignments
  children :users
[...]
```

After you run the migration fire up the app:

```
> rails server
```

Here is what your app should look like:

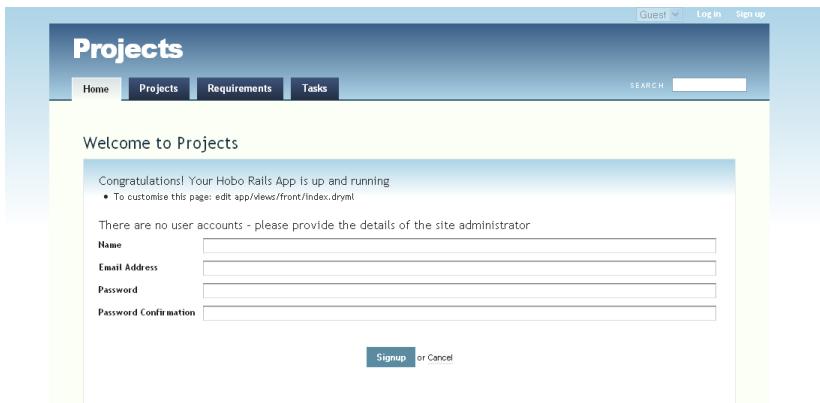
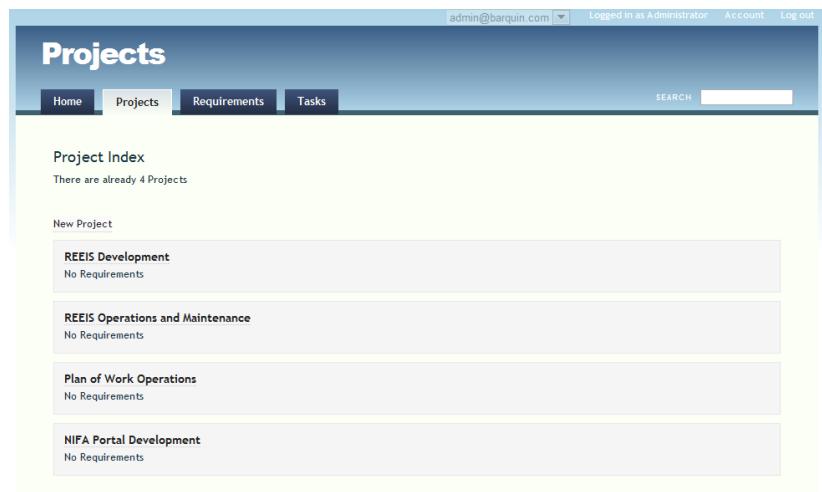


Figure 179: The default Home page for the Projects application

Make sure you create a first user, which will by default have administrator rights. Then remember to stay in as an administrator (e.g., the user who signed up first), and spend a few minutes populating the app with projects, requirements and tasks.

Enter a few projects like this:

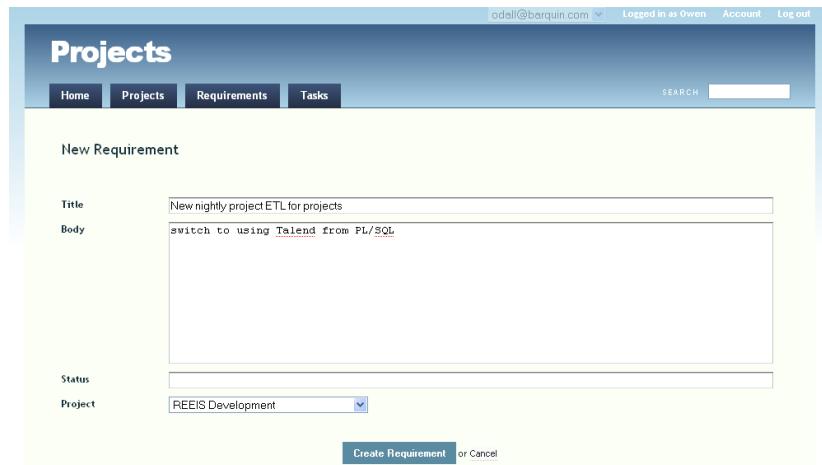


The screenshot shows the 'Projects' index page. At the top, there is a navigation bar with tabs: Home, Projects, Requirements (which is selected), and Tasks. On the right side of the header, there are links for 'admin@barquin.com' (logged in as Administrator), 'Account', and 'Log out'. Below the header, the title 'Projects' is displayed. Underneath it, a section titled 'Project Index' shows a message: 'There are already 4 Projects'. A 'New Project' button is present. Below this, four project cards are listed:

- REEIS Development**: No Requirements
- REEIS Operations and Maintenance**: No Requirements
- Plan of Work Operations**: No Requirements
- NIFA Portal Development**: No Requirements

Figure 180: The Projects index page

Enter a couple of requirements for one of your projects:



The screenshot shows the 'New Requirement' page. At the top, there is a navigation bar with tabs: Home, Projects, Requirements (selected), and Tasks. On the right side of the header, there are links for 'odall@barquin.com' (logged in as Owen), 'Account', and 'Log out'. Below the header, the title 'Projects' is displayed. Underneath it, a section titled 'New Requirement' shows a form with fields:

Title	New nightly project ETL for projects
Body	switch to using Talend from PL/SQL
Status	(empty field)
Project	REEIS Development

At the bottom of the form is a 'Create Requirement' button.

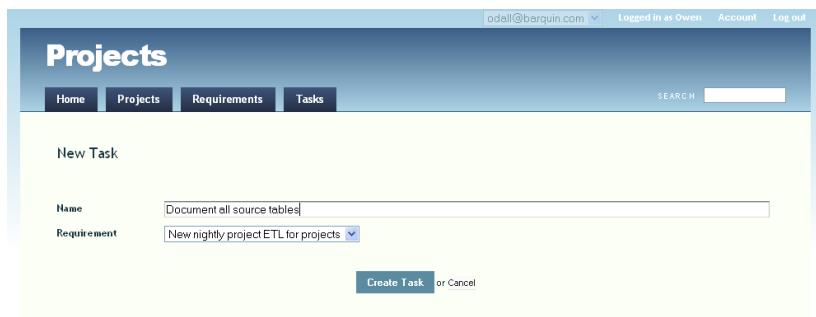
Figure 181: New Requirement page



The screenshot shows the 'Requirements' section of the application. At the top, there is a header bar with links for Home, Projects, Requirements (which is currently selected), and Tasks. On the right side of the header are links for odall@barquin.com (Logged in as Owen), Account, and Log out. Below the header is a search bar labeled 'SEARCH'. The main content area is titled 'Requirements' and displays a message: 'There are 2 Requirements'. It lists two items: 'New nightly project ETL for projects' (0 Tasks) and 'Add institution filter to the data marts' (0 Tasks). Each item has a small description and a link to its details.

Figure 182: Index view for Requirements

Enter some tasks for one of the requirements:



The screenshot shows the 'New Task' page. At the top, there is a header bar with links for Home, Projects, Requirements (selected), and Tasks. On the right side of the header are links for odall@barquin.com (Logged in as Owen), Account, and Log out. Below the header is a search bar labeled 'SEARCH'. The main content area is titled 'New Task'. It has fields for 'Name' (containing 'Document all source tables') and 'Requirement' (containing 'New nightly project ETL for projects'). At the bottom is a blue 'Create Task' button with the text 'or Cancel' next to it.

Figure 183: New Task page



Figure 184: Index view for Tasks

Using the “Application Summary” page. A handy new feature starting with Hobo 0.9.0 is the Application Summary page. If you are an administrator you can access this page by entering the following URL in your browser:

<http://localhost:3000/dev/summary>

This summary provides you quick access to information on:

- Application Name
- Application Location
- Rails Version
- Change Control (e.g., Git)
- Bundled Gems
- Plugins
- Environments
- Models/Tables
- Model Associations

The following are screen shots of the Projects application. Notice that the development environment we have been using is Oracle.

Note: The Application Summary is refreshed each time a hobo g migration is executed.

The screenshot shows the 'Projects' application summary page. At the top, there's a navigation bar with 'Guest', 'Login', and 'Signup' links, and a search bar. Below the navigation is a section titled 'Application Summary' containing the following information:

Application Name	Projects
Application Location	C:/_WORK1/hobo/rails3/projects
Rails Version	3.0.3
Mode	development

Below this is a 'Change Control' section with a single entry: 'Method other'. The next section is 'Bundled Gems', which lists various gems and their dependencies:

	Version	Dependencies
abstract	1.0.0	
actionmailer	3.0.3	actionpack (= 3.0.3, runtime) mail (>~ 2.2.9, runtime)
actionpack	3.0.3	activemodel (= 3.0.3, runtime) activesupport (= 3.0.3, runtime) builder (>~ 2.1.2, runtime) erubis (>~ 2.6.6, runtime) i18n (>~ 0.4, runtime) rack (>~ 1.2.1, runtime) rack-mount (>~ 0.6.13, runtime) rack-test (>~ 0.5.6, runtime) tzinfo (>~ 0.3.23, runtime)
activemodel	3.0.3	activesupport (= 3.0.3, runtime) builder (>~ 2.1.2, runtime) i18n (>~ 0.4, runtime)
active-record	3.0.3	activemodel (= 3.0.3, runtime) activesupport (= 3.0.3, runtime) arel (>~ 2.0.2, runtime) tzinfo (>~ 0.3.23, runtime)
activeresource	3.0.3	activemodel (= 3.0.3, runtime) activesupport (= 3.0.3, runtime)

Figure 185: Part 1 of the Application Summary page

activesupport	3.0.3	
arel	2.0.6	
builder	2.1.2	
dryml	1.3.0.pre25	actionpack (>= 3.0.0, runtime) hobo_support (= 1.3.0.pre25, runtime)
erubis	2.6.6	abstract (>= 1.0.0, runtime)
hobo	1.3.0.pre25	dryml (= 1.3.0.pre25, runtime) hobo_fields (= 1.3.0.pre25, runtime) hobo_support (= 1.3.0.pre25, runtime) rails (>= 3.0.0, runtime) will_paginate (= 3.0.pre, runtime)
hobo_fields	1.3.0.pre25	hobo_support (= 1.3.0.pre25, runtime) rails (>= 3.0.0, runtime)
hobo_support	1.3.0.pre25	rails (>= 3.0.0, runtime)
i18n	0.5.0	
mail	2.2.12	activesupport (>= 2.3.6, runtime) i18n (>= 0.4.0, runtime) mime-types (>= 1.16, runtime) treetop (> 1.4.8, runtime)
mime-types	1.16	
polyglot	0.3.1	
rack	1.2.1	
rack-mount	0.6.13	rack (>= 1.0.0, runtime)
rack-test	0.5.6	rack (>= 1.0, runtime)
rails	3.0.3	actionmailer (= 3.0.3, runtime) actionpack (= 3.0.3, runtime) activerecord (= 3.0.3, runtime) activeresource (= 3.0.3, runtime) activesupport (= 3.0.3, runtime) bundler (> 1.0, runtime) railties (= 3.0.3, runtime)
railties	3.0.3	actionpack (= 3.0.3, runtime) activesupport (= 3.0.3, runtime) rake (>= 0.8.7, runtime) thor (> 0.14.4, runtime)
rake	0.8.7	
sqlite3-ruby	1.3.2	
thor	0.14.6	
treetop	1.4.9	polyglot (>= 0.3.1, runtime)
tzinfo	0.3.23	
will_paginate	3.0.pre2	

Plugins

Environments

database		
development	oracle	XE
test	oracle	projects_test
production	oracle	projects_production

Models

Class	Table
Guest	
Project	projects
Requirement	requirements
Task	tasks
TaskAssignment	task_assignments
User	users

Project

Column	Type	
name	string	
created_at	datetime	
updated_at	datetime	
Association	Macro	Class
requirements	has_many	Requirement

Requirement

Column	Type	
title	string	
body	text	
status	string	
created_at	datetime	
updated_at	datetime	
Association	Macro	Class
project	belongs_to	Project
tasks	has_many	Task

Task		
Column	Type	
name	string	
created_at	datetime	
updated_at	datetime	
Association	Macro	Class
requirement	belongs_to	Requirement
task_assignments	has_many	TaskAssignment
users	has_many :through	User

TaskAssignment		
Column	Type	
created_at	datetime	
updated_at	datetime	
Association	Macro	Class
user	belongs_to	User
task	belongs_to	Task

User		
Column	Type	
encrypted_password	string	
salt	string	
remember_token	string	
remember_token_expires_at	datetime	
name	string	
email_address	string	
administrator	boolean	
created_at	datetime	
updated_at	datetime	
state	string	
key_timestamp	datetime	
Association	Macro	Class
task_assignments	has_many	TaskAssignment
tasks	has_many :through	Task

Figure 187: Part 4 of the Application Summary page

Removing actions

By default Hobo has given us a full set of restful actions for every single model/controller pair. However, many of these page flows (“routes”) are not optimal for our application.

For example, why would we want an index page listing every task in the database? We only really want to see *tasks listed against related requirements and users*. We need to disable the routes we don’t want.

Note: There’s an interesting change of approach here that often crops up with Hobo development. Normally you’d expect to have to build everything yourself. With Hobo, you often are given everything you want and more besides. Your job is to take away the parts that you *don’t* want

Step 1. Here’s how we would remove, for example, the index action from TasksController.

In app/controllers/tasks_controller.rb, change

```
auto_actions :all
```

To

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

Step 2. Next, refresh the browser and you'll notice that "Tasks" has been removed from the main nav-bar.



Figure 188: Effect of removing the "index" action from the Tasks controller

Note: Hobo's page generators adapt to changes in the actions that you make available.

Here's another similar trick. Browse to one of your projects that do not have related requirements. You'll see the page text says "No requirements to display":



Figure 189: View of "No Requirements to display" message

There is an "Edit Project" link, but no obvious way to add a requirement related to this project. Hobo has support for this—but we need to switch it on.

Step 3. Add the following declaration to the requirements controller:

```
auto_actions_for :project, [:new, :create]
```



Figure 190: The "New Requirement" link now appears

Hobo's page generators will respond to the existence of these routes and add a "New Requirement" link to the project page, and an appropriate "New Requirement" page:

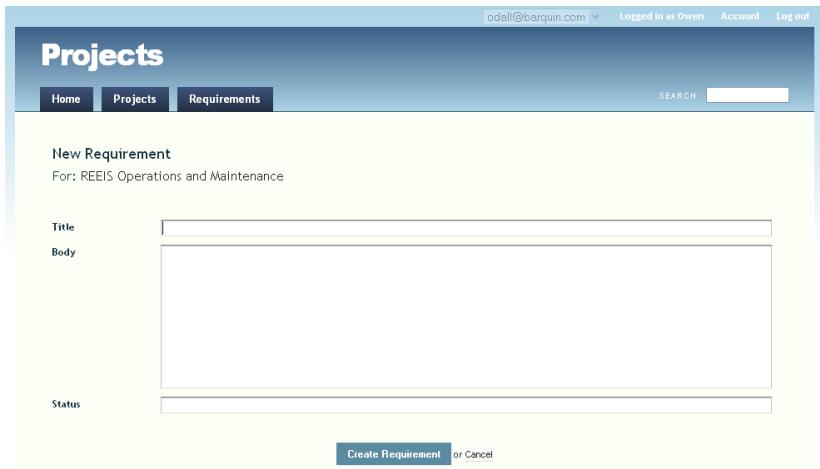


Figure 191: View of the "New Requirement" page

Step 4. Create a requirement and you'll see the requirement has the same issue with an associated task – there is no obvious way to create one. Again, we can add the `auto_actions_for` declaration to the tasks controller, but this time we'll only ask for a `:create` action, and not a new action:

```
auto_actions_for :requirement, :create
```

Hobo's page generator can support the lack of a 'New Task' page – it gives you an in-line form on the requirement page!

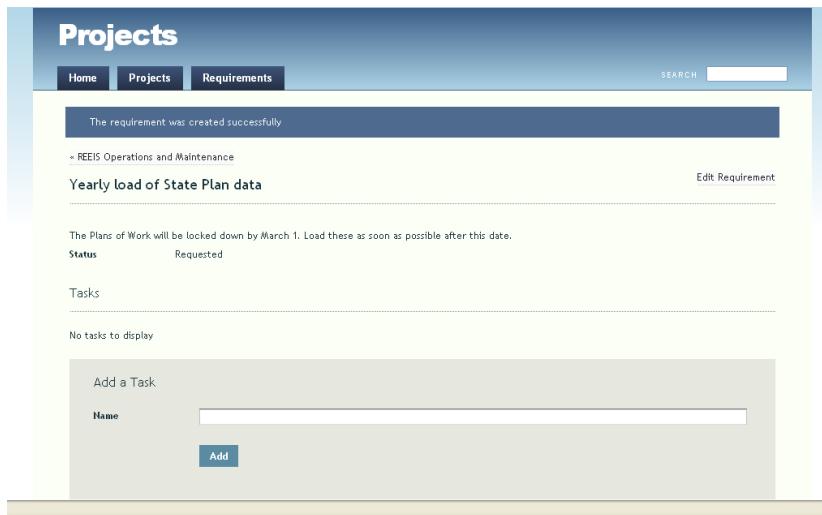


Figure 192: View of the in-line "Add a Task" form

We can continue to configure the available actions for all of the controllers. So far we've seen the “black-list” style where you list what you don't want:

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

The “white-list” style lists what you do want, e.g.:

```
auto_actions :index, :show
```

There's also a handy shortcut to get just the read-only routes (i.e., the ones that don't modify the database):

```
auto_actions :read_only
```

The opposite shortcut is handy for things that are manipulated by AJAX, but never viewed directly:

```
auto_actions :write_only
# short for -- :create, :update, :destroy
```

Step 5. Now edit each of the controllers as listed below:

```
class ProjectsController < ApplicationController
  hobo_model_controller
  auto_actions :all
end
```

```
class TasksController < ApplicationController
  hobo_model_controller
  auto_actions :write_only,:edit
  # Add the following to put an in-place editor within the
  # Requirement page
  auto_actions_for :requirement, :create
end
```

```
class RequirementsController < ApplicationController
  hobo_model_controller
  # add this to remove the Requirement tab from the main
  # navigation bar
  auto_actions :all, :except=> :index

  # add this line to get a New Requirement link for the
  # Project page
  auto_actions_for :project, [:new, :create]
end
```

Notice the Task listing within a Requirement, and the “Add a Task” in-page editor:

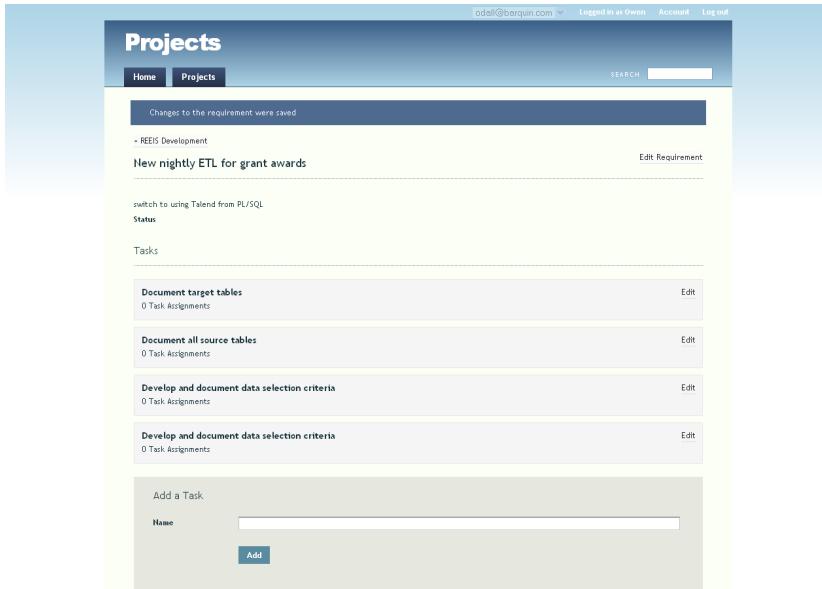


Figure 193: Requirement page after modifying controller definitions

Permissions

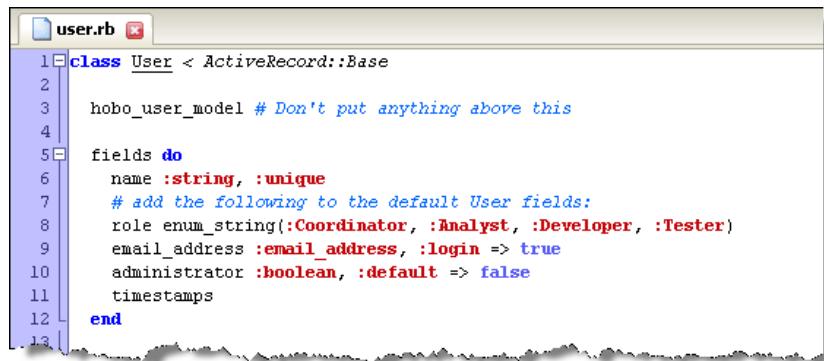
So far we've done two major things with our app:

- Created models and specified associations
- Modified controllers to specify which actions are available

One more thing we do when creating a new Hobo app. Before we even touch the view layer. Here we modify permissions in the model layer.

Adding Roles

Let's do a simple addition to the User model. Below we have taken the simple route, and created a new field called "role" along with the list of acceptable values using the Ruby `enum_string` method:

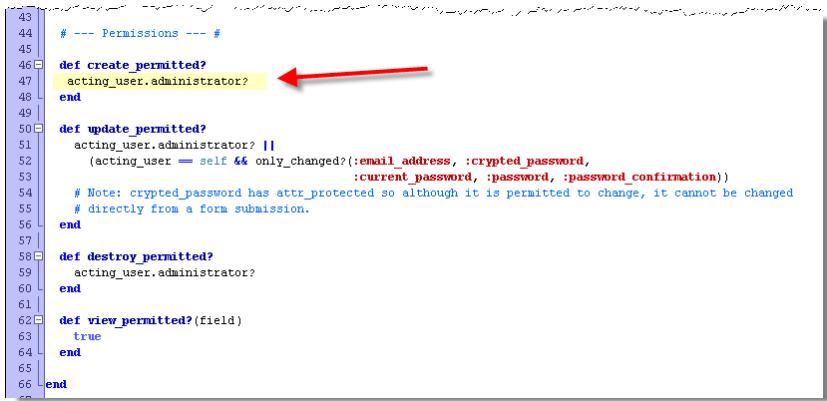


```
1 class User < ActiveRecord::Base
2
3   hobo_user_model # Don't put anything above this
4
5   fields do
6     name :string, :unique
7     # add the following to the default User fields:
8     role enum_string(:Coordinator, :Analyst, :Developer, :Tester)
9     email_address :email_address, :login => true
10    administrator :boolean, :default => false
11    timestamps
12  end
13
```

Figure 194: Defining available roles using "enum_string"

Step 1. Run a `hobo g migration` to add this field to the database.

Step 2. Modify the create permission to allow an Administrator to create a new user:



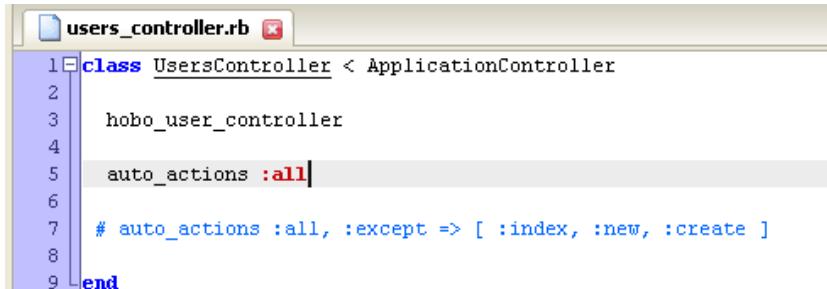
```

43 # --- Permissions --- #
44
45
46 def create_permitted?
47   acting_user.administrator? ←
48 end
49
50 def update_permitted?
51   acting_user.administrator? ||
52     (acting_user == self && only_changed?(:email_address, :encrypted_password,
53       :current_password, :password, :password_confirmation))
54   # Note: encrypted_password has attr_protected so although it is permitted to change, it cannot be changed
55   # directly from a form submission.
56 end
57
58 def destroy_permitted?
59   acting_user.administrator?
60 end
61
62 def view_permitted?(field)
63   true
64 end
65
66 end
67

```

Figure 195: Modifying the "create_permitted" method to the User model

Step 3. Modify your Users Controller as follows:



```

1 class UsersController < ApplicationController
2
3   hobo_user_controller
4
5   auto_actions :all
6
7   # auto_actions :all, :except => [ :index, :new, :create ]
8
9 end

```

Figure 196: Users Controller with "auto actions :all":

Step 4. Run the server again and then refresh your browser:



Figure 197: The Users tab is now active

Step 5. Now we can edit a user and add a role:



The screenshot shows a user interface for editing a user named 'Owen'. At the top, there's a navigation bar with 'Home', 'Projects', and 'Users' tabs, and a search bar. Below the navigation is a header 'Edit User'. On the right, there's a 'Remove This User' button. The main form contains fields for 'Name' (Owen), 'Role' (set to 'Analyst'), 'Email Address' (odell@berquin.com), and 'Administrator' (checked). At the bottom are 'Save' and 'Cancel' buttons.

Figure 198: The Edit User page with the new Role field

I have selected the “Analyst” option. So I have

1. A Hobo system permission as an Administrator
2. An Application role as Analyst.

Let’s see how to use this information.

Customizing the Permissions by Role

Here is what we would like to implement:

- Only an administrator can delete projects, requirements, or tasks
- Only an administrator or coordinator can create and edit projects, requirements, tasks or task assignments

Change your permissions in `project.rb` as follows:

```

project.rb
1 class Project < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   fields do
6     name :string
7     timestamps
8   end
9
10  has_many :requirements, :dependent=> :destroy
11  |
12  # --- Permissions --- #
13
14  def create_permitted?
15    # Make sure the user is 1) Signed up and a Coordinator or 2) is an Administrator
16    (acting_user.signed_up? && acting_user.role=="Coordinator") || acting_user.administrator?
17  end
18
19  def update_permitted?
20    # Make sure the user is 1) Signed up and a Coordinator or 2) is an Administrator
21    (acting_user.signed_up? && acting_user.role=="Coordinator") || acting_user.administrator?
22  end
23
24  def destroy_permitted?
25    false
26  end
27
28  def view_permitted?(field)
29    true
30  end
31
32 end
33

```

The code editor shows the `project.rb` file with several annotations:

- A red arrow points to the condition in the `create_permitted?` method: `(acting_user.signed_up? && acting_user.role=="Coordinator") || acting_user.administrator?`
- A red arrow points to the condition in the `update_permitted?` method: `(acting_user.signed_up? && acting_user.role=="Coordinator") || acting_user.administrator?`
- A red arrow points to the value in the `destroy_permitted?` method: `false`.

Figure 199: Adding the use of Role in Permissions

To create a project, the active user must be an administrator OR:

- The user must be signed up (not a guest) or
- The signed up user must have the role “Coordinator”

Also notice that we have entered “false” in the `destroy_permitted?` Definition. In this case, *no user can erase a project*. Deleting projects would have to be done behind the scenes in the database, or the permissions changed to clean up unwanted projects.

Now enter the same permissions (except for the `destroy_permitted?` permission) for requirements, tasks, and task assignments.

Here is the code for `project.rb` listed in the figure above:

```
class Project < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    name :string
    timestamps
  end
  has_many :requirements, :dependent=> :destroy

  # --- Permissions ---
  def create_permitted?
    # Make sure the user is 1) Signed up and a Coordinator
    # or 2) is an Adminstrator
    (acting_user.signed_up? &&
     acting_user.role=="Coordinator") ||
     acting_user.administrator?
  end
  def update_permitted?
    # Make sure the user is 1) Signed up and a Coordinator
    # or 2) is an Adminstrator
    (acting_user.signed_up? &&
     acting_user.role=="Coordinator") ||
     acting_user.administrator?
  end
  def destroy_permitted?
    false
  end
  def view_permitted?(field)
    true
  end
end
```

Permissions for data integrity

The permissions system is not just for providing operations to some users and not others. It is also used to prevent operations that don't make sense for anyone. For example, notice default UI allows requirements to be moved from one project to another. This may or may not be a sensible operation for anyone to be doing. If you want to stop this from happening, change the "update_permitted?" method in requirement.rb:

```

requirement.rb
class Requirement < ActiveRecord::Base
  hobo_model # Don't put anything above this

  fields do
    title :string
    body :text
    status :string
    timestamps
  end

  belongs_to :project, :index => 'requirement_project_index'
  has_many :tasks, :dependent => :destroy

  # --- Permissions ---
  def create_permitted?
    acting_user.administrator?
  end

  def update_permitted?
    # Make sure the user is: A. Signed up AND is a Coordinator OR B. Is an Administrator
    # If the user is a Guest, the other checks are NOT done, avoiding a fatal error
    # Prevent Requirement from being moved from one Project to another
    (acting_user.signed_up? && acting_user.role == "Coordinator") || acting_user.administrator? && !project_changed?
  end

  def destroy_permitted?
    acting_user.administrator?
  end

  def view_permitted?(field)
    true
  end
end

```

Figure 200: Modifying the “update_permitted?” method in the Requirement model

Refresh the browser and you’ll see that menu was removed from the form automatically.

Now make a similar change to prevent tasks being moved from one requirement to another in task.rb:

```

def update_permitted?
  ((acting_user.signed_up? &&
    acting_user.role == "coordinator") or
   acting_user.administrator?) &&
   !requirement_changed?
end

```

Associations

Although we have modeled the assignment of tasks to users, at the moment there is no way for the user to set these assignments. We’ll add that to the task edit page. Create a task and browse to the edit page. Notice that only the description is editable.

Hobo does provide support for “multi-model” forms, but it is not active by default. To specify that a particular association should be accessible to updates from the form, you need to declare `:accessible => true` on the association.

In `task.rb`, edit the `has_many :users` association as follows:

```
has_many :users, :through => :task_assignments,  
:accessible => true
```

Note: Without that declaration, i.e. the permission system was reporting that the association was not editable. Now that the association is “accessible”, the permission system will check for create and destroy permissions on the join model `TaskAssignment`. As long as the current user has those permissions, the task edit page will include a nice JavaScript powered control for assigning users in the edit-task page. Notice you can continue to assign users to a task and not leave the page:

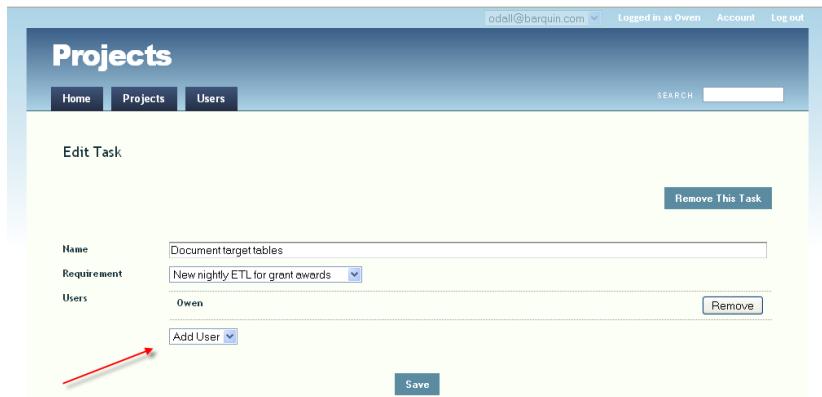


Figure 201: Assigning multiple Users to a Task in the Edit Task page

Renaming Fields and Field Help

Hobo has a great facility that makes it easy to modify the display of a field name and the field help that is displayed in the edit form. As of Hobo 1.3, this functionality has been merged into the Hobo internationalization module: i18n.

For this tutorial, we will rename the Project name field and also specify some help text for the field that will appear on the Project edit form. To do this, we need add the following to the `config/locales/app.en.yml` file:

```
en:  
  activerecord:  
    attributes:  
      project:  
        name: Project Name  
      attribute_help:  
        project:  
          name: Enter a name for the project. Make it short \  
and descriptive
```

Refresh your browser and enter a new project:



Figure 202: The New Project page using “ProjectHints”

Customizing views

It's surprising how far you can get without touching the view layer. That's the way we like to work with Hobo – get the models and controllers right and the view will probably get close to what you want. From there you can override the parts of the view that you need to.

We do that using the DRYML template language, which is part of Hobo. DRYML is tag based – it allows you to define and use your own tags right alongside the regular HTML tags. Tags are like helpers, but a lot more powerful. DRYML is quite different to other tag-based template languages, to features like the implicit context and nestable parameters. DRYML is also an extension of ERB so you can still use the ERB syntax if you are familiar with Rails.

DRYML is probably the single best part of Hobo. It's very good at high-level re-use. It allows you to make very focused changes, if a given piece of pre-packaged HTML is not exactly what you want.

Changing the Front Page

The first thing we are going to do is to change the front page. Let's change the title of the app and the default message:



Figure 203: The default application name and welcome message

To change the application name, edit `/config/application.rb`:

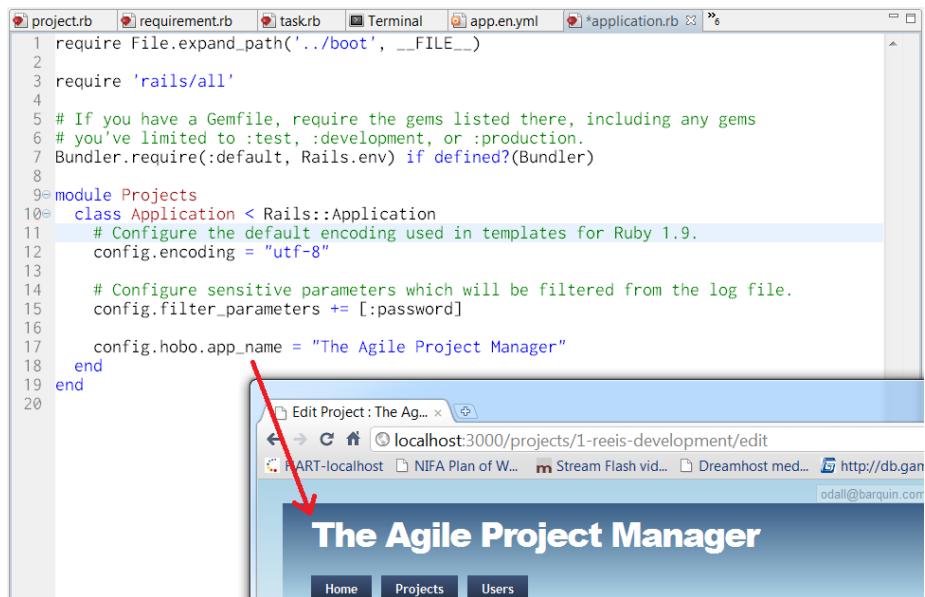


Figure 204: Changing the application name in "config/application.rb"

Changing the `app-name` value for the `app-name` tag here will change it anywhere that tag is used throughout the application.

Now let's change the rest of the page...

Bring up /app/views/front/index.dryml in your editor:



```
1 <page title="Home">
2   <body: class="front-page"/>
3
4   <content:>
5     <header class="content-header">
6       <h1>Welcome to <app-name/></h1>
7       <section class="welcome-message">
8         <h3>Congratulations! Your Hobo Rails App is up and running</h3>
9         <ul>
10          <li>To customise this page: edit app/views/front/index.dryml</li>
11        </ul>
12      </header>
13
14      <% if User.count == 0 -%>
15        <h3 style="margin-top: 20px;">There are no user accounts - please provide the details of the site administrator</h3>
16        <form with=&quot;&lt;this || User.new&gt;&quot; without-cancel>
17          <field-list: fields="name, email_address, password, password_confirmation"/>
18          <submit: label="Register Administrator"/>
19        </form>
20      <% end -%>
21
22
23      <section>
24        </header>
25
26      <section class="content-body">
27        </section>
28    </content:>
29
30  </page>
31
```

Figure 205: Modifying “front\index.dryml”

This is what it looks like before you change it. Change it to the following:

```
<page title="Home">
  <body: class="front-page"/>
  <content:>
    <header class="content-header">
      <h1>Powered by Hobo</h1>
      <section class="welcome-message">
        <h3>Here is what you can do:</h3>
        <ul>
          <li>Create and maintain any number of Projects</li>
          <li>Associate Requirements to each Project</li>
          <li>Assign Tasks and assign people to complete each Task</li>
        </ul>
      </section>
    </header>
  </content:>
</page>
```

Refresh your browser:



Figure 206: Home page modified by changing "/front/index.dryml"

Add Assigned Users to the Tasks

Currently the only way to see who's assigned to a task is to click the task's edit link. It would be better to add a list of the assigned users to each task when we're looking at a requirement.

DRYML has a feature called "polymorphic" tags. These are tags that are defined differently for different types of objects. Rapid makes use of this feature with a system of "cards". The tasks that are displayed on the requirement page are rendered by the <card> tag.

You can define custom cards for particular models. Furthermore, if you call the base <card> you can define your card by tweaking the default, rather than starting from scratch. This is what DRYML is all about. It's like a smart-bomb, capable of taking out little bits of unwanted HTML with pin-point strikes and no collateral damage.

The file app/views/taglibs/application.dryml is a place to put tag definitions that will be available throughout the site. Add this definition to that file:

```
<extend tag="card" for="Task">
  <old-card merge>
    <append-body:>
      <div class="users">
        Assigned users:
        <repeat:users join=","><a>/</a></repeat>
        <else>None</else>
      </div>
    </append-body:>
  </old-card>
</extend>
```



```

1 <include src="rapid" plugin="hobo"/>
2
3 <include src="taglibs/auto/rapid/cards"/>
4 <include src="taglibs/auto/rapid/pages"/>
5 <include src="taglibs/auto/rapid/forms"/>
6
7 <set-theme name="clean"/>
8
9 <extend tag="card" for="Task">
10 <old-card merge>
11   <append-body:>
12     <div class="users">
13       Assigned users: <repeat:users join="," ><a/></repeat><else>None</else>
14     </div>
15   </append-body:>
16 </old-card>
17 </extend>
18

```

Figure 207: Extending the card tag for Task in "application.dryml"

Refresh the requirement page. You'll see that in the cards for each task there is now a list of assigned users. The users are clickable - they link to each user's home page (which doesn't have much on it at the moment).

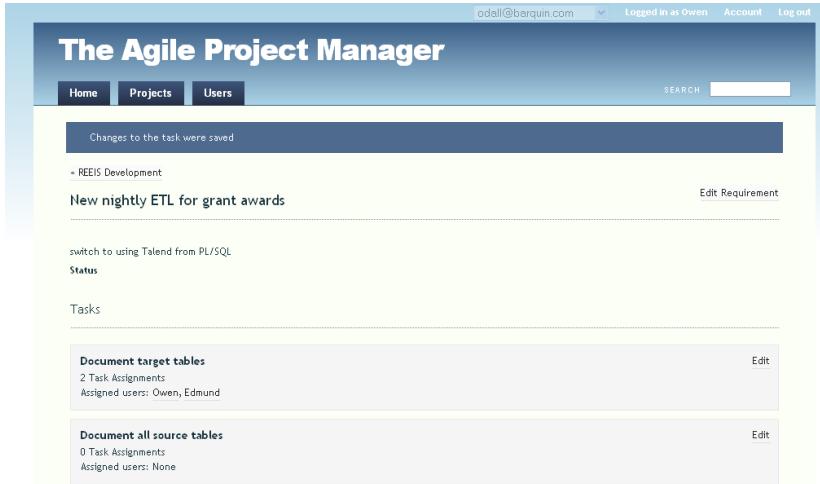


Figure 208: Viewing assigned users on a the Task card

The `<extend>` tag is used to extend any tag that's already defined. The body of `<extend>` is our new definition. It's very common to want to base the new definition on the old one. For example, we often want to insert a bit of extra content as we've done here.

We can do that by calling the “old” definition, which is available as `<old-card>`. We've passed the `<append-body:>` parameter to `<old-card>` which is used to append content to the body of the card.

Some points to note:

The <repeat> tag provides a join attribute that we use to insert the commas. The link is created with a simple empty <a/>. It links to the ‘current context’, which, in this case, is the user. The :users in <repeat:users> switches the context. It selects the users association of the task.

DRYML has a multi-purpose <else> tag. When used with repeat, it provides a default for the case when the collection is empty.

Add a Task Summary to the User’s Home Page

Since each task provides links to the assigned users, the user’s page is not looking great. Rapid has rendered cards for the task-assignments but there’s no meaningful content in them. What we’d like to see is a list of all the tasks the user has been assigned to. Having them grouped by requirement would be helpful too.

To achieve this we want to create a custom template for users “show” page. If you look in app/views/users you’ll see that it’s empty. When a page template is missing, Hobo tries to fall back on a defined tag. For a ‘show’ page, that tag is <show-page>. The Rapid library provides a definition of <show-page>, so that’s what we’re seeing at the moment.

As soon as we create app/views/users/show.dryml, that file will take over from the generic <show-page> tag. Try creating that file and just throw “Hello!” in there for now. You should see that the user’s show page now displays just “Hello!” and has lost all of the page styling.

If you now edit show.dryml to read ”<show-page/>” you’ll see we’re back where we started. The <show-page> tag is just being called explicitly instead of by convention.

Rapid has generated a custom definition of <show-page for="User">. You can find this in app/views/taglibs/auto/rapid/pages.dryml.

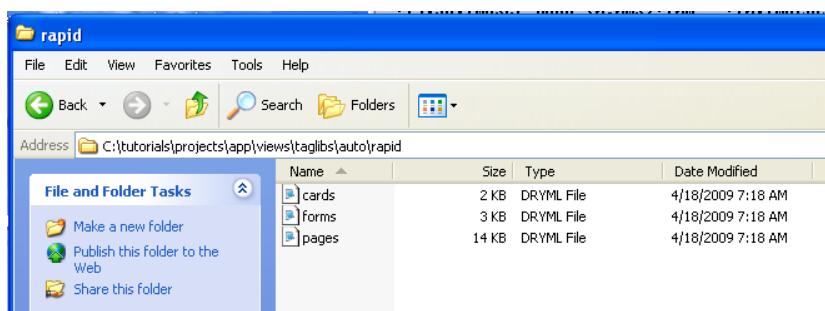


Figure 209: Listing the contents for the "app\views\taglibs\auto\rapid" folder

Warning: *Don't edit this file!* Your changes will be overwritten. Instead, use this file as a reference so you can see what the page provides, and what parameters there are (the param attributes).

Here is the top of the file:

```
1 [!-- AUTOMATICALLY GENERATED FILE - DO NOT EDIT -->
2
3 <!-- ===== Main Navigation ===== -->
4
5 <def tag="main-nav">
6   <navigation class="main-nav" merge-attrs param="default">
7     <nav-item href="#{base_url}/home</nav-item>
8     <nav-item with="#{&Project}"><ht key="project.nav_item" count="100"><model-name-human count="100"/></ht></nav-item>
9     <nav-item with="#{&User}"><ht key="user.nav_item" count="100"><model-name-human count="100"/></ht></nav-item>
10   </navigation>
11 </def>
12
13
14
15
16 <!-- ===== Project Pages ===== -->
17
18 <def tag="index-page" for="Project">
19   <page merge title="#{ht 'project.index.title', :default=>[model.model_name.human(:count=>100)] }">
20     <body class="index-page project" param>
21
22       <content: param>
23         <header param="content-header">
24           <h2 param="heading">
25             <ht key="projekt:index.heading">
26               <model-name-human model="#{&model}"/>
27             </ht>
28           </h2>
29         </header>
30       </content:>
31     </body>
32   </page>
33 </def>
```

Figure 210: contents of the pages.dryml file

Find the “show-page” tag for User:

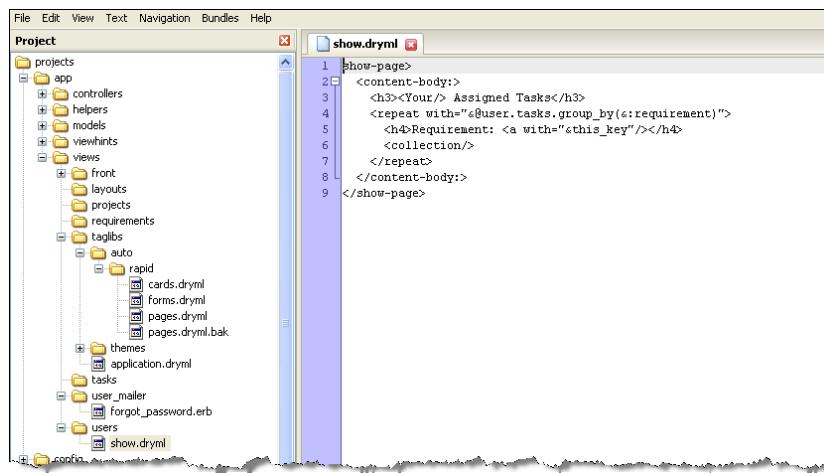
```
| application.dyml | pages.dyml |
```

```
530
531
532 <def tag="show-page" for="User">
533   <page merge title="#(ht 'user.show.title', :default>['User'])">
534
535     <body class="show-page user" param>
536
537       <content param="args">
538         <header param="content-header">
539           <a:tasks param="parent-link">&laquo; <ht key="user.actions.back_to_parent" parent="Task" name="&this">Back to <name></ht><a:tasks>
540
541           <h2 param="heading">
542             <key="user.show.heading" name="&this.respond_to?(:name) ? this.name : ''">
543               <name>
544             </name>
545           </h2>
546
547           <record-flags fields="administrator" param>
548
549             <a action="edit" if="&can_edit?" param="edit-link">
550               <ht key="user.actions.edit" name="&this.respond_to?(:name) ? this.name : ''">
551                 Edit User
552               </ht>
553             </a>
554           </header>
555
556           <section param="content-body">
557             <field-list fields="role, email_address, state" param>
558           </section>
559         </content>
560
561       </page>
562     </page>
563   </def>
564
```

Figure 211: The auto-generated "show-page" tag for User in "pages.dryml"

Now let's get the content we're after - the user's assigned tasks, grouped by requirement. It's only five lines of markup to put in a file `\views\users\show.dryml`.

```
<show-page>
  <content-body:>
    <h3><Your/> Assigned Tasks</h3>
    <repeat with="@user.tasks.group_by(:requirement)">
      <h4>Requirement: <a with="this_key"/></h4>
      <collection/>
    </repeat>
  </content-body:>
</show-page>
```



This will override the definition in `pages.dryml` and display a page similar to the following:

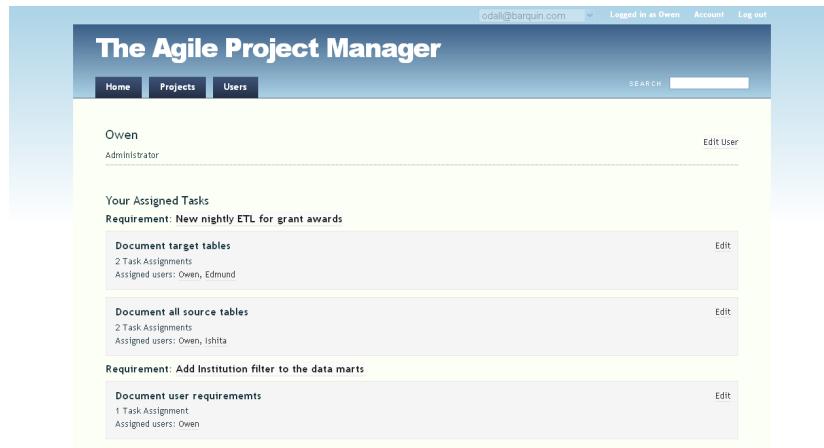


Figure 212: View of the enhanced User "show-page"

The <Your> tag is a handy gadget. It outputs “Your” if the context is the current user. Otherwise it outputs the user’s name. You’ll see “Your Assigned Tasks” when looking at yourself, and “Fred’s Assigned Tasks” when looking at Fred.

We’re using <repeat> again, but this time we’re setting the context to the result of a Ruby expression (with="&...expr..."). The expression:

```
@user.tasks.group_by(&:requirement)
```

gives us the grouped tasks. Inside the “repeat this” (the implicit context) will be an array of tasks, and `this_key` will be the requirement.

So gives us a link to the requirement. <collection> is used to render a collection of anything in a list. By default it renders <card> tags. To change this, just provide a body to the <collection> tag. Click on the Users tab to see a summary of tasks for all users:



Figure 213: The Users tab showing all assignments

Now, you can get the big picture of *all* user assignments.

This is a lot to take in all at once. The main idea here is to give you an overview of what’s possible. See The DRYML Guide for more in-depth information:

<http://cookbook.hobocentral.net/manual/dryml-guide>

Improve the Project Page with a Searchable, Sortable table

The project page is currently workable, but we can easily and vastly improve it. Hobo Rapid provides a tag called <table-plus> which:

- Renders a table with support for sorting by clicking on the headings
- Provides a built-in search bar for filtering the rows displayed

- Searching and sorting are done server-side so we need to modify the controller as well as the view for this enhancement.

As with the user's show-page, to get started put a simple call to <show-page/> in app/views/projects/show.dryml

To see what this page is doing, take a look at

```
<def tag="show-page" for="Project">
```

in pages.dryml. (app/views/taglibs/auto/rapid).

Notice this tag:

```
<collection:requirements param/>
```

That's the part we want to replace with the table. Note: that when a param attribute doesn't give a name, the name defaults to the same name as the tag.

Here's how we would replace that <collection> with a simple list of links:

```
<show-page>
  <collection: replace>
    <div>
      <repeat:requirements join=" " ><a/></repeat>
    </div>
  </collection:>
</show-page>
```

You should see that in place of the requirement cards, we now get a simple comma-separated list of links to the requirements. Not what we want of course, but it illustrates the concept of replacing a parameter. Here's how we get the "table-plus":

```
<show-page>
  <collection: replace>
    <table-plus:requirements fields="this, status">
      <empty-message:>
        No requirements match your criteria
      </empty-message:>
    </table-plus>
  </collection:>
</show-page>
```

The fields attribute to <table-plus> lets you specify a list of fields that will become the columns in the table. We could have specified fields="title, status" which would have given us the same content in the table, but by saying this, the first column contains *links* to the requirements, rather than just the title as text.

We could also add a column showing the number of tasks in a requirement. Change to `fields="this, tasks.count, status"` and see that a column is added with a readable title “Tasks Count”.

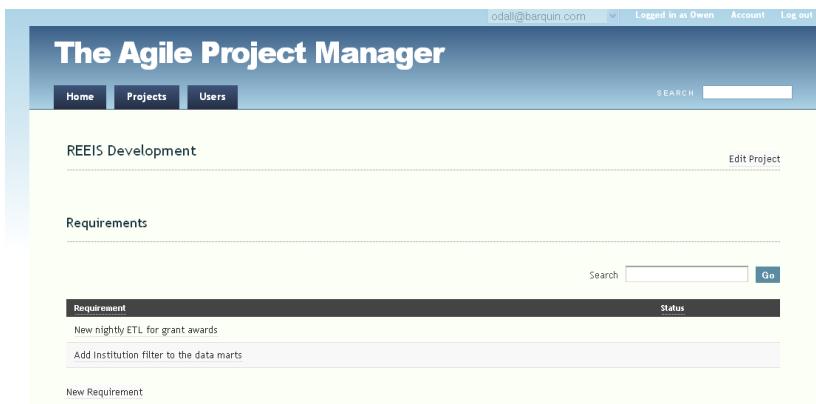


Figure 214: Using the Hobo “<table-plus>” feature to enhance the Requirements listing

To get the search feature working, we need to update the controller side. Add a show method to `app/controllers/projects_controller.rb` like this:

```
def show
  @project = find_instance
  @reqlist = @project.requirements.where(["title like ?",
    "%#{params[:search]}%"]).order(parse_sort_param(:title,
      :status).join(' '))
end
```

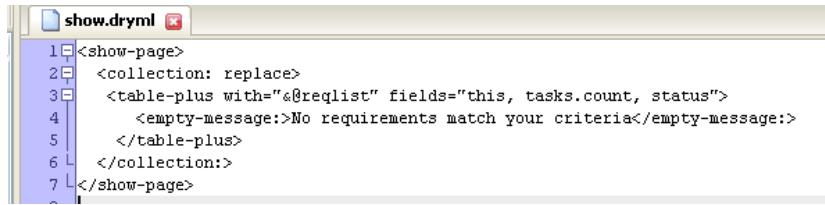
What we are doing is creating two instance variables that will hold the values in memory between the controller and view.

- `@project` = Holds the information for the project that has just been clicked
- `@reqlist` = A variable name we chose to hold the list of projects returned by the `apply_scopes` method.

If there are no values in the search `params`, all requirements for that project are returned. The first time the projects page is loaded `params` will be null.

Then get the `<table-plus>` to use `@requirements`:

```
<table-plus with="&@reqlist" fields="this, tasks.count, status">
```



```

1 <show-page>
2   <collection: replace>
3     <table-plus with="@reqlist" fields="this, tasks.count, status">
4       <empty-message:>No requirements match your criteria</empty-message:>
5     </table-plus>
6   </collection:>
7 </show-page>

```

Figure 215: Enhancing the <table-plus> listing

Enter a word in the Search box and see how the requirement list is filtered:



The screenshot shows a web application titled "The Agile Project Manager". The top navigation bar includes links for Home, Projects, Users, and a search bar. The main content area is for a project named "REEIS Development". A sub-header "Requirements" is followed by a table with three columns: Requirement, Tasks Count, and Status. A red arrow points from the search bar to the first requirement row, which contains the text "New nightly ETL for grant awards". The search bar has the word "grant" typed into it.

Figure 216: Using a search within the Requirements listing

Other Enhancements

We're going to work through some easier, but very valuable enhancements to the application. We will add:

- A menu for requirement statuses. We'll do this with a hard-wired set of options, and then add the ability to manage the set of available statuses.
- Filtering of requirements by status on the project page
- Drag and drop re-ordering of tasks for easy prioritization.
- Rich text formatting of requirements. This is implemented by changing one symbol in the source code and adding the CKEditor plugin.

Requirement Status Menu

We're going to do this in two stages. First using a fixed menu that will require a source-code change, if you ever need to alter the available statuses. We'll then remove

that restriction by adding a `RequirementStatus` model. We'll also see the migration generator in action again.

The fixed menu is very simple. Locate the declaration of the status field in `requirement.rb` (it's in the `fields do ... end` block), and change it to this:

```
status enum_string(:proposed, :accepted, :rejected,
                   :reviewing, :developing, :completed)
# etc..
```

Now, the “Edit Requirement” page looks like this, with a select list:

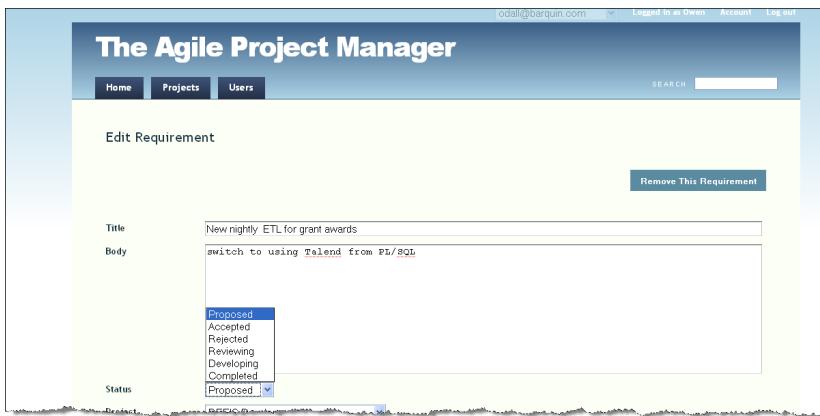


Figure 217: The Edit Requirement form with selectable status codes

The menu is working in the “edit requirement” now. It would be nice though if we had an “AJAX-ified” editor on the requirement page. Edit the file `app/views/requirements/show.dryml` to be:

```
<show-page>
  <field-list: tag="editor"/>
</show-page>
```

Now, the page has an in-place editor that does not require a submit button update.

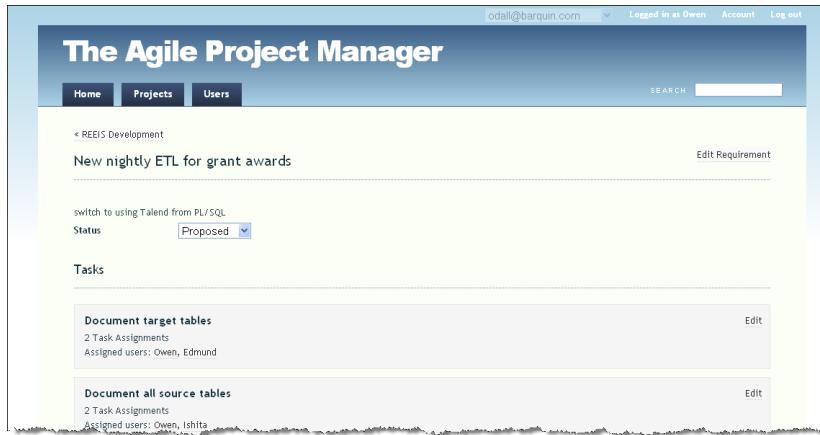


Figure 218: Creating an AJAX status update for Requirements

Simply select the new status, and a save is automatically executed via an AJAX call.

How did Hobo do that? <show-page> uses a tag called <field-list> to render a table of fields. DRYML's parameter mechanism allows the caller to customize the parameters that are passed to <field-list>.

On our requirement page the field-list contains only the status field. By default <field-list> uses the <view> tag to render read-only views of the fields, but that can be changed by passing a tag name to the tag attribute. We're passing the name "editor" which is a DRYML tag for creating AJAX-style in-place editors.

Create a Configurable Status List

In order to support management of the statuses available, we'll create a Requirement Status model:

```
> hobo g resource requirement_status name:string
```

Whenever you create a new model and controller with Hobo, get into the habit of thinking about permissions and controller actions.

In this case, we probably want only administrators to be able to manage the permissions. As for actions, we probably only want the write actions, and the index page:

```
auto_actions :write_only, :new, :index
```

Remove the status field from the fields do ... end block in the "Requirement" model and add the following association declaration:

```
belongs_to :status, :class_name => "RequirementStatus",
:index => 'requirement_status_index'
```

Run the migration generator

```
> hobo g migration
```

You'll see that the migration generator considers this change to be ambiguous and will prompt you for an action.

Note: Whenever there are columns removed and columns added, the migration generator can't tell whether you're actually removing one column and adding another, or if you are renaming the old column. It's also pretty fussy about what it makes you type. We really don't want to play fast and loose with precious data.

To confirm that you want to drop the 'status' column, you have to type in full: "drop status".

Once you've done that you'll see that the generated migration includes the creation of the new foreign key and the removal of the old status column.

That's it! The page to manage the requirement statuses should appear in the main navigation.

We've decided to revise our list while entering them using the "New Requirement Status" page:



The Agile Project Manager

odall@barquin.com Logged in as Owen Account Log out

The requirement status was created successfully.

Requirement Statuses

There are 8 Requirement Statuses

New Requirement Status

Accepted	X
Proposed	X
Reviewing	X
Rejected	X
Accepted	X
Developing	X
Testing	X
Completed	X

Figure 219: Requirement Status view

Now that we've got more structured statuses, let's do something with them...

Reordering Tasks

We're going to add the ability to re-order a requirement's tasks by drag-and-drop. There's support for this built into Hobo, so there's not much to do. First, we need the `acts_as_list` plugin:

```
> rails plugin install git://github.com/rails/acts_as_list.git
```

We will make two changes to our models:

Task needs:

```
acts_as_list :scope => :requirement
```

Requirement needs a modification to the `has_many :tasks` declaration:

```
has_many :tasks, :dependent => :destroy, :order => :position
```

The migration generator knows about the `acts_as_list` plugin. Run it and you'll get the new position column on Task which is needed to keep track of ordering for you.

```
> hobo g migration
```

Now refresh the application...

You'll notice a slight glitch – the tasks position has been added to the new-task and edit-task forms. Fix this by customizing the Task form.

In `application.dryml` add:

```
<extend tag="form" for="Task">
  <old-form merge>
    <field-list: fields="name, users"/>
  </old-form>
</extend>
```

On the “task edit” page you might also have noticed that Hobo Rapid didn't manage to figure out a destination for the “cancel” link. You can fix that by editing `tasks/edit.dryml` to be:

```
<edit-page>
  <form:>
    <cancel: with="&this.requirement"/>
  </form:>
</edit-page>
```

This is a good demonstration of DRYML's nested parameter feature. The `<edit-page>` makes its form available as a parameter, and the form provides a `<cancel:>` parameter.

We can drill down from the edit-page to the form and then to the cancel link to pass in a custom attribute. You can do this to any depth.

Adding a “Due Date” to Tasks

Let's first add a good library of date and time validations:

- Add the following line to the end of the Gemfile file in your application root directory:

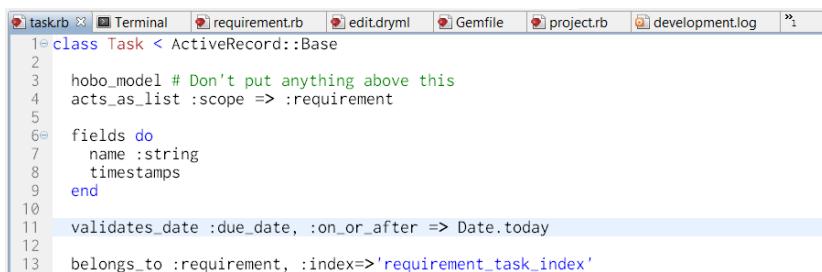
```
gem "validates_timeliness"
```

- Then run the following command in your console:

```
> bundle install
```

Now update your “Task” model with a due date and add this validation for that date field:

```
validates_date :due_date, :on_or_after => Date.today
```



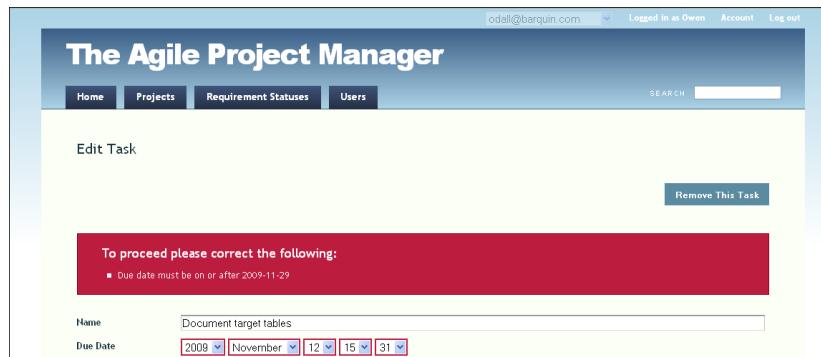
A screenshot of a terminal window titled "taskrb". The window contains the following Ruby code for the Task model:

```
1< class Task < ActiveRecord::Base
2
3  hobo_model # Don't put anything above this
4  acts_as_list :scope => :requirement
5
6< fields do
7  name :string
8  timestamps
9 end
10
11 validates_date :due_date, :on_or_after => Date.today
12
13 belongs_to :requirement, :index=>'requirement_task_index'
```

Figure 220: Task model with "due_date" and a validation for the date

In application.dryml add the new “due_date” field:

```
<extend tag="form" for="Task">
  <old-form merge>
    <field-list: fields="name, due_date, users"/>
  </old-form>
</extend>
```



The screenshot shows a web application titled "The Agile Project Manager". The top navigation bar includes links for Home, Projects, Requirement Statuses, and Users, along with a search bar and user account information. A red error message box displays the text: "To proceed please correct the following:" followed by a single bullet point: "Due date must be on or after 2009-11-29". Below the message box, there are input fields for "Name" (set to "Document target tables") and "Due Date" (set to "2009 November 12"). A "Remove This Task" button is located in the top right corner of the main content area.

Figure 221: Error message from trying to enter a date earlier than today

Tutorial 18

Using CKEditor (Rich Text) with Hobo

By Tola Awofolu

Tutorial Application: projects

Overview

CKEditor is the new rich text editor that replaces the popular FCKeditor used by many web developers.

To use CKEditor (3.x):

Download CKEditor from the download website: <http://www.ckeditor.com>

Extract the downloaded zip file, (ckeditor_3.5.zip or ckeditor_3.5.tar.gz at the time of this writing) to a new directory, public/javascripts/ckeditor in your Hobo application from the website:

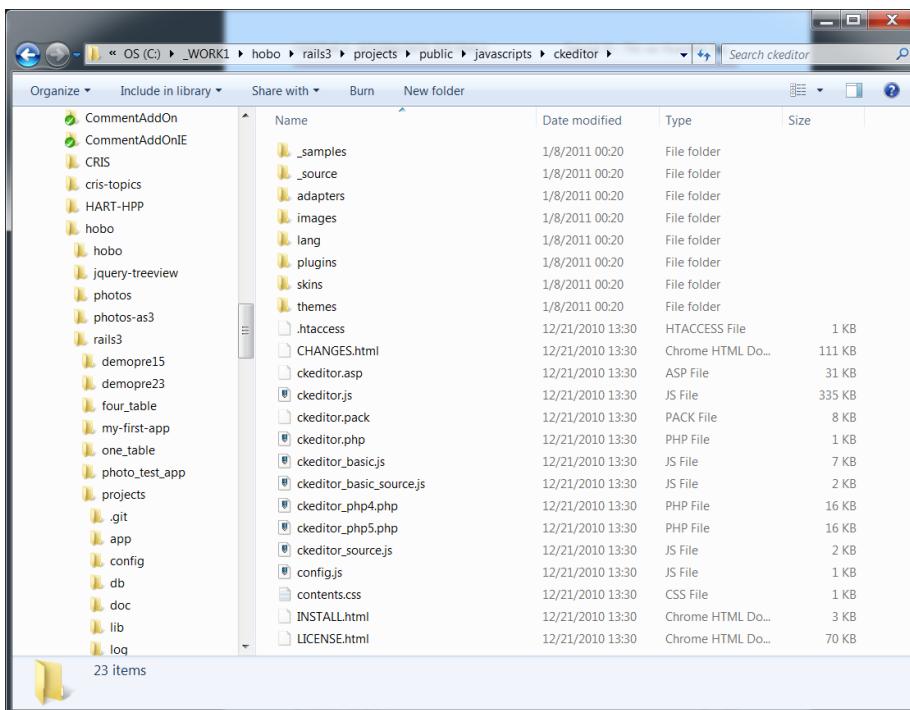


Figure 222: CKEditor source folder listing

Add the following file, `load_ckeditor.js`, to the `public/javascripts` directory of your Hobo application:

```
HoboCKEditor = {
    newEditor : function(elm, buttons) {
        if (elm.name != "") {
            oInstance = CKEDITOR.replace( elm.name ,
                { toolbar : HoboCKEditor.standardToolbarConfig || buttons }
            );
            oInstance.setData( elm.value );
            oInstance.resetDirty();
        }
        return oInstance;
    },
    makeEditor : function(elm) {
        if (!elm.disabled && !elm.readOnly){
            HoboCKEditor.newEditor(elm);
        }
    },
    standardToolbarConfig: [ ['DocProps','-', 'Preview', ' ', 'Templates'],
        ['Cut','Copy','Paste','PasteText','PasteWord','-', 'Print','SpellCheck'],
        ['Undo','Redo','-', 'Find','Replace','-', 'SelectAll','RemoveFormat'],
        [],
        '/',
        ['Bold','Italic','Underline','StrikeThrough','-', 'Subscript','Superscript'],
        ['OrderedList','UnorderedList','-', 'Outdent','Indent','Blockquote'],
        ['JustifyLeft','JustifyCenter','JustifyRight','JustifyFull'],
        ['Link','Unlink'],
        ['Image','Rule','SpecialChar','PageBreak'],
        '/',
        ['Style','FontFormat','FontName','FontSize'],
        ['TextColor','BGColor'],
        ['FitWindow','ShowBlocks','-', 'About'] ]
    }
}
Hobo.makeHtmlEditor = HoboCKEditor.makeEditor
```

Note: The code listed above has line wrapping because of the width of the paper.
Please remove any hard return characters in your code.

Notice that the “standardToolbarConfig” portion of this JavaScript customizes the CKEditor toolbar options. Read the CKEditor documentation if you wish to add more.

This code also replaces the normal text box with the rich-text editor, as long as the text box is an HTML “`textarea`” tag that includes this HTML attribute in the tag definition.

Here’s an example of HTML markup that is created by Hobo:

```
<textarea id=“contact[notes]” class=“contact large”/>
```

This HTML markup is automatically generated by Hobo for fields defined with the `:html` symbol in the model. Open your `app/models/requirement.rb` file and

change the ‘body’ field to :html:



```
1 class Requirement < ActiveRecord::Base
2   hobo_model # Don't put anything above this
3
4   fields do
5     title :string
6     body :html
7     timestamps
8   end
9
10 belongs_to :project, :index => 'requirement_project_index'
11
```

Figure 223: Using the “:html” field option to trigger rich-text editing

Add the following lines of code to app/views/taglibs/application.dryml:

```
<extend tag="page">
  <old-page merge>
    <after-scripts:>
      <javascript name="ckeditor/ckeditor"/>
      <javascript name="load_ckeditor"/>
    </after-scripts:>
  </old-page>
</extend>
```

Now access the “Edit” page for any requirement you have defined and the body field should look as follows:

The screenshot shows a Hobo application interface titled "The Agile Project Manager". The main title bar includes the application name and navigation links for Home, Projects, Requirement statuses, and Users. A search bar and user account information (odall@barquin.com, Logged in as Owen, Account, Log out) are also present. The current page is titled "Edit Requirement". On the left, there are fields for "Title" (containing "New nightly project ETL for projects") and "Body" (containing "switch to using Talend from PL/SQL"). The "Body" field is a CKEditor rich text editor, indicated by its toolbar and the ability to switch between rich text and plain text modes. Below the CKEditor is a dropdown menu for "Status" with the option "(No Requirement status)". At the bottom right of the form are buttons for "Save Requirement" and "Cancel".

Figure 224: Sample Hobo form using CKEditor

Tutorial 19

Using FusionCharts with Hobo

By *Marcelo Giorgi*

Overview

Presenting data in a visual informative way is a powerful. A widely-used charting and graphing library that includes lots of special effects is FusionCharts from InfoSoft (<http://www.fusioncharts.com/>).

FusionCharts offers a wide range of flash components for rendering data-driven charts, graphs, and maps. The way to feed those flash components with our data is to create an XML file (with a specific format and semantics understood by FusionCharts) and then set the URL for that file so that the Flash component (running on the client browser) can reach it.

In this tutorial we will continue with the `four_table` project completed earlier, so we can leverage the existing models and focus on the chart's functionality.

We'll be adding two charts to the project:

- Recipes By Country (which counts the number of recipes for each country)
- Recipes by Category (which counts the number of recipes in each category)

Configuring FusionCharts for our Hobo application

The first thing we need to do is download the trial version of FusionCharts Version 3. Go to the URL <http://www.fusioncharts.com/Download.asp> and submit the form as shown below:

Figure 225: Registration form to request FusionCharts

Figure 226: Download page for FusionCharts

Click on the ‘Download FusionCharts v3 (Evaluation)’ link, save and unzip the file into a safe location such as, c:\FusionChartsDistribution.

Next:

1. Create a new folder under the Hobo “public” folder called FusionCharts. Copy all the swf files contained in the folder c:\FusionChartsDistribution\Charts to folder you created: \four_table\public\FusionCharts

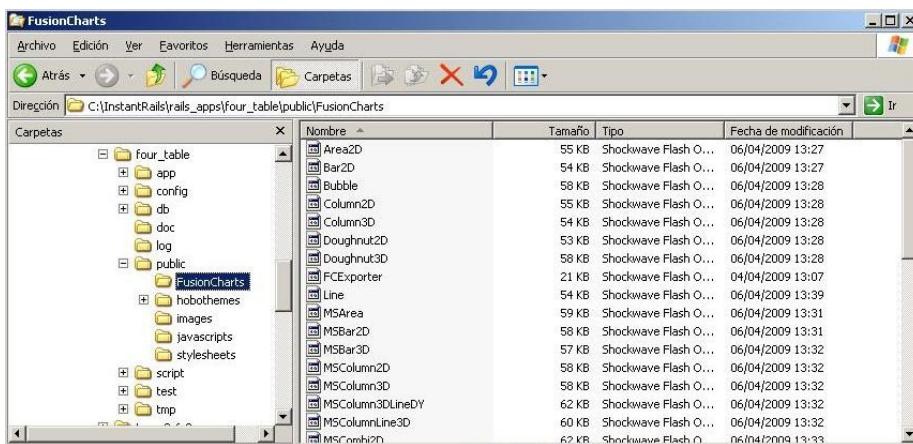
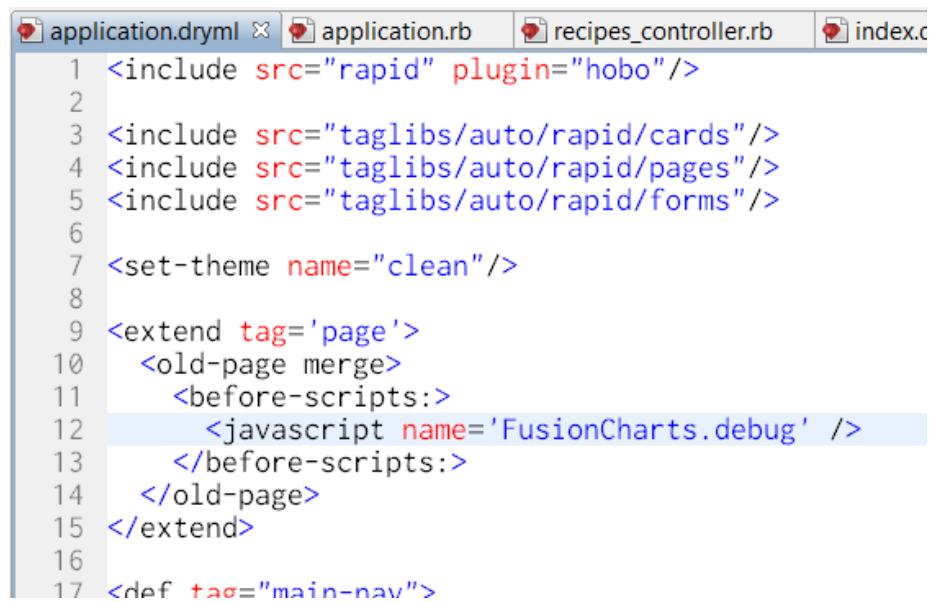


Figure 227: Target location for the FusionCharts SWF files

2. Copy the file c:\FusionChartsDistribution\JSClass\FusionCharts.debug.js to \four_table\public\javascripts
3. Now we are ready to reference the JavaScript file (copied in Step 2) in our application.dryml file, as follows:



The screenshot shows a code editor window with the tab 'application.dryml' selected. The code in the editor is as follows:

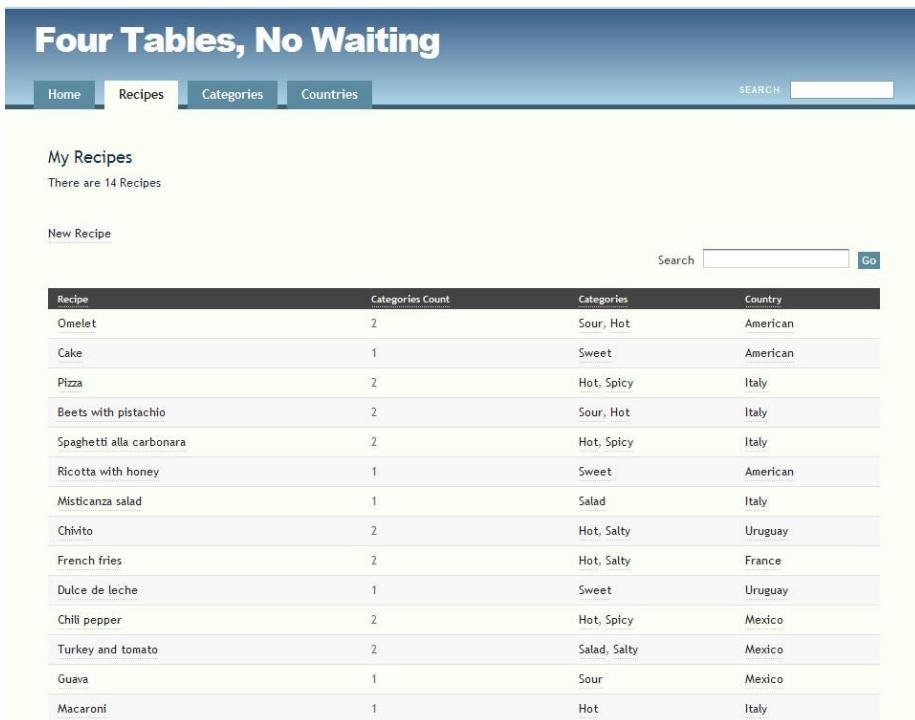
```
1 <include src="rapid" plugin="hobo"/>
2
3 <include src="taglibs/auto/rapid/cards"/>
4 <include src="taglibs/auto/rapid/pages"/>
5 <include src="taglibs/auto/rapid/forms"/>
6
7 <set-theme name="clean"/>
8
9 <extend tag='page'>
10   <old-page merge>
11     <before-scripts:>
12       <javascript name='FusionCharts.debug' />
13     </before-scripts:>
14   </old-page>
15 </extend>
16
17 <def tag="main-nav">
```

Figure 228: Adding the required <extend tag='page'> definition in application.dryml

As you can see from the code of application.dryml, we extended the 'page' view so that we always included the JavaScript file FusionCharts.debug.js. We could include this JavaScript at a page level, but for this tutorial's purpose it was more practical to do it this way.

Adding sample data

Before implementing the chart functionality, create sample data to use:



The screenshot shows a web application titled "Four Tables, No Waiting". The top navigation bar includes links for Home, Recipes, Categories, and Countries, along with a search bar. The main content area is titled "My Recipes" and displays a list of 14 recipes. Each recipe entry includes its name, the number of categories it belongs to, its category names, and its country of origin. The data is presented in a table format with a dark header row.

Recipe	Categories Count	Categories	Country
Omelet	2	Sour, Hot	American
Cake	1	Sweet	American
Pizza	2	Hot, Spicy	Italy
Beets with pistachio	2	Sour, Hot	Italy
Spaghetti alla carbonara	2	Hot, Spicy	Italy
Ricotta with honey	1	Sweet	American
Misticanza salad	1	Salad	Italy
Chivito	2	Hot, Salty	Uruguay
French fries	2	Hot, Salty	France
Dulce de leche	1	Sweet	Uruguay
Chili pepper	2	Hot, Spicy	Mexico
Turkey and tomato	2	Salad, Salty	Mexico
Guava	1	Sour	Mexico
Macaroni	1	Hot	Italy

Figure 229: Screen shot of sample recipe data for the tutorial

It is better to use the data presented above to make sure your charts will look the same as the tutorial.

Recipes By Country

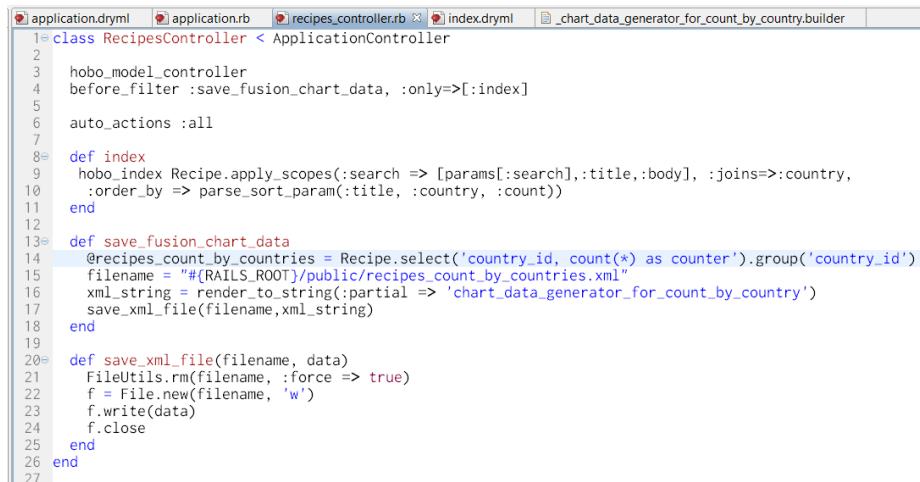
In order to implement a “Recipes By Country” chart we need to complete two steps:

1. Programmatically save the data to an XML file
2. Configure the Flash Component to retrieve the generated data.

Save the data to an XML file

For our first chart we need to modify the `recipes_controller.rb`. This will save the data (XML file) needed by the FusionCharts Flash component. We will activate the chart within the `recipes/index.dryml` file view since the needed data will be derived from the collection `Recipes`.

To get this to work, we will need to add a filter method to the controller, as well as a method to render XML. Take a look at the completed program below:



```

1= class RecipesController < ApplicationController
2
3  hobo_model_controller
4  before_filter :save_fusion_chart_data, :only=>[:index]
5
6  auto_actions :all
7
8= def index
9  hobo_index Recipe.apply_scopes(:search => [params[:search],:title,:body], :joins=>:country,
10   :order_by => parse_sort_param(:title, :country, :count))
11 end
12
13= def save_fusion_chart_data
14  @recipes_count_by_countries = Recipe.select('country_id, count(*) as counter').group('country_id')
15  filename = "#{RAILS_ROOT}/public/recipes_count_by_countries.xml"
16  xml_string = render_to_string(:partial => 'chart_data_generator_for_count_by_country')
17  save_xml_file(filename,xml_string)
18 end
19
20= def save_xml_file(filename, data)
21  FileUtils.rm(filename, :force => true)
22  f = File.new(filename, 'w')
23  f.write(data)
24  f.close
25 end
26 end
27

```

Figure 230: Enhancements to RecipesController to provide data to FusionCharts

(modifications are highlighted in bold italics below). We added a new filter to store the XML file, but only when receiving a request for the index page.

```

class RecipesController < ApplicationController
  hobo_model_controller
  before_filter :save_fusion_chart_data, :only => [:index]
  auto_actions :index, :show, :new, :edit, :create, :update, :destroy
  ...

```

We must define the ruby method `save_fusion_chart_data` for this controller. For now ignore the “private” method that encloses the code:

Let’s go through this code:

```

@recipes_count_by_countries = Recipe.select(
  'country_id, count(*) as counter').group('country_id')

```

In the above line we defined an instance variable (`@recipes_count_by_countries`) that resolves the query of how many recipes there are for each country.

```

filename = "#{RAILS_ROOT}/public/recipes_count_by_countries.xml"

```

In the above line we define the local path (from the Server point of view) where the XML data file will be stored. As you can see, we are pointing to the public directory of the Hobo application,. That’s necessary because the file must be available so that the FusionCharts Flash component (on the client side) can load it.

```
xml_string = render_to_string(:partial =>
    'chart_data_generator_for_count_by_country')
```

The line above implements the Rails “render_to_string” method using the template and needed semantics by FusionCharts included in the “chart_data_generator_for_count_by_country”. This will be discussed below.

```
save_xml_file(filename, xml_string)
```

The final line above calls the save_xml_file method passing the filename and the string stored in the variable `xml_string` (which represent an XML file)

Now, it’s time to review the implementation of the Rails’ “partial¹” that generates the XML string. Add the code below to the specified file.

```
app/views/recipes/_chart_data_generator_for_count_by_country.builder
```

```
xml.instruct!
xml.chart :caption => 'Recipes Count by Country' do
  @recipes_count_by_countries.each do |recipe|
    xml.set(:label => recipe.country.name,
            :value => recipe['counter'])
  end
end
```

Rails “partials” that end with the extension “.builder” instruct rails to use “Builder”, which is the XML generator. The API documentation can be found at:

<http://api.rubyonrails.org/classes/Builder/XmlMarkup.html>

This code defines a *chart* XML element (line #2), and then for each instance of the collection `@recipes_count_by_countries` it adds (within XML chart element) a *set of* XML elements that contain both the name of the Country and a counter for the number of recipes for the related country.

The following is a sample file generated by that Builder code:

¹To learn more about Rails partials, please see http://guides.rubyonrails.org/layouts_and_rendering.html. Here is a quote: —it’s important to know that the file extension on your view controls the choice of template handler. In Rails 2, the standard extensions are `.erb` for ERB (HTML with embedded Ruby), `.rjs` for RJS (javascript with embedded ruby) and `.builder` for Builder (XML generator). You’ll also find `.rhtml` used for ERB templates and `.rxml` for Builder templates, but those extensions are now formally deprecated and will be removed from a future version of Rails.||

```
<?xml version="1.0" encoding="UTF-8"?>
<chart caption="Recipes Count by Country">
    <set label="American" value="3"/>
    <set label="Uruguay" value="2"/>
    <set label="Mexico" value="3"/>
    <set label="Italy" value="5"/>
    <set label="France" value="1"/>
</chart>
```

Configure the Flash Component to retrieve the generated data

Now that we have the data needed by our FusionCharts Flash Component, we need to instruct our FusionCharts Flash Component, by using the available JavaScript API (thanks to the included file `FusionCharts.debug.js`), to load it.

See the figure below that includes the code from `recipes/index.dryml` that demonstrating how we can accomplish that:



```
• index.dryml
<index-page>
    <collection: replace>
        <div>
            <table-plus fields="this, categories.count, categories, country"/>
        </div>
    </collection>
    <after-content>
        <div id='recipes_count_by_countries'></div>
        <div id='recipes_count_by_categories'></div>
        <script>
            var chart_recipes_by_countries = new FusionCharts('http://localhost:3000/FusionCharts/Bar2D.swf', 'Recipes_Countries_Chart', '1000', '400');
            chart_recipes_by_countries.setDataURL('http://localhost:3000/recipes_count_by_countries.xml');
            chart_recipes_by_countries.render('recipes_count_by_countries');
        </script>
    </after-content>
</index-page>
```

Figure 231: Content of `recipes/index.dryml` used to render FusionCharts

- We define a `div` element (with `id` equal to `recipes_count_by_countries`), at line #8, intended to be the placeholder of the chart.
- Next, we make use of the FusionCharts JavaScript API o by creating a FusionCharts object at line #11.
 - The first parameter for the constructor is the particular Chart type that we are going to use. In this particular case, we will be using a Bar chart.
 - The second parameter is used to identify this Chart by name if you are going to use advanced features of the JavaScript API.
 - The third and forth parameters indicate the dimensions (width and height respectively) of the chart.
 - Finally, in line #13, we instruct FusionCharts to render the chart within the DOM element with “`id`” = to `recipes_count_by_countries`.

And that's it!!! Just go to the browser and request the URL: <http://localhost:3000/recipes>, and you'll see, at the bottom of the view, a chart similar to the following:

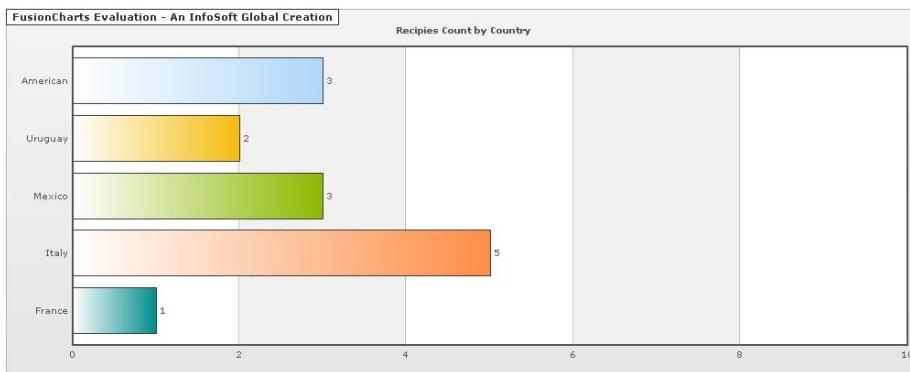


Figure 232: Screen shot of rendered FusionCharts bar chart

Recipes By Category

Let's try a different type of Chart. A typical choice would be a pie chart. The good news is that it's pretty much the same effort as the previous chart, because it uses the same type of XML data as input. For that reason, I'll be focusing on the differences of the new chart.

Save the data to an XML file

We are going to use the same mechanism presented earlier in this tutorial to store the XML file. In fact, we'll be modifying the method `save_fusion_chart_data` of `recipes_controller.rb`, this way:

```

1  private
2  def save_fusion_chart_data
3      @recipes_count_by_countries = Recipe.find(:all,
4          :select => 'country_id, count(*) as counter',
5          :group => 'country_id')
6      filename = "#{RAILS_ROOT}/public/recipes_count_by_countries.xml"
7      xml_string = render_to_string(:partial =>
8          'chart_data_generator_for_count_by_country')
9      save_xml_file(filename, xml_string)
10     @recipes_count_by_categories = CategoryAssignment.select(
11         'category_id, count(*) as counter').group('category_id')
12     filename = "#{RAILS_ROOT}/public/recipes_count_by_categories.xml"
13     xml_string = render_to_string(:partial =>
14         'chart_data_generator_for_count_by_categories')
15     save_xml_file(filename, xml_string)
16 end
17 def save_xml_file(filename, data)
18     FileUtils.rm(filename, :force => true)
19     f = File.new(filename, 'w')
20     f.write(data)
21     f.close
22 end

```

Again, the statement above marked with italics represent the modifications to the previous code. These new lines implement the same functionality as before, but by using a different collection as input, this time we are using `recipes_count_by_categories`.

Next, as we did for the previous chart, we define an XML builder as shown below:

`recipes/_chart_data_generator_for_count_by_categories.builder`

```

1 xml.instruct!
2 xml.chart :caption => 'Recipes Count by Category' do
3     @recipes_count_by_categories.each do |category_assignment|
4         xml.set(:label => category_assignment.category.name,
5                 :value => category_assignment['counter'])
5     end
6 end

```

You can tell that the only significant difference (apart from the caption description), is the way we invoke the model description. This is different in both cases because the queries are different.

After adding this we'll be generating both XML data files each time a request to the "Recipes" index arrives.

Configure the Flash Component to retrieve the generated data

The only thing missing to render the second chart is a placeholder for the flash and the proper JavaScript. Below we show the last piece of the puzzle:

```

File Edit Search View Tools Options Language Buffers Help
1 index.dryml 2 recipes_controller.rb 3 chart_data_generator_for_count_by_category.builder
1 <index-page>
2   <collection: replace>
3     <div>
4       <table-plus fields="this, categories.count, categories, country"/>
5     </div>
6   </collection:>
7   <after-content:>
8     <div id='recipes_count_by_countries'>
9     </div>
10    <div id='recipes_count_by_categories'>
11    </div>
12    <script>
13      var chart_recipes_by_countries = new FusionCharts("http://localhost:3000/FusionCharts/Bar2D.swf", 'Recipes_Countries_Chart', '1000', '400');
14      chart_recipes_by_countries.setDataURL('http://localhost:3000/recipes_count_by_countries.xml');
15      chart_recipes_by_countries.render('recipes_count_by_countries');
16      var chart_recipes_by_categories = new FusionCharts("http://localhost:3000/FusionCharts/Pie3D.swf", 'Recipes_Categories_Chart', '1000', '400');
17      chart_recipes_by_categories.setDataURL('http://localhost:3000/recipes_count_by_categories.xml');
18      chart_recipes_by_categories.render('recipes_count_by_categories');
19    </script>
20  </after-content:>
21 </index-page>
22
23

```

Figure 233: The recipe/index.dryml file to render a FusionCharts pie chart and bar chart

And then, we're done!! Here is the final result:

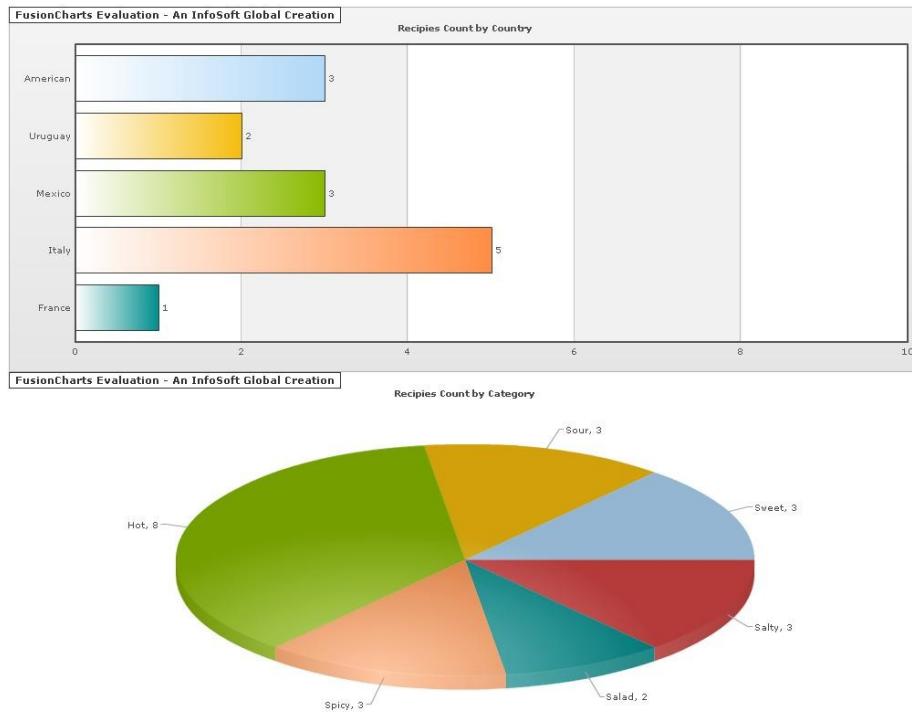


Figure 234: Screen shot of the rendered FusionCharts bar and pie charts

Have fun with FusionCharts!! Explore the different options here:

<http://www.fusioncharts.com/OnlineDocs.asp>

Tutorial 20

Adding User Comments to Models

By Kevin Potter

Tutorial Application: comments_recipe

Say you're developing some big social site with a ton of different models that *all* need to have comments. The question is, how can we do this so we don't have to repeat any code? Also, how can we make it so that adding it to a new model is easy when that happens down the road?

Let's see what sort of requirements we have:

- Comments will come from a signed in user and be attached to them
- Comments should be able to be attached to *any* model now or down the road
- Guests can't comment but can see comments
- Users can't edit or delete their comments but admins can

There's several different potential approaches but we'll go with *polymorphism*. The comments model will have a polymorphic association to the parent, the one that `has_many :comments`.

So, let's start with a blank app.

```
> hobo new comment_test_app
```

After, we'll need a run of the mill Post model for testing.

```
> cd comment_test_app
> hobo g resource Post subject:string body:text
> hobo g migration
```

The Model

```
hobo g resource Comment
```

Now, let's set the fields and the association up in in the model file, `app/models/comment.rb`

```
class Comment < ActiveRecord::Base
  hobo_model # Don't put anything above this
  fields do
    body :text, :required
    timestamps
  end
  belongs_to :commentable, :polymorphic => true
  belongs_to :owner, :class_name => "User",
    :creator => true
  set_default_order "created_at desc"
  ...
end
```

Fairly standard stuff here except the `:polymorphic => true` bit. This is actually from rails, but really can shine with some DRYML magic attached. What this does, once you migrate, is adds a 'type' column to the comments table in addition to the standard commentable_id column. Now, when you attach a comment to another model, in addition to the model's id, it also stores the type (id on its own is not enough to guarantee finding the right model with a polymorphic association).

"How does one setup the other side of the association though?" you might ask. That's the `:commentable` part. To add the comments association to another model you just add:

```
has_many :comments, :as => :commentable
```

Note: It doesn't have to be comments but for our example, it's going to be a requirement for proper activation in the code.

So, let's run that migration:

```
hobo g migration
```

Also, we have an owner for each comment, which is actually a User. The `:creator => true` flag takes care of setting the owner association as the `current_user` when creating a comment.

Let's go ahead and take care of the permissions while we're thinking about the model:

```
# --- Permissions ---
def create_permitted?
  owner_is? acting_user
end
```

We don't need to change the other permissions as they're already what we want (only admins can edit or delete comments). Here, we're using a helper method from the

permissions system, `owner_is?`, which is letting us bypass loading the owner model (if we did `owner == acting_user`, it'd load the owner association unnecessarily) or having the less clear code of `owner_id == acting_user.id`. Also, by having it so that the only allowed creation is when the owner is the `acting_user`, Rapid will remove the owner selector from the form. Pretty slick.

The Controller

True to Hobo style, we're just popping in the comment controller to take out the un-needed actions changing:

```
app/controllers/comments_controller.rb
```

```
auto_actions :all
```

to

```
auto_actions :write_only
```

We'll be embedding the very simple comment form in our commentable's show-page, but I'm getting ahead of myself.

The Target(s)

Now add `has_many :comments, :as => :commentable` to both the Post and User model.

We don't need to migrate as there's no new columns on either Post or User.

Note: If you doing this on an existing app, you can add this to any model you want to be commentable.

The View

We have the data side setup but no way of adding comments or seeing them currently. Let's remedy that. In `app/views/taglibs/application.dryml` add this extension:

```
<extend tag="show-page">
  <old-show-page merge>
    <append-content-body:>
      <do:comments if="&this.respond_to? :comments">
        <h4 param="comment-header">Comments</h4>
        <collection part="comments-collection" />
        <h5 param="comment-form-label">
          Add a comment
        </h5>
        <form with="&this.user_new(current_user)" update="comments-collection" reset-form param>
          <field-list: skip="commentable"
            without-body-label />
        </form>
      </do>
    </append-content-body:>
  </old-show-page>
</extend>
```

That's a lot to take in there, but let's break it down. First, we're extending the show-page. Since we're not targetting a particular model's show-page (with the `for` attribute) this is modifying the show-page that *every* model's show-page is defined against. So, we'll see the next part inside the end of the content-body param on every page... if, it responds to the comments method.

```
<do:comments if="&this.respond_to? :comments">
```

So, this switches context to comments if the current context (`this`) has a `comments` method, which the `has_many :comments` provides.

Note: We couldn't just use an `if :comments` shortcut since `if` tests for *blankness*, not *response*. We need the form to be visible even if there are no comments.

The header is pretty standard with an added `param` call so if need be it can be changed on specific pages. Unfortunately we can't add `param` on the next line:

```
<collection part="comments-collection" />
```

We're defining the collection as a part. A part is a way to mark content for later update via AJAX callbacks. Since it's using the current definition to make a javascript updatable piece, you can't have the `param` flexibility with parts like you do with other dryml content.

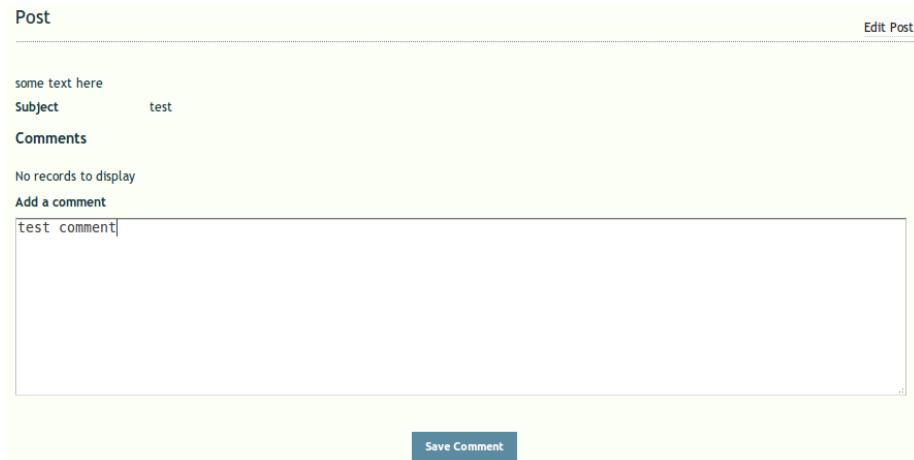
The form:

```
<form with="&this.user_new(current_user)"  
      update="comments-collection" reset-form>
```

The with, creates a new comment (unsaved) using the current_user as the owner. The update attribute, tells the form what *DOM id* (*not part name* which can be confusing as they're usually the same) to stick the updated content. The last bit, tells it to reset the form to a blank state after successful submission.

```
<field-list: skip="commentable" without-body-label />
```

We didn't want the body label since it's only the one field. Also, because the commentable field isn't your standard fields, it breaks the standard dryml form trying to render them. We only need the body input anyway.



A screenshot of a web-based comment form. At the top left is a 'Post' button and at the top right is an 'Edit Post' link. Below this is a text area containing 'some text here'. Underneath is a 'Subject' field with 'test' and a 'Comments' section. The comments section contains a message 'No records to display' and a 'Add a comment' button. A text input field contains the text 'test comment'. At the bottom right is a blue 'Save Comment' button.

Figure 235: The comment form



A screenshot of the same comment form after a comment has been posted. The 'Comments' section now lists the comment 'test comment kevin' with a small 'X' icon to its right, indicating it can be deleted.

Figure 236: Comment posted

Wow, almost there...

The Card

We just need to update the comment card so that it shows the appropriate information in a more logical layout.

```
<extend tag="card" for="Comment">
  <old-card merge>
    <creator-link: replace>
      <h5><a:a:owner><You /></a>
        posted at <view:created_at />
      </h5>
    </creator-link:>
  </old-card>
</extend>
```

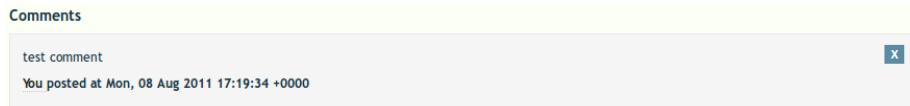


Figure 237: Updated comment card

Notice too that we already have comments on user pages from this as well.

A screenshot of a user profile page for "User kevin". The page includes basic user information like email address and state, and a "Comments" section which displays a message: "No records to display". There is also a link to "Add a comment".

Figure 238: User page comments

In Closing

As you add new models, you can add comment support simply by declaring that the model has_many :comments, :as => :commentable. No view/controller changes required!

There's still room for improvement but I'll leave that up to you. Some suggestions:

- Change the date format
- Add a summary of a User's comments with links to the commented items on the user page
- Add avatar support (hint: it's quite simple with the gravatar tag.)
- We didn't put in edit support for admins yet (hint: think controller)
- The point it's inserting the comment list and form is potentially problematic depending on what other customizations you have on any particular show-page
- Think about how this design pattern could be used elsewhere. Personally I've used something similar with tags, categories and ratings. And don't forget you can use it with lifecycle transitions, such as by replacing delete buttons with a lifecycle link on edit-pages when something like a merge_and_delete method is present.

Tutorial 21

Replicating the Look and Feel of a Site

By Tom Locke

Introduction

Let's say we want a new Hobo app to have the same look-and-feel of an existing site. The really big win is if we can have the same look and feel happen app almost 'automatically'. We want to be able to develop at "Hobo" speed, and have the look-and-feel "just happen". This is not easy to set-up, but once set-up, the payback in terms of development agility is more than worth it.

We'll invoke the standard web design used throughout all agencies within the U.S. Department of Agriculture. The authors have done substantial work with NIFA, The Cooperative State Research, Education, and Extension Service, so we will use their website (www.nifa.usda.gov) as an example:



Figure 239: Screen shot of the nifa.usda.gov home page

Note that, for now, this recipe will document how to create a *close approximation* to the above theme. We're going to skip some of the details that cannot be implemented without resorting to images to keep the recipe from getting too complex and lengthy.

This will be as much a guide to general web-development best practices as it will be a lesson in Hobo and DRYML. The mantra when working with themes in Hobo is already familiar to skilled web developers:

Separate content from presentation

The vast majority of common mistakes that are made in styling a web-app come under this heading. If separating content from presentation can be understood and applied, you're well on your way to:

- Having the look-and-feel “just happen” as your site changes and evolves
- Being able to change the theme in the future, without having to modify the app

Since CSS has been widely adopted, most web developers are familiar with separating content from presentation here is a reminder about how this works:

“Content” describes what is on the page, but not what it will look like. In a Hobo app content comes from tag definitions, page templates and the application’s data of course.

“Presentation” describes how the page should look. That is, it describes fonts, colors, margins, borders, images and ect.. In a Hobo app the presentation is handled essentially the same way as with any app, using CSS stylesheets and image assets.

Having said that, we need to inject a note of pragmatism:

- Humans beings are visual animals. Information can never truly be separated from the way it is displayed. The line is sometimes blurred and there are often judgment calls to be made.
- The technologies we’ve got to work with, in particular cross-browser support for CSS, are far from perfect. Sometimes we have to compromise.

There’s probably an entire PhD thesis lurking in that first point, but let’s move on!

The current site

To begin look at the elements of the existing site we need to replicate. The main ones are:

A banner image:



Figure 240: The NIFA banner image

A photo image that fits below the banner image:



Figure 241: The NIFA photo image

The main navigation bar:



Figure 242: The NIFA main navigation bar

A couple of styles of navigation panels:

Search NIFA

Go

- Search Help

Browse by Audience

Information for ...

Browse by Subject

- ▷ Agricultural Systems
- ▷ Animals
- ▷ Biotechnology & Genomics
- ▷ Economics & Community Development
- ▷ Education
- ▷ Environment & Natural Resources
- ▷ Family, Youth & Communities
- ▷ Food, Nutrition & Health
- ▷ International
- ▷ Pest Management
- ▷ Plants
- ▷ Technology & Engineering

Grants

- [Agriculture and Food Research Initiative](#)
- [Small Business Innovation Research](#)
- [More...](#)

- [Request for Application \(RFAs\)](#)
- [Application Information](#)

Quick Links

- [A-Z Index](#)
- [Local Extension Office](#)
- [Jobs & Opportunities](#)
- [State & National Partners](#)
- [NIFA Staff Directory](#)
- [Programs](#)
- [Program Impacts](#)
- [CRIS \(funded projects\)](#)
- [Visiting NIFA](#)
- [Budget Information](#)

Figure 243: NIFA navigation panels

And more navigation in the page footer:

NIFA | USDA.gov | Site Map | Policies | Grants.gov | CRIS | REEIS | Leadership Management Dashboard | eXtension | RSS
FOIA | Accessibility Statement | Privacy Policy | Non-Discrimination Statement | Information Quality | USA.gov | White House

Figure 244: NIFA footer navigation

An important thing to notice is that this site is *not* just a “theme” in the Hobo sense of the word. Hobo themes are purely about presentation, whereas the “look and feel” of this site is a mixture of contents, elements, and presentation.

That means we’re going to be creating three things to capture the site’s look-and-feel:

- Tag Definitions
- A CSS stylesheet
- Some image assets.

The current markup

The existing site makes extensive use of HTML tables for layout. The various images in the page are present in the markup as `` tags. In other words, the existing markup is very *presentational*.

Rather than create tag definitions out of the existing markup, we’ll be recreating the site using clean, semantic markup and CSS.

The other advantage of re-creating the markup is that it will be easier to follow Hobo conventions. There’s no particular need to do this, but it will be easier to jump from one Hobo app to the next.

Building the new app

Let’s do this properly and follow along in a blank Hobo app. At the end of the recipe we’ll see how we could package this look-and-feel up and re-use it another app. To follow along, use Firefox and the Firebug extension you can find at <http://getfirebug.com>.

```
> hobo nifa-demo
> cd nifa-demo
> hobo g migration
```

If you fire up the server, you’ll see the default Hobo app of course:

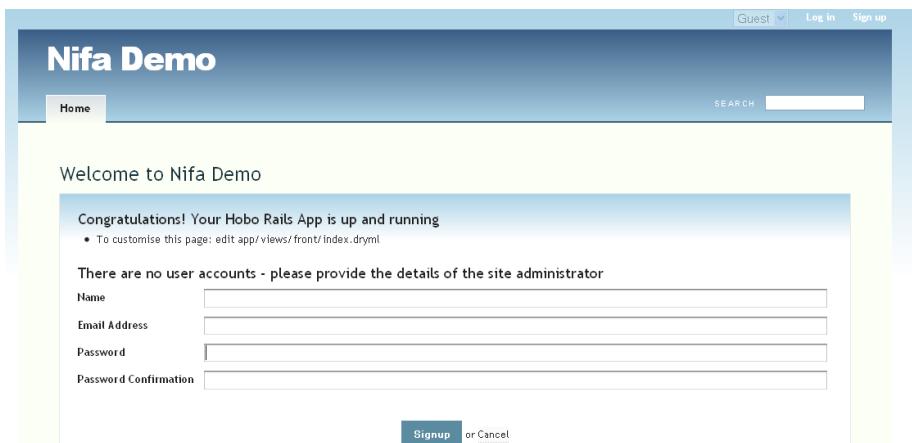


Figure 245: The NIFA Demo default home page

First thing to do is change the heading “Nifa” to “NIFA” in \views\taglibs\application1.dryml since it is an acronym for the National Institute of Food and Agriculture:

```

1 <include src="rapid" plugin="hobo"/>
2
3 <include src="taglibs/auto/rapid/cards"/>
4 <include src="taglibs/auto/rapid/pages"/>
5 <include src="taglibs/auto/rapid/forms"/>
6
7 <set-theme name="clean"/>
8
9 <def tag="app-name">NIWA Demo</def>
10

```

Figure 246: Using the "app-name" tag to change the default application name

Now, we can start to make it look like the page we’re after. We’ll take it step by step.

Main background and width

With the Firebug add-on for Firefox I can tell that the NIFA background color is #A8ACB7:

TUTORIAL 21 REPLICATING THE LOOK AND FEEL OF A SITE

CHAPTER 5 ADVANCED TUTORIALS

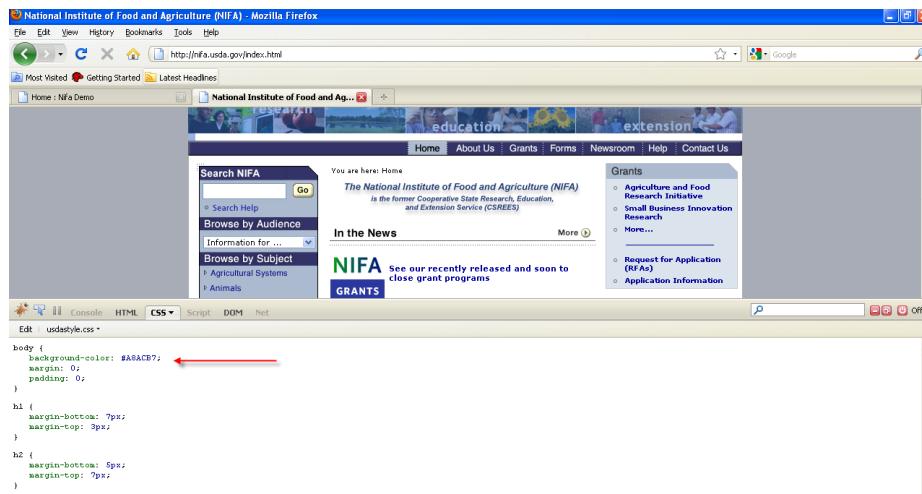


Figure 247: Using Firebug to locate the background color

Switching to the Hobo Nifa Demo application, Firebug tells us (click the inspect button, then click on the background) that the CSS rule that sets the current background comes from `clean.css` and looks like:

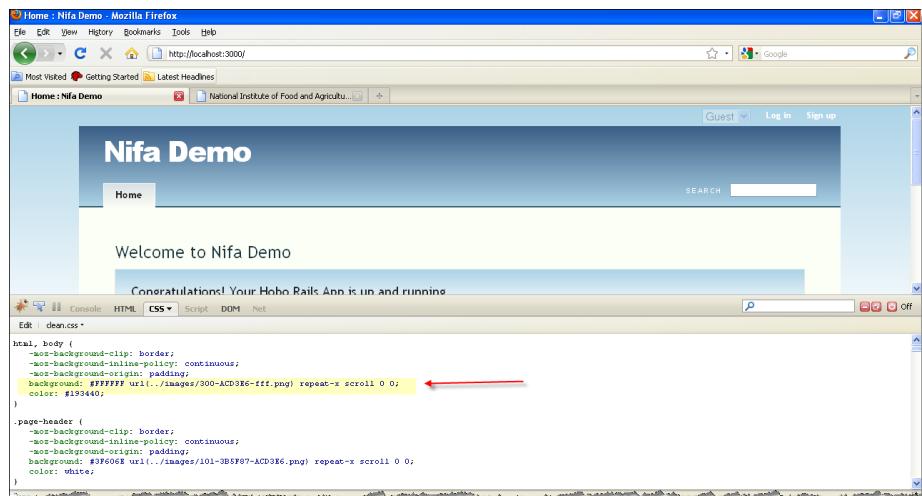


Figure 248: Using Firebug to find the images used by Hobo for the default background

Anything we add to `application.css` (it is empty by default) will override `clean.css`. So I'm going to add this rule to `public/stylesheets/application.css`:

```
html, body { background:#A8ACB7 }
```

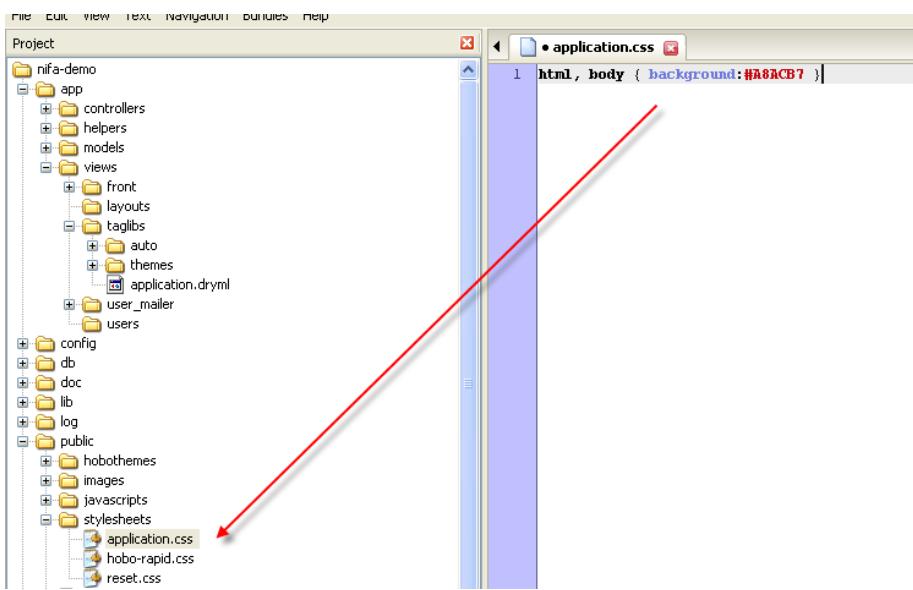


Figure 249: Adding the new background color to "application.css"

Again, using Firebug on the NIFA Demo app (by clicking on the <body> tag in the HTML window) I can see that the width is set on the body tag:

```
body { ... width: 960px; ... }
```

Back in NIFA, I can right click the banner image and chose “View Image”, and Firefox tells me its width is 766 pixels. In `application.css` I add

```
body { width: 766px; }
```

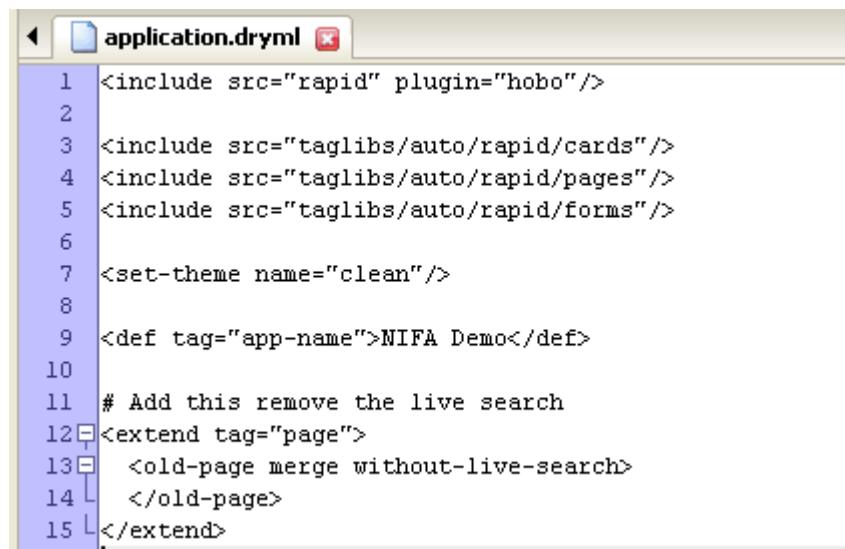
Note: we've not changed any markup - that's how we like it.

Account navigation

The log-in and sign-up links are in the top right. They are not on the NIFA site. The app needed them, the place they are in now would be fine, so we'll leave them where they are.

Search

The page header has a search-field that we don't want. To get rid of this we'll customize the <page> tag. We need to do this in `application.dryml`:



```

1 <include src="rapid" plugin="hobo"/>
2
3 <include src="taglibs/auto/rapid/cards"/>
4 <include src="taglibs/auto/rapid/pages"/>
5 <include src="taglibs/auto/rapid/forms"/>
6
7 <set-theme name="clean"/>
8
9 <def tag="app-name">NIFA Demo</def>
10
11 # Add this remove the live search
12 <extend tag="page">
13   <old-page merge without-live-search>
14   </old-page>
15 </extend>

```

Figure 250: First pass at modifying "application.dryml"

```

<extend tag="page">
  <old-page merge without-live-search>
  </old-page>
</extend>

```

Now we *have* made a change to the markup, but that makes perfect sense because here we wanted to change *what's on the page* not *what stuff looks like*.

The Banner

Again, using Firefox's "View Image", it turns out that the existing banner is in fact two images.



Figure 251: The two images used in NIFA's top banner

To add these images without changing the markup, we need to use CSS's background-image feature. One major limitation of CSS is that you can only have one background image per element. That won't be a problem. To understand our approach, first take a look at a simplified view of the page markup we're working with:

```
<html>
  <head>...</head>
  <body>
    <div class="page-header">
      <h1 class="app-name">NIFA Demo</h1>
    </div>
    ...
  </body>
</html>
```

Notice the image below is essentially a graphical version of that `<h1>` tag. We'll use CSS to render that same `<h1>` as an image. The existing text can be hidden, by moving it way out of the way with a `text-indent` rule. First, we need to save that image into our `public/images` folder.

The CSS to add to `application.css` is:

```
div.page-header { padding: 0; }
div.page-header h1.app-name {
  text-indent: -10000px;
  background: url(..\images\banner_nifa.gif) no-repeat;
  padding: 0; margin: 0;
  height: 62px;
}
```

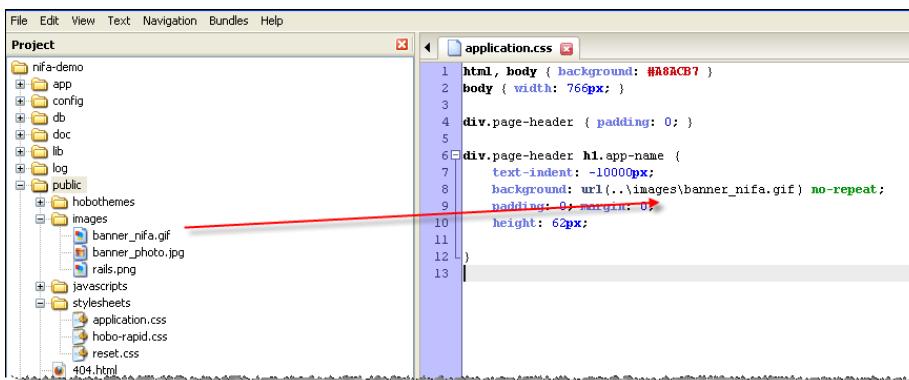


Figure 252: How to reference the banner gif in "application.css"

OK that was a bit of a leap! Why `padding: 0px` for the page-header. The fact is that working with CSS is all about trial and error. Whether you are to figure out what rules are currently in effect or flipping back and forth between the stylesheet in your editor and the browser. Try experimenting by taking some rules out, and you'll see why each rule is needed.

Now for the photo part of the banner. Again, save it to `public/images`, then add some extra properties to the `div.page-header` selector, so it ends up like:

```
div.page-header {  
    padding: 0;  
    background: url(..\images\banner_photo.jpg) no-repeat 0px 62px;  
    height: 106px;  
}
```

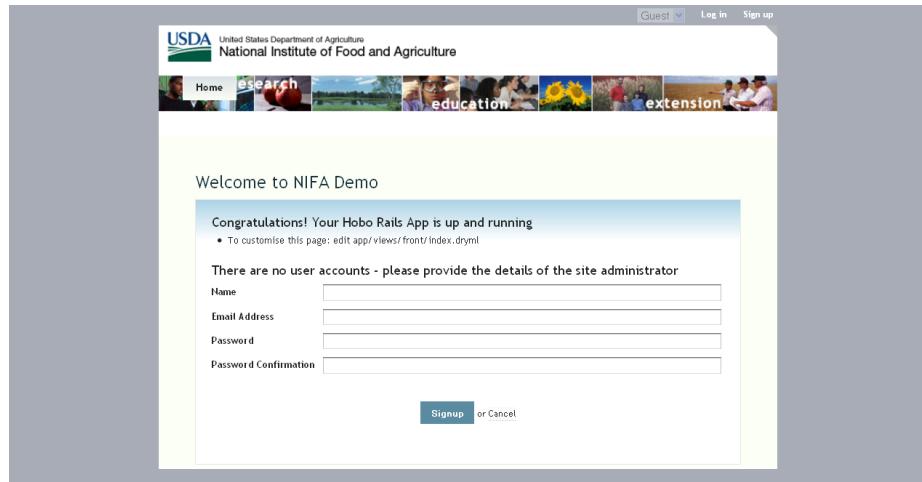


Figure 253: View of the NIFA Demo login page

The main navigation panel (“Home” tag) is hovering on top of the photos:



Figure 254: The Navigation Panel before refactoring

Navigation

The existing navigation bar is created entirely with images. It's quite common to do this, as it gives total control over fonts, borders, and other visual effects such as color

gradients. The downside is that you have to fire up your image editor every time there's a change to the navigation.

This doesn't sit very well with our goal to be able to make changes quickly and easily. For this recipe we're going to go implement the navigation bar without resorting to images. We'll lose the bevel effect, but some might think the end result is actually cleaner, clearer, and more professional.

Our app only has a home page, so first let's define a fake navigation bar. In application.dryml:

```
<def tag="main-nav">
  <navigation class="main-nav">
    <nav-item href="">Home</nav-item>
    <nav-item href="">About Us</nav-item>
    <nav-item href="">Grants</nav-item>
    <nav-item href="">Forms</nav-item>
    <nav-item href="">Newsroom</nav-item>
    <nav-item href="">Help</nav-item>
    <nav-item href="">Contact Us</nav-item>
  </navigation>
</def>
```

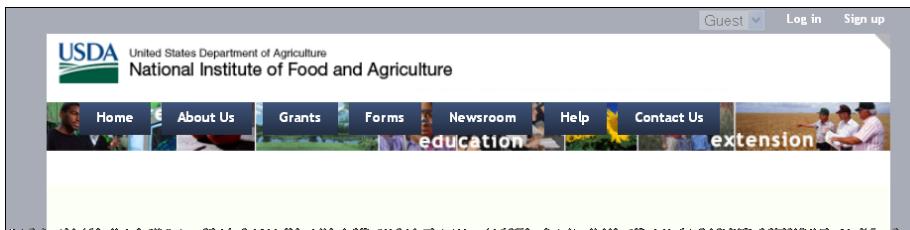


Figure 255: View of our first pass at the main navigation menu

Use Firebug's "Inspect" button to find the navigation bar. You'll see that it's rendered as a `` list, which is generally considered good practice. It is a list of links after all. There are several things wrong with the appearance of the navigation:

- It's in the wrong place - we want to move it down and to the right.
- Needs to be shorter and the spacing of the items need to be fixed.
- Smaller font not bold
- The background color needs to change, as do the colors when you mouse-over a link

This is not a CSS tutorial, so we're not going to explain every last detail. We'll build it up in a few steps which will help to illustrate what does what. First, update the rules for `div.page-header` in `application.css` so they look like:

```
div.page-header {  
    padding: 0;  
    background: white url(..\images\banner_photo.jpg) no-repeat 0px 62px;  
    height: 138px;  
}
```

And add:

```
div.page-header .main-nav {  
    position: absolute; bottom: 0; right: 0;  
}
```

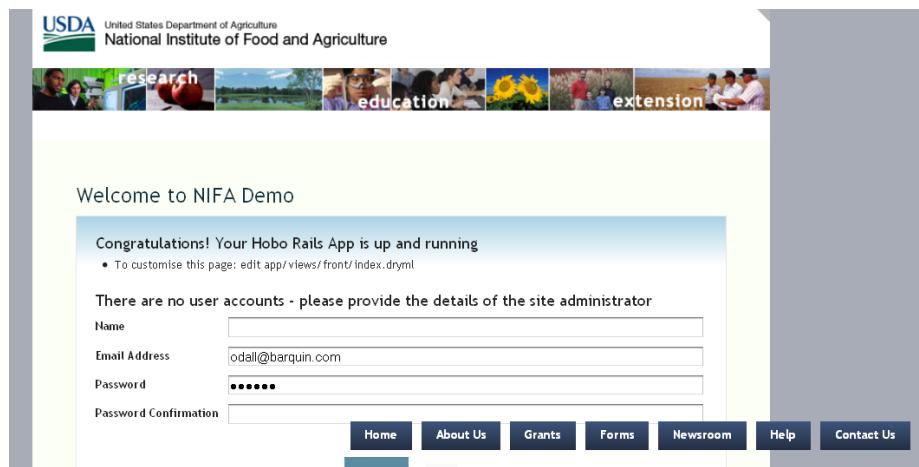


Figure 256: Still need more to fix the top navigation menu...

The nav-bar still looks wrong, so we'll fix the sizing and placement. Update the new rule (`div.page-header .main-nav`) and add new ones and colors. The entire `application.css` looks like this so far:

```
html, body { background:#A8ACB6 }
body { width: 766px; }

div.page-header {
    padding: 0;
    background: white url(..../images/banner_photo.jpg) no-repeat 0px 62px;
    height: 138px;
}

div.page-header h1.app-name {
    text-indent: -10000px;
    background: url(..../images/banner_nifa.gif) no-repeat;
    padding: 0; margin: 0;
    height: 55px;
}

div.page-header .main-nav {
    position: relative;
    top: 63px;
    height: 21px;
    width: 100%;
    line-height: 21px;
    padding: 0;
    text-align: right;
    background: #313367;
}

div.page-header .main-nav li {
    margin: 0;
    padding: 0 0 0 4px;
    display:inline;
    float:none;
    border-left: 1px dotted #eee;
    background: #313367;
    color: silver;
}

div.page-header .navigation.main-nav li a {
    padding: 3px 8px;
    margin: 0;
    font-weight: normal;
    display:inline;
    font-size: 12px;
    background: #313367;
    color: silver;
}
div.page-header .navigation.main-nav li.current a {
    background: #313367;
    color: white;
}

div.page-header .navigation.main-nav li a:hover {
    background: #A9BACF;
    color: black;
}
```

Note: that we had to make the last two selectors a bit more specific, in order to ensure

that they take precedence over rules in the “Clean” theme.

The page header should be done at this point:

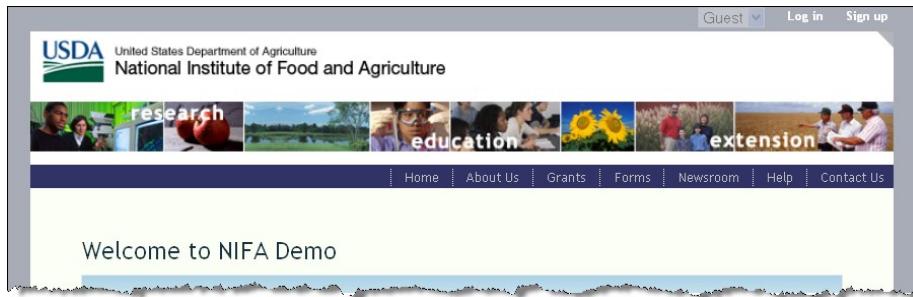


Figure 257: The fixed NIFA man navigation bar

The sidebars

The existing site has both left and right sidebars. Which we will add those now. First step is to add the three content sections to the <page> tag in application.dryml. We’ve already extended <page>, so modify the DRYML you already have to look like:

```
<extend tag="page">
  <old-page merge without-live-search>
    <content: replace>
      <section-group class="page-content">
        <aside param="aside1"/>
        <section param="content"/>
        <aside param="aside2"/>
      </section-group>
    </content:>
  </old-page>
</extend>
```

We’ve replaced the existing <content:> with a <section-group> that contains our two <aside> tags and the main <section>.

To try this out, insert some dummy content in app/views/front/index.dryml. Edit that file as follows:

```
<page title="Home">
  <body: class="front-page"/>
  <aside1:>Aside 1</aside1:>
  <content:>Main content</content:>
  <aside2:>Aside 2</aside2:>
</page>
```

You should see something like:

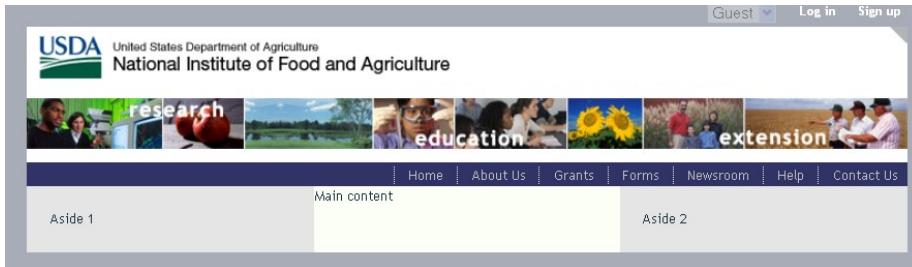


Figure 258: View of the default three-column formatting

Obviously, there is a bunch of styling to do. First, though, let's add the content for the left sidebar. This is the "search and browse" panel which is on every page of the site. Let's define it as a tag in `application.dryml`:

```
<def tag="search-and-browse" attrs="current-subject">
  <div class="search-and-browse">
    <div param="search">
      <h3>Search NIFA</h3>
      <form action="">
        <input type="text" class="search-field"/>
        <submit label="Go"/>
      </form>
      <p class="help"><a href="">Search Help</a></p>
    </div>
    <div param="browse-by-audience">
      <h3>Browse by Audience</h3>
      <select-menu first-option="Information for..." options="&[]"/>
    </div>
    <div param="browse-by-subject">
      <h3>Browse by Subject</h3>
      <navigation current="&current_subject">
        <nav-item href="/">Agricultural & Food Biosecurity</nav-item>
        <nav-item href="/">Agricultural Systems</nav-item>
        <nav-item href="/">Animals & Animal Products</nav-item>
        <nav-item href="/">Biotechnology & Geneomics</nav-item>
        <nav-item href="/">Economy & Commerce</nav-item>
        <nav-item href="/">Education</nav-item>
        <nav-item href="/">Families, Youth & Communities</nav-item>
      </navigation>
    </div>
  </div>
</def>
```

A few points to note about that markup:

- We've tried to make the markup as "semantic" as possible – it describes what the content *is*, not what it looks *like*.

- We've added a few params, so that individual pages can customize the search-and-browse panel. Each param also gives us a CSS class of the same name, so we can target those in our stylesheet.
- We've used <navigation> for the browse-by-subject links. This gives us the ability to highlight the current page as the user browses.

Because the search-and-browse panel appears on every page, let's call it from our master page tag (<extend tag="page">). Change:

```
<aside param="aside1"/>
```

To:

```
<aside param="aside1"><search-and-browse/></aside>
```

Remove the <aside1>Aside 1</aside1> parameter from front/index.dryml.

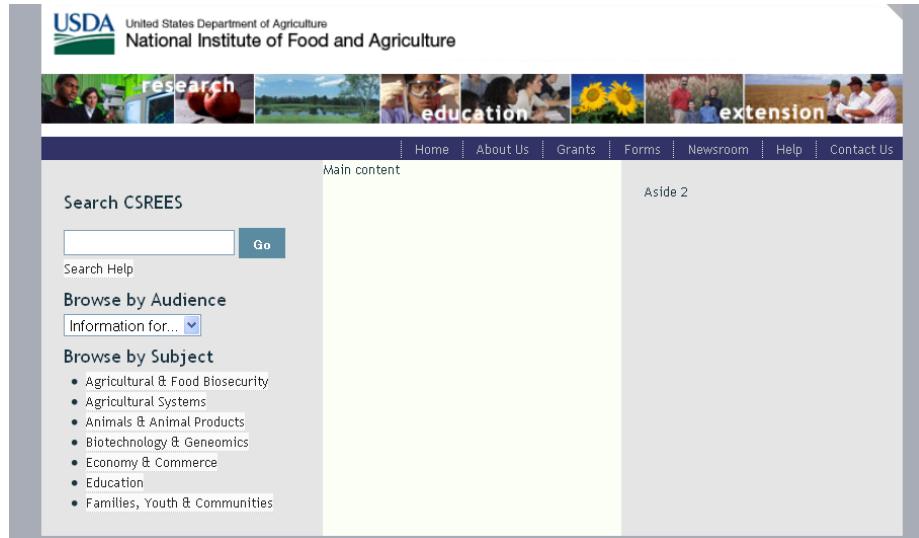


Figure 259: View of the left panel contact without styling

We need to style this panel. After a good deal of experimentation, we get to the following CSS:

```

div.page-content, div.page-content .aside { background: white; }
.aside1 { width: 173px; padding: 10px; }
.search-and-browse {
    background: #A9BACF;
    border: 1px solid #313367;
    font-size: 11px;
    margin: 4px;
}
.search-and-browse h3 {
    background: #313367; color: white;
    margin: 0; padding: 3px 5px;
    font-weight: normal; font-size: 13px;
}
.search-and-browse a { background: none; color: #000483; }
.search-and-browse .navigation { list-style-type: circle; }
.search-and-browse .navigation li {
    padding: 3px 0; font-size: 11px; line-height: 14px; }
.search-and-browse .navigation li a { border:none; }
.search-and-browse .search form { margin: 0 3px 3px 3px; }
.search-and-browse .search p { margin: 3px; }
.search-and-browse .search-field { width: 120px; }
.search-and-browse .submit-button { padding: 2px; }
.search-and-browse .browse-by-audience select {
    margin: 5px 3px; width: 92%; }

```

With that added to application.css you should see:

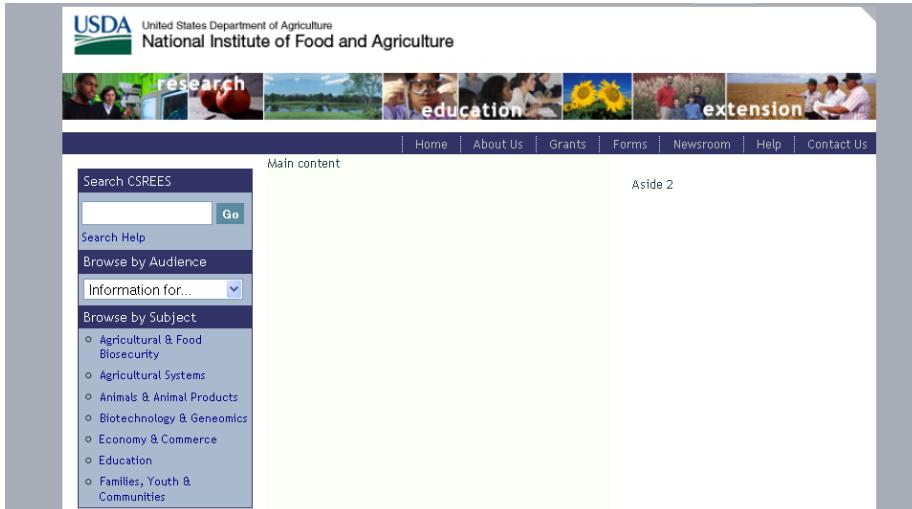


Figure 260: View of the left panel content with correct styling

OK - let's switch to the right-hand sidebar.

If you click around **the site** you'll see the right sidebar is always used for navigation panels, like this one:

You'll also notice it's missing from some pages, which is as easy as:

```
<page without-aside2/>
```

It seems like a good idea to define a tag that creates one of these panels:

```
<nav-panel>
  <heading:>Quick Links</heading:>
  <items:>
    <nav-item href="/">A-Z Index</nav-item>
    <nav-item href="/">Local Extension Office</nav-item>
    <nav-item href="/">Jobs and Opportunities</nav-item>
  </items:>
</nav-panel>
```

We've re-used the `<nav-item>` tag as it gives us an `` and an `<a>` which is just what we need.

Add the definition of `<nav-panel>` to your `application.dryml`:

```
<def tag="nav-panel">
  <div class="nav-panel" param="default">
    <h3 param="heading"></h3>
    <div param="body">
      <ul param="items"/>
    </div>
  </div>
</def>
```

Notice that we defined two parameters for the body of the panel. Callers can either provide the `<items:>` parameter, in which case the `` wrapper is provided, or, in the situation where the body will not be a single ``, they can provide the `<body:>` parameter.

OK let's throw one of these things into our page. Here's what `front/index.dryml` needs to look like:

```
<page title="Home">
  <body: class="front-page"/>
  <content:>Main content</content:>
  <aside2:>
    <nav-panel>
      <heading:>Grants</heading:>
      <items:>
        <nav-item href="/">
          National Research Initiative
        </nav-item>
        <nav-item href="/">
          Small Business Innovation Research
        </nav-item>
        <nav-item href="/">More...</nav-item>
      </items:>
    </nav-panel>
    <nav-panel>
      <heading:>Quick Links</heading:>
      <items:>
        <nav-item href="/">A-Z Index</nav-item>
        <nav-item href="/">Local Extension Office</nav-item>
        <nav-item href="/">Jobs and Opportunities</nav-item>
      </items:>
    </nav-panel>
  </aside2:>
</page>
```

And here's the associated CSS – add this to the end of your application.css:

```
.aside2 { margin: 0; padding: 12px 10px; width: 182px; }
.nav-panel {border: 1px solid #C9C9C9; margin-bottom: 10px; }
.nav-panel h3 {
  background:#A9BACF;
  color: #313131;
  font-size: 13px;
  padding: 3px 8px;
  margin: 0;}
.nav-panel .body {
  background: #DAE4ED;
  color: #00059A;
  padding: 5px;}
.nav-panel .body a {color: #00059A; background: none;}
.nav-panel ul {list-style-type: circle;}
.nav-panel ul li { margin: 5px 0 5px 20px;}
```

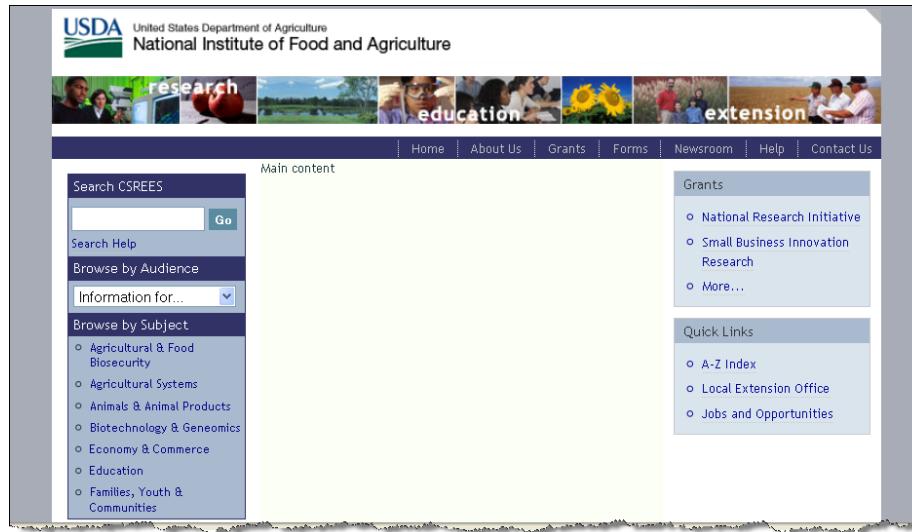


Figure 261: View of the right panel content with styling

Main content

The main content varies a lot from page to page, so let's just make sure that the margins are OK. First we need content to work with, so in front/index.dryml, replace:

```
<content:>Main content</content:>
```

With:

```
<content:>
  <h2>National Institute of Food and Agriculture</h2>
  <p>Main content goes here...</p>
</content:>
```

On refreshing the browser it seems there's nothing else to do. This looks fine:



Figure 262: View of the main content panel

The footer

The footer is the same throughout the site. Define it as a tag and add it to our main <page> tag. Here's the definition for application.dryml:

```
<def tag="footer-nav">
  <ul>
    <nav-item href="/">NIFA</nav-item>
    <nav-item href="/">USDA.gov</nav-item>
    <nav-item href="/">Site Map</nav-item>
    <nav-item href="/">Policies and Links</nav-item>
    <nav-item href="/">Grants.gov</nav-item>
    <nav-item href="/">CRIS</nav-item>
    <nav-item href="/">REEIS</nav-item>
    <nav-item href="/">Leadership Management Dashboard</nav-item>
    <nav-item href="/">eXension</nav-item>
    <nav-item href="/">RSS</nav-item>
  </ul>
</def>
```

Add this parameter to the <extend tag="page">:

```
<footer: param><footer-nav/></footer:>
```

Note: Since Hobo already includes a page-footer div out-of-the-box, we don't need to create this div in DRYML. If we did, we would end up with a duplicate and this would distort the footer.

Finally we have the CSS. To get the corner graphic that we've used. you need to right-click and “Save Image As” on the bottom left corner in the existing site:

```
.page-footer {
    background: white url(images/footer_corner_left.gif) no-repeat bottom left;
    overflow: hidden; height: 100%;
    border-top: 1px solid #B8B8B8;
    font-size: 12px; line-height: 10px;
    padding: 5px 0px 10px 40px;
}
.page-footer ul { list-style-type: none; }
.page-footer ul li {
    float: left;
    border-right: 1px solid #2A049A;
    margin: 0;
    padding: 0 5px;
}
.page-footer ul li a {border:none; color: #2A049A;}
```

There's one CSS trick in that is worth mentioning. In the .page-footer section, we've specified:

```
overflow: hidden; height: 100%;
```

This is the famous "self clearing" trick. Since all the content in the footer is floated, the footer loses its height, without the "self clearing" trick.



Figure 263: NIFA Demo with final footer styling

That brings us to the end of how to reproduce a look-and-feel. We should now be able to build out our application, and it will look right "automatically". In practice you always run into small problems and need to dive back into CSS to tweak things, but the bulk of the job is done.

The next question is - how could we make several apps look like this without repeating all the code? That is the subject of our next tutorial.

Tutorial 22

Creating a “Look and Feel” Plugin

By Tom Locke

In this tutorial we will start with the results of Tutorial 21. To re-use this work across many apps, we’ll use the standard Rails technique - create a plugin.

The plugin will contain:

- A DRYML taglib with all of our tag definitions
- A Public directory, containing our images and stylesheets

The idea of “creating a plugin” seems like a big deal, but there’s really nothing to it. All we’re going to do is move a few files into different places.

Here is the content of a batch file to create the folders and move the files:



```
create_plugin.bat
1 md vendor\plugins\nifa
2 cd vendor\plugins\nifa
3 md taglibs
4 md public
5 md public\nifa
6 md public\nifa\stylesheets
7 md public\nifa\images
8 cd ..\..\..
9 copy app\views>taglibs\application.dryml vendor\plugins\nifa\taglibs\nifa.dryml
10 copy public\stylesheets\application.css vendor\plugins\nifa\public\stylesheets\nifa.css
11 copy public\images\* vendor\plugins\nifa\public\images
12 |
```

Figure 264: Batch file with commands to create the plugin folders and content

Or as individual commands:

```
> md vendor\plugins\nifa
> cd vendor\plugins\nifa
> md taglibs
> md public
> md public\nifa
> md public\nifa\stylesheets
> md public\nifa\images
> cd ..\..\..
> copy app\views>taglibs\application.dryml \
  vendor\plugins\nifa\taglibs\nifa.dryml
> copy public\stylesheets\application.css \
  vendor\plugins\nifa\public\stylesheets\nifa.css
> copy public\images\* vendor\plugins\nifa\public\images
```

(That last command will also copy `rails.png` into the plugin. Which you probably want to delete).

We've copied the whole of `application.dryml` into our plugin because nearly everything in there belongs in the plugin. It does need some editing:

At the top, remove all of the includes, the `<set-theme>` and the definition of `<app-name>`

We need to make sure our stylesheet gets included, so add the following parameter to the call to `<old-page>`

```
<append-stylesheets:>
  <stylesheet name="\nifa\stylesheets\nifa.css"/>
</append-stylesheets:>
```

The new `nifa.dryml` will be:

```
# Add this remove the live search and add sidebars
<extend tag="page">
  <old-page merge without-live-search>

    # need this to acces the nifa.css stylesheet
    <append-stylesheets:>
      <stylesheet name="\nifa\stylesheets\nifa.css"/>
    </append-stylesheets:>
    #

    <content: replace>
      <section-group class="page-content">
        <aside param="aside1"><search-and-browse/></aside>
        <section param="content"/>
        <aside param="aside2"/>
      </section-group>
    </content:>
    <footer: param><footer-nav/></footer:>
  </old-page>
</extend>
# Replace the default navigation bar
<def tag="main-nav">
  <navigation class="main-nav">
    <nav-item href="">Home</nav-item>
    <nav-item href="">About Us</nav-item>
    <nav-item href="">Grants</nav-item>
    <nav-item href="">Forms</nav-item>
    <nav-item href="">Newsroom</nav-item>
    <nav-item href="">Help</nav-item>
    <nav-item href="">Contact Us</nav-item>
  </navigation>
</def>
# new tag
<def tag="search-and-browse" attrs="current-subject">
  <div class="search-and-browse">
    <div param="search">
      <h3>Search CSREES</h3>
      <form action="">
        <input type="text" class="search-field"/>
        <submit label="Go"/>
      </form>
      <p class="help"><a href="">Search Help</a></p>
    </div>
    <div param="browse-by-audience">
      <h3>Browse by Audience</h3>
      <select-menu first-option="Information for..." options="&[]"/>
    </div>
    <div param="browse-by-subject">
      <h3>Browse by Subject</h3>
      <navigation current="&current_subject">
        <nav-item href="/">Agricultural & Food Biosecurity</nav-item>
        <nav-item href="/">Agricultural Systems</nav-item>
        <nav-item href="/">Animals & Animal Products</nav-item>
        <nav-item href="/">Biotechnology & Geneomics</nav-item>
        <nav-item href="/">Economy & Commerce</nav-item>
        <nav-item href="/">Education</nav-item>
        <nav-item href="/">Families, Youth & Communities</nav-item>
      </navigation>
    </div>
  </div>
</def>
```

```
# Parameterized panel
<def tag="nav-panel">
  <div class="nav-panel" param="default">
    <h3 param="heading"></h3>
    <div param="body">
      <ul param="items"/>
    </div>
  </div>
</def>
# Footer parameterized tag
<def tag="footer-nav">
  <ul>
    <nav-item href="/">NIFA</nav-item>
    <nav-item href="/">USDA.gov</nav-item>
    <nav-item href="/">Site Map</nav-item>
    <nav-item href="/">Policies</nav-item>
    <nav-item href="/">Grants.gov</nav-item>
    <nav-item href="/">CRIS</nav-item>
    <nav-item href="/">REEIS</nav-item>
    <nav-item href="/">Leadership Management Dashboard</nav-item>
    <nav-item href="/">eXension</nav-item>
      <nav-item href="/">RSS</nav-item>
  </ul>
</def>
```

Using the plugin

To try out the plugin, create a new blank Hobo app. There are then three steps to install and setup the plugin:

Step 1. Copy vendor\plugins\nifa from nifa-demo into vendor\plugins in the new app.

Step 2. To install the taglib add:

```
<include src="nifa" plugin="nifa"/>
```

to application.dryml. It must be added after the <set-theme> tag.

Step 3. To install the public assets:

```
> copy vendor\plugins\nifa\public\* public
```

That should be it. Your new app will now look like the NIFA website, and the tags we defined, such as <nav-panel> will be available in every template.

Tutorial 23

Using Hobo Lifecycles for Workflow

By Venka Ashtakala

Now we have our “Four Table” application working the way we want. Let’s add an approval process, so that new recipes require approval by a user before they are published to the web.

To do this we can take advantage of ‘Hobo Lifecycles’, which is the Hobo answer to creating a workflow. The workflow that we will define for this application is that a Recipe can exist in one of 2 states: “Not Published” and “Published” and that there will be two transitions: “Publish” and “Not Publish” which will move the Recipe from one state to the other.

The “Publish” transaction will move the Recipe from the “Not Published” to “Published” state, while the “Not Publish” transaction will do the opposite. Lastly, we’ll make controller and view changes as necessary.

Tutorial Application: four_table

Topic: HOBO Lifecycles

Steps

Step 1. Setup the lifecycle.

Now that we know the functional requirements for the Recipe workflow we wish to implement we can start modifying our Four Table application. We are going to add the Hobo Lifecycle definition to our Recipe model. Let’s open up the /app/model/recipe.rb file and add the `lifecycle do...end` block:

```
[...]
belongs_to :country
lifecycle :state_field => :lifecycle_state do
  state :not_published, :default => :true
  state :published

  transition :publish, { :not_published => :published },
    :available_to => "acting_user if acting_user.signed_up?"
  transition :not_publish, { :published => :not_published },
    :available_to => "acting_user if acting_user.signed_up?"

end
# --- Permissions --- #
[...]
```

So what exactly did we add exactly? The `lifecycle do..end` block defines the lifecycle for a given model. The `:state_field` argument specifies that we want the lifecycle to save the current state to a ‘`lifecycle_state`’ column in the table. Within the block we have to define our states and transition actions.

We define our states by using the ‘`state`’ keyword, which takes the state name and options as arguments. In this manner we have defined two states:

```
:not_published  
:published
```

The `:default => :true` argument to the `:not_published` state, means that when the state is not defined, such as when the recipe is created, its initial state will be `:not_published`.

After the state declarations, we have defined two transition actions using the ‘`transition`’ keyword. The transition keyword requires a name, a hash that specifies the state transition and then options. The first transition, `:publish`, specifies that when this action is executed, the Recipe’s state will go from `:not_published` to `:published`. The `:available_to` argument specifies that this action can only be executed by a user that has signed up, so guests are not allowed to execute this action. The second transition, `:not_publish`, changes the state from `:published` to `:not_published`, and limits the action to be available only to signed up users.

By adding the lifecycle behaviour to our model, we’ll need to generate and run a hobo `g migration` since a new ‘`lifecycle_state`’ column will be added to our recipes table. At the command line, in your application directory, execute the following:

```
> hobo g migration
```

Select ‘m’ when prompted to migrate now, and then specify a name for this migration.

Step 2. Setup the lifecycle controls in your view.

Now that we have setup the lifecycle for our Recipe model, we need to expose the transition actions to our users. HOBO makes this very easy by giving us a predefined dryml tag called `<transition-buttons/>` We’ll use this tag on our Recipe listing page.

Open up the `views/recipes/index.dryml` page and change this code:

```
<table-plus fields="this, categories.count, categories,country"/>
```

to:

```
<table-plus fields="this, categories.count, categories, country">  
  <controls:>  
    <transition-buttons/>  
  </controls:>  
</table-plus>
```

By using the <controls:> parameter tag in table-plus, it allows us to insert an extra column at the end of the table where we can place action buttons or links. There we use the <transition-buttons/> tag to specify that lifecycle transition buttons should show for any actions that are available for the current user.

Step 3. Setup the lifecycle actions in the controller.

We need to make a couple of changes to our Recipes controller:

The lifecycle actions need to be added to the controller so that the transition-buttons added above work correctly. To do this open up: /app/controllers/recipes_controller and replace the existing auto_actions list with this:

```
auto_actions :all
```

Specifying :all will also add support for the lifecycle actions.

Step 4. Modify the Recipes Index page.

The Recipes index page needs to be modified so that it only shows published recipes when the user is a Guest, and all the Recipes for logged in users. So we need to do add the following named_scope to the Recipe model:

```
named_scope :viewable, lambda {|acting_user| if acting_user.signed_up?
  {}
  else
    { :conditions => "lifecycle_state='published'" }
  }
}
```

... which returned all Recipes for logged-in users and only published recipes to Guest users.

Note: The lambda block is used so that we can pass in a parameter to a named_scope, which in this case is a reference to the logged in user.

- The Recipe controller index action needs to be modified so that when a Guest user is viewing the Recipe listing page, only “published” Recipes will be shown. To do this, change the following line by inserting in the red italic text:

Original:

```
hobo_index Recipe.apply_scopes(:search => [params[:search],
  :title, :body], :order_by => parse_sort_param(:title,
  :country))
```

To:

```
hobo_index Recipe.viewable(current_user).apply_scopes(  
  :search => [params[:search], :title, :body],  
  :order_by => parse_sort_param(:title, :country))
```

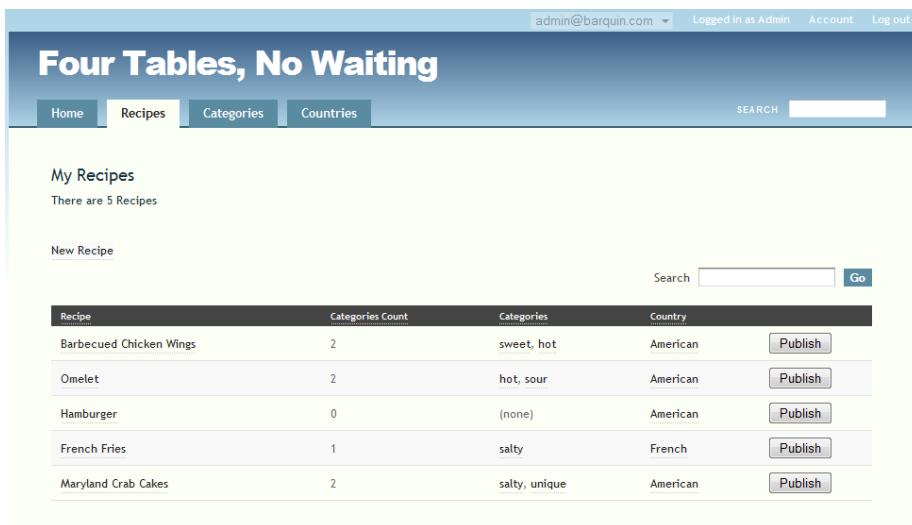
Step 5. Try it out.

Restart your server to see the changes. Following that, access the Recipe listing page as a Guest and you should see that there aren't any Recipes showing (this is because all the Recipes are in a state of 'Not Published'):



Figure 265: Guest view Recipes - All recipes are in state "Not Published"

If you login as a user, you should see your recipes showing with 'Publish' buttons next to each row:

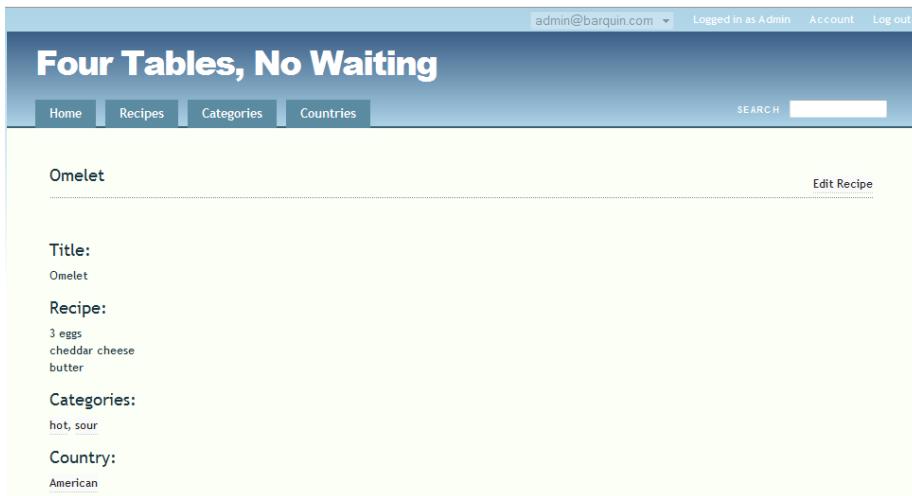


The screenshot shows a web application interface titled "Four Tables, No Waiting". At the top, there is a navigation bar with links for "Home", "Recipes" (which is the active tab), "Categories", and "Countries". On the right side of the header, there are links for "admin@barquin.com", "Logged in as Admin", "Account", and "Log out". Below the header, the main content area is titled "My Recipes" and displays a message "There are 5 Recipes". A "New Recipe" button is located at the top left of the list. To the right of the list is a search bar with a "Go" button. The list itself is a table with columns: "Recipe", "Categories Count", "Categories", and "Country". Each row contains a "Publish" button. The data in the table is as follows:

Recipe	Categories Count	Categories	Country	Action
Barbecued Chicken Wings	2	sweet, hot	American	<input type="button" value="Publish"/>
Omelet	2	hot, sour	American	<input type="button" value="Publish"/>
Hamburger	0	(none)	American	<input type="button" value="Publish"/>
French Fries	1	salty	French	<input type="button" value="Publish"/>
Maryland Crab Cakes	2	salty, unique	American	<input type="button" value="Publish"/>

Figure 266: Recipes ready to Publish.

To publish a Recipe just click on the 'Publish' button. For this example, let's publish the Omelet recipe. After clicking on the button, we will get the show page for the Omelet.

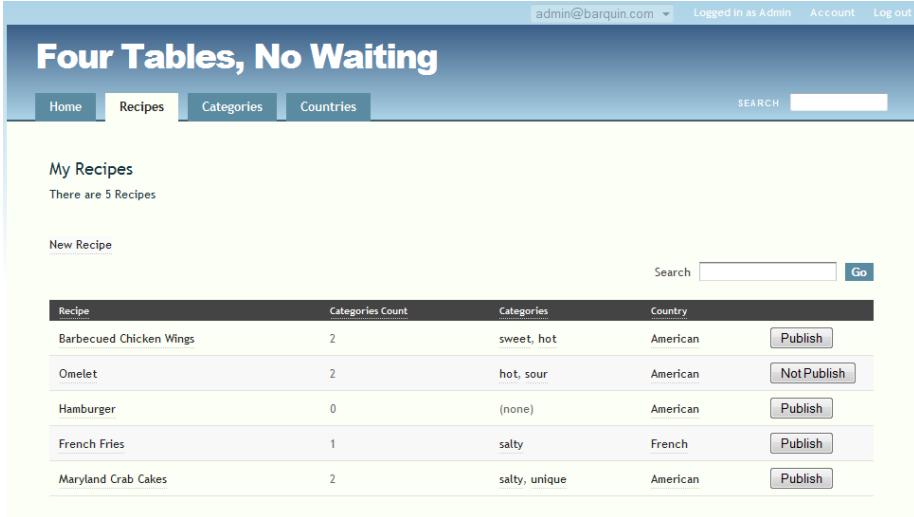


The screenshot shows the "Omelet" recipe details after it has been published. The title of the page is "Omelet". On the right side, there is a link "Edit Recipe". The recipe information is listed under sections: "Title:", "Recipe:", "Categories:", and "Country:". The details are as follows:

- Title:** Omelet
- Recipe:** 3 eggs
cheddar cheese
butter
- Categories:** hot, sour
- Country:** American

Figure 267: Omelet recipe after being placed in the "Published" state

Going back to the Recipe listing page the following is listed.

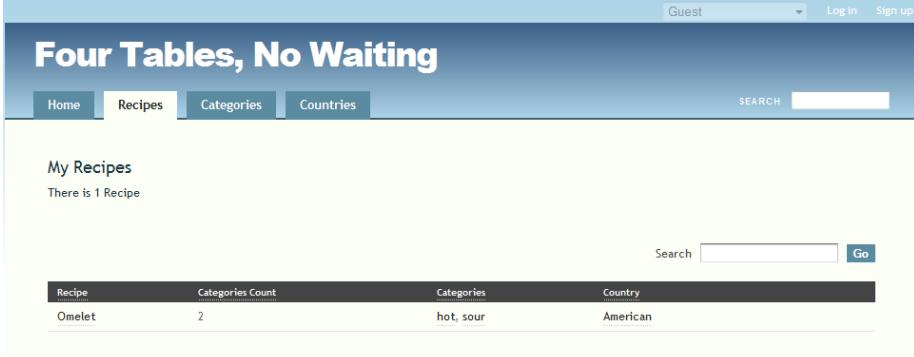


Recipe	Categories Count	Categories	Country	Action
Barbecued Chicken Wings	2	sweet, hot	American	Publish
Omelet	2	hot, sour	American	Not Publish
Hamburger	0	(none)	American	Publish
French Fries	1	salty	French	Publish
Maryland Crab Cakes	2	salty, unique	American	Publish

Figure 268: Recipe index with buttons for "Publish" and "Not Publish"

Since the Omelet recipe has been published, the only available action that is left is to 'Not Publish' it.

Going to the Recipe listing page as a Guest user, the Omelet recipe is visible:



Recipe	Categories Count	Categories	Country
Omelet	2	hot, sour	American

Figure 269: Guest user can only see the published Recipe

Step 6. Improve the navigation.

At this point we are able to either Publish or Not Publish our recipes. Our workflow is behaving as we expect. However, the navigation can be improved for clarity, if after on a transition button the page would just refresh instead of taking us to the show screen

for the recipe. To do this, we will need to override the default lifecycle actions in the Recipes controller.

For each defined transition, Hobo creates 2 controller actions: 1 for a GET request and 1 for a PUT request. For the Publish transition action, Hobo creates a publish action for GET requests and a do_publish action for PUT requests. The publish action would be used if you wanted to show a form before executing the transition action, i.e. If you wanted to collect comments from the user before he/she Publishes or Not Publishes, you could show a form with a comments box and a Publish/Not Publish submit button. However, in this example, only configure the application so that after a Recipe is Published or Not Published, the browser redirects back to the Recipe listing page. To do this add the following 2 actions to the Recipe controller just after the index action:

```
def do_publish
  do_transition_action :publish do
    redirect_to recipes_path
  end
end
def do_not_publish
  do_transition_action :not_publish do
    redirect_to recipes_path
  end
end
```

These actions override the default Hobo actions so that the page redirect can be specified after the transition has been executed. Once these actions are decided, if you access the Recipe index page and click on a Publish or Not Publish button, you'll see the page get refreshed. Once these actions are added, the page will be refreshed, if the recipe indexpage has been accessed, and the "publish" or "not publish" buttons have been selected.

So now you have a working Publish/Not Publish workflow for Recipes in the Four Tables application.

Note: This example is a basic implementation of Hobo lifecycles, but, it does serve as a good introduction to its various features. It is possible to implement workflows with numerous states and transitions, have the ability to implement more fine-grained security for each transition using the :available_to argument. Consult the full Hobo Lifecycles overview at <http://cookbook.hobocentral.net>

Tutorial 24

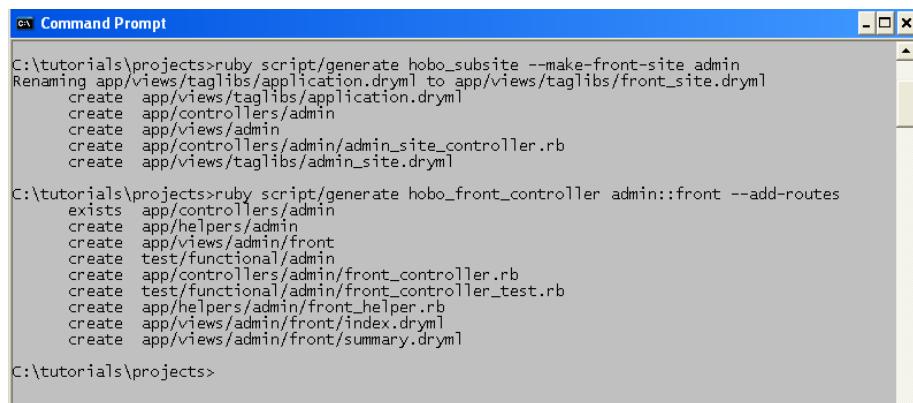
Creating an Administration Sub-Site

By Bryan Larsen

This tutorial will show how you can create an administrative sub-site for a Hobo. This will allow the administrator to create, update and destroy any database row without writing any view code.

Let's add an admin sub-site to the project created in the "Agile Project Manager" tutorial.

```
\projects> hobo g admin_subsite --make-front-site  
\projects> hobo g front_controller admin::front --add-routes
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The console output is as follows:

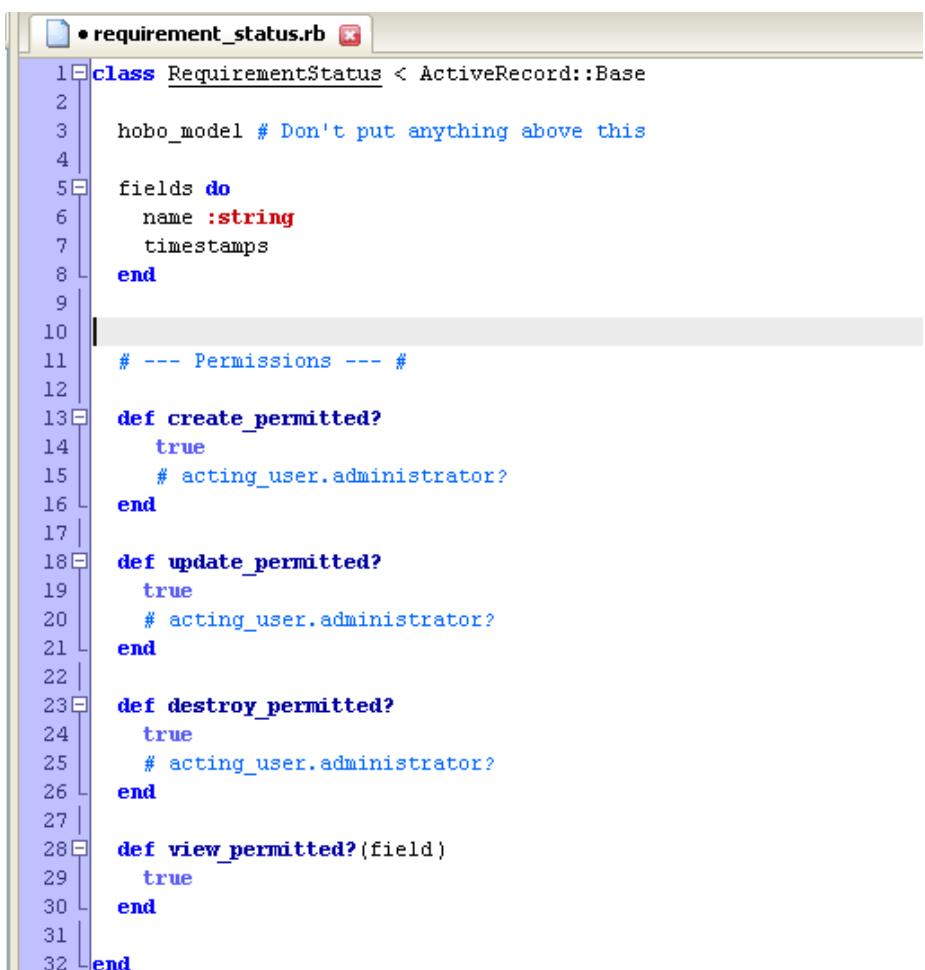
```
C:\tutorials\projects>ruby script/generate hobo_subsite --make-front-site admin  
Renaming app/views/taglibs/application.dryml to app/views/taglibs/front_site.dryml  
create app/views/taglibs/application.dryml  
create app/controllers/admin  
create app/views/admin  
create app/controllers/admin/admin_site_controller.rb  
create app/views/taglibs/admin_site.dryml  
C:\tutorials\projects>ruby script/generate hobo_front_controller admin::front --add-routes  
exists app/controllers/admin  
create app/helpers/admin  
create app/views/admin/front  
create test/functional/admin  
create app/controllers/admin/front_controller.rb  
create test/functional/admin/front_controller_test.rb  
create app/helpers/admin/front_helper.rb  
create app/views/admin/front/index.dryml  
create app/views/admin/front/summary.dryml  
C:\tutorials\projects>
```

Figure 270: Generator console output for creating an admin sub-site

Model Modifications

We would like to "hide" our code table maintenance the admin sub-site. Currently we have one code table, `requirement_statuses` (model = `RequirementStatus`).

First change all of the permissions for this model to "true", as only an administrator will be able to access this sub-site:



The screenshot shows a code editor window with the file 'requirement_status.rb' open. The code defines a class 'RequirementStatus' that inherits from 'ActiveRecord::Base'. It includes fields for a string name and timestamps. The file also contains permission logic for create, update, destroy, and view operations, all returning true unless the acting user is not an administrator.

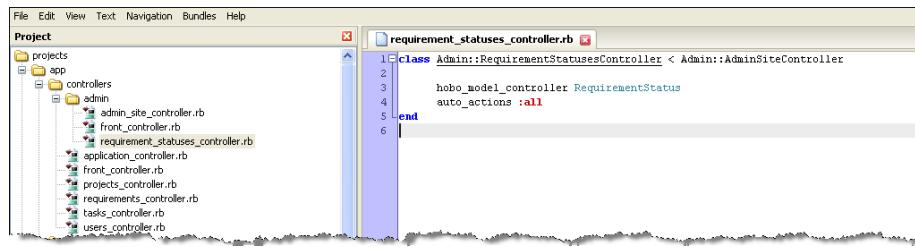
```
1 class RequirementStatus < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   fields do
6     name :string
7     timestamps
8   end
9
10  # --- Permissions --- #
11
12  def create_permitted?
13    true
14    # acting_user.administrator?
15  end
16
17  def update_permitted?
18    true
19    # acting_user.administrator?
20  end
21
22  def destroy_permitted?
23    true
24    # acting_user.administrator?
25  end
26
27  def view_permitted?(field)
28    true
29  end
30
31
32 end
```

Figure 271: Requirement Status Permissions

Controller Modifications

We need to move the controller for RequirementStatus to the admin folder and modify to:

```
class Admin::RequirementStatusesController < Admin::AdminSiteController
  hobo_model_controller RequirementStatus
  auto_actions :all
end
```



A screenshot of a code editor window. The left pane shows a project tree with a 'Project' header, followed by 'app' and 'admin' folders. Inside 'admin', there are several files: admin_site_controller.rb, front_controller.rb, requirement_statuses_controller.rb (which is currently selected), application_controller.rb, projects_controller.rb, requirements_controller.rb, tasks_controller.rb, and users_controller.rb. The right pane displays the contents of requirement_statuses_controller.rb:

```
1 class Admin::RequirementStatusesController < Admin::AdminSiteController
2   hobo_model_controller RequirementStatus
3   auto_actions :all
4
5 end
6
```

Figure 272: View of the Admin folder contents

At this stage you should be able to run your application. If you browse to "/admin", you can create, remove, update and destroy any requirement status:

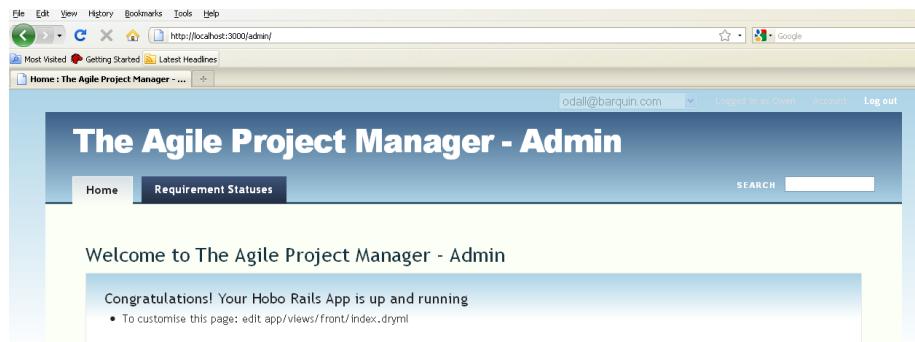


Figure 273: View of the Admin Sub-Site

Tutorial 25

Using Hobo Database Index Generation

By Matt Jones

Defining effective indexes on your data can give massive database performance benefits in any application. To further this goal, Hobo's migration generator attempts to provide useful indices without any additional code, using shorthand to defining indices.

The :index Option

Throughout the index generator API, the :index parameter is used to switch indexing on/off and specify an explicit name for an index. The convention is:

- :index => true will switch on indexing for a field not indexed by default; the name used is the default name generated by Rails.
- :index => false will switch off automatic indexing for a field.
- :index => 'name' will specify a name for the generated index. Note: that some databases require that index names be unique across the entire database, not just the individual table.

Note: Oracle's 30-character limit for entity names causes problems with the default naming scheme that Rails uses for indices. The Oracle driver for ActiveRecord attempts to mitigate this by shortening overlong index names in add_index. Unfortunately, this will break the generated down migrations (which rely on the original index names). The best short-term solution is to pass a manual index name parameter wherever possible.

Automatic Indexing

The belongs_to associations will automatically declare an index on their foreign key field; polymorphic belongs_to will declare a multi-field index on [association_type, foreign_key].

Example:

```
class SomeModel < ActiveRecord::Base
  hobo_model
  belongs_to :other_model
  belongs_to :another_model, :index => 'some_random_name'
  belongs_to :fooable, :polymorphic => true
end
```

Will generate the following in an up migration:

```
add_index :some_models, :other_model_id
add_index :some_models, :another_model_id, :name => 'some_random_name'
add_index :some_models, [:fooable_type, :fooable_id]
```

Lifecycle state fields will also be automatically indexed, as will the `inheritance_column` of an STI parent class.

Indexing in the ‘fields do’ block

Within the standard fields block, indexes can be declared as part of a field just like the `:required` or `:unique` options. Fields that also have the `:unique` option will automatically declare a unique index.

Example:

```
class SomeModel < ActiveRecord::Base
  fields do
    name :string, :index => true
    unique_field :string, :unique, :index => 'foo'
  end
end
```

Will generate the following in an up migration:

```
add_index :some_models, :name
add_index :some_models, :unique_field, :name => 'foo', :unique => true
```

Indexing in the model

More complicated indexes may need to be declared outside the fields block. For instance, specific slow-running SQL queries may benefit from a multi-field index. The `index` method provides a simple interface for specifying any type of index on the model.

Example:

```
class SomeModel < ActiveRecord::Base
  fields do
    last_name :string
    first_name :string
  end
  index [:last_name, :first_name]
end
```

Will generate the following in an up migration:

```
add_index :some_models, [:last_name, :first_name]
```

When declaring a multi-field index, the order is relevant - consult your database's manual for more detail (for example, section 7.4.3 of the MySQL 5.0 Reference).

The index method currently supports two options:

- `:name` - use to specify the name of the index. If not given, the Rails default will be used.
- `:unique` - passing `:unique => true` will specify the creation of a unique index.

Chapter 6

DEPLOYING YOUR APPLICATIONS

Introductory Comments

Tutorial 26 – Installing and using the Git Version Control System

Tutorial 27 – Rapid Deployment Using Heroku.com

Introductory Concepts and Comments

There isn't much use in developing an application that you don't put into production. This chapter is devoted to helping you put together the tools necessary to use one of the most innovative cloud computing sites today—heroku.com

Once you configure your computer to work with the source code configuration management software called “git” and create your subscription with Heroku, you will be able to publish a new app in a manner of minutes.

Of course, if you are an experienced Rails developer you can publish any Hobo app on your existing infrastructure. If you haven't tried Heroku, I encourage you to do so. This is the wave of the future.

Tutorial

Installing and Using Git

Git has become the standard distributed version control system for Ruby and Rails applications, in part due the success of the social coding site, <http://github.com>.

On Github you will find thousands of public and private projects aided by the extremely useful Web 2.0 user interface designed with distributed coding in mind. Hobo's code

base is located there. You can access the source, view the change history, and view the branching and merging of code as members of the open source community participate:

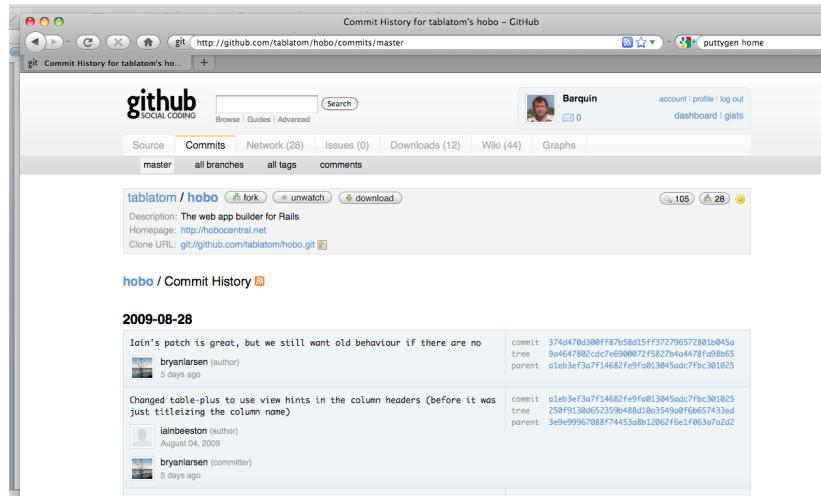


Figure 274: Hobo source code on github.com

It is also where the Hobo gems are stored:

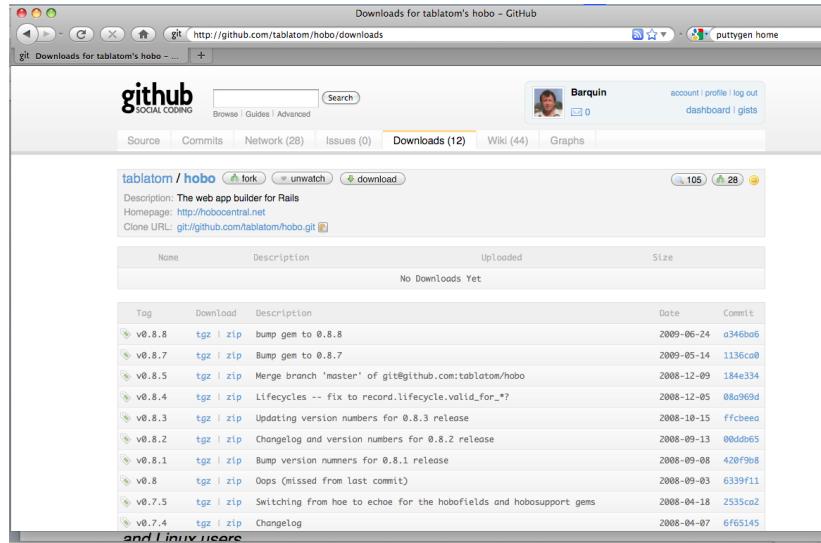


Figure 275: Hobo gems are also available on github.com

Barquin International also uses Github as the central hub for developing several large-scale Hobo projects that involve participants from several countries.

In this tutorial we will focus on the Windows user, as git is much easier for Mac OS X and Linux users. You only need to learn a few commands for basic usage. There are many outstanding resources for more in-depth understanding, including the excellent <https://peepcode.com/products/git-internals-pdf> by Scott Chacon.

There is an excellent tutorial for Mac users:

<http://help.github.com>

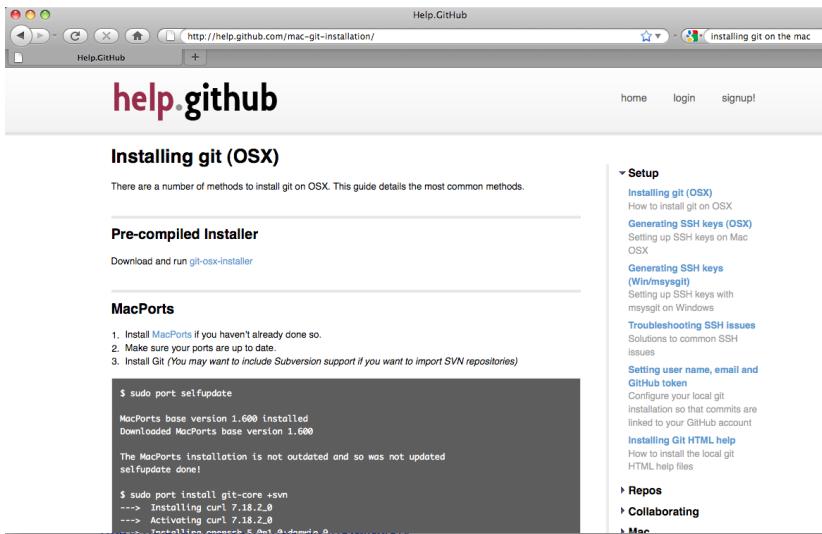


Figure 276: Installing Git for Mac OSX

Ok. Let's get the software we need for Git:

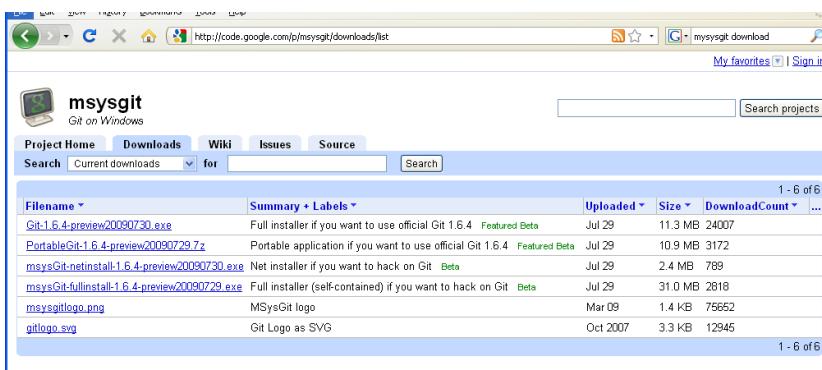


Figure 277: Download the msysgit installer for Windows

Download and run the git installer for windows:

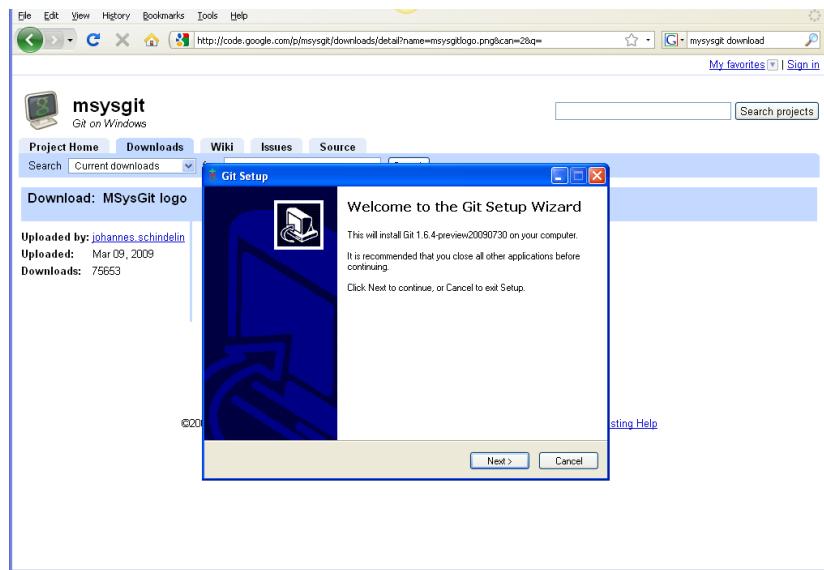


Figure 278: Running the Git Setup Wizard

Select the following options:

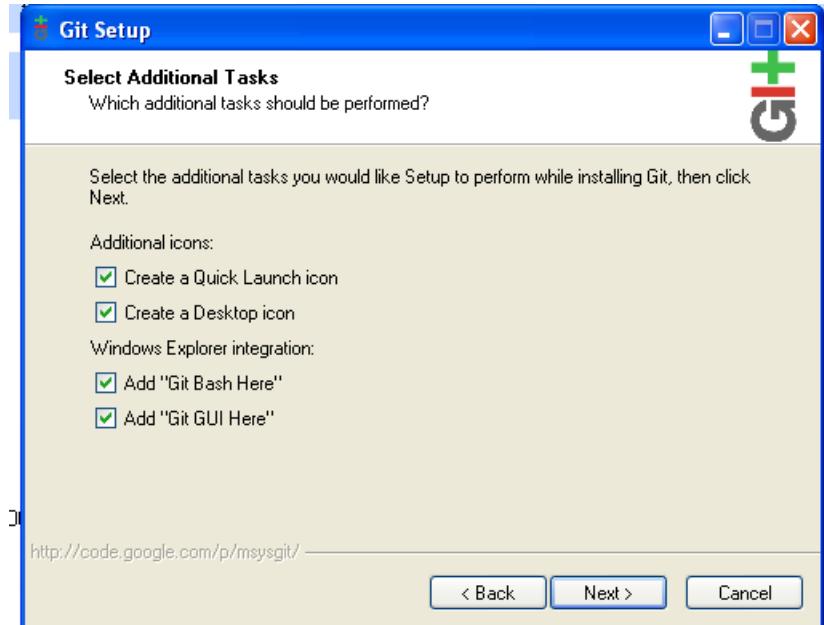


Figure 279: Git setup options

Select the “Use OpenSSH” option:

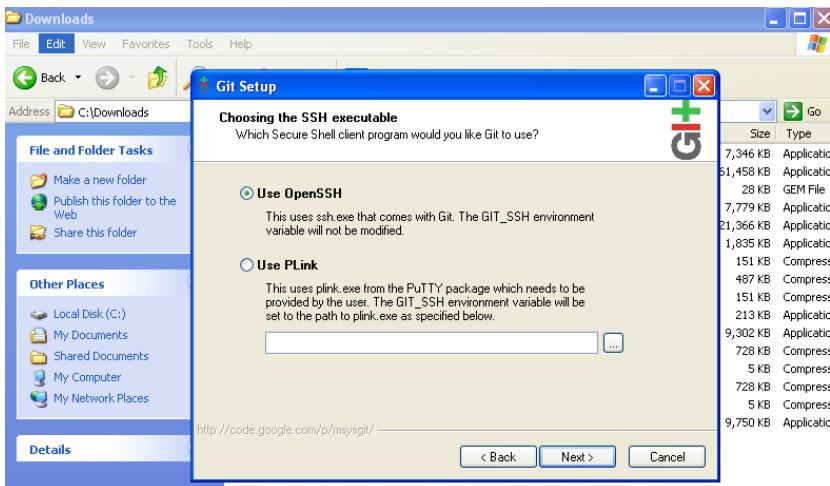


Figure 280: Select the OpenSSH option

Allow the installer to configure running git from the Windows command prompt:

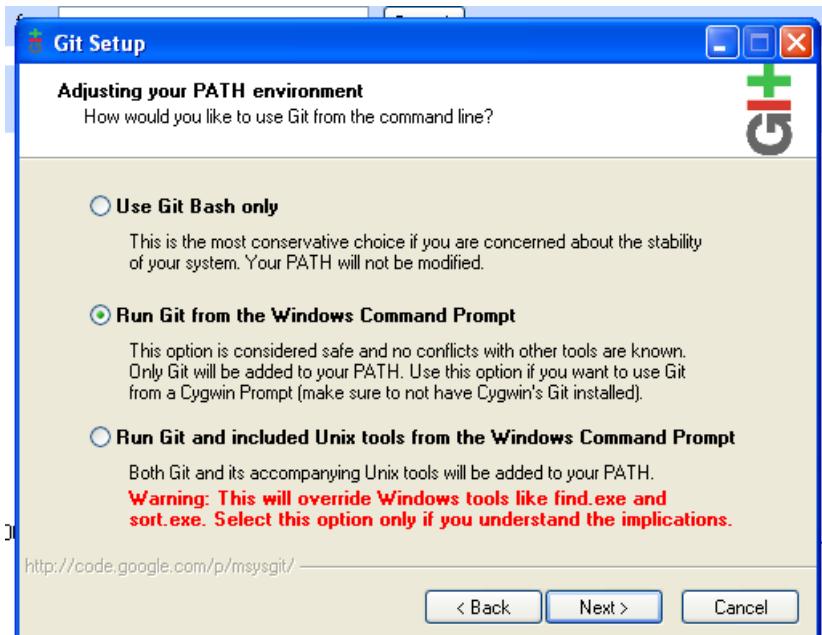


Figure 281: Select to option to run Git from the Windows command prompt

Next select the CR/LF behavior option:

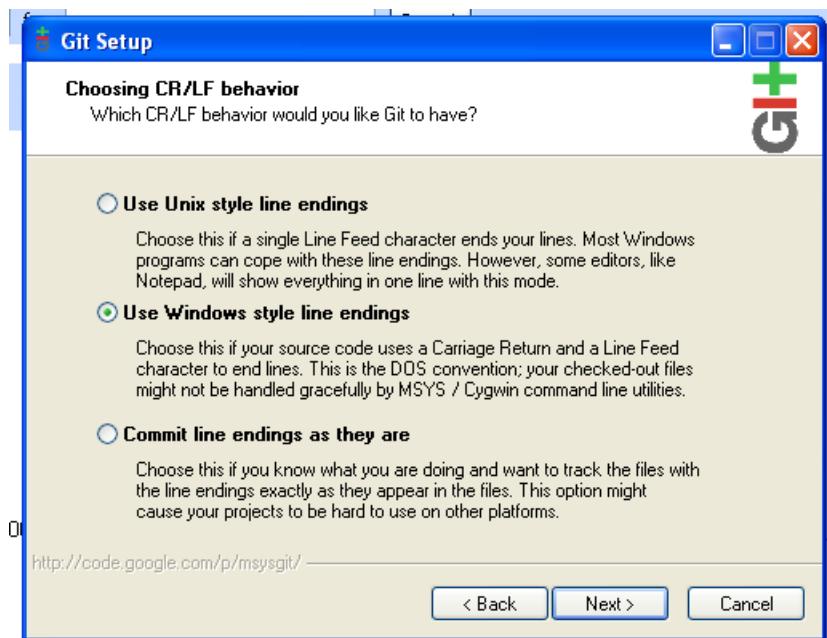


Figure 282: Select Windows style line endings

After the installation is complete, the release notes will be displayed.

Now, download the PuTTYgen RSA/DSA secure key generator from this URL:

<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

Run the downloaded `puttygen.exe` file to install:

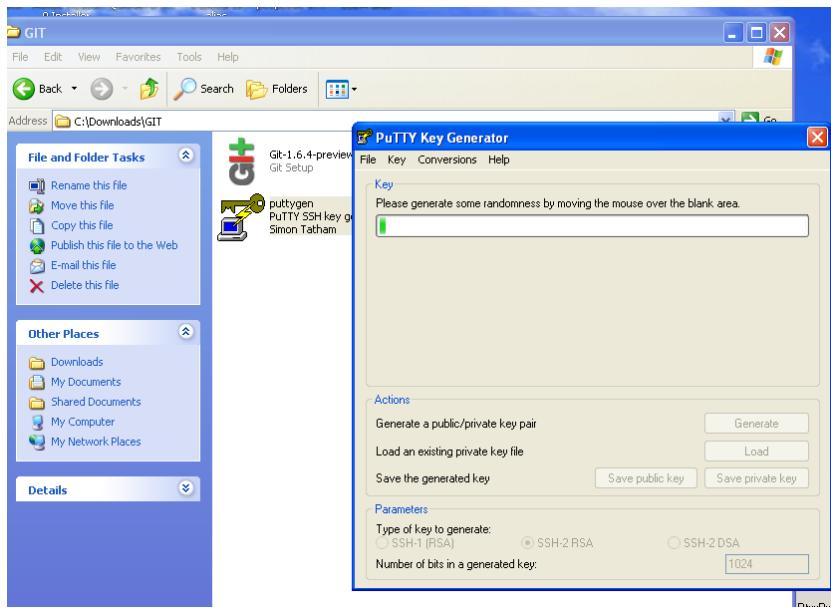


Figure 283: Running the PuTTY Key Generator install

Open up the application and start the process of generating key pairs:

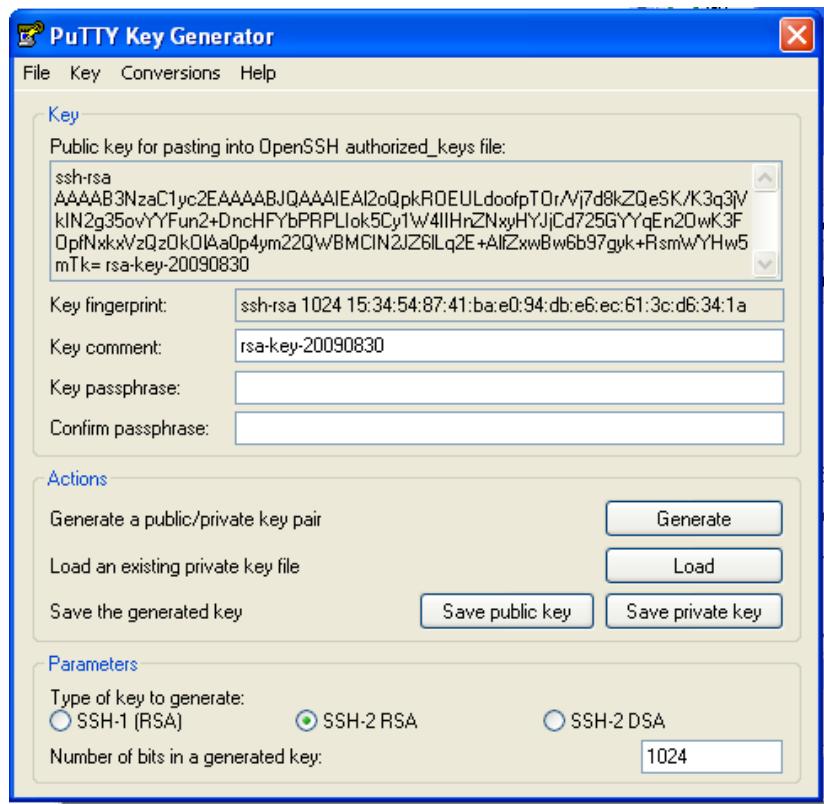


Figure 284: Generate SSH key pairs for use with Git

Saving the files with default names:

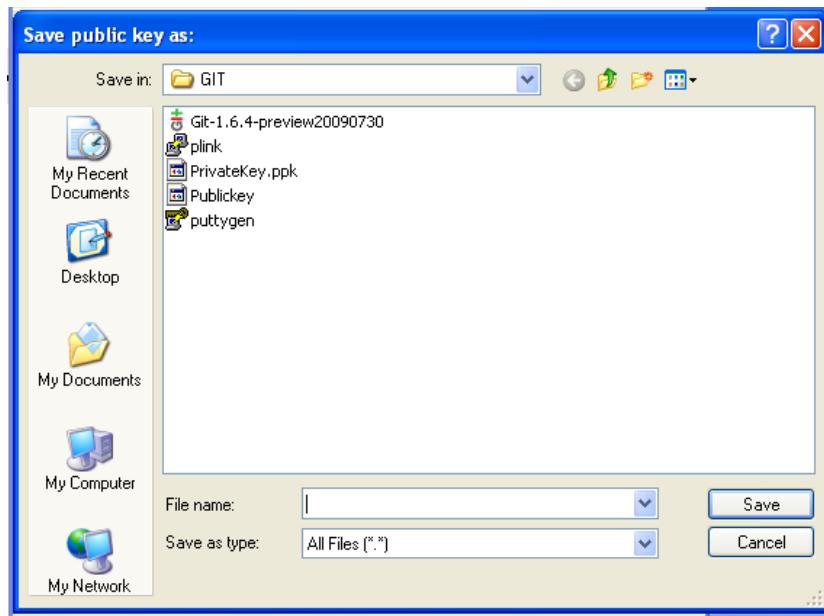
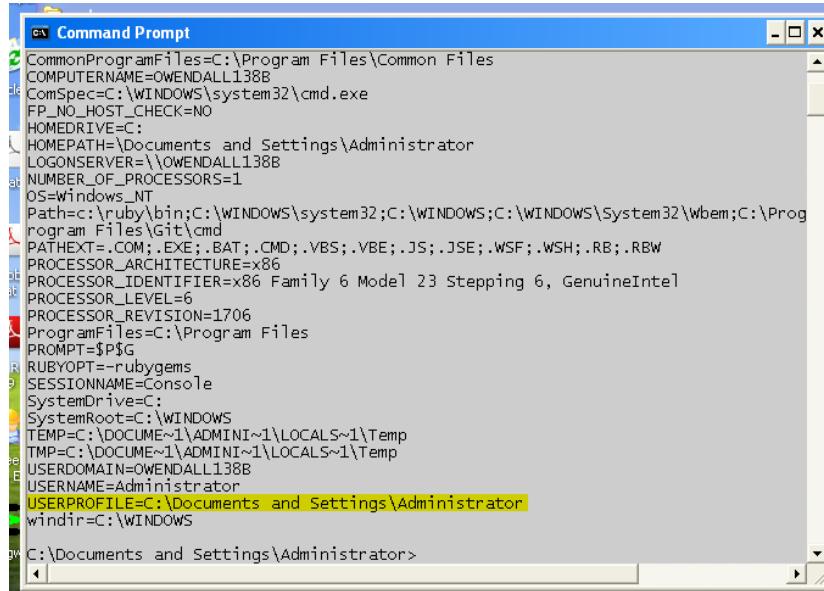


Figure 285: The default file names generated by PuTTYGen

Private key = “PrivateKey.ppk”

Public Key = “Publickey”

You will need to rename these and put them in the USERPROFILE environment setting default location that most systems will look.



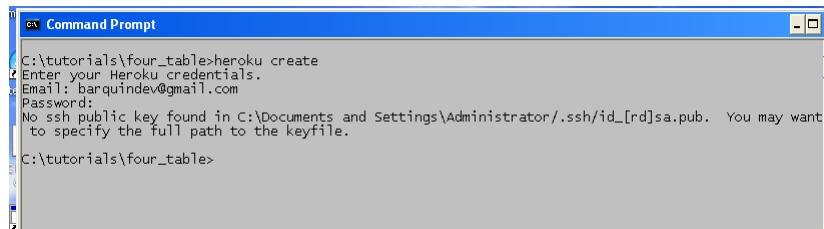
```

CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=OWENDALL138B
ComSpec=C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK=NO
HOMEDRIVE=C:
HOMEPATH=\Documents and Settings\Administrator
LOGONSERVER=\OWENDALL138B
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Path=c:\bin;c:\WINDOWS\system32;c:\WINDOWS;c:\WINDOWS\System32\wbem;c:\Program Files\Git\cmd
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.RB;.RBW
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 23 Stepping 6, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=1706
ProgramFiles=C:\Program Files
PROMPT=$P$G
RUBYOPT=-rubygems
SESSIONNAME=Console
SystemDrive=C:
SystemRoot=C:\WINDOWS
TEMP=C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp
TMP=C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp
USERDOMAIN=OWENDALL138B
USERNAME=Administrator
USERPROFILE=C:\Documents and Settings\Administrator
windir=C:\WINDOWS
C:\Documents and Settings\Administrator>

```

Figure 286: Locating your USERPROFILE setting

I was logged in as the user “Administrator” in windows when I tried to use the Heroku gem (see next chapter):



```

C:tutorials\four_table>heroku create
Enter your Heroku credentials.
Email: barquindev@gmail.com
Password:
No ssh public key found in C:\Documents and Settings\Administrator/.ssh/id_[rd]sa.pub. You may want
to specify the full path to the keyfile.
C:tutorials\four_table>

```

Figure 287: View of "no ssh public key found" error

Heroku was looking for the file `id_rsa.pub` (since I was used the RSA option with PuttyGen) in the default folder:

C:\Documents and Settings\Administrator\.ssh

We can move the keys as follows:

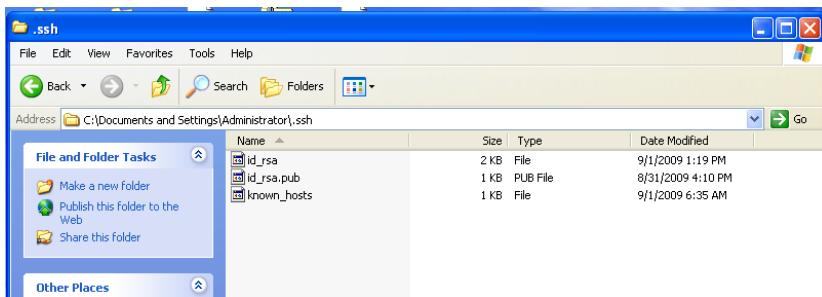


Figure 288: Naming your SSH key pairs

(The known_hosts file will be created and updated automatically when you connect to Heroku.)

Now you are ready to use Git. See the next tutorial to learn how Git is used to deploy your application to Heroku.com.

Tutorial

Rapid Deployment with Heroku

We have been following with great interest the development of Heroku for almost two years. I recently tracked down my initial “Invitation to Heroku Beta” email invitation:

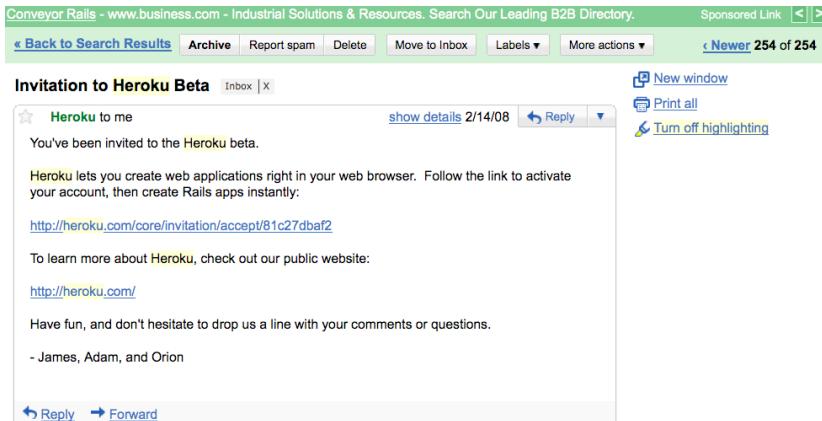


Figure 289: The original Heroku beta invitation

According to Wikipedia, it has been in development since June of 2007, with an initial investment of about \$3 million dollars. It was one of the first to use the new Amazon Elastic Compute Cloud (EC2) as its infrastructure. <http://aws.amazon.com/ec2/>

For more details on this innovative architecture, see:

<http://heroku.com/how/architecture>

For information for pricing and options:

<http://heroku.com/pricing#blossom-1>

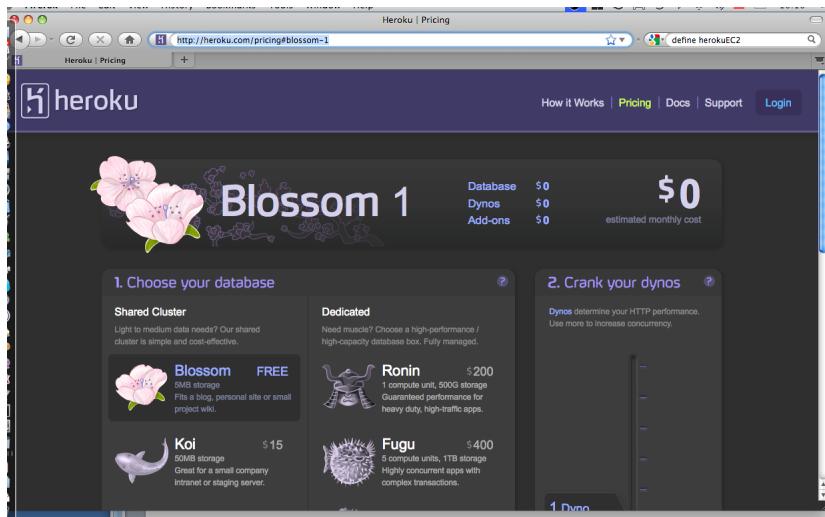


Figure 290: Using the free "Blossom" database hosting option on Heroku.com

For this tutorial, we are going to use the free “Blossom” version for apps under 5MB in size. In addition to choosing more storage capacity, you can add “Dynos” (processing power) to suit your needs and choose your replication and backup options. The database backend provided by Heroku is PostgreSQL, a rock-solid choice in the open source world.

Of course, you can always host your database elsewhere and use Heroku for your Hobo or Rails front end. The nice thing about Heroku is the database migration and setup is transparent, so you can develop your app using SQLite and then deploy your app to Heroku’s PostgreSQL back-end.

For this tutorial we will use the “four_table” application built into the earlier tutorials and deploy it to Heroku.

Step 1: Install and Configure git

If you haven’t done so already, please follow the instructions in Tutorial 23 – Installing and Using git.

Step 2: Create an Account at Heroku.com

Go to <http://heroku.com/signup>:

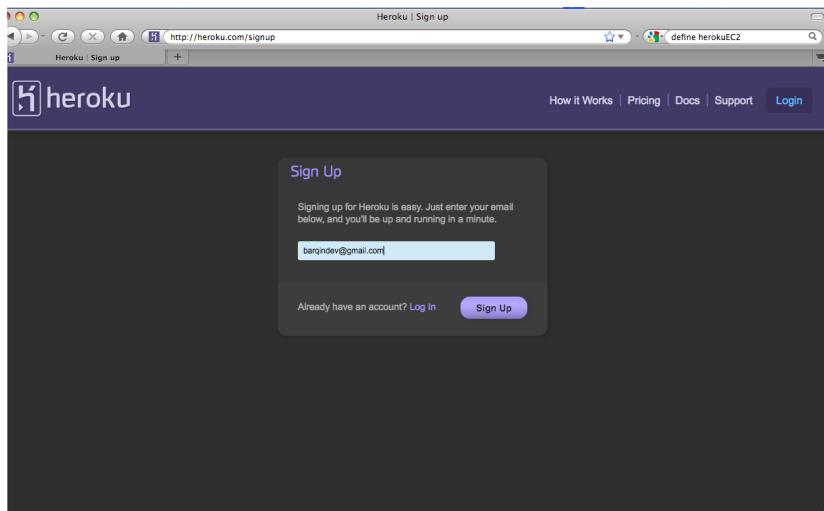


Figure 291: Sign Up for a Heroku account

Enter the email address you wish to use for communication with Heroku. Heroku will send a confirmation email with a link to access your account.

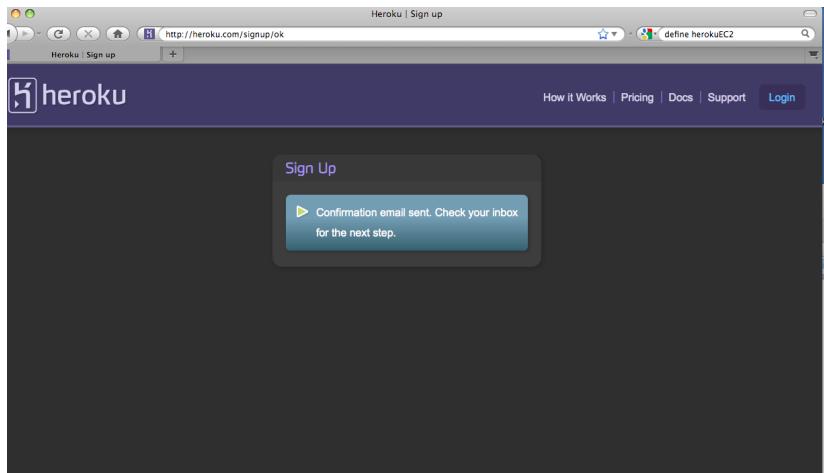


Figure 292: Heroku notification that "Confirmation email sent"

Going to your email to access the confirmation link you will need:

TUTORIAL

RAPID DEPLOYMENT WITH HEROKU

CHAPTER 6

DEPLOYING YOUR APPLICATIONS

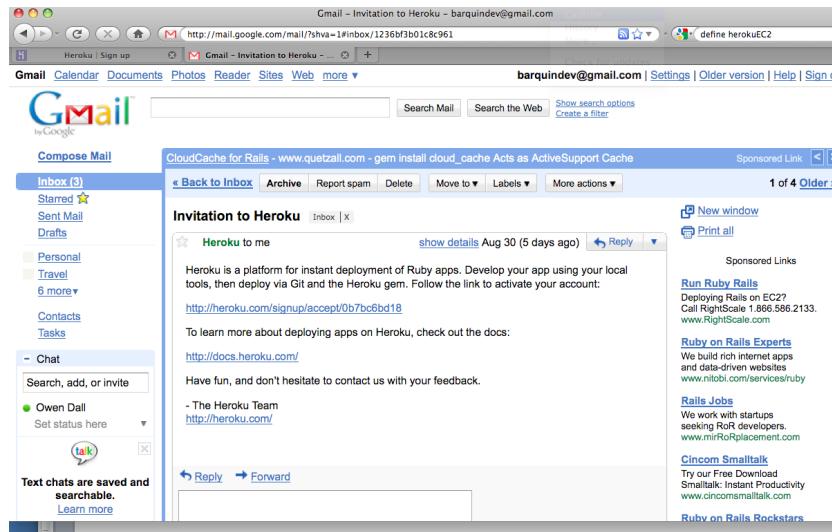


Figure 293: Locating your "Invitation to Heroku" email

When you click the confirmation link, you should see a screen similar to the following:

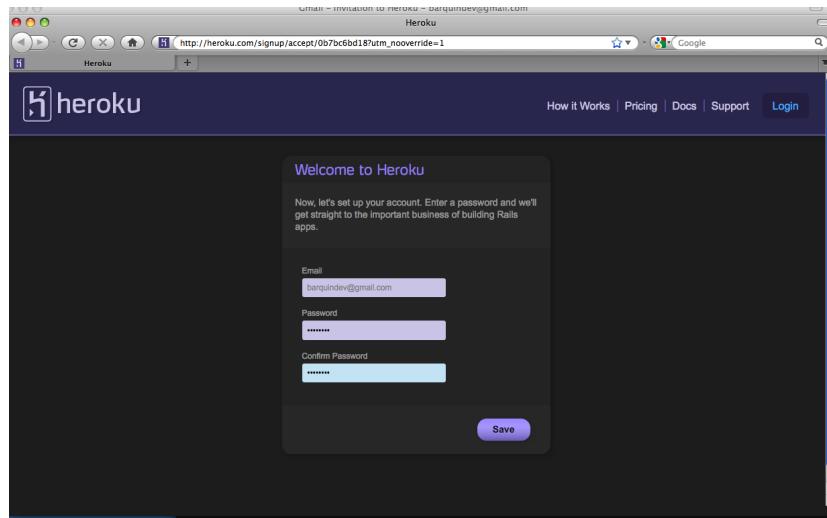


Figure 294: The "Welcome to Heroku" signup page

And this when you finish:

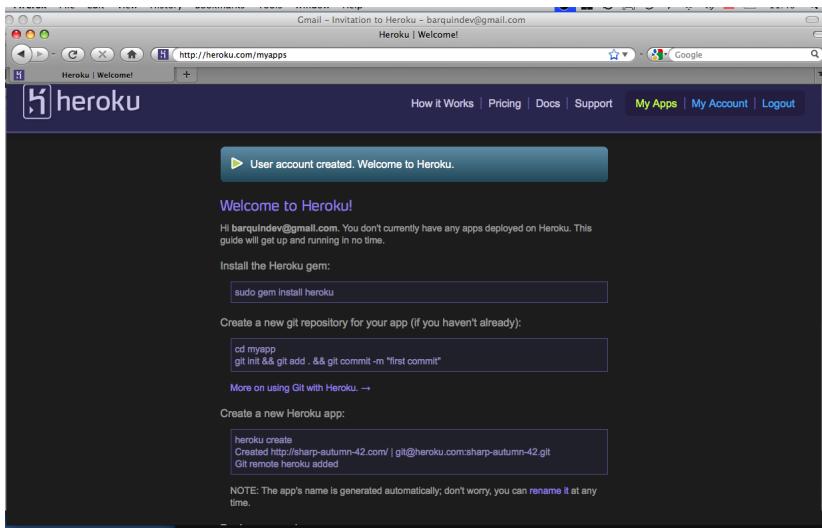


Figure 295: The "Account Created" message at Heroku.com

The instructions that are displayed on the “Welcome to Heroku!” splash screen are tailored for the Mac or Linux user. We’ll provide the Windows equivalents below.

Step 3: Install the Heroku Gem

Go to command prompt and type the following command:

```
C:\Sites\tutorials>gem install heroku
```

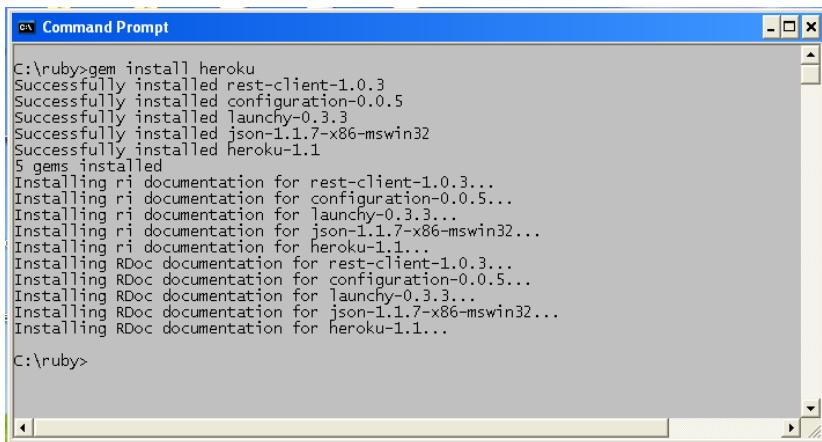


Figure 296: Installing the Heroku Ruby gem

Note: The other four gems that are installed along with the Heroku gem.

Step 4: Use git to package your application

Initialize git for your app:

```
\four_table> git init
```

Tell git to add all the files in all folders to the project:

```
\four_table> git add .
```

Tell git to commit these additions and enter an optional message that helps for version control:

```
\four_table> git commit -m "My first Commit"
```

Step 5: Use the “heroku create” command to Initialize your application

Change your directory to c:\tutorials\my-first-app and then execute the command while in the root directory of the app.

```
\four_table> heroku create four_table
```

Note: Heroku application names need to be globally unique. As such, you won't be able to use four_table. If you leave off the name, it will generate a random one consisting of 3 words. Substitute your application name wherever tutorial refers to four_table (in the context of Heroku, your local files will remain the same).

```

drwxr-xr-x 10064  ch_1c/performance/benchmark
create mode 100644 script/performance/profiler
create mode 100644 script/plugin
create mode 100644 script/runner
create mode 100644 script/server
create mode 100644 test/fixtures/categories.yml
create mode 100644 test/fixtures/category_assignments.yml
create mode 100644 test/fixtures/countries.yml
create mode 100644 test/fixtures/names.yml
create mode 100644 test/fixtures/users.yml
create mode 100644 test/functional/categories_controller_test.rb
create mode 100644 test/functional/countries_controller_test.rb
create mode 100644 test/functional/front_controller_test.rb
create mode 100644 test/functional/recipes_controller_test.rb
create mode 100644 test/functional/users_controller_test.rb
create mode 100644 test/performance/browsing-test.rb
create mode 100644 test/test_helper.rb
create mode 100644 test/unit/category_assignment_test.rb
create mode 100644 test/unit/category_test.rb
create mode 100644 test/unit/country_test.rb
create mode 100644 test/unit/recipe_test.rb
create mode 100644 test/unit/user_test.rb

C:\tutorials\four_table>heroku create four_table
Name must start with a letter and can only contain letters, numbers, and dashes
C:\tutorials\four_table>heroku create four-table
Created http://four-table.herokuapp.com/ | git@heroku.com:four-table.git
Git remote heroku added
C:\tutorials\four_table>

```

Figure 297: Console output from the "heroku create" command

Note: The first time you try to create using the heroku gem you will be prompted to enter the user name and password you provided heroku when creating your account:

Looking at the output you can see that we could not create the application “four_table”, as Heroku does not allow an underscore in a name. We need to change the name of our app and try again:

```
\four_table> heroku create four-table
```

And then:

```
\four_table> git push heroku master
```

```

C:\tutorials\four_table>git push heroku master
Counting objects: 151, done.
Compressing objects: 100% (128/128), done.
Writing objects: 100% (151/151), 167.89 KiB, done.
Total 151 (delta 19), reused 0 (delta 0)

-----> Heroku receiving push
-----> Rails app detected
      Compiled slug size is 120K
-----> Launching...
      !     App crashed during startup
      !     Visit http://four-table.herokuapp.com to see the crashlog
To git@heroku.com:four-table.git
 * [new branch]    master -> master
C:\tutorials\four_table>

```

Figure 298: Using heroku git push

OK. So our app launched, but then crashed. What we forgot to do is to inform Heroku to add the Hobo gem to our application. We can do this by adding an instruction:

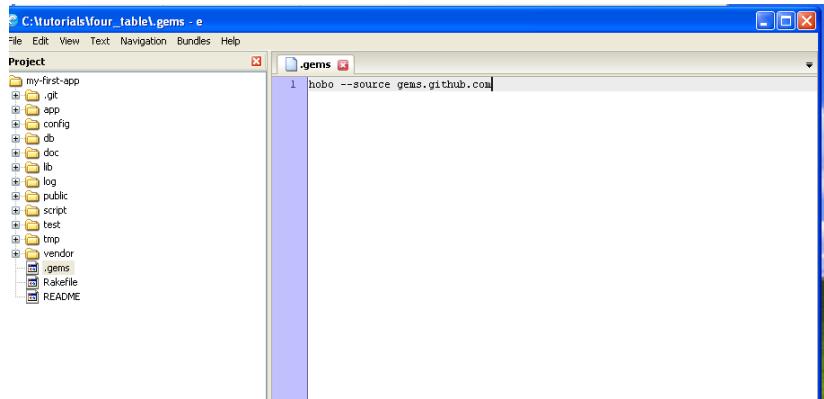


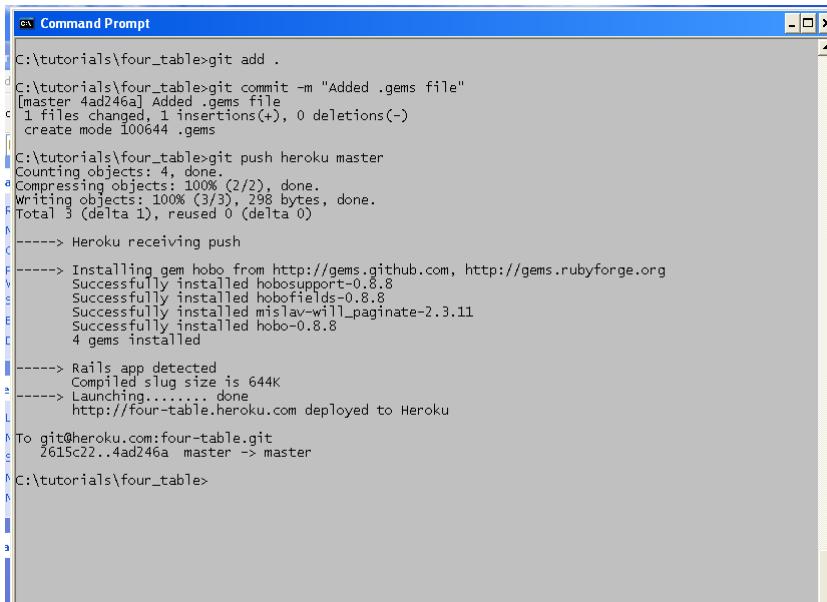
Figure 299: Telling Heroku where to find your application's gems

Create a text file with the name `.gems` in the application's root folder. Add the following text:

```
hobo -source gems.github.com
```

Now, we need to use GIT again to add these changes and push them to Heroku:

```
\four_table> git add .
\four_table> git commit -m "Added .gems definition file"
\four_table> git push heroku master
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command history is as follows:

```
C:\tutorials\four_table>git add .
C:\tutorials\four_table>git commit -m "Added .gems file"
[master 4ad246a] Added .gems file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 .gems
C:\tutorials\four_table>git push heroku master
Counting objects: 4, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 298 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)

----> Heroku receiving push
----> Installing gem hobo from http://gems.github.com, http://gems.rubyforge.org
      Successfully installed hobosupport-0.8.8
      Successfully installed hobofields-0.8.8
      Successfully installed mislav-will_paginate-2.3.11
      Successfully installed hobo-0.8.8
      4 gems installed
----> Rails app detected
      Compiled slug size is 644K
----> Launching..... done
      http://four-table.herokuapp.com deployed to Heroku
To git@heroku.com:four-table.git
 2615c22..4ad246a master -> master
C:\tutorials\four_table>
```

Figure 300: Adding your “.gems” config file to your git repository

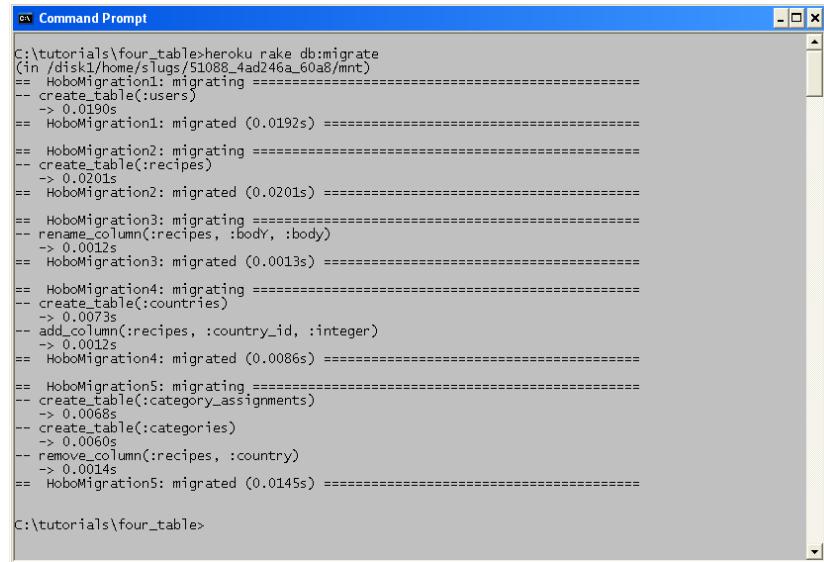
Note: The additional gems that Hobo uses (dependencies) were automatically installed as well.

Step 6: Migrate your database schema to Heroku

Your UI is up and running, but your database has not been migrated until you do this:

```
\four_table> heroku rake db:migrate
```

TUTORIAL **CHAPTER 6**
RAPID DEPLOYMENT WITH HEROKU **DEPLOYING YOUR APPLICATIONS**



```
C:\tutorials\four_table>heroku rake db:migrate
(in /disk1/home/r7ugs/51088_4ad246a_60a8/mnt)
== HboMigration1: migrating =====
-- create_table(:users)
-> 0.0190s
== HboMigration1: migrated (0.0192s) =====
== HboMigration2: migrating =====
-- create_table(:recipes)
-> 0.0201s
== HboMigration2: migrated (0.0201s) =====
== HboMigration3: migrating =====
-- rename_column(:recipes, :bodyv, :body)
-> 0.0012s
== HboMigration3: migrated (0.0013s) =====
== HboMigration4: migrating =====
-- create_table(:countries)
-> 0.0073s
-- add_column(:recipes, :country_id, :integer)
-> 0.0012s
== HboMigration4: migrated (0.0086s) =====
== HboMigration5: migrating =====
-- create_table(:category_assignments)
-> 0.0068s
-- create_table(:categories)
-> 0.0060s
-- remove_column(:recipes, :country)
-> 0.0014s
== HboMigration5: migrated (0.0145s) =====
C:\tutorials\four_table>
```

Figure 301: Migrating your database schema to Heroku.com

Step 7: Test your application

Log into Heroku.com to see the application URL:

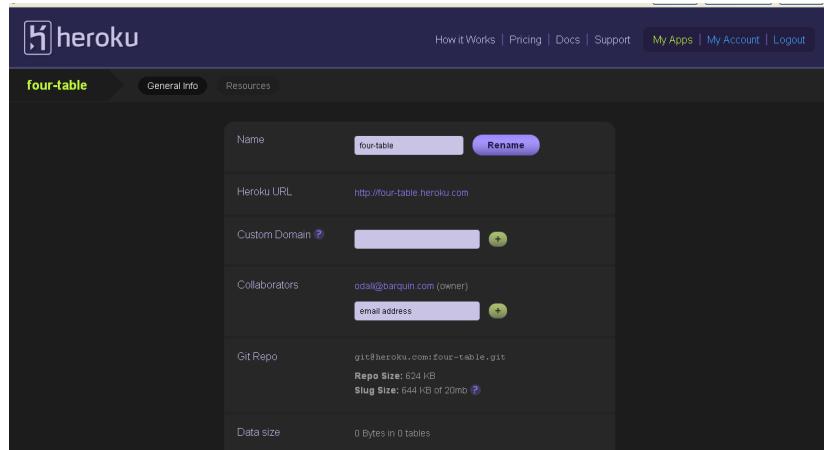


Figure 302: Testing your Heroku app

<http://four-table.herokuapp.com>

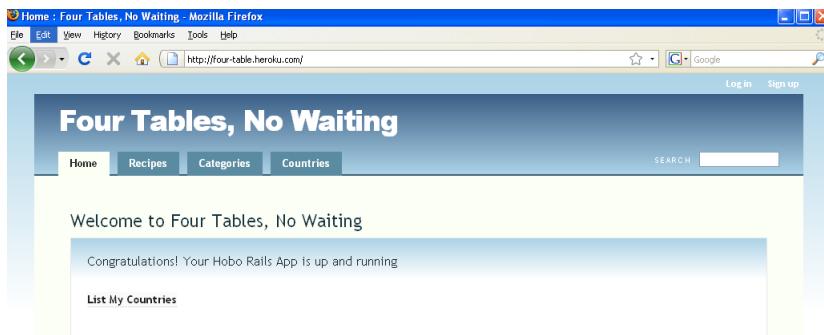


Figure 303: Running the "Four Table" app on Heroku.com

Note: You can set up your application to use an existing domain name instead of heroku.com. See the information located on this link:

<http://docs.heroku.com/custom-domains>

Step 8: Use the Taps gem to push data to your app on Heroku

The data we created in earlier tutorials has not yet been loaded to Heroku. However, we can easily do this with Heroku by installing the “taps” gem:

```
\four_table> gem install taps
```

The screenshot shows a Command Prompt window with the title 'Command Prompt'. The window displays the output of the command 'gem install taps'. The output shows the successful installation of several gems: sinatra-0.9.2, activesupport-2.2.2, activerecord-2.2.2, thor-0.9.9, sequel-3.0.0, and taps-0.2.19. It also shows the installation of documentation for these gems, including ri documentation and RDoc documentation for each installed gem.

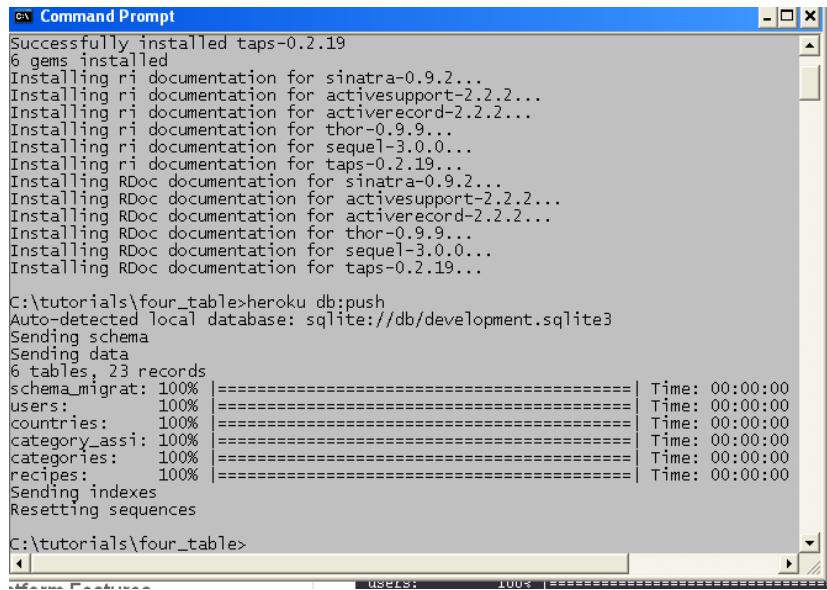
```
C:\tutorials\four_table>gem install taps
Successfully installed sinatra-0.9.2
Successfully installed activesupport-2.2.2
Successfully installed activerecord-2.2.2
Successfully installed thor-0.9.9
Successfully installed sequel-3.0.0
Successfully installed taps-0.2.19
6 gems installed
Installing ri documentation for sinatra-0.9.2...
Installing ri documentation for activesupport-2.2.2...
Installing ri documentation for activerecord-2.2.2...
Installing ri documentation for thor-0.9.9...
Installing ri documentation for sequel-3.0.0...
Installing ri documentation for taps-0.2.19...
Installing RDoc documentation for sinatra-0.9.2...
Installing RDoc documentation for activesupport-2.2.2...
Installing RDoc documentation for activerecord-2.2.2...
Installing RDoc documentation for thor-0.9.9...
Installing RDoc documentation for sequel-3.0.0...
Installing RDoc documentation for taps-0.2.19...
C:\tutorials\four_table>
```

Figure 304: Installing the Taps gem to upload data to Heroku.com

Note: Several other dependencies are also installed along with Taps.

You can use the following single command to upload your existing (local) data to your version on Heroku:

```
\four_table> heroku db:push
```



```
C:\tutorials\four_table> heroku db:push
Successfully installed taps-0.2.19
6 gems installed
Installing ri documentation for sinatra-0.9.2...
Installing ri documentation for activesupport-2.2.2...
Installing ri documentation for activerecord-2.2.2...
Installing ri documentation for thor-0.9.9...
Installing ri documentation for sequel-3.0.0...
Installing ri documentation for taps-0.2.19...
Installing RDoc documentation for sinatra-0.9.2...
Installing RDoc documentation for activesupport-2.2.2...
Installing RDoc documentation for activerecord-2.2.2...
Installing RDoc documentation for thor-0.9.9...
Installing RDoc documentation for sequel-3.0.0...
Installing RDoc documentation for taps-0.2.19...

C:\tutorials\four_table> heroku db:push
Auto-detected local database: sqlite://db/development.sqlite3
Sending schema
Sending data
6 tables, 23 records
schema_migrat: 100% |=====| Time: 00:00:00
users: 100% |=====| Time: 00:00:00
countries: 100% |=====| Time: 00:00:00
category_assi: 100% |=====| Time: 00:00:00
categories: 100% |=====| Time: 00:00:00
recipes: 100% |=====| Time: 00:00:00
Sending indexes
Resetting sequences

C:\tutorials\four_table>
```

Figure 305: Using "heroku db:push" to push data to your app on Heroku.com

The log indicates that six tables with a total of 23 records were sent. Let's look at the live app:

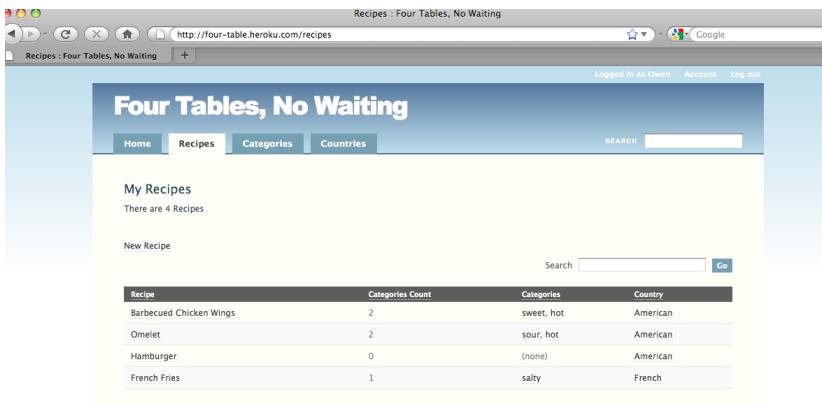


Figure 306: The "Four Table" app on Heroku.com with data

Let's add a recipe for "Crab Cakes":

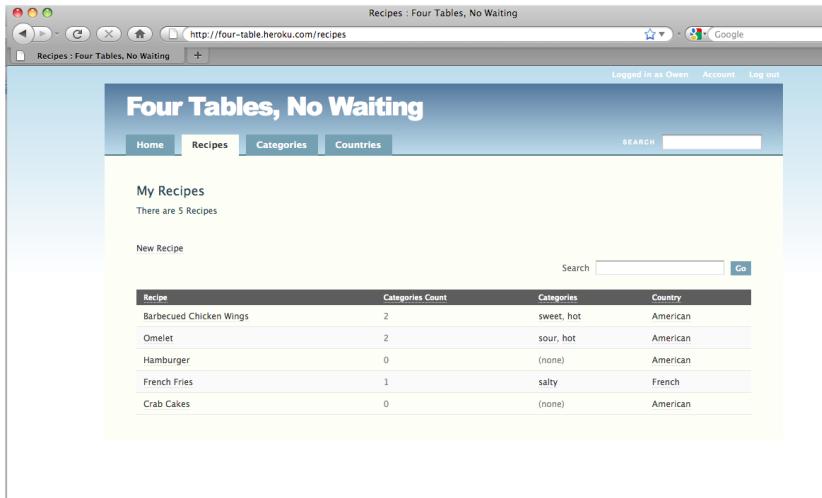
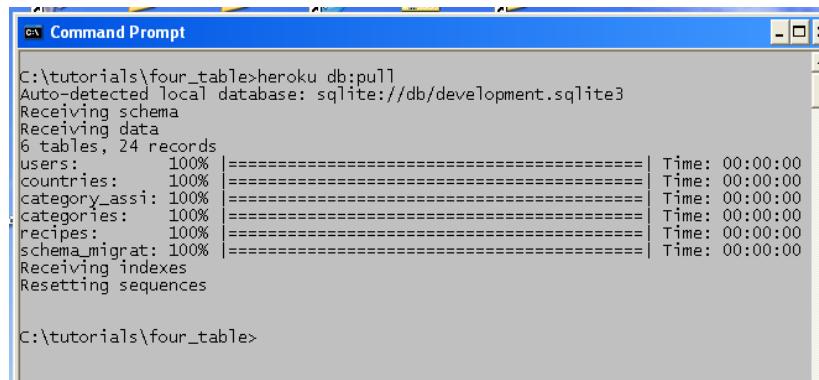


Figure 307: Add a recipe on Heroku.com

Step 9: Pull changed data from Heroku

I can use the "pull" option to backup my change on Heroku to my local database:

```
\four_table> heroku db:pull
```



```
C:\tutorials\four_table>heroku db:pull
Auto-detected local database: sqlite:///db/development.sqlite3
Receiving schema
Receiving data
6 tables, 24 records
users:    100% =====| Time: 00:00:00
countries: 100% =====| Time: 00:00:00
category_assi: 100% =====| Time: 00:00:00
categories: 100% =====| Time: 00:00:00
recipes:   100% =====| Time: 00:00:00
schema_migrat: 100% =====| Time: 00:00:00
Receiving indexes
Resetting sequences

C:\tutorials\four_table>
```

Figure 308: Pull changed data from Heroku.com to your local app

Pretty slick! I now have 24 records on the local version—including my precious recipe for crab cakes.

Part III

HOBO UNDER THE HOOD

Chapter 7

HOBO GENERATORS

Changes from Hobo 1.0 to 1.3

Commands

hobo g migration	hobo g migration
ruby script/generate hobo_model_resource	hobo g resource
ruby script/generate hobo_model	hobo g model

Generator names

hobo_migration	migration
hobo_model_resource	resource
hobo_model	model

The Hobo generators have been rewritten from scratch to match the new Rails 3 generator standards.

The Hobo generators have also been reorganized to fit to the new `setup_wizard`, and to be more DRY. Indeed when needed they often invoke each other from the inside.

Hobo Generators

The following command in any hobo application directory will show all the generators available:

```
> hobo g --help  
(or)  
> rails g --help
```

Under the 'Hobo' group we find the hobo generators:

```
Hobo:  
hobo:admin_subsite  
hobo:assets  
hobo:controller  
hobo:front_controller  
hobo:i18n  
hobo:migration  
hobo:model  
hobo:rapid  
hobo:resource  
hobo:routes  
hobo:setup_wizard  
hobo:subsite  
hobo:subsite_taglib  
hobo:test_framework  
hobo:user_controller  
hobo:user_mailer  
hobo:user_model  
hobo:user_resource
```

You can have more help about any generator (e.g., the "hobo:resource" generator) by doing:

```
> hobo g resource --help  
(which is the same as)  
> rails g hobo:resource --help
```

Note: If you use the hobo command, you can omit the 'hobo:' namespace prepended to hobo generators.

The following options are common options for all the generators:

```
Runtime options:  
-q, [--quiet]      # Suppress status output  
-f, [--force]     # Overwrite files that already exist  
-s, [--skip]       # Skip files that already exist  
-p, [--pretend]    # Run but do not make any changes
```

Note: The -p or -pretend option that can be passed to all generators. It will run the generator without actually producing any change, so it is very useful to see what a generator would do without eventually messing up your app.

hobo:setup_wizard

The above command is the generator that the hobo command invokes after creating the Rails infrastructure. It calls others generators in the background, in order to customize the new application.

You can eventually invoke it manually (when you choose the -skip-setup options of the Hobo command) or re-invoke it in order to overwrite any file of the original installation. Its common use is to be called internally by the Hobo command.

In its default mode (interactive/wizard mode) the developer will be asked for any feature he possibly wants in his new application.

Here are the static options as they come from the -help option:

```
Options:  
[--migration-migrate]  
# Generate migration and migrate  
# Default: true  
[--fixture-replacement=FIXTURE_REPLACEMENT]  
# Use a specific fixture replacement  
[--fixtures]  
# Add the fixture option to the test  
# framework - default: true  
[--wizard]  
# Ask instead using options  
# Default: true  
[--update]  
# Run bundle update to install the missing gems  
-i, [--invite-only]  
# Add features for an invite only website  
[--git-repo]  
# Create the git repository with the  
[--front-controller-name=FRONT_CONTROLLER_NAME]  
# Front Controller Name: Default: front  
[--gitignore-auto-generated-files]  
# Add the auto-generated files to .gitignore  
# Default: true  
[--default-locale=DEFAULT_LOCALE]  
# Sets the default locale  
[--admin-subsite-name=ADMIN_SUBSITE_NAME]  
# Admin Subsite Name  
[--user-resource-name=USER_RESOURCE_NAME]  
# Default: admin  
# User Resource Name  
--t, [--test-framework=TEST_FRAMEWORK]  
# Use a specific test framework  
# Default: test_unit  
--locales=one two three]  
# Choose the locales  
# Default: en  
[--private-site]  
# Make the site unaccessible to non-members  
[--activation-email]  
# Send an email to activate the account
```

```
[--migration-generate]  
[--main-title]  
# Generate migration only  
# Shows the main title  
# Default: true
```

Here they are in the order they get invoked by the `setup_wizard`:

`-main-title`

This shows the main title

hobo:assets

No option / prompt: this generator copies a few asset files.

hobo:test_framework generator -test-framework=TEST_FRAMEWORK interactively set by: "Do you want to customize the test_framework?" and "Choose your preferred test framework: <available frameworks>"

This gives you the opportunity to change the test framework, so the next generators invoked will use it for generating tests.

`-fixtures`

Interactively set by "Do you want the test framework to generate the fixtures?"

This allows you to choose fixtures generation.

`-fixture-replacement=FIXTURE_REPLACEMENT`

Interactively set by: "Type your preferred fixture replacement or <enter> for no replacement:"

You can use the fixture replacement that you prefer.

Site options:

`-invite-only`

Interactively set by: "Do you want to add the features for an invite only website?"

It will pass this option to the next invoked generators in order to create an admin site and the resources used to invite a new user.

`-private-site`

Interactively set by: "Do you want to prevent all access to the site to non-members? (Choose 'y' only if ALL your site will be private, choose 'n' if at least one controller will be public)"

If 'n' the wizard will print the instruction to do the same with single controllers

hobo:rapid generator

No option / prompt: this generator copies files

hobo:user_resource generator

-user-resource-name

Interactively set by: "Choose a name for the user resource [<enter>=user|<custom_name>]:"

You can choose a different name for the user model

-activation-email

Interactively set by: "Do you want to send an activation email to activate the user?"

The user will receive an activation email containing an activation link.

hobo:front_controller generator

-front-controller-name

Internally set by: "Choose a name for the front controller [<enter>=front|<custom_name>]:"

You can change the front controller name

hobo:admin_subsite generator

-admin-subsite-name

Interactively set by: "Choose a name for the admin subsite [<enter>=admin|<custom_name>]:"

You can change the admin site name and it will be passed to the next invoked generators.

hobo:migration generator

The following option are interactively set by: "Initial Migration: [s]kip, [g]enerate migration file only, generate and [m]igrate [<s|g|m>]:"

-migration-migrate

Generate and migrate.

-migration-generate

Generate only

hobo:i18n generator

-locales

Interactively set by: "Type the locales (space separated) you want to add to your application or <enter> for 'en':"

Copies the locale files for the supported locales

-default-locale

Interactively set by: "Do you want to set a default locale? Type the locale or <enter> to skip:"

set the config.i18n.default_locale in the config/application.rb file

Git repository options

-git-repo

Interactively set by: "Do you want to initialize a git repository now?"

This will initialize a git repository committing the new generated app

-gitignore-auto-generated-files

Interactively set by: "Do you want git to ignore the auto-generated files?\n(Choose 'n' only if you are planning to deploy on a read-only File System like Heroku)"

This will add the auto-generated files to .gitignore. In read-only file systems like Heroku, this feature would be counter-productive, so in that case is better committing also the auto generated files.

Note: You use any of the generators used by the Setup Wizard in order to override/re-store any modified/generated file

hobo:admin_subsite

```
> hobo g admin_subsite [NAME=admin] [options]
```

Options:

-i, [--invite-only] [--make-front-site]	# Add features for an invite only website # Rename application.dryml to front_site.dryml
-t, [--test-framework=NAME]	# Test framework to be invoked
[--user-resource-name=USER_RESOURCE_NAME]	# User Resource Name # Default: user

This generator is used internally by the `setup_wizard`, when the `-invite-only` option is true.

hobo:assets

```
> hobo g assets
```

Used by the `setup_wizard`, this generator copies asset files

hobo:controller

```
> hobo g controller NAME
```

Options:
-t, [--test-framework=NAME] # Test framework to be invoked
Default: test_unit

hobo:front_controller

```
> hobo g front_controller [NAME=front] [options]
```

Options:
-i, [--invite-only]
Add features for an invite only website
[--add-routes]
Modify config/routes.rb to support the front controller

Default: true
-t, [--test-framework=NAME]
Test framework to be invoked

Default: test_unit
-d, [--delete-index]
Delete public/index.html

Default: true
[--user-resource-name=USER_RESOURCE_NAME]
User Resource Name
Default: user

Used by the `hobo:setup_wizard` generator

hobo:i18n

```
> hobo g i18n [en it ...]
```

Used by the `hobo:setup_wizard` generator, it creates the locale files for i18n. You can use it to add any supported language at any point in the development lifecycle.

hobo:migration

```
$ hobo g migration [NAME] [options]
```

Options:

```
-g, [--generate]      # Don't prompt for action - generate the migration
-m, [--migrate]       # Don't prompt for action - generate and migrate
-n, [--default-name]  # Don't prompt for a migration name - just pick one
-d, [--drop]          # Don't prompt with 'drop or rename' - just drop everything
```

Used by the `setup_wizard` generator and by the user when needed. It is actually part of `hobo_fields`, so it can be used outside of hobo.

hobo:model

```
$ hobo g model NAME [field:type field:type] [options]
```

Options:

```
[--timestamps] # Indicates when to generate timestamps
```

Creates a new hobo model and all the related files. It actually invokes the ActiveRecord generator and injects the code needed to hobo.

hobo:rapid

```
> hobo g rapid
```

Used by the `setup_wizard`, this generator just copy a few assets in order to make Rapid work

hobo:resource

```
> hobo g resource NAME [field:type field:type] [options]
```

Options:

```
[--timestamps] # Indicates when to generate timestamps
```

Use this generator to create resource (model + controller + files) of called NAME, adding fields as needed. The value for NAME must be singular.

hobo:routes

```
> hobo g routes
```

This generator prepares the automatic routes for your application. It is automatically used internally, so you should not use it manually. You can customize the path where it will create the routes in config/application.rb setting the config.hobo.routes_path

hobo:subsite

```
> hobo g subsite [options]
```

Options:

```
-i, [--invite-only]
# Add features for an invite only website
[--user-resource-name=USER_RESOURCE_NAME]
# User Resource Name

# Default: user
-t, [--test-framework=NAME]
# Test framework to be invoked

# Default: test_unit
[--make-front-site]
# Rename application.dryml to front_site.dryml
```

You can use this generator to create a new custom subsite.

hobo:subsite_taglib

```
> hobo g subsite_taglib NAME [options]
```

Options:

```
-i, [--invite-only] # Add features for an invite only
[--user-resource-name=USER_RESOURCE_NAME] # User Resource Name
# Default: user
```

Used internally when a subsite is generated in order to generate the subsite tag library.

hobo:test_framework

```
$ hobo g test_framework NAME [options]
```

Options:

```
--fixture-replacement=FIXTURE_REPLACEMENT]
# Use a specific fixture replacement
[--update]
# Run bundler update to install missing gems
[--fixtures]
# Add the fixture option to the test framework

# Default: true
-t, [--test-framework=TEST_FRAMEWORK]
# Use a specific test framework

# Default: test_unit
```

Used internally by the `setup_wizard` (see the "invoke `hobo:test_framework` generator" step above)

The following generators are for the user resource, and are:

hobo:user_controller

```
> hobo g user_controller [NAME=users] [options]
```

Options:

```
-i, [--invite-only] # Add features for an invite only website
-t, [--test-framework=NAME] # Test framework to be invoked
# Default: test_unit
```

Used internally by the `setup_wizard` for the user resource.

hobo:user_mailer

```
$ hobo g user_mailer [NAME=user] [options]
```

Options:
-i, [--invite-only] # Add features for an invite only website
-t, [--test-framework=NAME] # Test framework to be invoked
Default: test_unit
[--activation-email] # Send an email to activate the account

Used internally by the setup_wizard for the user resource

hobo:user_model

```
$ hobo g user_model [NAME=user] [options]
```

Options:
-i, [--invite-only] # Add features for an invite only website
[--activation-email] # Send an email to activate the account
[--timestamps] # Indicates when to generate timestamps
[--admin-subsite-name=ADMIN_SUBSITE_NAME] # Admin Subsite Name
Default: admin

Used internally by the setup_wizard for the user resource.

hobo:user_resource

```
$ hobo g user_resource [NAME=user] [options]
```

Options:
-i, [--invite-only] # Add features for an invite only website
[--activation-email] # Send an email to activate the account

Used internally by the setup_wizard for the user resource

Automatically Generated DRYML Tags

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

```
<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">
    ...
</def>
```

There is a lot going on in the tag definitions in `pages.dryml` that you might not fully understand. 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>, <field-list></code> <code><collection> (for associated models)</code>	edit
<code><edit-page></code>	edit	<code><form></code>	update

Table 5: 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 that link to other actions.

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.

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`.

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 is the difference between tag definitions and tag usage.

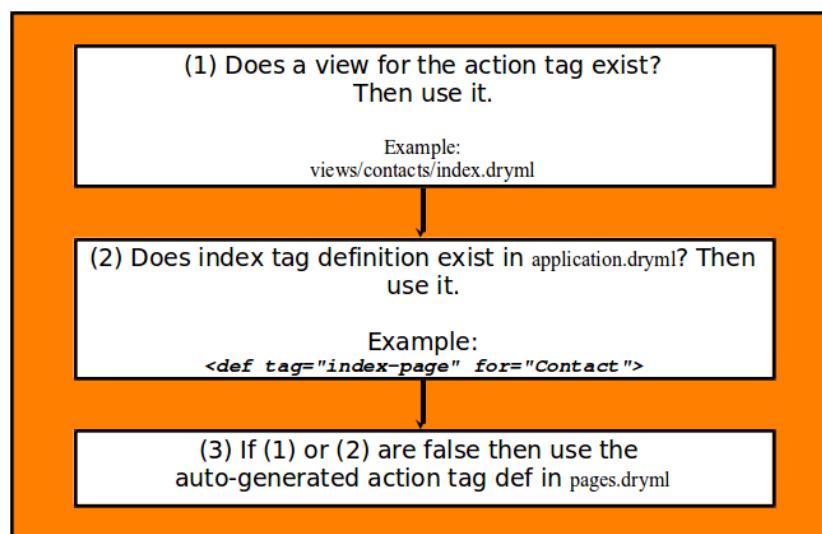


Figure 309: Hobo precedence logic for action tags

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 8

THE HOBO PERMISSION SYSTEM

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 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 be aware that edit permission is a bit of an anomaly.

Defining permissions

In a typical Hobo app, the permission system is most prominent in your own code in your permission declarations. These are methods, which you define on your models, known as “permission methods”. These methods let 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 the following four methods:

- 1 def create_permitted?
- 2 def update_permitted?
- 3 def destroy_permitted?
- 4 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 can access.

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 a 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.

Destroy permissions

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. 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. 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. Since 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. 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 notice 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 used 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. The permission system is able to cover 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.

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.createable_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 `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. It then restores it to its 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

We would often like to initialize some aspect of our model based on who is the `acting_user`. 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. If an attempt is made to change the group to a different one, that would fail.

The edit permission mechanism (described in a section) can detect this, so the end result is that (by default) your app will have the “New Event” form. However, 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:

- 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. On the hand UI may provide radio buttons to chose “Male” or “Female”. 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. 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 slightly a strong assumption. It fits for many applications, but might be totally inappropriate for many others.

Although you've probably seen this method, 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 “administrator” field, and of all calls to `administrator?` from your models’ permission methods. Those are 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 9

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. The down-side of having almost no code in the controllers is that there's nothing to tweak. Don't worry! Hobo's controllers have been built with customization in mind. The things you will commonly tweak are extremely easy and full customization is not hard.

Introduction

Here's a typical controller in a Hobo app. (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 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 seven 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, all the default actions 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.

The default routes created by Hobo are located in the `config/hobo_routes.rb` file. You can override these by placing entries in the `config/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 provides aren't what you want. They may 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. controllers should be thought of 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 differently 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`. 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 works 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 try:

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

The above example 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 will not respond so that 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 as a range of options. 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 will not 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)
```

... This 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 similar to the regular `hobo_index`, `hobo_new` and `hobo_create` except that they take an additional first argument – the name of the association. For example, the default implementation of, `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 needed.

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. 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 as a single declaration:

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

Since 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 autocomplete 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 autocomplete.
- :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. The implementation given to you is a simple call to the underlying method that does the actual work. You can call this underlying method directly. To illustrate on a UsersController 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.

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
```

This 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. This method 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 section

Chapter 10

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 appear 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. In a way they are, but with Hobo style. The big win comes from the fact that:

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. The bolded words capture the way we think about the friendship process much better. Of course, we *could* implement friendship in a RESTful style, but we’d be *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 being 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 310: Defining the Friendship model

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

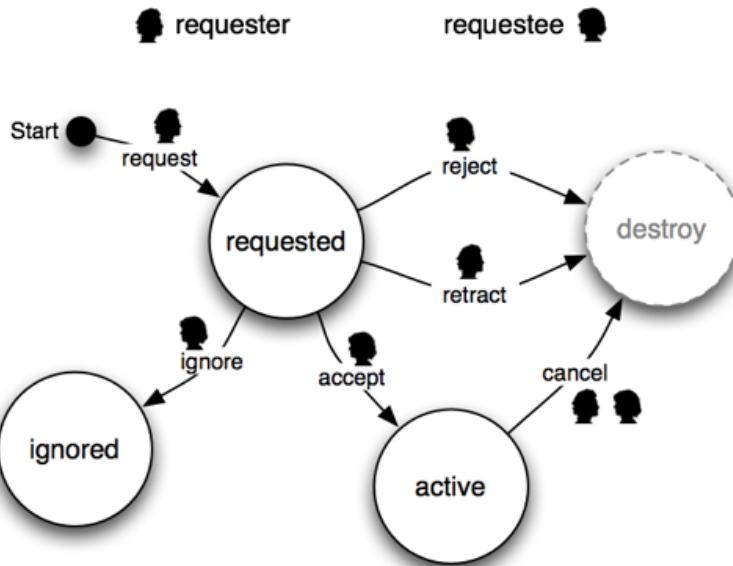


Figure 311: Lifecycle diagram

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"
```

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
- 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.

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. 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. They define actions that can start the lifecycle, setting the state to be some start-state.
- Zero or more *transitions*. Each has a name. 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. 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 the one below:

```
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: 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. A creator provides 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: The precondition is evaluated before any changes are made to the record using the parameters to the lifecycle step.

- `:new_key` – generates 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 in

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 to 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 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, 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

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`

VALIDATIONS

Some examples:

A model has an owner:

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

You can only 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:

```
: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. 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 for these that Hobo generates controller actions. 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.

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. Thus, 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 meta-data 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. Also, 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. 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 someone 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. 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 even more general. 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, 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::SHA1hexdigest{"#{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, 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_rest,  
           { :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. 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 an available URL helper. 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 11

HOBO VIEW HINTS AND LOCALES

Note: While Hobo View Hints still operate in version 1.3, we recommend using the new Internationalization (I18n) Locales functionality which will be discussed later in this chapter.

Introduction

One of the main attractions of Hobo is its ability to give you a decent starting point for your app's UI. This is 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.

This functionality used to be handled in the ViewHints module in Hobo 1. As of Hobo 1.3, these features have been migrated into the Internationalization module. For those of you who are upgrading your Hobo 1 applications, a section on migrating ViewHints to Internationalization is available at the end of this chapter.

Internationalization (I18n)

The old view hints renaming has been moved into the locale files. Locales files have been introduced to support Rails Internationalization (i18n) and have their own conventions. They are perfect and already working English to English customizations as well. Hobo now uses this Rails-standard tool and established conventions to provide

model, field name, and field help customization. For more detail see the Rails Internationalization (I18n) API Guide at <http://guides.rubyonrails.org/i18n.html>

Here is the uncommented default config/locale/app.en.yml file

```
en:
  hello: "Hello world"

  attributes:
    created_at: Created at
    updated_at: Updated at

  activerecord:
    models:
      user:
        one: User
        other: Users
    attributes:
      user:
        created_at: Created at
        name: Name
        password: Password
        password_confirmation: Password Confirmation
        email_address: Email Address
    attribute_help:
      user:
        email_address: We will never share your address with third parties
```

Figure 312: config/locale/app.en.yml

Note: YAML (.yml) files use a fixed number of spaces (2) to indent levels. Tabs are illegal. Quotation marks are optional.

You can rename models and attributes by *changing* or *adding* key/values in the specific activerecord.models and activerecord.attributes sections. You can also add any attributes help you might need in the activerecord.attribute_help section (which is a Hobo specific section). For example:

```

en:
  hello: "Hello world"

  attributes:
    created_at: Created at
    updated_at: Updated at

  activerecord:
    models:
      user:
        one: User
        other: Users
      contact:
        one: Friend
        other: Friends
    attributes:
      user:
        created_at: Created at
        name: Name
        password: Password
        password_confirmation: Password Confirmation
        email_address: eAddress
      contact:
        name: Nickname
        email_address: eAddress
    attribute_help:
      user:
        email_address: We will never share your address with third parties
      contact:
        name: This is the nickname

```

Figure 313: Name and input help

Just be careful: Don't mess up the indentation!

Note: The key/values inside the first attributes: group (marked in blue) are used as the default for all of the models

The Hobo-specific translation definitions can be seen in the `hobo.en.yml` file:

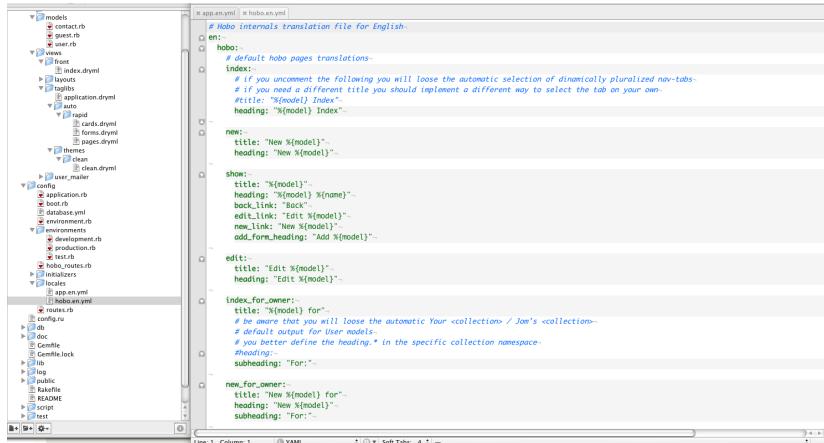


Figure 314: hobo.en.yml

Note: Members of the Hobo community have created versions for Spanish, Italian, German, Russian, etc. Check out the latest by joining the [Hobo Google Users Group](#).

Child relationships

The `ViewHints.children` and the `ViewHints.inline_booleans` methods have been moved in the model from the View Hints, but they are used in the exactly same way they were used before.

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 User < ActiveRecord::Base
  fields do
    ...
  end
  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 User < ActiveRecord::Base
  fields do
    ...
  end
  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 User < ActiveRecord::Base
  fields do
    ...
  end
  inline_booleans :administrator, :moderator
end
```

As a shortcut...

```
class User < ActiveRecord::Base
  fields do
    ...
  end
  inline_booleans true
end
```

...will apply the option to all boolean fields in the model.

Note: You can always reach the `view_hints` class by accessing the `Model.view_hints` method, so for example `User.view_hints.children` will return the children of the `User` model. Same thing with `parent`, `parent_defined`, `sortable?` and `paginate?` which are the survived methods in the view hint class.

Locale (I18n) Friendly Tags

In Hobo 1.3.x all the code not compatible by the Ruby i18n standard has been removed. As a result a few Hobo tags have changed internally and accept more attributes or deprecate others. A few Hobo tags have been deprecated and replaced with relevant i18n-specific tags that are included in the auto-generated taglibs.

Note: Many internal Hobo tags have been transparently internationalized, i.e., the strings they use are automatically translated to the current locale language. Here, we are only documenting the tags that you might use in application development.

Changed tags

<view/>

The "view" tags for these types are localized (i.e. their values get formatted consistently with the locale)

- Date
- Time
- ActiveSupport::TimeWithZone
- Numeric
- boolean

<count/>

A convenience tag is used to output a count summary with a correctly localized and pluralized label. It 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>
```

You can pass a summary attribute which will generate a complete localized sentence. It allows two options:

- Boolean (e.g., `<count summary/>`): it will lookup the `tags.count.default` key in the locale file. If the lookup fails, it will fallback to the english default sentences consistent with the count.
- String (e.g., `<count summary="offer"/>`): it will lookup the `tags.count.offer` key in the locale file.

If the lookup fails, it will fall back to the English default sentences consistent with the count.

Examples

```
it:
tags:
count:
default:
    zero: "Non ci sono {{label}}"
    one: "C'è solo 1 {{label}}"
    other: "Ci sono {{count}} {{label}}"
choice:
    zero: "Non ci sono {{label}} da scegliere"
    one: "Puoi scegliere solo una {{label}}"
    other: "Puoi scegliere tra {{count}} {{label}}"
```

with :en locale and boolean summary (internal defaults)

```
<count:comments summary/>
-> <span class="count">There is 1 Comment</span>
<count:viewings summary/>
-> <span class="count">There are 3 Viewings</span>
```

Note: Add the locale English strings to use the following examples)

With :it locale and boolean summary (key "tags.count.default"):

```
<count:comments summary/>
0 -> <span class="count">Non ci sono Commenti</span>
1 -> <span class="count">C'è solo 1 Commento</span>
5 -> <span class="count">Ci sono 5 Commenti</span>
```

With :it locale and summary="choice" (key "tags.count.choice"):

```
<count:comments summary="choice"/>
0 -> <span class="count">
    Non ci sono Commenti da scegliere
</span>
1 -> <span class="count">
    Puoi scegliere solo 1 Commento
</span>
5 -> <span class="count">
    Puoi scegliere tra 5 Commenti
</span>
```

Additional Notes

- If the prefix attribute is deprecated: use summary instead.
- 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>
```

<you/>

Convenience tag to help with the common situation where you need to address the current user as "you" and refer to other users by name

Usage

The context should be a user object. If `this == current_user` the "you" form is rendered, otherwise use the form with the user's name:

```
<you have/> new mail
-> "you have new mail" or "Jim has new mail"
<you are/> now an admin
-> "you are now an admin" or "Jim is now an admin"
<you do/>n't want to go there
-> "you don't want to go there"
or "Jim doesn't want to go there"
```

The tag is also localized in the namespaces "tags.you.current_user" and "tags.you.other_user".

Each namespace can contain the legacy keys "have," "are," "do" used for the respective attributes. "Nothing" is used when no attribute is passed. However, you can also use your own keys, providing that you add the keys in the correct namespaces.

Examples

```
it:
tags:
you:
    current_user:
        nothing: "Tu"
        have: "Hai"
        are: "sei"
        can: "Puoi"
    other_user:
        nothing: "{{name}}"
        have: "{{name}} ha"
        are: "{{name}} è"
        can: "{{name}} può"
```

```
<you have/> un nuovo messaggio.
-> "Hai un nuovo messaggio." or "Jim ha un nuovo messaggio."
Adesso <you are/> amministratore.
-> "Adesso sei amministratore."
or "Adesso Jim è amministratore."
<you can/> scrivere.
-> "Puoi scrivere." or "Jim può scrivere."
```

Note: The symbol `:name` is added by default as an interpreted variable)

Attributes

- **capitalize:** the first letter of the resulting sentence will be capitalized

Additional Notes

The "titleize" attribute is deprecated—use "capitalize" instead.

<your/>

Similar to <you>, but renders "Your" or "Fred's" or equivalent localized strings

Attributes

- **capitalize:** the first letter of the resulting sentence will be capitalized
 - **count:** used in pluralization. If omitted it will be set to 1.
 - **key:** used to lookup the translation in the locale file. It allows 3 different options:
 - A single key like 'message': simple translation in 'tags.your.message.current_user' or 'tags.your.message.other_user'
 - A Composite key like 'any.namespace.message': translation as for the previous case, but it will translate also the 'any.namespace.message' and will interpolate the variable <key> (in this case :message) in the translation
 - When the key is omitted it will be set to "default" and will do the translation with that key.
- NEED CLARIFICATION:
Pass other meaningful attributes to achieve a dynamic usage
- any other attribute passed to the tag will be used as a variable for interpolation

Notes

- The :name variable is added by default as an interpolable variable.
- If no translation is found an automatic (only english) default is generated:
 - the Your/Jim's string, joined to the tag content.
 - If you pass an explicit 'default' attribute, you will override the automatic default.

Examples

```

it:
tags:
your:
message:
current_user:
one: "Tuo Messaggio"
other: "Tuoi Messaggi"
other_user:
one: "Messaggio di {{name}}"
other: "Messaggi di {{name}}"
entry:
current_user:
one: "Tua {{entry}}"
other: "Tue {{entry}}"
other_user: "{{entry}} di {{name}}"

```

```

<your key="message" count=>"&messages.count"/>
1 -> "Tuo Messaggio" or "Messaggio di Jim"
5 -> "Tuoi Messaggi" or "Messaggi di Jim"
<your key="activerecord.models.entry"
count=>"&this.entries.count"/>
1 -> "Tua Entrata" or "Entrata di Jim"
5 -> "Tue Entrate" or "Entrate di Jim"
<your>Posts</your>
-> "your Posts" or "Jim's Posts"

```

<select-menu/>

A simple wrapper around the <select> tag and options_for_select helper

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.
- key: the key used to lookup in the locale file or 'default' by default. If you pass it hobo will lookup in the namespace "tags.select_menu.#{key}" in order to find

options, first_option and first_value. The passed attributes are used as a default if the lookups fail. (See the documentation of filter-menu tag for a similar example),

<filter-menu/>

A <select> menu intended to act as a filter for index pages.

Example

Filtering on state is a common use. Here's the dryml for Order:

```
<filter-menu param-name="state"
              options="&Order::Lifecycle.states.keys" />
```

And the controller action:

```
def index
  # always validate data given in URL's!!!
  params[:state]=nil unless
    Order::Lifecycle.states.include?(params[:state]._.?.to_sym)
  finder = params[:state] ? Order.send(params[:state]) : Order
  hobo_index finder
end
```

See [Filtering stories by status](#) in the [Agility Tutorial](#).

Attributes

- param-name: the name of the HTTP parameter to use for the filter
- options: an array of options or an array of arrays (useful for localized apps) for the menu. It can be omitted if you provide the options as an array or array of arrays in the locale file.
- no-filter: The text of the first option which indicates no filter is in effect. Defaults to 'All'

<I18n/>

It lookups the options attributes in filter_menu.`#{param_name}.options`.

The passed options are used as a default in case the lookup fails.

Besides the `tags.filter_menu.default.no_filter` key is used as default of the attribute no-filter

(or "All" if no default is found)

Example

```
es:
tags:
filter_menu:
period:
no_filter: Todos Períodos
options:
- [ "Hoy", "today" ]
- [ "Ayer", "yesterday" ]
```

Code:

```
TIME_PERIODS = %w[today yesterday]
<t-filter-menu param-name="period"
options=&TIME_PERIODS"
no-filter="All Periods"/>
```

with I18n.locale == :es

```
<select name="period">
<option value="">Todos Periodos</option>
<option value="today">Hoy</option>
<option value="yesterday">Ayer</option>
</select>
```

with I18n.locale == :en (i.e no locale file)

```
<select name="period">
<option value="">All Periods</option>
<option value="today">today</option>
<option value="yesterday">yesterday</option>
</select>
```

i18n tags

These are the tag from the i18n_rapid taglib.

<t/>

Simple wrapper around I18n.t.

The tag content is used as the :default option. It is overridden by an explicit 'default' attribute.

There is a default :count => 1

Attributes

- key: the key to lookup
- all the attributes accepted by the wrapped method

<ht/>

This is a helper and a tag. Uses RoR native I18n.translate.

Adds some conventions for easier hobo translation.

- Assumes the first part of the key is a model name (e.g.: users.index.title -> user)
- Tries to translate the model by lookup for: (e.g.: user-> activerecord.models.user)
- Adds a default fallback to the beginning of the fallback chain by replacing the first part of the key with "hobo" and uses the translated model name as additional attribute. This allows us to have default translations (e.g.: hobo.index.title: "{{model}} Index")

It is also used as a tag in the dryml-view files. The syntax is:

```
<ht key="my.app">My Application</ht>
```

Will lookup the my.app key for your locale and replaces the "My Application" content if found. :

```
<ht key="my" app="Program">My Application</ht>
```

Will look up both the my and app key for your locale, and replaces the "My Application" with the my key contents (interpolated using the app key).

sample.no.yml file:

```
"no":  
  my: "Mitt {{app}}"
```

The output should be: **Mitt Program**

<model-name-human/>

Wrapper around ActiveModel::Name#human

Attributes

- model: (optional) should be a model class or a record object (default to this)
- count: (optional) used to pick the inflected string for the model. It should be an integer.

<human-attribute-name/>

Wrapper around ActiveRecord::Base.human_attribute_name.

Attributes

- attribute: the attribute to lookup
- model: (optional) should be a model class or a record object (default to this)
- count: (optional) should be an integer

<human-collection-name/>

Used to localize and pluralize collection names.

A collection name is a special case of an attribute name. You should store the collection names as attribute names in the locale file. This tag internally uses human_attribute_name to return them.

With the your attribute and in the special case the context is a Hobo::Model::User instance it automatically embeds the your tag functionality. (note: :name is added by default as an interpolable variable)

Attributes

- collection: the attribute/collection key to lookup in the activerecord.attributes namespace. (e.g. 'roles')
- count: used to pick the inflected string for the collection. It should be an integer.
- your: wraps the collection name in a Your tag

Example

```

it:
  activerecord:
    attributes:
      post:
        comments:
          one: "Commento"
          other: "Commenti"
      user:
        roles:
          one: "Ruolo"
          other: "Ruoli"
      tags:
        your:
          roles:
            current_user:
              one: "Il tuo Ruolo"
              other: "I tuoi Ruoli"
            other_user:
              one: "Ruolo di {{name}}"
              other: Ruoli di {{name}}"

```

Context is a Post instance ('your' is ignored)

```

<human-collection-name collection="comments"
  count="&user.comments.count" your/>
I18n.locale = :en => "Comment" or "Comments"
I18n.locale = :it => "Commento" or "Commenti"

```

Context is an User instance

```

<human-collection-name collection="roles"
  count="&user.roles.count" your/>
I18n.locale = :en => "Your Role" or "Jim's Role"
  or "Your Roles" or "Jim's Roles"
I18n.locale = :it => "Il tuo Ruolo"
  or "Il Ruolo di Jim" or "I tuoi Ruoli" or "I Ruoli di Jim"

```

(output is the same as <Your key="roles" count=>"&user.roles.count"/>)

```

<human-collection-name collection="roles"
  count="&user.roles.count"/>
I18n.locale = :en => "Role" or "Roles"
I18n.locale = :it => "Ruolo" or "Ruoli"

```

Deprecated/Renamed tags

These tags have been deprecated because they were not compliant with i18n or they have been renamed in a more appropriate way.

<collection-name/>

deprecated: use <human-collection-name/>

<preview-with-more/>

renamed: use <collection-preview/> instead

Migrating ViewHints to the Internationalization Module

In the ViewHints module, there are 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.
- Inline Booleans: allows you to force Hobo to show boolean fields in the field list on model show pages.

Renaming the model name, field names, and specifying field help can be specified in the localization file. In this section's examples, the localization file we will be using is the config/locales/app.en.yml

Here is an example of migrating the model name rename, field rename and field help from your ViewHints to the config/locales/app.en.yml file:

Existing ViewHints file:

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

Corresponding config/locales

```
en:
  activerecord:
    models:
      answer:
        one: Response
        other: Responses
      attributes:
        answer:
          body: ""
          recipe: See recipe
        attribute_help:
          answer:
            recipe: Enter keywords from the name of a recipe
```

Since different languages handle pluralization in different ways, simply specifying another name for a model is no longer feasible. Instead, we have to specify at least the models's new name for it in a single instance and for plural instances. Above we have renamed 'Answer' to 'Response' when there is only 1 model present and 'Responses' in all other cases.

To specify Child relationships, this declaration should be moved from the "ViewHints" file to the "Model". :

```
class UserHints < Hobo::ViewHints
  children :recipes, :questions, :answers
end
```

would be changed to:

```
class User < ActiveRecord::Base
  fields do
    ...
  end
  children :recipes, :questions, :answers
end
```

If there are any inline_booleans specified in your ViewHints file, these can be specified in the Model declaration. So:

```
class UserHints < Hobo::ViewHints
  inline_booleans :administrators
end
```

would be changed to:

```
class User < ActiveRecord::Base
  fields do
    ...
  end
  inline_booleans :administrators
end
```

Tim Griffin ran into an issue with localization that is worth noting here:

- If you don't explicitly name the `enum_string` with the `:name =>` parameter or with a call to `set_name`, the main key in your locale file must include the model name in which the `enum_string` is defined, as well as the tableized name of the `enum_string` (lower-case/ pluralized).

BUT the `enum_string`'s main key can *ONLY* use the "/" notation.

So, if you do this:

```
class Document < ActiveRecord::Base
  hobo_model
  Region = HoboFields::Types::EnumString.for(:ab, :bc)
```

you must use this notation:

```
en:
  document/regions:
    ab: "Alberta"
    bc: "British Columbia"
```

and *NOT* this notation:

```
en:
  document:
    regions:
      ab: "Alberta"
      bc: "British Columbia"
```

- If you DO explicitly name the `enum_string` with the "`:name =>`" parameter or with a call to `set_name`, the keys in your locale file must *not* include the model name.

If you do this:

```
class Document < ActiveRecord::Base
  hobo_model
  Region = HoboFields::Types::EnumString.for(:ab, :bc,
    :name => "Region")
```

Your locale file's `enum_string` key must match the tableized version of the name you assigned to the `enum_string`. (Note the lower-case first letter and the pluralization.)

```
en:  
regions:  
    ab: "Alberta"  
    bc: "British Columbia"
```

Note: Tim Griffin submitted the following example of Hobo's new support for specifying a model name to enable different models with the same filter attribute to be properly represented in a locale file under the `activerecord.attributes` namespace.

The 'model' parameter on the `<filter-menu>` tag now enables you to include translated filter values for each model that requires a filter menu in your application.

As an example, consider a model representing packages that can be in different states. On the package index page using a `<table-plus>` tag to show packages, your `<filter-menu>` tag can show its options using translation keys from a locale file by including the 'model' parameter in the tag call:

```
<filter-menu model="package" param-name="state"/>
```

Your locale file will then need to include the keys for the filter menu as an entry under the `activerecord` namespace. The following example shows the options keys listed under the `filter_menu` key:

```
activerecord:  
  attributes:  
    package:  
      name: "Package Name"  
      id: "ID"  
      created_at: "Created"  
      updated_at: "Updated"  
      state: "Status"  
    lifecycle:  
      states:  
        draft: "Draft"  
        submitted: "Submitted"  
        received: "Received"  
        in_review: "In Review"  
        accepted: "Accepted"  
        rejected: "Rejected"  
        withdrawn: "Withdrawn"  
    filter_menu:  
      state:  
        no_filter: All  
        options:  
          - [ "Draft", "draft" ]  
          - [ "Submitted", "submitted" ]  
          - [ "In Review", "in_review" ]  
          - [ "Accepted", "accepted" ]  
          - [ "Rejected", "rejected" ]  
          - [ "Withdrawn", "withdrawn" ]  
          - [ "Archived", "archived" ]
```

Prior to the support for the 'model' parameter on the <filter-menu> tag, it was not possible to have more than one model using a filter-menu that had to show translated values from a locale file.

Chapter 12

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` (or `scope` in Hobo 1.3) 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
include
search
```

Association Scopes

```
with_
without_
_is
_is_not
```

Scoping Associations

Chaining

Preparation

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:

```
$ hobo g migration
```

Or with Hobo 1.0:

```
$ ./script/generate hobo_migration
```

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. No function is 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 also 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 13

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 easily be made very 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 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 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.

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.

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>
```

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><def tag="page">
  <html>
    <head>
      <title>My Blog</title>
    </head>
    <body param/>
  </html>
</def>
```

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: 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. However, that's not required. 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>
  </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. 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 in this regard is 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 be learned 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. 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 properly document DRYML the language. That will possibly be a source of confusion, if you’re used to working with Rapid. If you keep in mind that we’re not allowed to use any Rapid tags in this guide and you’ll be fine.

In order to clearly see the implicit context, 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 comfortingly 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). 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 before. In the case of `<view/>` which is a self-closing tag familiar from XML that happens immediately. The `<1:author>` tag is more interesting. We set the context to be the author, so that the link goes to the right place. Inside the `<1: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, but using only the `with` attribute:

```
<page>
  <content:>
    <h2><view with="@blog_post.title"/></h2>
    <div class="details">
      Published by <1
        with="@blog_post.author"
        <view with="this.name"/></1>
      on <view with="@blog_post.published-at"/>.
    </div>
    <div class="post-body">
      <view with="@blog_post.body"/>
    </div>
  </content:>
</page>
```

Note: 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=&#34;&i;&#34; />
```

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: 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? Blankness 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>
</h2>
</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="..."> ...
  </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"/>
```

CHAPTER 13

THE ~~IF~~ AND PSEUDO PARAMETERS - BEFORE, AFTER, APPEND, PREPEND, AND REPLACE

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 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 params 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>
```

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-params`

We'll introduce `merge-params` 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-params`, much as we use `merge-attrs`:

```
<def tag="linked-card">
  <card merge-attrs merge-params>
    <heading: param>
      <a href="#">&object_url this">
        <%= h this.to_s %>
      </a>
    </heading:>
  </card>
</def>
```

You can read `merge-params` 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 it is 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-head-` 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 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 %></a>
  </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/>
```

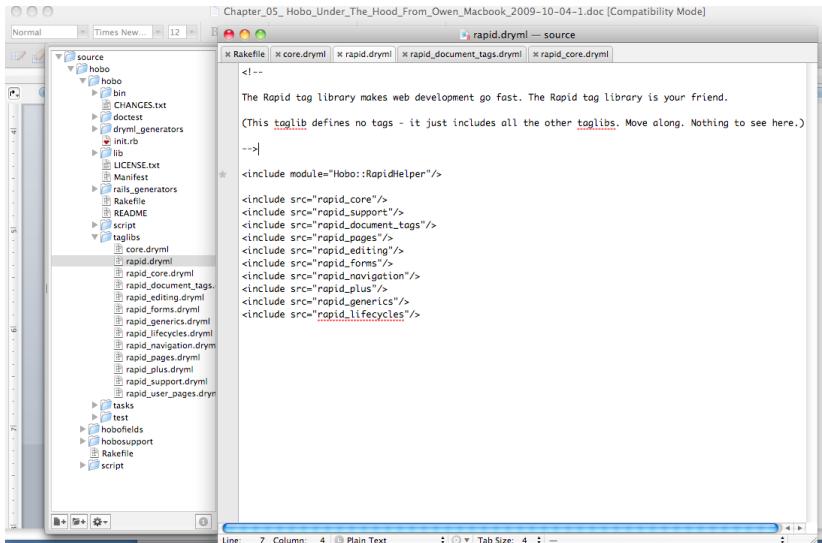
... is valid DRYML.

Chapter 14

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 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`



The screenshot shows a Microsoft Word document titled "Chapter_05_Hobo_Under_The_Hood_From_Owen_Macbook_2009-10-04-1.doc [Compatibility Mode]". The document contains the source code for the `rapid.dryml` file. The code includes XML declarations, comments, and multiple `<include>` statements that import various Hobo Rapid tag library modules such as `rapid_core`, `rapid_support`, `rapid_document_tags`, `rapid_pages`, `rapid_forms`, `rapid_navigation`, `rapid_plus`, `rapid_generics`, and `rapid_lifecycles`.

```
<!--
The Rapid tag library makes web development go fast. The Rapid tag library is your friend.
(This taglib defines no tags - it just includes all the other taglibs. Move along. Nothing to see here.)-->
<include module="Hobo::RapidHelper"/>
<include src="rapid_core"/>
<include src="rapid_support"/>
<include src="rapid_document_tags"/>
<include src="rapid_pages"/>
<include src="rapid_forms"/>
<include src="rapid_navigation"/>
<include src="rapid_plus"/>
<include src="rapid_generics"/>
<include src="rapid_lifecycles"/>
```

Figure 315: `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.
Rapid	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.

```
<call-tag>
<wrap>
<partial>
<repeat>
<do>
<with>
<if>
<else>
<unless>
```

<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.administrator?">
  Logged in as administrator
</if>
<else>
  Logged in as normal user
</else>
```

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`, and `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 controls parameter, the default controls column is enabled 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 already includes-

<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, then 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>
```

```
-> <a href="/blog_posts/1">My Blog Post</a>
```

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, it 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> </sp
```

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.

```
<section-group>
<section>
<aside>
<header>
<footer>
```

<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.

```
<has-many-editor>
<belongs-to-editor>
<select-one-editor>
<string-select-editor>
<boolean-checkbox-editor>
<integer-select-editor>
<editor>
```

<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 is a *DOM ID*, *not a part name*. 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: That is a *DOM ID* and not a *part name*. 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.

- message: A message to display in the AJAX-progress spinner. Default: “Saving...”

Note: 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.

<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: That is a *DOM ID*, not a *part name*. 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. <editor> 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 [hoboui](#)

<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.

```
<or-cancel>
<form>
<submit>
<remote-method-button>
<update-button>
<delete-button>
<create-button>
<select-one>
<name-one>
<select-input>
<error-messages>
<select-many>
<after-submit>
<select-menu>
<check-many>
<hidden-id-field>
<input-many>
<input-all>
<input>
<collection-input>
```

<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 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 autocomplete 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 autocomplete being defined in a controller. A simple example:

```
<form with=&quot;&ProjectMembership.new&quot;>
  <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 autocomplete 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

- o 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 chose 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

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"`

```
<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="
```

...

```
</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='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.

```
<card>
<search-card>
<empty-collection-message>
<collection>
<record-flags>
```

<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 not 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.

```
<transition-button>
<transition-buttons>
```

<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.

```
<navigation>
<nav-item>
<account-nav>
<page-nav>
```

<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, and page title, standard stylesheet JavaScript includes, the AJAX progress spinner, default header with app-name, account navigation, main navigation, 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-hide search feature. At the moment this tag is not very flexible. It is not easy to use 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 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:

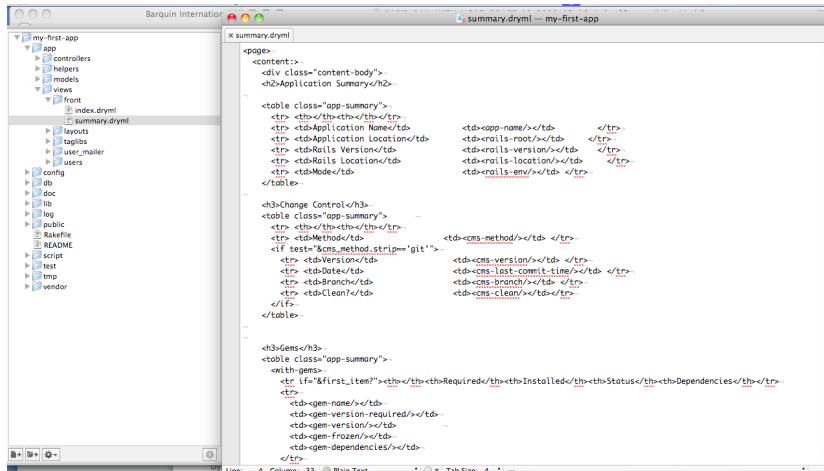


Figure 316: The contents of the "summary.dryml" file

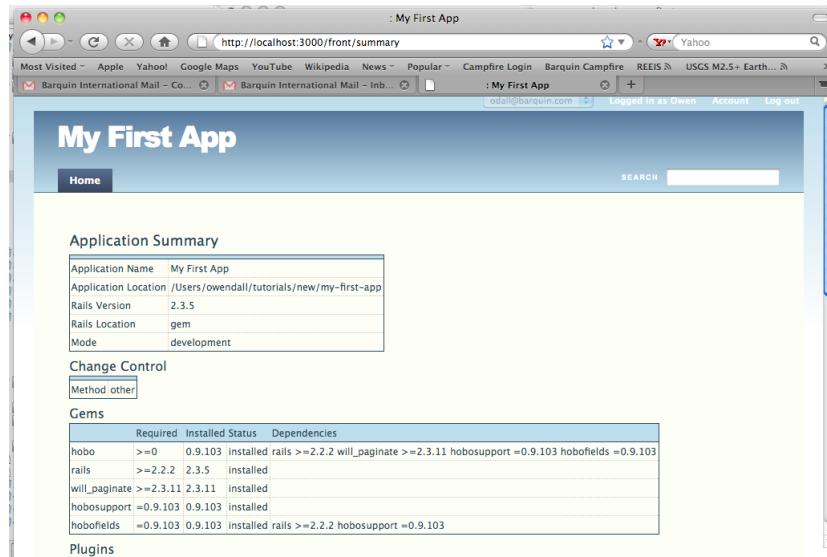


Figure 317: Sample view of the first section of an application summary page

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><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>Mode</td>
  <td><rails-env/></td>
</tr>
</table>

<h3>Change Control</h3>
<table class="app-summary">
  <tr> <th></th><th></th><th></th></tr>
  <tr>
    <td>Method</td>
    <td><cms-method/></td>
  </tr>
  <if test=&quot;&cms_method.strip==&quot;git&quot;>
    <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>Bundled Gems</h3>
<table class="app-summary">
  <with-gems>
    <tr if=&quot;&first_item?&quot;>
      <th></th>
      <th>Version</th>
      <th>Dependencies</th>
    </tr>
    <tr>
      <td><gem-name/></td>
      <td><gem-version/></td>
      <td><gem-dependencies/></td>
    </tr>
```

```
</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>
<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>
```

```
<th></th>
</tr>
<with-models>
<tr>
  <td><model-name/></td>
  <td><model-table-name/></td>
  <td><model-table-comment/></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>
        <th></th>
      </tr>
      <tr>
        <td><model-column-name/></td>
        <td><model-column-type/></td>
        <td><model-column-comment/></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>
```

```
</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>, tries to 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>, determines 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.

```
<with-fields>
<with-field-names>
```

<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 above.

Parameters

- 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:

```
<table fields='...'/>
```

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.

```
<simple-page>
<login-page>
<forgot-password-page>
<forgot-password-email-sent-page>
<account-disabled-page>
<account-page>
```

<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-header
 - * heading
 - content-body
 - * form
 - list-item-list
 - email-address-label
 - email-address-input
 - actions
 - submit

<forgot-password-email-sent-page>

The 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 the privacy concern of the forgotten-password mechanism being used to tell if a given email is associated with an account or not.

Parameters

- body
 - 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

Index

#, 91
:accessible => true, 227
:dependent => destroy, 117
:polymorphic => true, 263

acts_as_list plugin, 377
acts_as_state_machine plugin, 379
Adding
 controller actions, 368
 model fields, 66
 Roles, 222
administrator?, 88, 353, 362
after_submit, 376
after_user_new, 361
app.en.yml, 72
Application name, 141
Application Summary, 214
application.dryml, 63, 101, 134
 ApplicationController, 91
apply_scopes, 177, 239
Associations, 227
attr_accessible, 355
attr_protected, 355
attr_readonly, 355
Attribute input help, 74
Auto Actions
 :redirect parameter, 375
 Customizing, 375
auto-generated tags, 100, 127, 128, 162
 cards.dryml, 63, 100, 126
 editing, 128
 forms.dryml, 63, 100, 126
 pages.dryml, 63, 100, 126, 234
 show.dryml, 179
auto_actions, 91, 218, 365

:read_only, 99, 366
:write_only, 99, 366
hobo_create, 373
hobo_create_for, 374
hobo_destroy, 374
hobo_index, 177, 371, 373
hobo_index_for, 374
hobo_new, 371, 373
hobo_new_for, 374
hobo_show, 370, 373
hobo_update, 371, 374
 Owner actions, 366, 374
auto_actions_for, 219, 367
Autocompleters, 376
Automatic Indexing, 305
Automatic redirects, 376
Automatic Routes, 368

belongs_to, 108, 115, 117, 123, 305
 :creator => true attribute, 361

Change tracking helpers, 352
Changing Field Names, 70
Changing field names, 72
Charts, 251
Child relationships, 400
children, 121, 210
CKEditor, 247
Controller collection selection options, 373
Controller Instance Variables, 367
Controllers, 90, 176
Create a Hobo application, 53
create a plugin, 291
create_permitted?, 89
current context, 234

Database Index, 14, 305
 Database index, 111, 207
 destination_after_submit, 376
 destroy_permitted?, 89
 directory structure, 54
 Drag and drop reordering, 244, 377
 DRYML, 48, 126, 229, 269, 429
 <controls:>, 297
 Aliasing tags, 459
 all_attributes local variable, 443
 all_parameters local variable, 457
 attributes local variable, 443
 Automatically Generated Tags, 346
 Conditionals, 446
 Default Parameter, 434
 default parameter attribute, 147, 149
 Even/odd classes, 447
 Extending a tag, 458
 extending tags, 455
 field attribute, 439
 Field chains, 440
 first_item?, 447
 Flag attributes, 442
 Implicit Context, 437
 implicit context, 448
 last_item?, 447
 Merge, 458
 merge, 157
 merge attribute, 151, 444
 merge-attrs, 444
 merge-params, 456
 Merging Attributes, 443
 Merging selected attributes, 444
 Nested parameters, 435, 452
 nested parameters, 244
 Nested scoped variables, 465
 Numeric field indices, 441
 parameter tag, 433
 Parameters, 432
 parameters local variable, 457
 Polymorphic tags, 460
 Pseudo parameters, 449
 Repeating over hashes, 447
 Repetition, 446
 scoped variables, 463
 Selectively merging parameters, 458
 Tag attributes, 441
 Taglibs, 465
 Template Processing Order, 348
 The class attribute, 445
 The Default Parameter, 436
 with attribute, 439
 Wrapping content, 462
 DRYML parts, 265
 DRYML tag, 143
 DRYML taglib, 291
 enum_string, 105, 241
 ERB, 127
 Field Help, 228
 Field Validation, 76
 find_instance, 373
 finite state machine, 381
 Flash messages, 375
 flash[:notice], 375
 Foreign keys, 104, 109
 FusionCharts, 251
 gem list, 23
 generators, 335
 -pretend, 337
 generator options, 336
 hobo g admin_subsite, 302
 hobo g migration, 63, 67, 208, 243
 hobo g model, 63
 hobo g resource, 63, 85, 205, 242
 hobo resource generator, 107
 hobo:admin_subsite, 336, 339, 340
 hobo:assets, 336, 338, 341
 hobo:controller, 336, 341
 hobo:front_controller, 336, 339, 341
 hobo:i18n, 336, 340, 342
 hobo:migration, 336, 339, 342
 hobo:model, 336, 342
 hobo:rapid, 336, 342
 hobo:rapid generator, 339
 hobo:resource, 336, 343
 hobo:routes, 336, 343
 hobo:setup_wizard, 336, 337
 hobo:subsite, 336, 343
 hobo:subsite_taglib, 336, 344

hobo:test_framework, 336, 338, 344
 hobo:user_controller, 336, 344
 hobo:user_mailer, 336, 345
 hobo:user_model, 336, 345
 hobo:user_resource, 336, 339, 345
 private-site option, 338
 git, 309
 git add ., 324
 git commit, 324
 git init, 324
 git push, 325
 github.com, 309
 group_by, 237
 has_many, 108, 114, 117, 123
 has_many :through, 114, 117, 123
 Heroku, 309, 319, 340
 heroku rake, 327
 Hobo, 11
 Hobo automatic actions, 90
 Hobo Controller action, 99
 Hobo Enhancement, 13
 Hobo Fields, 13
 Hobo migrations, 116
 Hobo Model Controller, 13
 Hobo Rapid, 13
 Hobo Rapid Library, 127
 Hobo scopes, 419
 _after, 424
 _before, 423
 _between, 424
 _contains, 422
 _does_not_contain, 422
 _does_not_end, 423
 _does_not_start, 422
 _ends, 423
 _is, 421, 426
 _is_not, 422, 426
 _starts, 422
 by_most_recent, 424
 Chaining, 428
 include, 425
 Lifecycle scopes, 424
 limit, 425
 not_, 423
 order_by, 425
 recent, 425
 Scoping Associations, 426
 search, 425
 with_, 425
 without_, 426
 Hobo Setup Wizard, 54
 Hobo themes, 269
 hobo.en.yml, 72
 Hobo::Lifecycles::Lifecycle, 385
 hobo_model_controller, 91, 365
 home_page, 376
 I18n, 397, 402
 i18n, 72, 101
 id_rsa.pub, 318
 implicit context, 165, 168
 Index actions, 369
 inheritance_column, 306
 install MySQL, 26
 Installing, 18
 Integrated Development Environments, 26
 Internationalization, 397
 internationalization, 228
 lambda, 297
 layouts, 431
 Lifecycles, 295, 301, 379
 :available_to, 296, 301, 389
 :available_to attribute, 383
 :become attribute, 383
 :state_field, 385
 :user_becomes attribute, 383
 Controller actions, 390
 create, 382
 Create controller actions, 391
 creator options, 386
 creators, 385
 Defining, 385
 Defining creators, 386
 Defining states, 386
 Defining transitions, 387
 Keys, 394
 Repeated transition names, 389
 state, 295, 306, 382, 384
 transition, 295, 296, 301, 383, 385
 Transition controller actions, 392

transition options, 388
validation extensions, 390
loop, 140
Migrating ViewHints to i18n, 413
Model Relationships, 114
multi-field index, 306
MySQL, 26
mysysgit, 311
Navigation tabs, 100, 140
never_show, 354
not_found action, 378
Oracle, 39, 207
oracle_enhanced, 40
override auto-generated tags, 158
owner_is?, 264
Pagination, 373
param, 143, 238
parameter tag, 145, 184
params, 143, 177
parse_sort_param, 177
permission_denied action, 378
Permissions, 84, 87, 90, 222, 226, 349
 acting_user, 88, 351
 all_changed?, 352
 any_changed?, 352
 API, 358
 associations, 357
 Change tracking, 351
 creatable_by?, 359
 create_permitted?, 87
 defining, 350
 defining edit, 356
 Destroy, 353
 destroy_permitted?, 87
 destroyable_by?, 360
 Direct permission tests, 359
 Edit, 354
 edit, 350
 editable_by?, 360
 errors, 378
 guest?, 88, 351
 method_callable_by?, 360
none_changed?, 352
only_changed?, 352
signed_up?, 88, 351
updatable_by?, 360
update_permitted?, 87, 89, 351
user_create, 359
user_destroy, 359
user_find, 359
user_new, 359
user_save, 359
user_update_attributes, 359
user_view, 359
validations, 362
View, 354
View helpers, 363
 can_call?, 363
 can_create?, 363
 can_delete?, 363
 can_edit?, 363
 can_update?, 363
 viewable_by?, 360
polymorphic, 305
polymorphic associations, 262
polymorphic tag, 189, 232
Private key, 317
Public Key, 317
Rails
 layouts, 348
Rapid Generator, 15
Rapid library, 161
Rapid Parameter Tag, 184
Rapid Tag Library, 15
redirect, 371
redirect_to, 371, 375
removing database tables, 70
Removing model fields, 68
Renaming Fields, 228
render, 371
reorder action, 378
replace attribute, 173, 187
Replacing tag parameters, 151
respond_to, 372
Response types
 Handling non-HTML, 372
Routes, 90, 92, 132

Ruby, 47
 Ruby comment, 91
 RubyGems Documentation Server, 22
 schema, 60
 search, 66
 setup wizard
 bypassing, 84
 setup_wizard, 335
 Show actions, 368
 SSH key pairs, 316
 sub-site, 302
 Tag nesting, 185
 taglibs folder, 63
 Tags
 <A-or-An>, 482
 <I18n>, 408
 <JavaScript>, 510
 <You>, 481
 <Your>, 237, 481
 <a>, 158, 164, 198, 478
 action attribute, 199
 <account-disabled-page>, 528
 <account-nav>, 506
 <account-page>, 528
 <after-submit>, 376, 496
 <ajax-progress>, 511
 <app-name>, 140
 <aside>, 485
 <belongs-to-editor>, 486
 <boolean-checkbox-editor>, 487
 <call-tag>, 471
 <card>, 132, 139, 232, 503
 <check-many>, 497
 <collection-input for='ActiveRecord::Base'>
 502
 <collection-input>, 502
 <collection-name>, 413
 <collection-name>, 478
 <collection>, 140, 158, 161, 237, 504
 <comma-list>, 482
 <count/>, 402
 <count>, 480
 <create-button>, 494
 <def>, 143, 431
 <delete-button>, 493
 <dev-user-changer>, 473
 <do>, 472
 <doctype>, 509
 <edit-page>, 132, 189, 192, 346
 <editor for='BigDecimal'>, 490
 <editor for='HoboFields::EnumString'>, 488
 <editor for='big_integer'>, 490
 <editor for='boolean'>, 489
 <editor for='date'>, 489
 <editor for='datetime'>, 489
 <editor for='float'>, 489
 <editor for='html'>, 489
 <editor for='integer'>, 489
 <editor for='password'>, 489
 <editor for='string'>, 489
 <editor for='text'>, 489
 <editor>, 488
 <else>, 158, 171, 234, 472
 <empty-collection-message>, 503
 <error-messages>, 496
 <extend>, 151, 153, 186, 194, 233, 459
 <field-list>, 179, 182, 189, 191, 193, 194, 242, 473
 <field-list> fields attribute, 182
 <field-list> labels, 183
 <fieldname-label>, 179
 <filter-menu/>, 408
 <filter-menu>, 511
 <flash-message>, 510
 <flash-messages>, 510
 <footer>, 486
 <forgot-password-email-sent-page>, 527
 <forgot-password-page>, 527
 <form>, 132, 140, 189, 191, 491
 <gravatar>, 513
 <has-many-editor>, 486
 <header>, 485
 <hidden-id-field>, 498
 <hobo-rapid-javascripts>, 477
 <ht/>, 410
 <html>, 509
 <human-attribute-name/>, 411
 <human-collection-name/>, 411

<if-ie>, 510
 <if>, 158, 171, 472
 <image>, 476
 <include>, 465
 <index-page>, 132, 346
 <input for='BigDecimal'>, 502
 <input for='HoboFields::EnumString'>, 500
 <input for='big_integer'>, 502
 <input for='boolean'>, 501
 <input for='date'>, 501
 <input for='datetime'>, 501
 <input for='float'>, 502
 <input for='integer'>, 502
 <input for='password'>, 501
 <input for='string'>, 502
 <input for='text'>, 500
 <input for='time'>, 501
 <input-all>, 499
 <input-many>, 498
 <input>, 189, 194, 499
 <integer-select-editor>, 488
 <links-for-collection>, 482
 <live-search>, 511
 <login-page>, 526
 <main-nav>, 101, 140
 <model-name-human/>, 410
 <name-one>, 494
 <name/>, 169
 <name>, 165, 477
 <nav-item>, 506
 <navigation>, 505
 <new-page>, 132, 189, 192, 346
 <nil-view>, 474
 <not-found-page>, 378, 509
 <or-cancel>, 490
 <page-nav>, 507
 <page-scripts>, 508
 <page>, 507
 <param-content>, 462
 <partial>, 471
 <permission-denied-page>, 378, 508
 <preview-with-more/>, 413
 <preview-with-more>, 513
 <record-flags>, 504
 <remote-method-button>, 492
 <repeat>, 158, 168, 234, 237, 471
 <search-card>, 503
 <section-group>, 485
 <section>, 485
 <select-input>, 495
 <select-many>, 496
 <select-menu/>, 407
 <select-menu>, 497
 <select-one-editor>, 486
 <select-one>, 494
 <set-scoped>, 463
 <set>, 463
 <show-page>, 132, 179, 183, 234, 242, 346
 <simple-page>, 526
 <sortable-collection>, 512
 <spinner>, 476
 <string-select-editor>, 487
 <stylesheet>, 510
 <submit>, 492
 <t/>, 409
 <table-plus>, 174, 237, 512
 <table-plus> fields attribute, 174, 238
 <table>, 475
 <theme-stylesheet>, 481
 <transition-button>, 504
 <transition-buttons/>, 296
 <transition-buttons>, 505
 <type-name>, 477
 <unless>, 473
 <update-button>, 493
 <view for='ActiveRecord::Base'>, 484
 <view for='ActiveSupport::TimeWithZone'>, 484
 <view for='Date'>, 484
 <view for='Numeric'>, 484
 <view for='Time'>, 484
 <view for='boolean'>, 484
 <view for='string'>, 484
 <view/>, 402
 <view>, 482
 <with-field-names>, 525
 <with-fields>, 524
 <with>, 472
 <wrap>, 471
 <you/>, 404

<your/>, 406
param attribute, 145
polymorphism, 192
TextMate, 26
this keyword, 174
this_field, 440
this_key, 237
this_parent, 440
to attribute, 199

User model, 59

Validate acceptance, 82
validation errors, 371
validation helpers, 83
Validations, 14, 82
 length, 81
 uniqueness, 80
 validates_date, 245
 validates_length_of, 83
 validates_numericality_of, 78
 validates_presence_of, 77
 validates_size_of, 83
View templates, 348
view_permitted, 354
view_permitted?(field), 87
ViewHints, 120, 397
 Inline Booleans, 401

web server, 56
Webrick, 48