



RAPID RAILS WITH

HOB

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

OWEN DALL • JEFF LAPIDES • TOM LOCKE

with contributions by

**VENKA ASHTAKALA • TOLA AWOFOLU • TIAGO FRANCO
MARCELO GIORGI • MATT JONES • BRYAN LARSEN**



CONTENTS

| | |
|--|-----|
| LIST OF FIGURES | iii |
| AUTHORS..... | x |
| CONTRIBUTORS | xi |
| PREFACE | xii |
| CHAPTER 1 – INTRODUCTION | 1 |
| What is Hobo? | 1 |
| CHAPTER 2 – INSTALLATION..... | 8 |
| Introductory Concepts and Comments..... | 8 |
| Installing Ruby, Rails, Hobo..... | 9 |
| Using SQLite with Hobo..... | 14 |
| Using MySQL with Hobo..... | 16 |
| Using Oracle with Hobo..... | 22 |
| CHAPTER 3 - INTRODUCTORY TUTORIALS | 32 |
| Introductory Concepts and Comments..... | 33 |
| Tutorial 1 – Directories and Generators | 34 |
| Tutorial 2 – Changing Field Names with View Hints | 48 |
| Tutorial 3 – Field Validation..... | 52 |
| Tutorial 4 – Permissions | 60 |
| Tutorial 5 – Controllers | 67 |
| Tutorial 6 – Navigation Tabs | 76 |
| Tutorial 7 – Model Relationships: Part 1 | 79 |
| Tutorial 8 – Model Relationships: Part II..... | 90 |
| CHAPTER 4 – INTERMEDIATE TUTORIALS..... | 101 |
| Introductory Concepts and Comments..... | 102 |
| Tutorial 9 – Editing Auto-Generated Tags..... | 104 |
| Tutorial 10 – DRYML I: A First Look at DRYML..... | 118 |
| Tutorial 11 – DRYML II: Creating Tags from Tags | 125 |
| Tutorial 12 – Rapid, DRYML and Record Collections | 133 |
| Tutorial 13 – Listing Data in Table Form..... | 146 |
| Tutorial 14 – Working with the Show Page Tag..... | 152 |
| Tutorial 15 – New and Edit Pages with The Form Tag | 161 |
| Tutorial 16 – The <a> Hyperlink Tag | 169 |

| | |
|--|------------|
| CHAPTER 5 – ADVANCED TUTORIALS..... | 173 |
| Introductory Concepts and Comments..... | 174 |
| Tutorial 17 – The Agile Project Manager | 175 |
| Tutorial 18 – Using CKEditor (Rich Text) with Hobo | 223 |
| Tutorial 19 – Using FusionCharts with Hobo..... | 227 |
| Tutorial 20 – Adding User Comments to Models | 237 |
| Tutorial 21 – Replicating the Look and Feel of a Site | 245 |
| Tutorial 22 - Creating a “Look and Feel” Plugin for Hobo | 268 |
| Tutorial 23 – Using Hobo Lifecycles for Workflow | 272 |
| Tutorial 24 – Creating an Administration Sub-Site | 279 |
| Tutorial 25 – Using Hobo Database Index Generation..... | 282 |
| CHAPTER 6 – DEPLOYING YOUR APPLICATIONS..... | 285 |
| Introductory Concepts and Comments..... | 286 |
| Tutorial 26 – Installing and Using Git..... | 287 |
| Tutorial 27 – Rapid Deployment with Heroku | 297 |
| INDEX..... | 310 |

LIST OF FIGURES

| | |
|---|----|
| Figure 1: Download Site for Ruby..... | 9 |
| Figure 2: Installing Ruby | 10 |
| Figure 3: Ruby Installation Options..... | 10 |
| Figure 4: Setup Wizard Complete..... | 11 |
| Figure 5: Sample console output after installing the Hobo gem..... | 12 |
| Figure 6: Summary of Installed gems | 12 |
| Figure 7: Sample console output from the "gem env" command | 13 |
| Figure 8: Sample console output from installing the sqlite3-ruby gem..... | 14 |
| Figure 9: SQLite3 download website | 14 |
| Figure 10: Target location for the SQLite3 DLL..... | 15 |
| Figure 11: Site location for the SQLite DLL | 15 |
| Figure 12: Download site for MySQL | 16 |
| Figure 14: Choose the installation type | 17 |
| Figure 13: Using the .msi file to install MySQL on Windows | 17 |
| Figure 15: MySQL Server Setup Wizard..... | 18 |
| Figure 16: Configure MySQL Server | 18 |
| Figure 17: Choose Standard Configuration | 19 |
| Figure 18: Install as Windows Service | 19 |
| Figure 19: Launch MySQL from the command prompt..... | 20 |
| Figure 20: Create the database from the command line | 20 |
| Figure 21: Console output from the Hobo command | 21 |
| Figure 22: The MySQL format for the database.yml configuration file..... | 21 |
| Figure 23: Console output after installing Oracle gems for Ruby and Rails | 22 |
| Figure 24: The generated database.yml file for Oracle..... | 23 |
| Figure 25: Oracle database install download site | 24 |
| Figure 26: Running the Oracle XE installation..... | 25 |
| Figure 27: Specifying the database passwords | 25 |
| Figure 28: Launch the Database home page | 26 |
| Figure 29: Log in as SYS to configure your database..... | 26 |
| Figure 30: Creating a schema/user to use with Hobo | 27 |
| Figure 31: The tnsnames.ora file created during installation..... | 27 |
| Figure 32: Log into Oracle to view the created table..... | 28 |
| Figure 33: Access the Oracle Object Browser..... | 28 |
| Figure 34: Review the User table from within Oracle..... | 29 |
| Figure 35: Review the Indexes view for Users..... | 29 |
| Figure 36: Review the Constraints view for User..... | 30 |
| Figure 37: Hobo application folder structure..... | 35 |
| Figure 38: The default User model created by Hobo..... | 37 |
| Figure 39: Contents of the first Hobo migration file | 38 |
| Figure 40: Contents of the "schema.rb" file after the first migration | 38 |
| Figure 41: Home page for "My First App" | 39 |
| Figure 42: Drop down selector for the active user..... | 40 |

| | |
|--|----|
| Figure 43: Location of the Rapid templates..... | 41 |
| Figure 44: Folder location for Models and Views | 42 |
| Figure 45: Migration file changes..... | 43 |
| Figure 46: Contacts tab on "My First App" | 44 |
| Figure 47: New Contact page for "My First App" | 45 |
| Figure 48: Remove field from contact model | 46 |
| Figure 49: Creating a Hobo "ViewHints" definition for the Contact model | 49 |
| Figure 50: View of field relabeled using the Hobo viewhints "field_names" method | 49 |
| Figure 51: Adding help text using the Hobo viewhints "field_help" method..... | 50 |
| Figure 52: Contact entry page with ViewHints enabled..... | 50 |
| Figure 53: CSS definitions for the input text fields | 51 |
| Figure 54: Modified entry in "application.css" to shorten text prompts | 51 |
| Figure 55: Page view of validating presence of name | 53 |
| Figure 56: Page view of double validation error | 53 |
| Figure 57: Adding "validates_numericality_of" validation..... | 54 |
| Figure 58: Page view of triggering the "validates_numericality_of" error..... | 55 |
| Figure 59: Page view of uniqueness validation error..... | 56 |
| Figure 60: Page view of triggering a range validation error | 57 |
| Figure 61: Page view of validation of text length error | 57 |
| Figure 62: Page view of "validates_acceptance_of" error..... | 58 |
| Figure 63: Welcome to One Table in the Permissions tutorial | 61 |
| Figure 64: Recipes tab | 62 |
| Figure 65: Page view of created recipes | 63 |
| Figure 66: Table of Hobo permission methods..... | 64 |
| Figure 67: Table of Hobo "acting_user" options | 64 |
| Figure 68: Page view of a Recipe | 66 |
| Figure 69: Making the Recipes tab disappear..... | 69 |
| Figure 70: Error message "The page you were looking for could not be found" | 70 |
| Figure 71: Setting the Hobo "name" attribute for a model | 71 |
| Figure 72: Creating your own custom "name" attribute | 71 |
| Figure 73: Viewing the edit URL | 72 |
| Figure 74: "Unknown action" error page..... | 73 |
| Figure 75: Hobo Controller action summary | 75 |
| Figure 76: Customizing the name of a tab | 77 |
| Figure 77: Removing the default Home tab..... | 78 |
| Figure 78: Renaming a copy of your application..... | 79 |
| Figure 79: Using "enum_string" to create a drop-down list of Countries | 81 |
| Figure 80: Index page for Countries | 86 |
| Figure 81: Selecting a Country for a Recipe..... | 86 |
| Figure 82: Active link on Country name in the Recipe show page | 87 |
| Figure 83: The Country show page accessed from the Recipe show page..... | 88 |
| Figure 84: Editing Hobo Permissions to remove the Country Edit link | 89 |
| Figure 85: The Categories tab on the Four Table app | 93 |
| Figure 86: The Index page for Categories | 94 |
| Figure 87: "Category Assignments" on the Recipe show page | 94 |
| Figure 88: Assignment multiple Categories to a Recipe | 95 |

| | |
|--|-----|
| Figure 89: Edit page view of a Recipe with multiple Categories assigned | 96 |
| Figure 90: Show page view of Categories assigned to a recipe..... | 97 |
| Figure 91: Using Hobo ViewHints to enhance the view of related records | 98 |
| Figure 92: Show page for a Category before using ViewHints | 98 |
| Figure 93: Category page view after adding ViewHints "children :recipes" declaration | 99 |
| Figure 93: Changing the display style within <collection> | 100 |
| Figure 93: Specifying what <collection> tag will display | 100 |
| Figure 94: Folder view of \taglibs\auto\rapid | 103 |
| Figure 96: Folder view of the rapid DRYML files | 105 |
| Figure 95: Front page view of the Four Table application | 105 |
| Figure 97: Content of the "pages.dryml" file | 107 |
| Figure 98: Hobo Page Action Tag definitions | 107 |
| Figure 99: The Hobo Rapid <index-page> tag definition in the pages.dryml file | 108 |
| Figure 100: The Recipes Index page | 109 |
| Figure 101 : View of the taglibs/auto/rapid folder..... | 109 |
| Figure 102: Adding the definition of index-page into the application.dryml file..... | 110 |
| Figure 103: Modifying the "heading" parameter the index-page definition..... | 111 |
| Figure 104: Page view of "My Recipes" after modifying the <index-page> tag..... | 112 |
| Figure 105: Adding the <index-page/> tag to index.dryml | 113 |
| Figure 106: How a change to the <index-page> tag affects a collection..... | 114 |
| Figure 107: Changing the tab order for the main navigation menus | 115 |
| Figure 108: Changing the application name with the app-name tag | 116 |
| Figure 109: The \views\front\index.dryml file after the first modification | 119 |
| Figure 110: The Home page with the first set of custom messages..... | 119 |
| Figure 111: Passing a parameter to the tag <messages> you created | 120 |
| Figure 112: How the passed parameter displays on the page | 121 |
| Figure 113: Passing three parameters to your <messages> tag..... | 121 |
| Figure 114: Page display using your custom <bd-it> tag | 123 |
| Figure 115: Calling <span:> explicitly within to your <bd-it> tag | 123 |
| Figure 116: Adding the custom <more-messages> tag to front\index.dryml | 126 |
| Figure 117: Page rendering with <more-messages>..... | 126 |
| Figure 118: Extending the tag <message> in application.dryml | 128 |
| Figure 119: Using the extended <message> tag | 128 |
| Figure 120: Page view of the next additions to <message> | 129 |
| Figure 121: Page view of the <more-messages> tag usage | 130 |
| Figure 122: Page view of overriding the default message 0..... | 131 |
| Figure 123: More parameter magic..... | 132 |
| Figure 124: The Four Tables application as we left it | 134 |
| Figure 125: Creating the /views/recipes/index.dryml file..... | 134 |
| Figure 126: page view of using a blank "<collection:></collection:>" tag..... | 137 |
| Figure 127: How the <collection> tag iterates..... | 137 |
| Figure 128: Using the <a> hyperlink tag within a collection | 138 |
| Figure 129: Specifying what <collection> tag will display | 139 |
| Figure 130: Changing the display style within <collection> | 140 |
| Figure 131: Changing the implicit context within <collection> | 141 |
| Figure 132: Creating comma-delimited multi-valued lists in a <collection> | 142 |

| | |
|--|-----|
| Figure 133: Adding the count of values in the <card> tag | 143 |
| Figure 134: Using "if--else" within a tag to display a custom message | 144 |
| Figure 135: Using <table-plus> to display a columnar list..... | 147 |
| Figure 136: Adding a "Categories Count" to <table-plus>..... | 148 |
| Figure 137: Adding a comma-delimited list within a <table-plus> column..... | 149 |
| Figure 138: adding a search facility to <table-plus> using Hobo's apply_scopes method | 150 |
| Figure 139: Found Recipes searching for "French"..... | 151 |
| Figure 140: The Recipe show page before modification | 153 |
| Figure 141: Recipe show page after removing three critical lines of code..... | 154 |
| Figure 142: Using the <field=list> tag to choose which fields to display | 154 |
| Figure 143: Using the <collection-heading:> tag | 155 |
| Figure 144: Using the <body-label:> parameter tag | 156 |
| Figure 145: Using the <country-label:> parameter to change the label on the page | 158 |
| Figure 146: A new show page for Recipes | 159 |
| Figure 147: Page view of using the replace attribute in the <content-body:> parameter tag | 160 |
| Figure 148: Default Hobo form rendering..... | 163 |
| Figure 149: Modifying the <field-list> tag to remove fields on a page..... | 164 |
| Figure 150: First step using the <input> tag | 166 |
| Figure 151: Adding the label for the filed "Title" | 167 |
| Figure 152: Adding the rest of the input fields | 168 |
| Figure 153: Generating an active link to a list of Countries | 170 |
| Figure 154: The Countries index page activated by your custom link | 170 |
| Figure 155: Constructing a custom link to the "New Country" page | 171 |
| Figure 156: Page view of custom <show-page> tag..... | 172 |
| Figure 157: Adding "has_many :requirements" to the Project class | 177 |
| Figure 158: Adding "belongs_to :project" and "has_many :tasks" to the Requirement model.. | 178 |
| Figure 159: Adding the "belongs_to" and "has_many" declarations to the Task model | 178 |
| Figure 160: Adding the two "belongs_to" definitions to the TaskAssignment model | 179 |
| Figure 161: Adding the "has_many" declarations to the User model..... | 179 |
| Figure 162: First Hobo migration for Projects..... | 180 |
| Figure 163: View of indexes created by the migration..... | 180 |
| Figure 164: The default Home page for the Projects application | 181 |
| Figure 165: The Projects index page | 182 |
| Figure 166: New Requirement page | 182 |
| Figure 167: Index view for Requirements | 183 |
| Figure 168: New Task page | 183 |
| Figure 169: Index view for Tasks | 184 |
| Figure 170: Part 1 of the Application Summary page | 185 |
| Figure 171: Part 2 of the Application Summary page | 185 |
| Figure 172: Part 3 of the Application Summary page | 186 |
| Figure 173: Part 4 of the Application Summary page | 186 |
| Figure 174: Effect of removing the "index" action from the Tasks controller | 187 |
| Figure 175: View of "No Requirements to display" message | 188 |
| Figure 176: The "New Requirement" link now appears..... | 188 |
| Figure 177: View of the "New Requirement" page | 189 |
| Figure 178: View of the in-line "Add a Task" form | 190 |

| | |
|---|-----|
| Figure 179: Requirement page after modifying controller definitions | 192 |
| Figure 180: Defining available roles using "enum_string" | 193 |
| Figure 181: Modifying the "create_permitted" method to the User model | 194 |
| Figure 182: Users Controller with "auto actions :all: | 194 |
| Figure 183: The Users tab is now active..... | 194 |
| Figure 184: The Edit User page with the new Role field | 195 |
| Figure 185: Adding the use of Role in Permissions | 196 |
| Figure 186: Modifying the "update_permitted?" method in the Requirement model | 198 |
| Figure 187: Assigning multiple Users to a Task in the Edit Task page..... | 199 |
| Figure 188: Contents of the \apps\viewhints folder..... | 200 |
| Figure 189: The default blank "project_hints.rb" file for the "ProjectHints" class..... | 200 |
| Figure 190: Defining "field_names" and "field_help" in ProjectHints | 201 |
| Figure 191: The New Project page using "ProjectHints" | 201 |
| Figure 192: The default application name and welcome message | 202 |
| Figure 193: Changing the application name in "application.dryml" | 203 |
| Figure 194: Modifying "\front\index.dryml" | 203 |
| Figure 195: Home page modified by changing "/front/index.dryml" | 204 |
| Figure 196: Newly modified home page..... | 204 |
| Figure 197: Extending the card tag for Task in "application.dryml" | 205 |
| Figure 198: Viewing assigned users on a the Task card..... | 206 |
| Figure 199: Listing the contents for the "\views>taglibs\auto\rapid" folder | 207 |
| Figure 201: The auto-generated "show-page" tag for User in "pages.dryml" | 208 |
| Figure 200: contents of the pages.dryml file | 208 |
| Figure 202: View of the enhanced User "show-page" | 210 |
| Figure 203: The Users tab showing all assignments..... | 211 |
| Figure 204: Using the Hobo "<table-plus>" feature to enhance the Requirements listing | 213 |
| Figure 206: Using a search within the Requirements listing | 214 |
| Figure 205: Enhancing the <table-plus> listing..... | 214 |
| Figure 207: The Edit Requirement form with selectable status codes..... | 216 |
| Figure 208: Creating an AJAX status update for Requirements..... | 217 |
| Figure 209: adding the "validates_timeliness" gem to "environment.rb" | 221 |
| Figure 210: Task model with "due_date" and a validation for the date..... | 222 |
| Figure 211: Error message from trying to enter a date earlier than today | 222 |
| Figure 212: CKEditor source folder listing | 223 |
| Figure 213: Using the ":html" field option to trigger rich-text editing | 225 |
| Figure 214: Adding the required CKEditor references in application.dryml | 225 |
| Figure 215: Sample Hobo form using CKEditor | 226 |
| Figure 216: Registration form to request FusionCharts..... | 227 |
| Figure 217: Download page for FusionCharts..... | 228 |
| Figure 218: Target location for the FusionCharts SWF files..... | 229 |
| Figure 219: Adding the required <extend tag='page'> definition in application.dryml | 229 |
| Figure 220: Screen shot of sample recipe data for the tutorial | 230 |
| Figure 221: Content of recipes/index.dryml used to render the FusionChart..... | 233 |
| Figure 222: Screen shot of rendered FusionCharts bar chart..... | 233 |
| Figure 223: recipe/index.dryml to render a FusionCharts pie chart and bar chart | 235 |
| Figure 224: Screen shot of the rendered FusionCharts bar and pie charts..... | 236 |

| | |
|--|-----|
| Figure 225: Editing the application name for the Comments Recipe | 237 |
| Figure 226: Home page for the Comments Recipe | 238 |
| Figure 227: Adding Body and Game to Comments | 238 |
| Figure 228: Permissions for the Comment model | 239 |
| Figure 229: The auto_actions for the comments_controller | 239 |
| Figure 230: Adding comments to the Game model | 240 |
| Figure 231: Posting comments about a game | 241 |
| Figure 232: Comments' Recipe with support for courts | 242 |
| Figure 233: Adding courts to comments | 242 |
| Figure 234: Adding comments to courts | 242 |
| Figure 235: Modifying auto_actions for the comments_controller (allow court) | 243 |
| Figure 236: Hiding court and game in the comment's form | 243 |
| Figure 237: View of the in-line "Add a Comment" form | 244 |
| Figure 238: Posting comments about a court | 244 |
| Figure 239: Screen shot of the nifa.usda.gov home page | 245 |
| Figure 241: The NIFA photo image | 246 |
| Figure 240: The NIFA banner image | 246 |
| Figure 242: The NIFA main navigation bar | 247 |
| Figure 243: NIFA navigation panels | 247 |
| Figure 244: NIFA footer navigation | 247 |
| Figure 245: The NIFA Demo default home page | 249 |
| Figure 246: Using the "app-name" tag to change the default application name | 249 |
| Figure 247: Using Firebug to locate the background color | 250 |
| Figure 248: Using Firebug to find the images used by Hobo for the default background | 250 |
| Figure 249: Adding the new background color to "application.css" | 251 |
| Figure 250: First pass at modifying "application.dryml" | 252 |
| Figure 251: The two images used in NIFA's top banner | 252 |
| Figure 252: How to reference the banner gif in "application.css" | 254 |
| Figure 253: View of the NIFA Demo login page | 255 |
| Figure 254: The Navigation Panel before refactoring | 255 |
| Figure 255: View of our first pass at the main navigation menu | 256 |
| Figure 256: Still need more to fix the top navigation menu... .. | 257 |
| Figure 257: The fixed NIFA main navigation bar | 258 |
| Figure 258: View of the default three-column formatting | 259 |
| Figure 259: View of the left panel contact without styling | 261 |
| Figure 260: View of the left panel content with correct styling | 262 |
| Figure 261: View of the right panel content with styling | 264 |
| Figure 262: View of the main content panel | 265 |
| Figure 263: NIFA Demo with final footer styling | 266 |
| Figure 264: Batch file with commands to create the plugin folders and content | 268 |
| Figure 265: Guest view Recipes - All recipes are in state "Not Published" | 275 |
| Figure 266: Recipes ready to Publish. | 275 |
| Figure 267: Omelet recipe after being placed in the "Published" state | 276 |
| Figure 268: Recipe index with buttons for "Publish" and "Not Publish" | 276 |
| Figure 269: Guest user can only see the published Recipe | 277 |
| Figure 270: Generator console output for creating an admin sub-site | 279 |

| | |
|--|-----|
| Figure 271: View of the Admin folder contents | 280 |
| Figure 272: View of the Admin Sub-Site | 281 |
| Figure 273: Hobo source code on github.com | 287 |
| Figure 274: Hobo gems are also available on github.com | 288 |
| Figure 275: Installing Git for Mac OSX | 289 |
| Figure 276: Download the mysysgit installer for Windows | 289 |
| Figure 277: Running the Git Setup Wizard | 290 |
| Figure 278: Git setup options..... | 290 |
| Figure 279: Select the OpenSSH option | 291 |
| Figure 280: Select to option to run Git from the Windows command prompt..... | 291 |
| Figure 281: Select Windows style line endings..... | 292 |
| Figure 282: Running the PuTTY Key Generator install | 292 |
| Figure 283: Generate SSH key pairs for use with Git | 293 |
| Figure 284: The default file names generated by PuTTYGen | 294 |
| Figure 285: Locating your USERPROFILE setting | 295 |
| Figure 286: View of "no ssh public key found" error..... | 295 |
| Figure 287: Naming your SSH key pairs..... | 296 |
| Figure 288: The original Heroku beta invitation | 297 |
| Figure 289: Using the free "Blossom" database hosting option on Heroku.com | 298 |
| Figure 290: Sign Up for a Heroku account..... | 299 |
| Figure 291: Heroku notification that "Confirmation email sent" | 300 |
| Figure 292: Locating your "Invitation to Heroku" email..... | 300 |
| Figure 293: The "Welcome to Heroku" signup page..... | 301 |
| Figure 294: The "Account Created" message at Heroku.com | 301 |
| Figure 295: Installing the Heroku Ruby gem | 302 |
| Figure 296: Console output from the "heroku create" command | 303 |
| Figure 297: Using heroku git push..... | 304 |
| Figure 298: Telling Heroku where to find your application's gems | 304 |
| Figure 299: Adding your ".gems" config file to your git repository | 305 |
| Figure 300: Migrating your database schema to Heroku.com..... | 306 |
| Figure 301: Testing your Heroku app..... | 306 |
| Figure 302: Running the "Four Table" app on Heroku.com..... | 307 |
| Figure 303: Installing the Taps gem to upload data to Heroku.com..... | 307 |
| Figure 304: Using "heroku db:push" to push data to your app on Heroku.com | 308 |
| Figure 305: The "Four Table" app on Heroku.com with data | 308 |
| Figure 306: Add a recipe on Heroku.com..... | 309 |
| Figure 307: Pull changed data from Heroku.com to your local app..... | 309 |

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 Lapides

Jeff Lapides 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 and 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 as early as 1996.

CONTRIBUTORS

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, search, reporting and data warehousing solutions for both structured and non structured data and implementing the latest in open source technologies. 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 and Tandberg LTD.

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 with 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 last year.

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.

Bryan Larsen

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

PREFACE

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 it would take much more focus and energy than we anticipated to complete this 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. OK, so we bit off more than we could chew, at least in the timeframe we envisioned. So 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 tentative moniker “Hobo Under the Hood.” Jeff liked the Acronym this created, “HuH?” . It really represents the awe (with initial confusion) we sometimes feel about the Ruby “spells” within Hobo and Rails that appear to be magical to the newly initiated, which includes us.

A brief history

The search for a new web development framework began with my frustration with the 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 move to 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.

This was particularly frustrating to me as I had experience with the “agile” tools of the 1980's and 1990's, which included Revelation and PowerBuilder, client-server technologies that didn't manage to survive into the Internet age. With Revelation we could build an application prototype that included complex business logic while sitting 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", and tried it out.

My first simple application worked, but I have to admit 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 basic things I had been used to for over 20 year. Among other things, I was 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 on one of the major Oracle sites about interesting add-ons to Rails, which led to a post by the renowned Ruby evangelist, Peter Cooper, in January of 2007. 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 *got it*. It was Tom Locke.

Following an open source project was something totally new to me. We I owned my own software business for a dozen years I used proprietary tools that I paid licenses for. I couldn't see the source code. Oracle and Microsoft weren't giving me the code to their database servers, applications servers, or WYSIWYG design tools. I paid support and expected THEM to fix the problems we invariably discovered 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 of our products were shrink-wrapped add-ons for other developers. This gave us clout for recommending priorities for new development and the ability to get the president on the phone if one of my very high profile customers was having an issue.

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

It was a welcome and pleasant surprise to find supportive, generous, and incredibly talented people willing to help. Even Tom Locke would answer my questions, patiently. Later I was lucky enough to spend time with Tom in person on a number of occasions, which increased 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 was Chief Systems Architect of the Barquin team, not one of our senior developers. So 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, and additional Barquin team members became interested in learning, it became more and more urgent to have a solid resource for training not only developers, but also our 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 development efforts underway using Hobo that will put in production this year. The new Leadership and Management Dashboard (LMD) led by Joe Barbano, and the REEport (Research, Education and Economics Reporting) project lifecycle reporting system under the direction of John Mingee. 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 “skunkworks” 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 built-in e a 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*

CHAPTER 1 – INTRODUCTION

What is Hobo?

By Tom Locke

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

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

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

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

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

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

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

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

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

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

Background

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

One fellow managed to voice his concerns much more loudly than anyone else, by showing a better way. In 2004 David Heinemeier Hansson released a different kind of framework for building web apps, using a then little-known language called Ruby. A video was released in which Hansson created a working database-driven Weblog application from scratch in less than 15 minutes. That video was impressive enough to rapidly circulate the globe, and before anyone really even knew what it was, the Ruby on Rails framework was famous.

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

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

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

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

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

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

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

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

I must say I find it easy to agree with this perspective, and many projects did seem, in hindsight, to have been chasing a mirage. But it's also a hugely dissatisfying position. Surely we don't have to resign ourselves to re-inventing the wheel forever? So while the incredibly talented team behind Rails 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, and the Python language has Django, a web framework with some similar features.

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

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

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

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

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

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

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

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

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

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

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

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

The Future

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

Bigger library

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

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

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

Performance improvements

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

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

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

Having said all that, it's always nice to give people more performance out-of-the-box and postpone the day that they have to resort to app-specific efforts. Just as Rails has focused a lot on performance in the last couple of years, this is definitely an area that we will focus on in the future.

Less magic

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

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

Even higher level

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

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

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

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

CHAPTER 2 – INSTALLATION

Introductory Concepts and Comments

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

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

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

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

Installing Ruby, Rails, Hobo

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

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

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

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

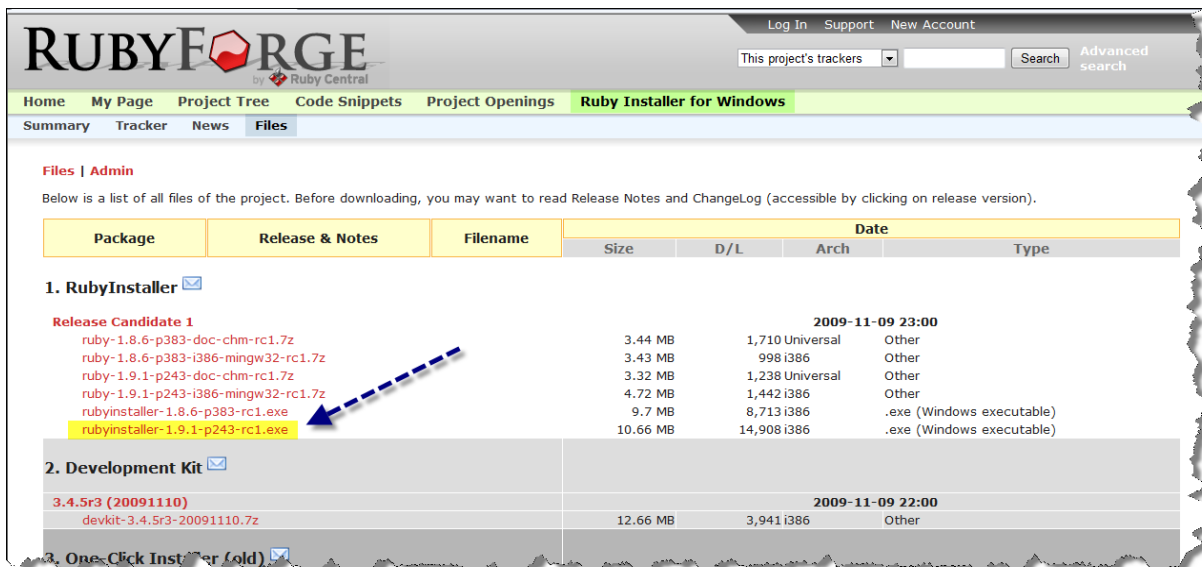
<http://hivelogic.com/articles/2008/02/ruby-rails-leopard>

For Linux aficionados:

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

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

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



The screenshot shows the RubyForge website interface. The top navigation bar includes links for Log In, Support, and New Account. Below the navigation bar, there are tabs for Home, My Page, Project Tree, Code Snippets, Project Openings, and Ruby Installer for Windows. The main content area displays a list of files for the Ruby Installer for Windows. The table has columns for Package, Release & Notes, Filename, Size, D/L, Arch, and Date. The first section is titled '1. RubyInstaller' and lists several release candidates. The file 'rubyinstaller-1.9.1-p243-rc1.exe' is highlighted with a blue arrow. The second section is titled '2. Development Kit' and lists a development kit file. The third section is titled '3. One-Click Installer (old)' and lists an old installer file.

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

Figure 1: Download Site for Ruby

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

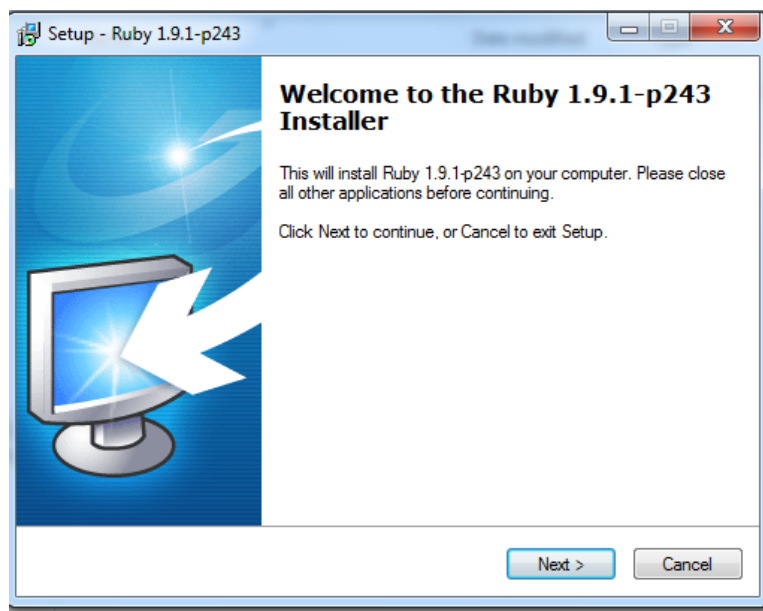


Figure 2: Installing Ruby

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

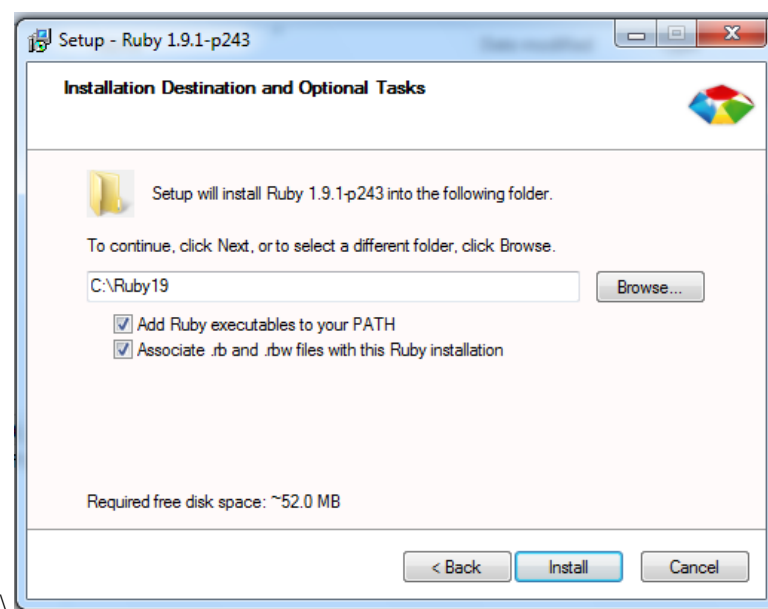


Figure 3: Ruby Installation Options

The installer will create a larger number of folders under the Ruby 19 folder, or an alternative folder if you specified on.

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



Figure 4: Setup Wizard Complete

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

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

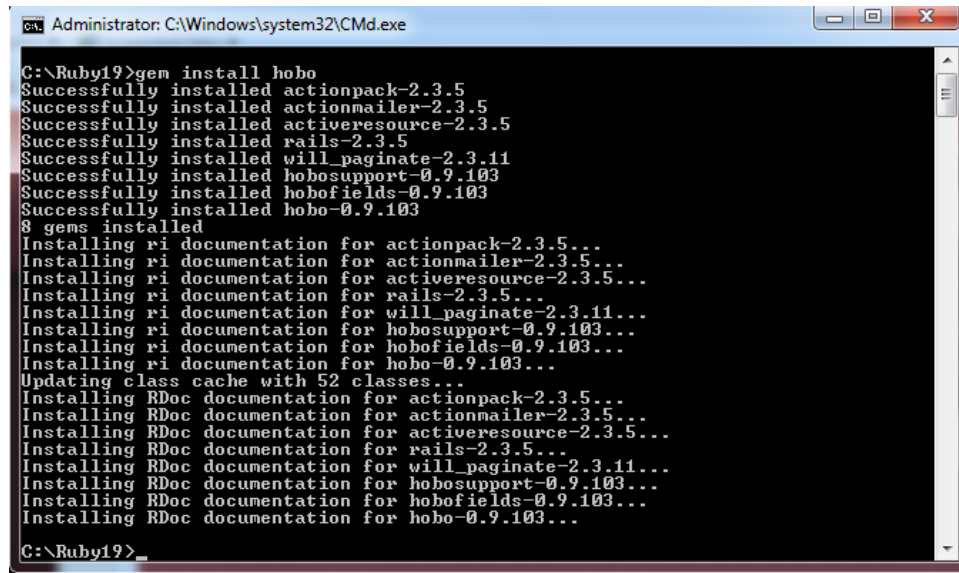
Also install “gemcutter” for future use:

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

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

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

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



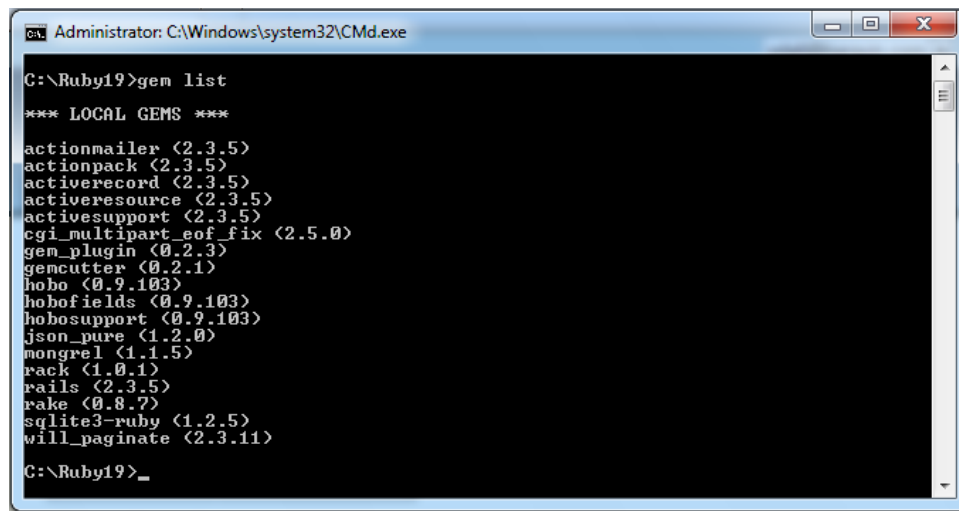
```
Administrator: C:\Windows\system32\Cmd.exe
C:\Ruby19>gem install hobo
Successfully installed actionpack-2.3.5
Successfully installed actionmailer-2.3.5
Successfully installed activerecord-2.3.5
Successfully installed rails-2.3.5
Successfully installed will_paginate-2.3.11
Successfully installed hobosupport-0.9.103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

Figure 5: Sample console output after installing the Hobo gem

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

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

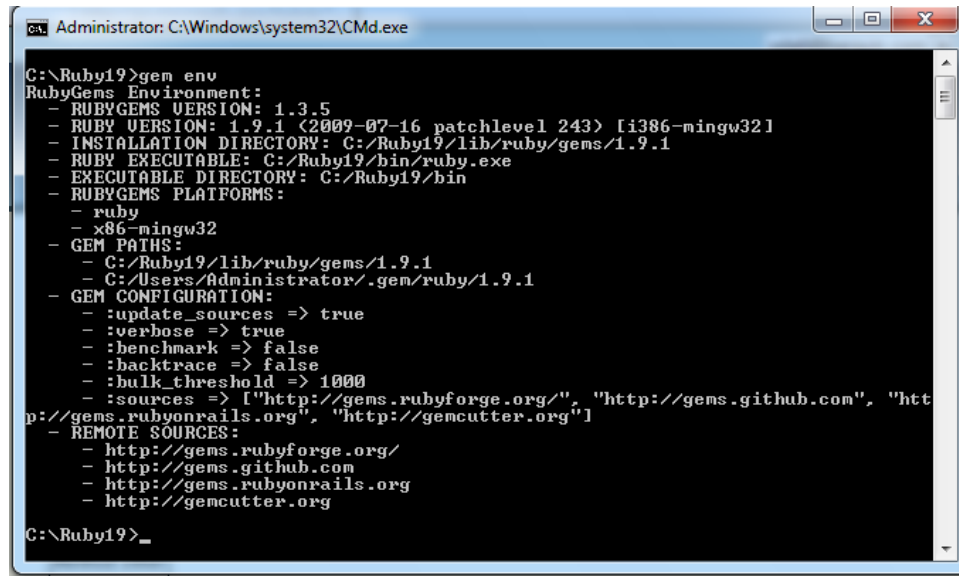
```
C:\ruby> gem list
```



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

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

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

```
Administrator: C:\Windows\system32\Cmd.exe
C:\Ruby19>gem env
RubyGems Environment:
- RUBYGEMS VERSION: 1.3.5
- RUBY VERSION: 1.9.1 (2009-07-16 patchlevel 243) [i386-mingw32]
- INSTALLATION DIRECTORY: C:/Ruby19/lib/ruby/gems/1.9.1
- RUBY EXECUTABLE: C:/Ruby19/bin/ruby.exe
- EXECUTABLE DIRECTORY: C:/Ruby19/bin
- RUBYGEMS PLATFORMS:
  - ruby
  - x86-mingw32
- GEM PATHS:
  - C:/Ruby19/lib/ruby/gems/1.9.1
  - C:/Users/Administrator/.gem/ruby/1.9.1
- GEM CONFIGURATION:
  - :update_sources => true
  - :verbose => true
  - :benchmark => false
  - :backtrace => false
  - :bulk_threshold => 1000
  - :sources => ["http://gems.rubyforge.org/", "http://gems.github.com", "http://gems.rubyonrails.org", "http://gemcutter.org"]
- REMOTE SOURCES:
  - http://gems.rubyforge.org/
  - http://gems.github.com
  - http://gems.rubyonrails.org
  - http://gemcutter.org
C:\Ruby19>
```

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

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

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

Using SQLite with Hobo

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

We used SQLite3 at the time of this writing.

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

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

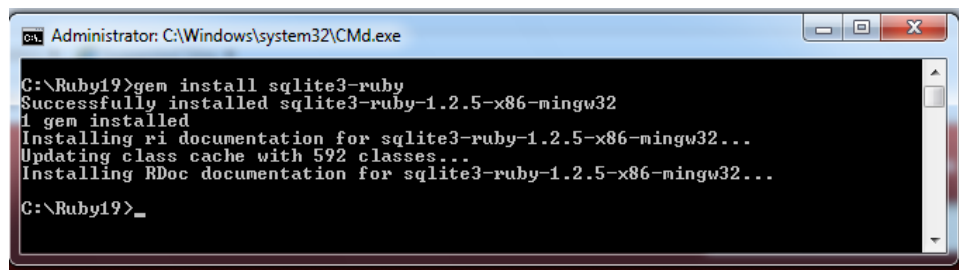


Figure 8: Sample console output from installing the sqlite3-ruby gem

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

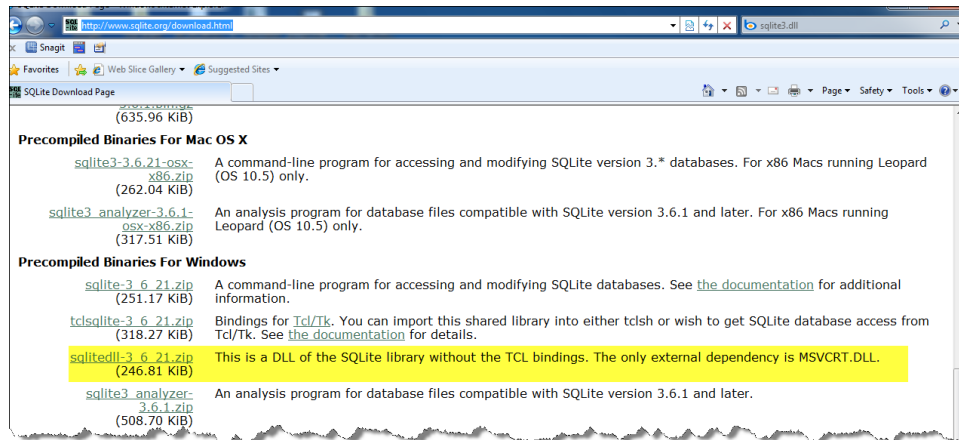


Figure 9: SQLite3 download website

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

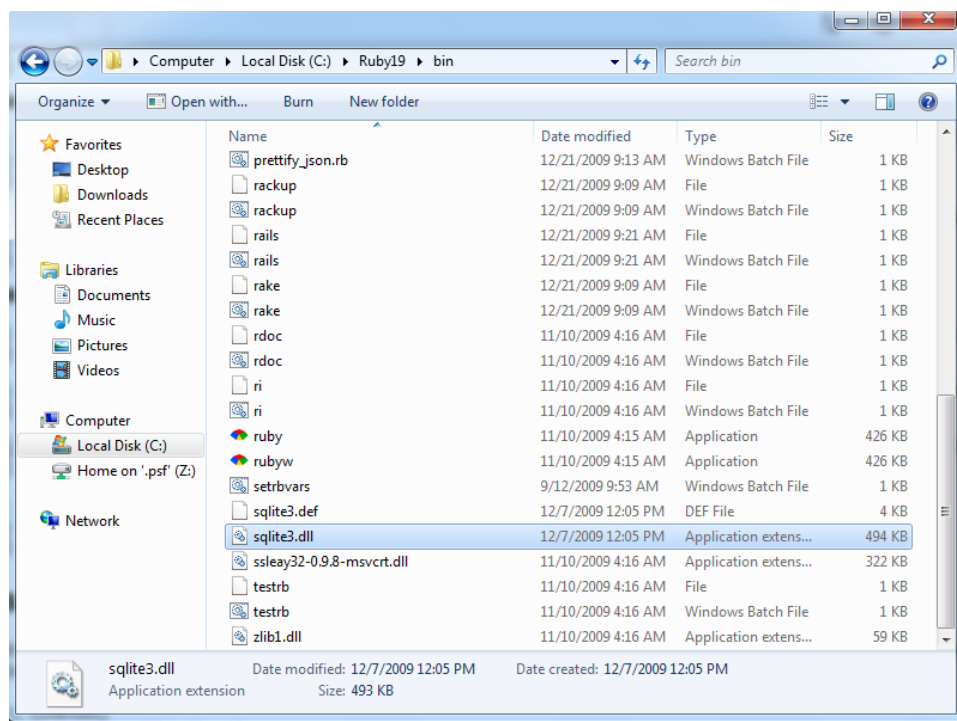


Figure 10: Target location for the SQLite3 DLL

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 the following detailed instructions are provided:

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

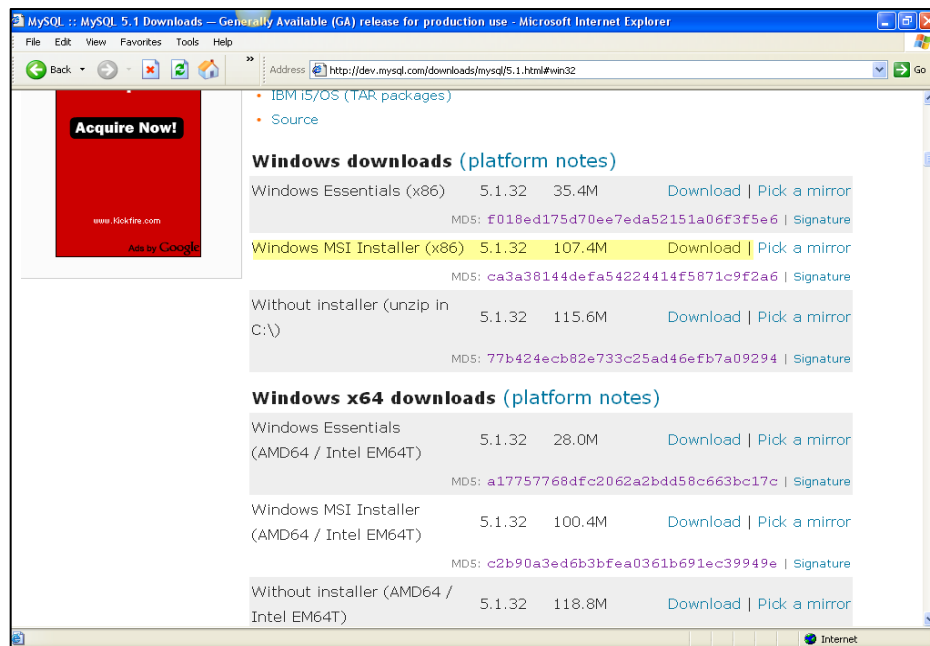


Figure 12: Download site for MySQL

Double-click on the MySQL MSI installation file:

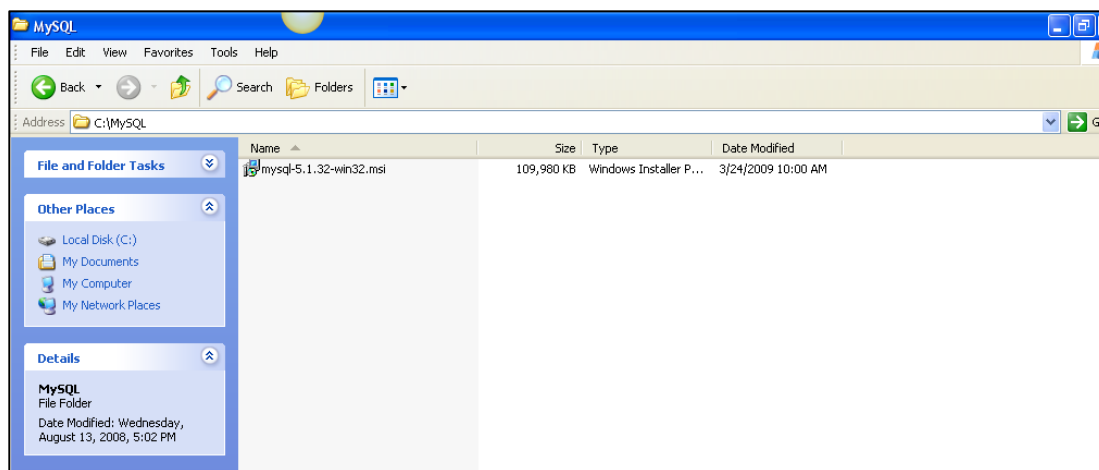


Figure 13: Using the .msi file to install MySQL on Windows

Choose the “Typical” option when prompted:

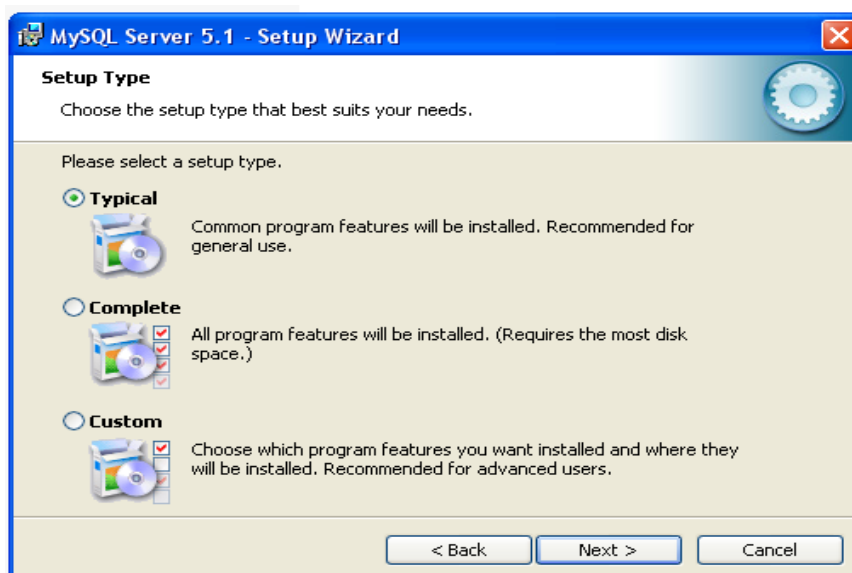


Figure 14: Choose the installation type

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

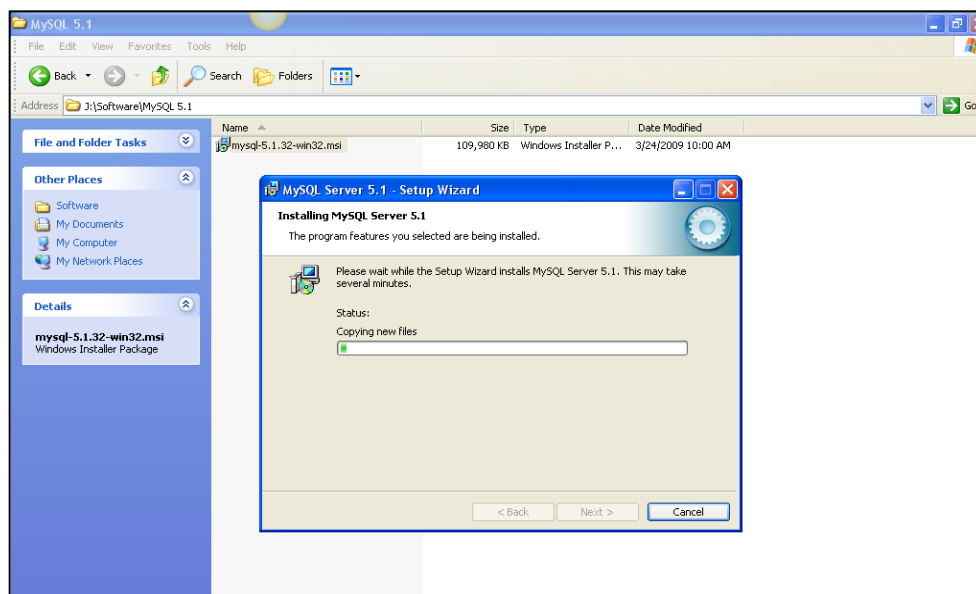


Figure 15: MySQL Server Setup Wizard

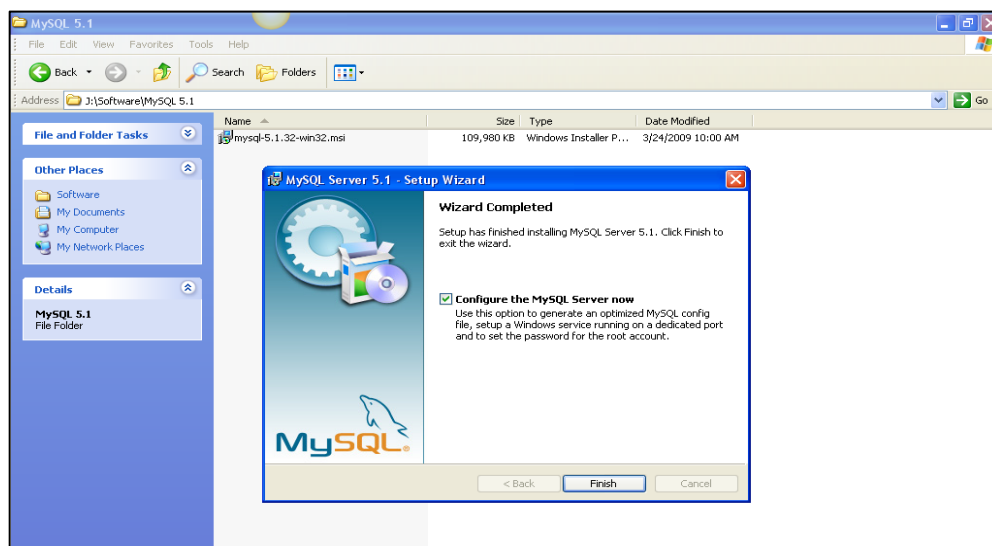


Figure 16: Configure MySQL Server

The next step is to configure the database instance:

We recommend choosing the “Standard Configuration” option.

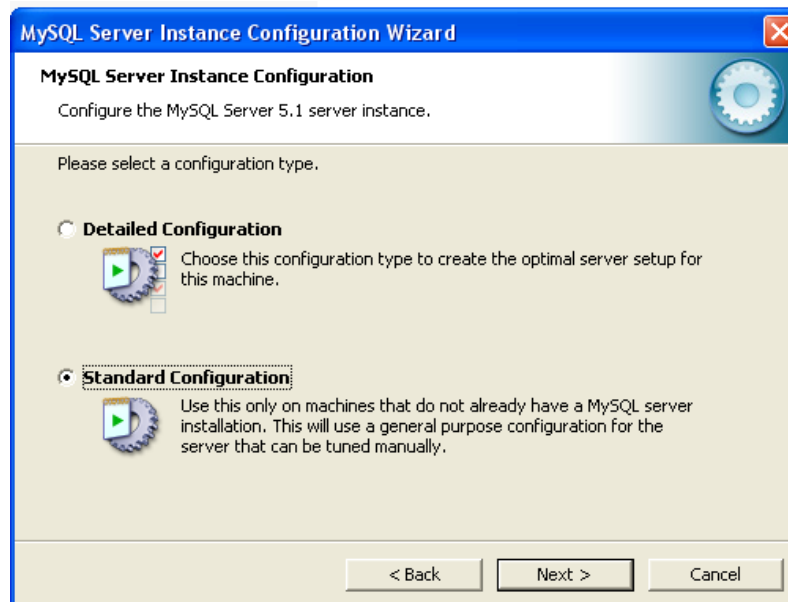


Figure 17: Choose Standard Configuration

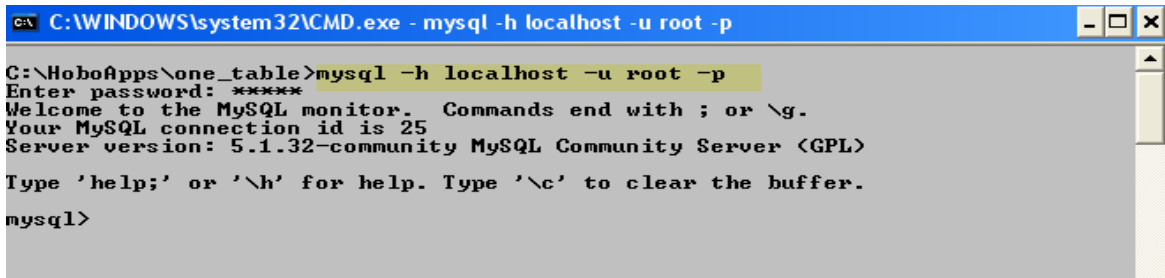
Select both “Install As Windows Service” and “Include Bin Directory in Windows PATH”:



Figure 18: Install as Windows Service

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

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



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

C:\HoboApps\one_table>mysql -h localhost -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25
Server version: 5.1.32-community MySQL Community Server (GPL)

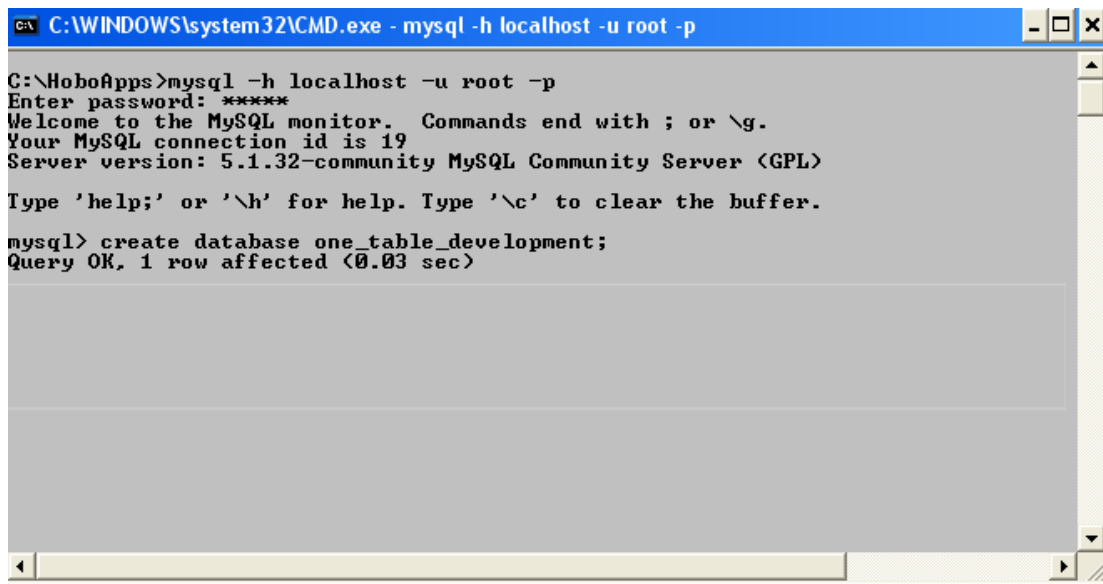
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Figure 19: Launch MySQL from the command prompt

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

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



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

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

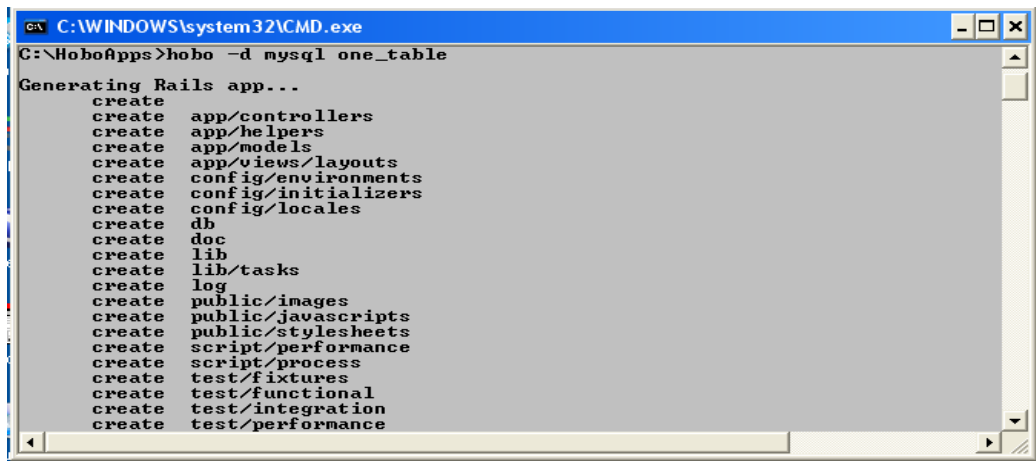
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

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

Figure 20: Create the database from the command line

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

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

```

C:\WINDOWS\system32\CMD.exe
C:\HoboApps>hobo -d mysql one_table

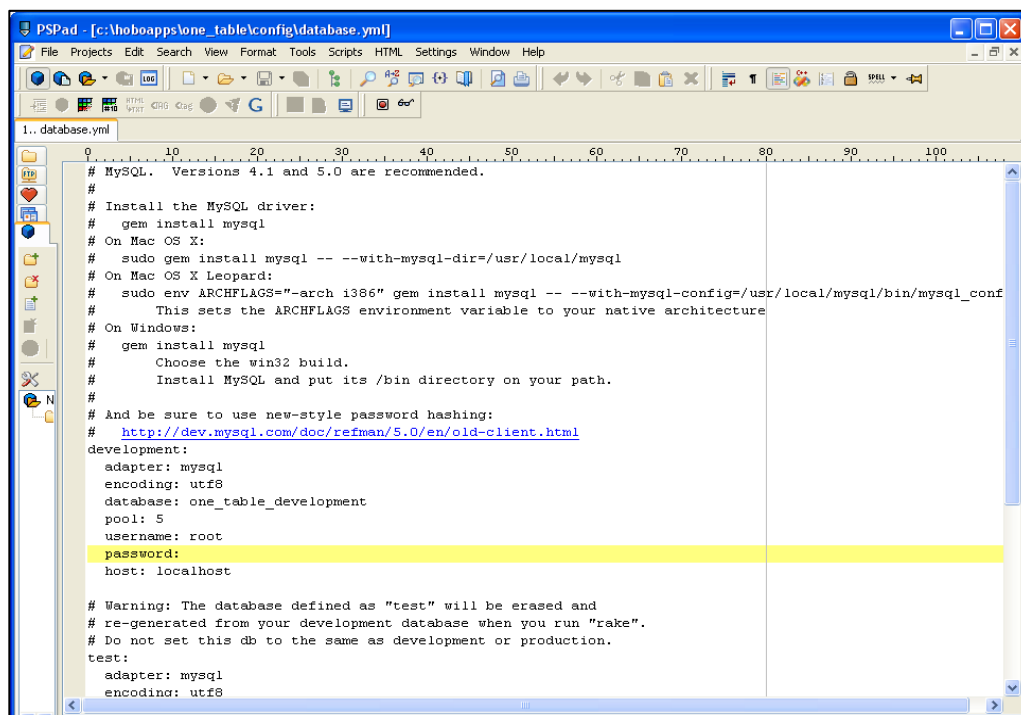
Generating Rails app...
create    app/controllers
create    app/helpers
create    app/models
create    app/views/layouts
create    config/environments
create    config/initializers
create    config/locales
create    db
create    doc
create    lib
create    lib/tasks
create    log
create    public/images
create    public/javascripts
create    public/stylesheets
create    script/performance
create    script/process
create    test/fixtures
create    test/functional
create    test/integration
create    test/performance

```

Figure 21: Console output from the Hobo command

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

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



```

PSPad - [c:\hoboapps\one_table\config\database.yml]
File Projects Edit Search View Format Tools Scripts HTML Settings Window Help

1.. database.yml
# MySQL. Versions 4.1 and 5.0 are recommended.
#
# Install the MySQL driver:
#   gem install mysql
# On Mac OS X:
#   sudo gem install mysql -- --with-mysql-dir=/usr/local/mysql
# On Mac OS X Leopard:
#   sudo env ARCHFLAGS="-arch i386" gem install mysql -- --with-mysql-config=/usr/local/mysql/bin/mysql_conf
#   This sets the ARCHFLAGS environment variable to your native architecture
# On Windows:
#   gem install mysql
#   Choose the win32 build.
#   Install MySQL and put its /bin directory on your path.
#
# And be sure to use new-style password hashing:
#   http://dev.mysql.com/doc/refman/5.0/en/old-client.html
development:
  adapter: mysql
  encoding: utf8
  database: one_table_development
  pool: 5
  username: root
  password:
  host: localhost

# 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: mysql
  encoding: utf8

```

Figure 22: The MySQL format for the database.yml configuration file

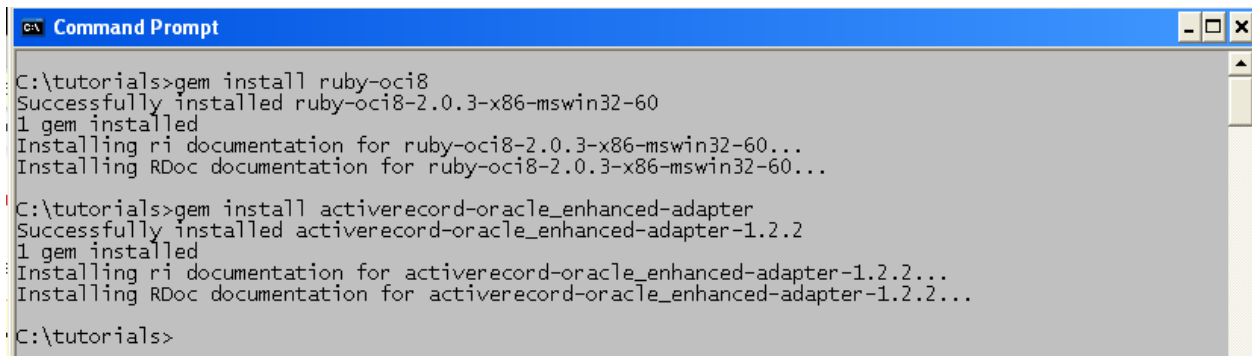
Using Oracle with Hobo

We will discuss the following two configuration options:

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

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

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



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

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

C:\tutorials>
```

Figure 23: 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:\tutorials> hobo two_table -d oracle
```

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

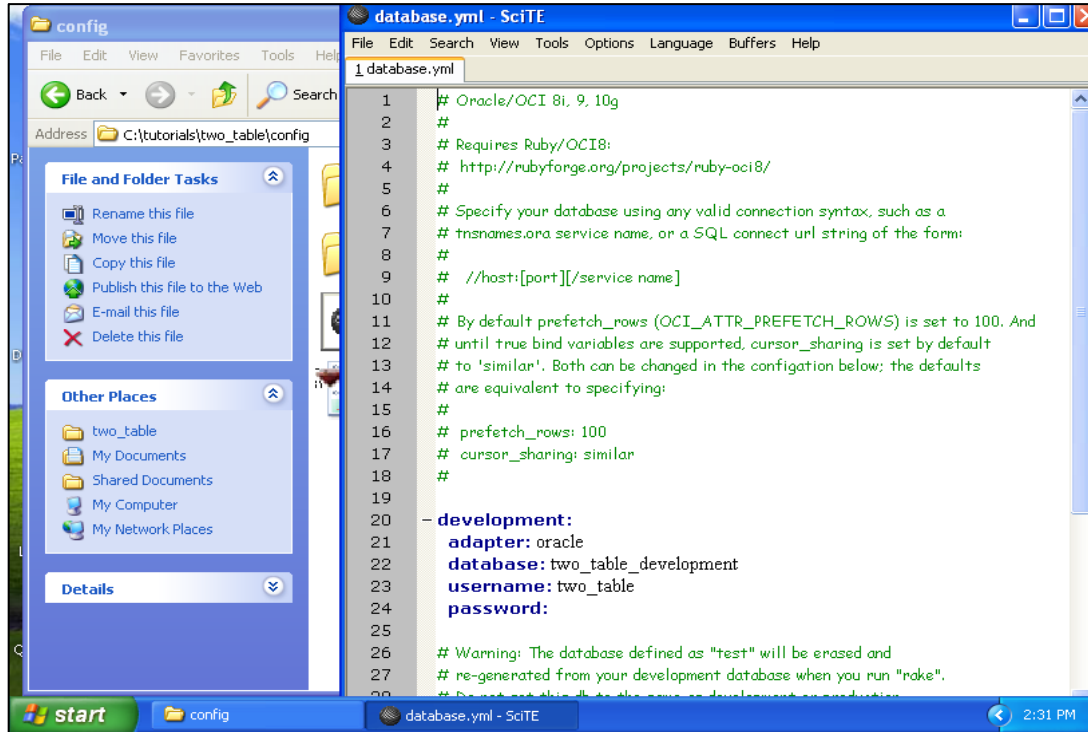


Figure 24: The generated database.yml file for Oracle

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

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

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

Option 2

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

<http://www.oracle.com/technology/software/products/database/xs/htdocs/102xeinsoft.html>

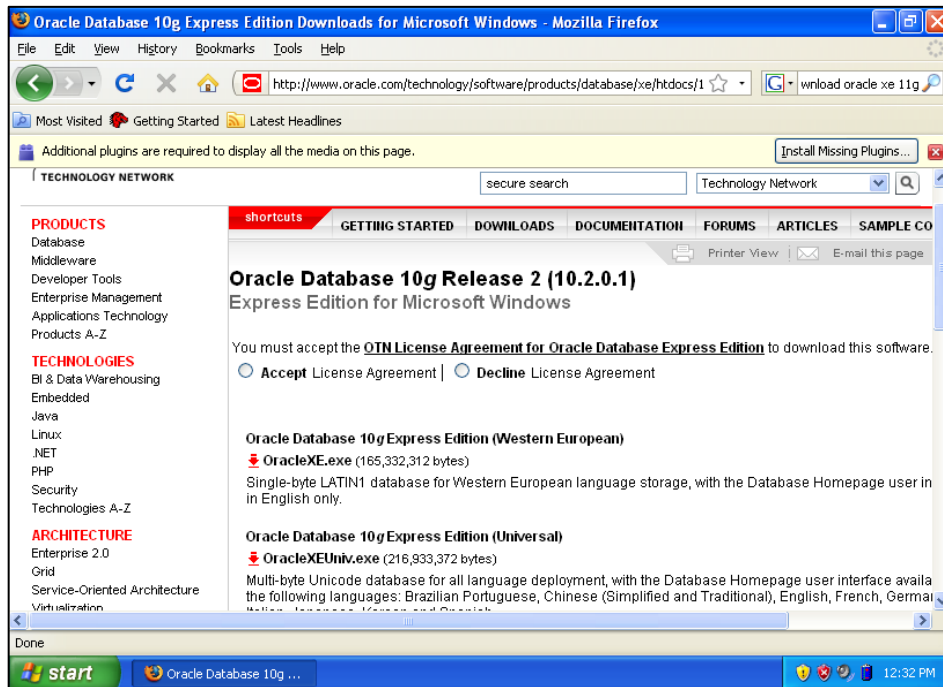


Figure 25: Oracle database install download site

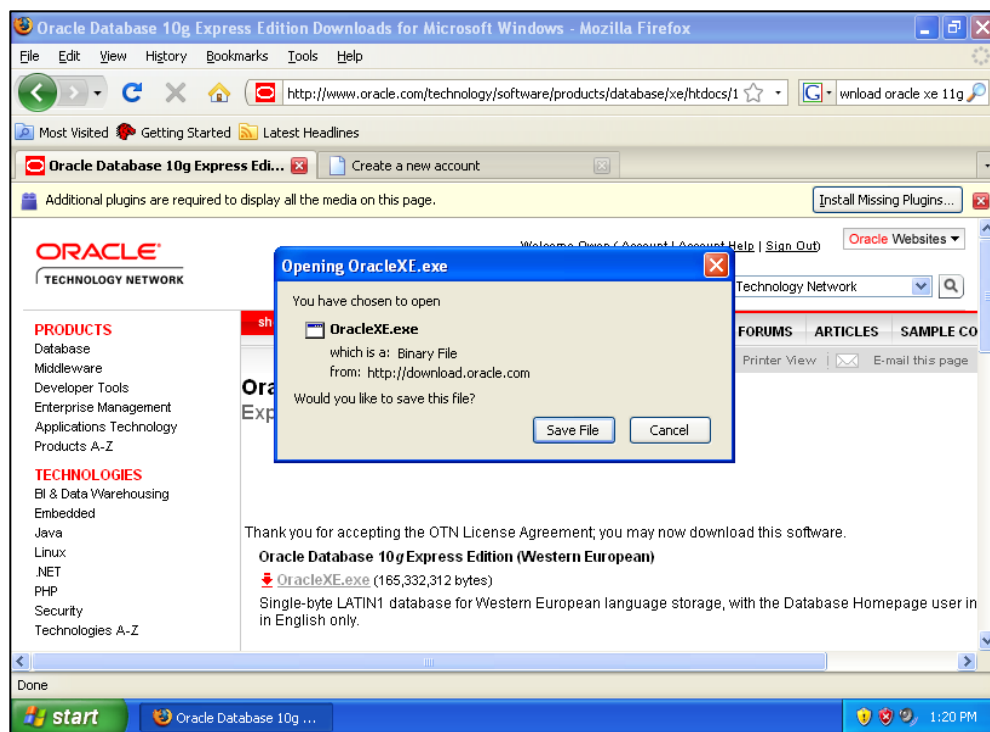


Figure 26: Running the Oracle XE installation

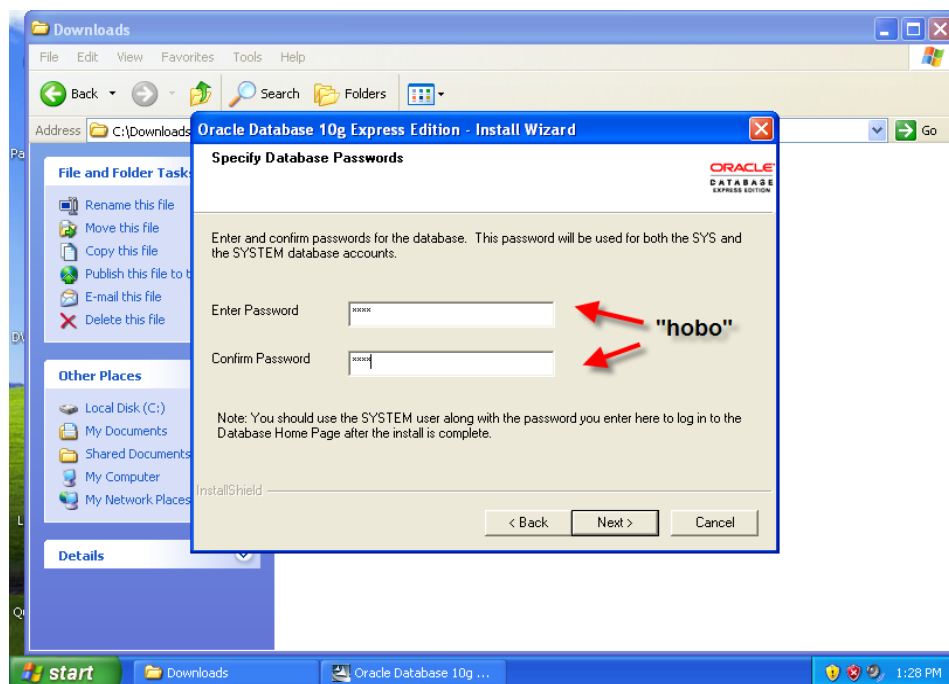


Figure 27: Specifying the database passwords

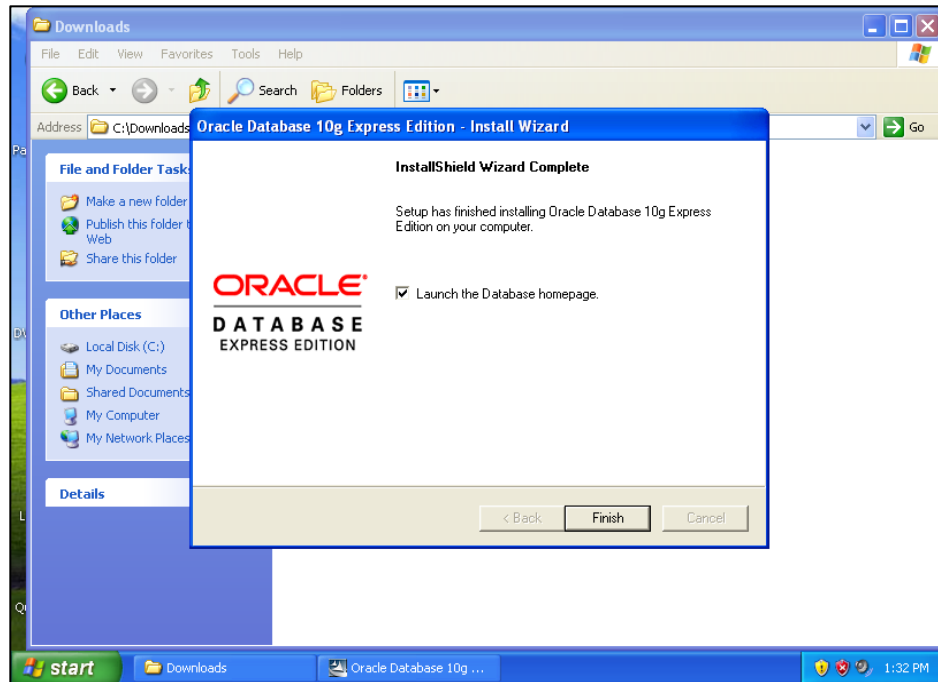


Figure 28: Launch the Database home page

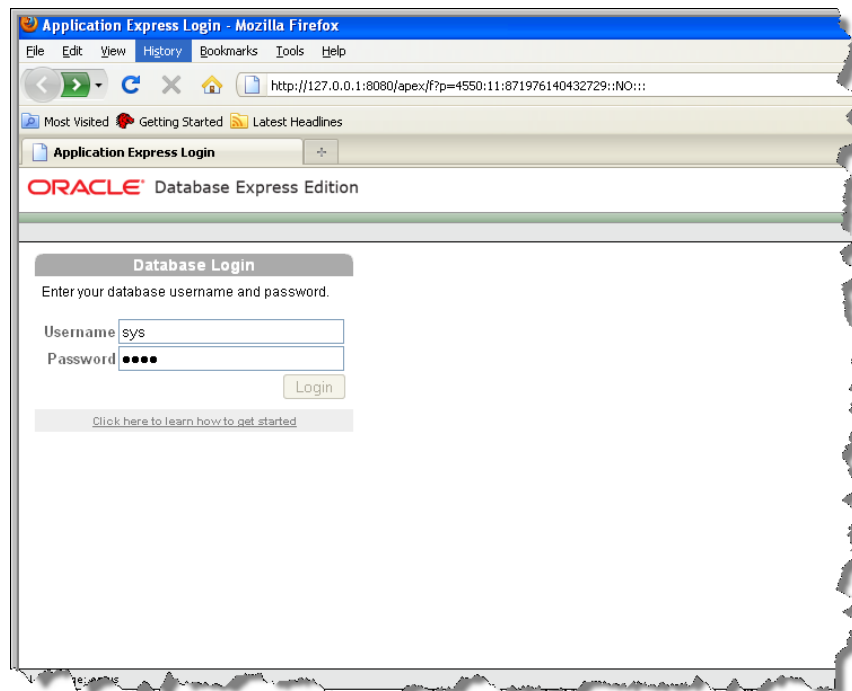


Figure 29: Log is as SYS to configure your database

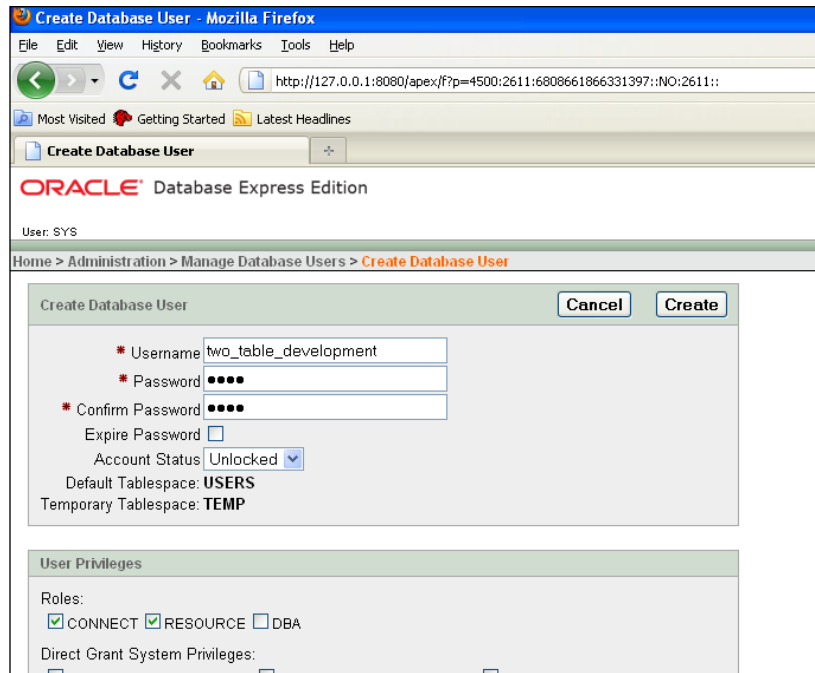


Figure 30: Creating a schema/user to use with Hobo

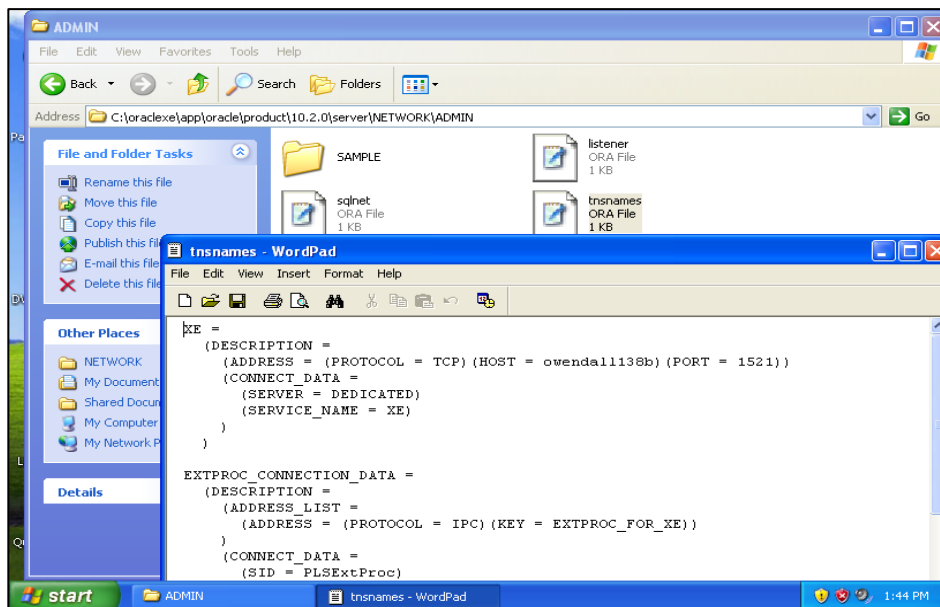


Figure 31: The tnsnames.ora file created during installation

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

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

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

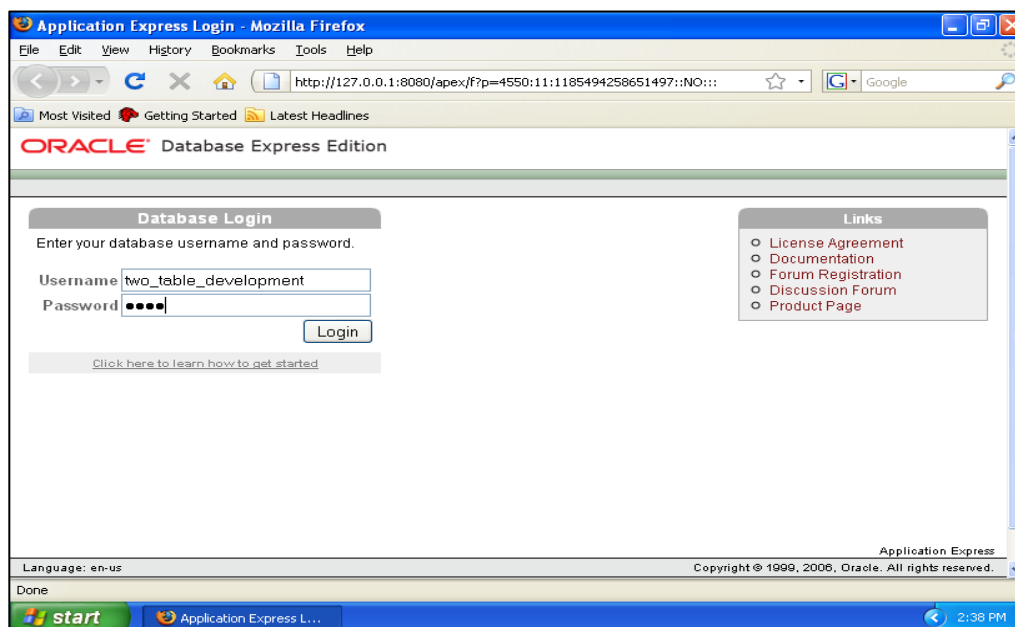


Figure 32: Log into Oracle to view the created table

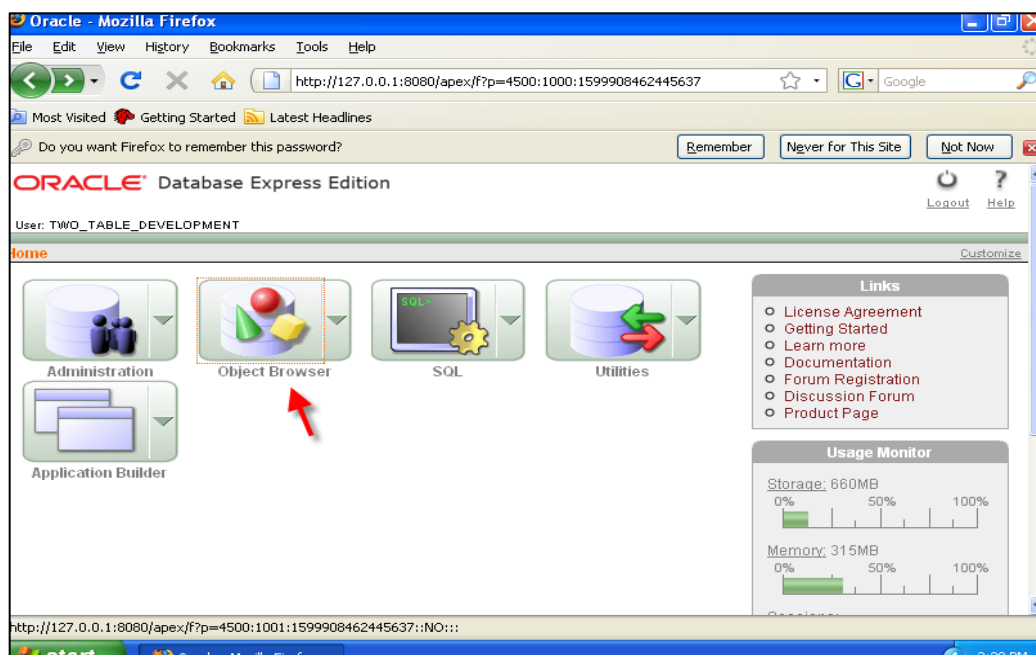


Figure 33: Access the Oracle Object Browser

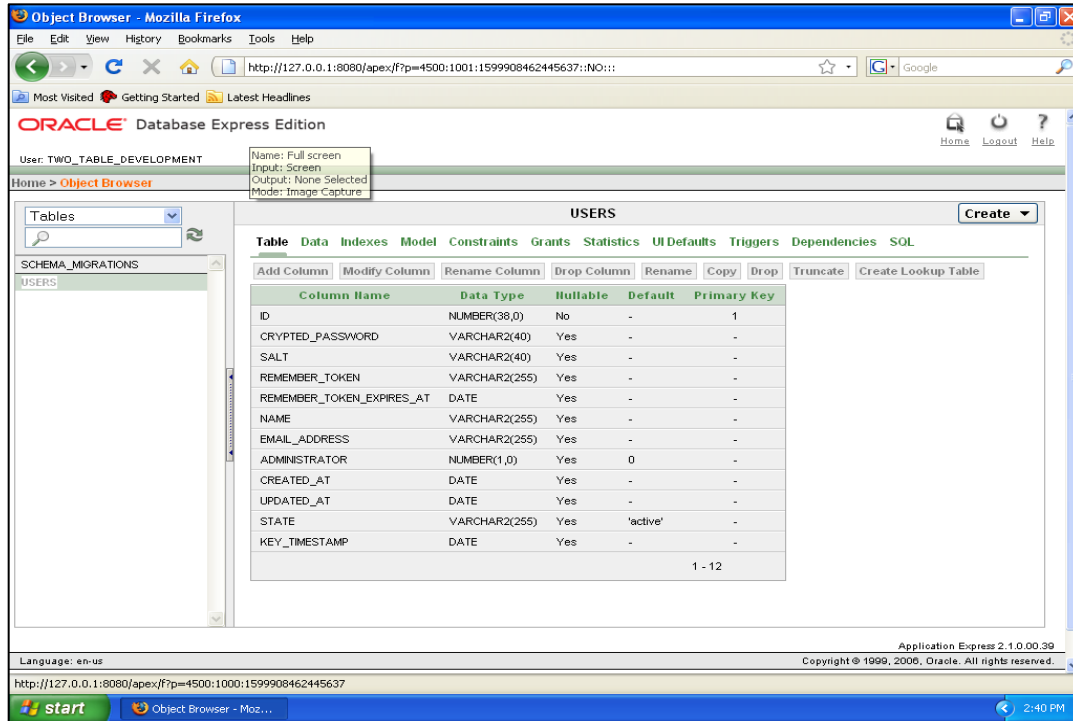


Figure 34: Review the User table from within Oracle

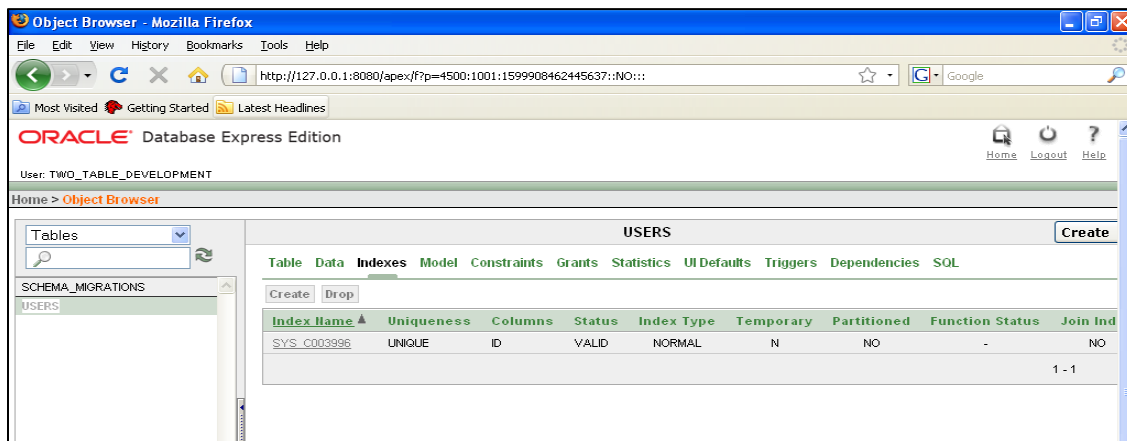


Figure 35: Review the Indexes view for Users

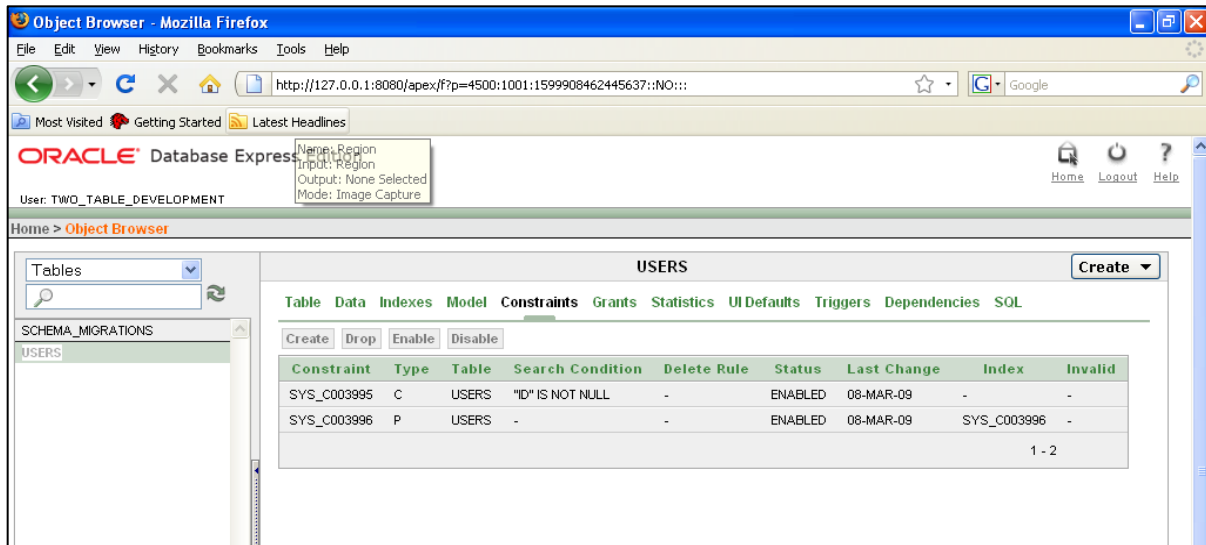


Figure 36: Review the Constraints view for User

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 development framework. Hobo is written in Ruby. One of Hobo’s secret weapons is the powerful and succinct DRYML (**Don’t Repeat Yourself Markup Language**).
- The **SQLite3** database and related Ruby gem (**sqlite3-ruby**) that makes prototyping quick and painless. SQLite3 is a robust and widely used database engine for embedded systems and is the repository used by the Firefox browser.
- The Webrick **HTTP/web** server written in Ruby. This is a basic server for desktop development work. Another popular one is Mongrel (Ruby 1.8 only) and Thin. For production implementation they’re a variety of options, including the popular Phusion Passenger (aka mod_rails), which can be used in conjunction with the Apache HTTP server. <http://www.modrails.com/>. With JRuby you can run on JBoss, Glassfish, etc.
- A variety of add-on gems (ruby modules or “libraries”) that each framework has included as “dependencies”. For example, **will_paginate** is used by Hobo for pagination of lists on web pages.

Now you are ready for the 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

If you explain a magic trick before it is performed, you risk spoiling the enjoyment. 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 right now to allow 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.

Tutorial Application: `my-first-app`

Topics

- Creating a Hobo application
- Learning the Hobo Directory structure
- Generating Hobo models and controllers with `hobo_model_resource`
- Generating Hobo models with `hobo_model`
- Generating Hobo controllers with `hobo_model_controller`
- Creating Migrations and Databases with `hobo_migration`
- Editing Models and propagating the changes with `hobo_migration`

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 (Mongrel 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 my-first-app
```

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.

Take a moment to confirm that no error messages were displayed. At this point, the main thing that can cause an error is an incomplete installation. So if you have an error, refer to Chapter 2's installation instructions and make sure you have completed all of them correctly.

Finish off this step by moving to your application's directory:

```
my-first-app>
```

Using Windows Explorer you should see a folder structure similar to the following:

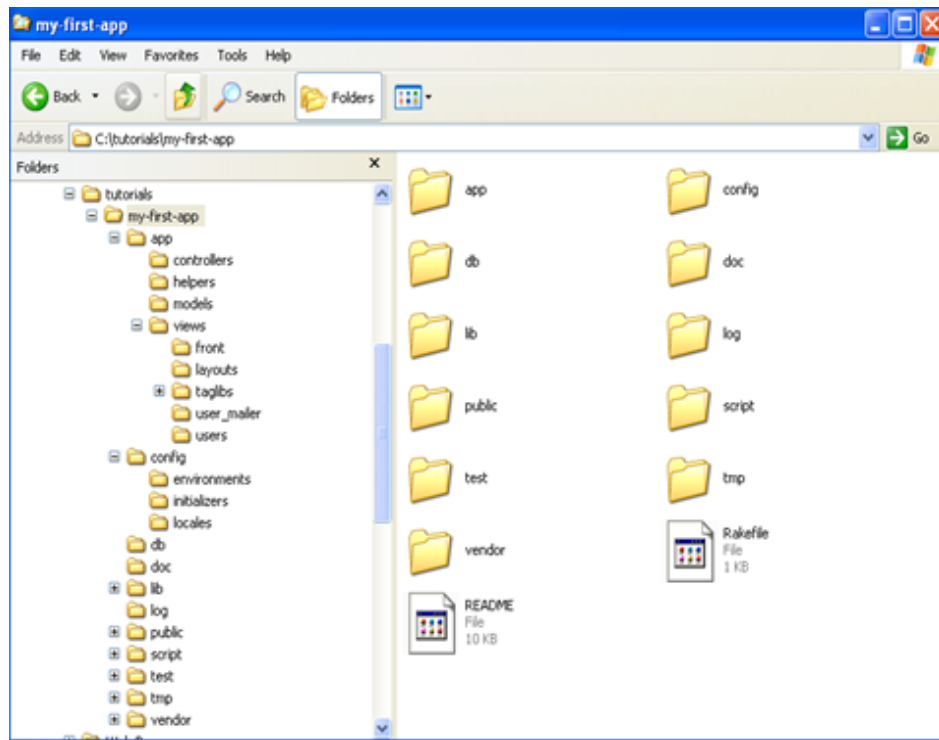


Figure 37: Hobo application folder structure

4. **Run a Hobo migration.** This step, the “migration”, creates the necessary database entities for the application. Execute your first hobo migration by issuing the following command.

```
my-first-app> ruby script/generate hobo_migration
```

After executing this command you will again see a log of what Hobo is doing in the shell. Again make sure there are no errors in the execution. If there is an error, the most common mistake will be a misspelling in the command above.

You will get this message when issuing the below command:

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

Answer with an 'm' (don't use the quotation marks) to generate a migration now.

You will be prompted with the following message:

```
Migration filename:
(you can type spaces instead of '_' -- every little helps)
Filename [hobo_migration_1]:
```

Respond by hitting the return key to accept the proposed filename. Hobo will log what it is doing and you should then be returned to your command prompt.

```
my-first-app>
```

Note for Rails developers: Any time you run a Rails generator that creates a model, it will also create a migration file. You execute the migration file by issuing the following command: **rake db:migrate**. With Hobo, the rake command is unnecessary to execute migrations because **hobo_migration** takes care of it when you select the 'm' option from above.

Hobo does not create the migration until you issue the **hobo_migration** command and will continue on to execute it if you choose the 'm' option as you did above.

The primary thing that **hobo_migration** does is to look at your models, and then both build and execute a migration as a result of this single command.

When you started my-first-app by issuing the Hobo command, Hobo generated a *User* model automatically for you. So the **hobo_migration** generator will create a migration file from the *User* model and a users table. Let's take a look now:

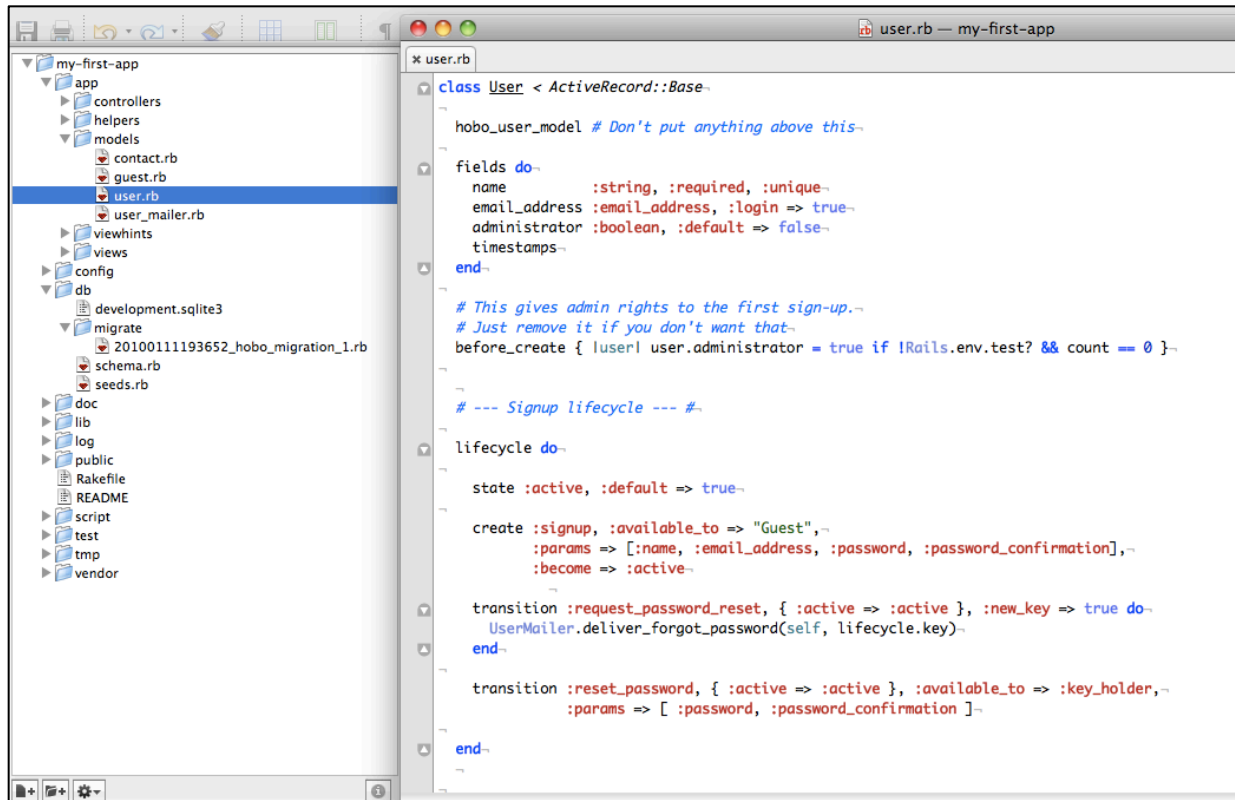


Figure 38: The default User model created by Hobo

You will learn in subsequent steps that when you edit a model the **hobo_migration** generator detects this change and creates a new migration file that is used to alter the database.

6. **Examine the directory changes after the first migration.** In the following figure, you can see that the db directory is now populated. The file, `development.sqlite3`, is the database file. The `hobo_migration_1.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.

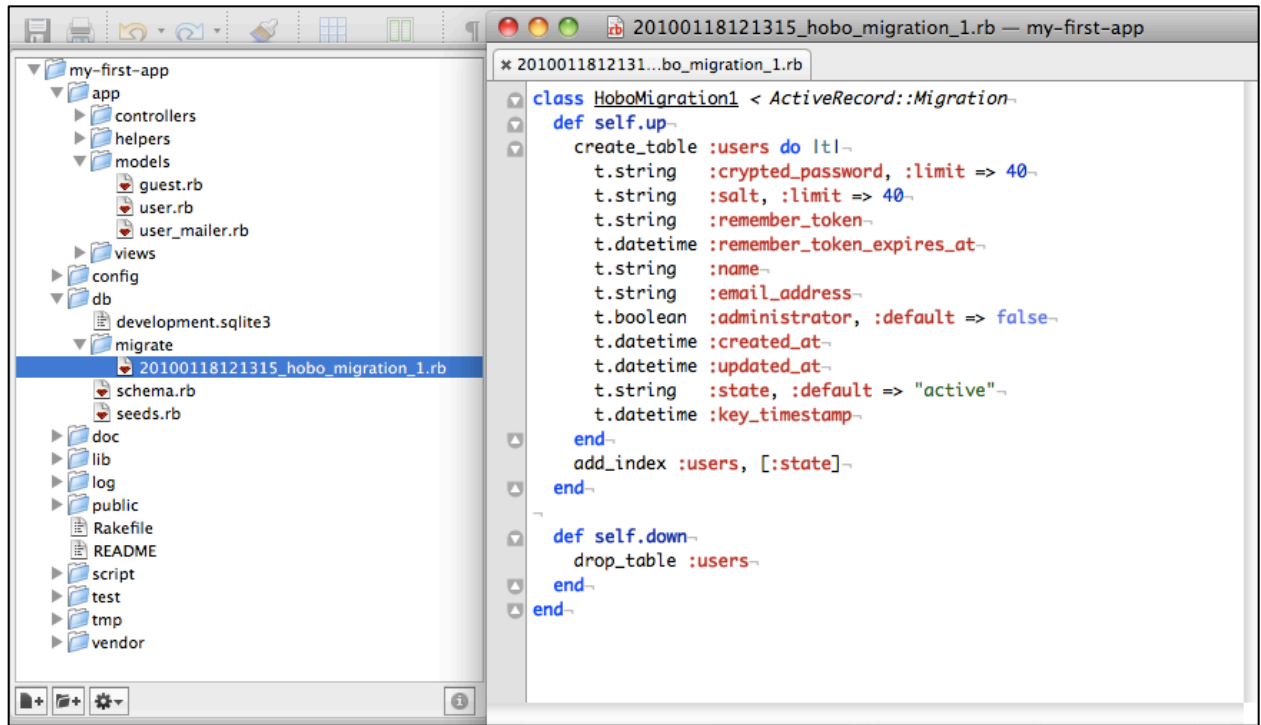


Figure 39: Contents of the first Hobo migration file

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

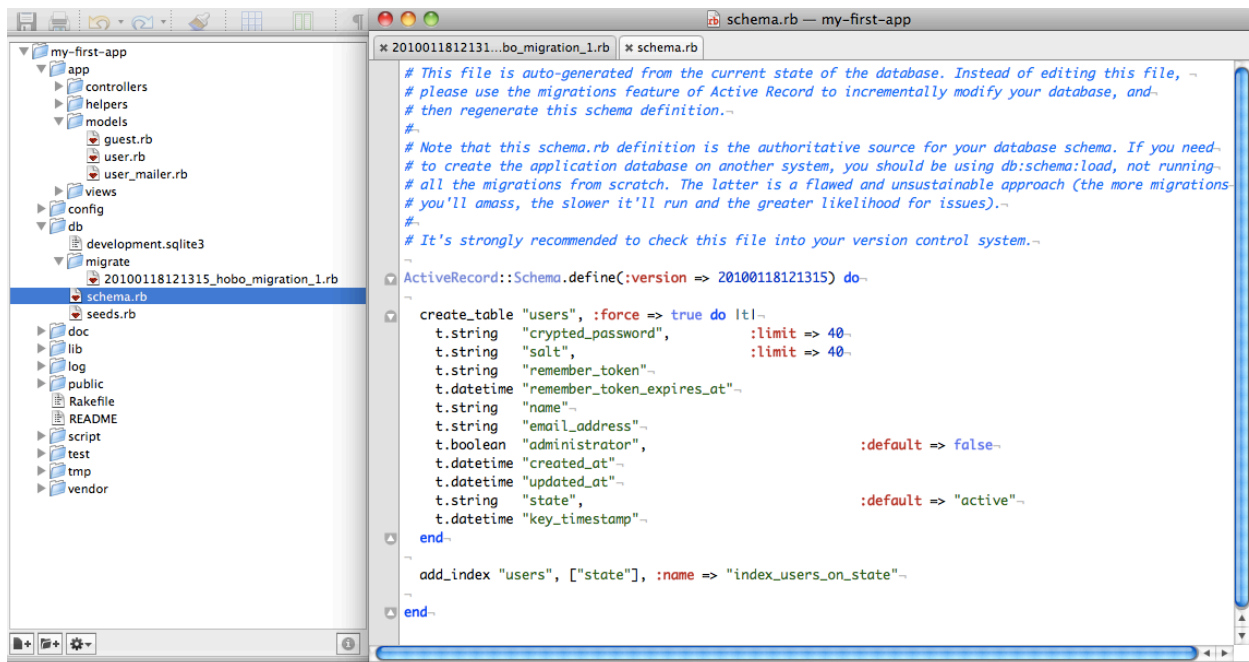


Figure 40: Contents of the "schema.rb" file after the first migration

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.

7. **Test out your application.** Create a second shell window (or tab).

You are now going to start a local web server called Mongrel 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 Mongrel web server by issuing the following command at your command prompt.

```
my-first-app> ruby script/server
```

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 Mongrel server by typing *control-c* and restart it the same way you started it above, but do not terminate the server.

8. **Open your application in a web browser.** Type the following URL into your browser URL window and you should see the following result. Note the User login in the upper right and the search capability.

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

Note the User login in the upper right and the search capability.

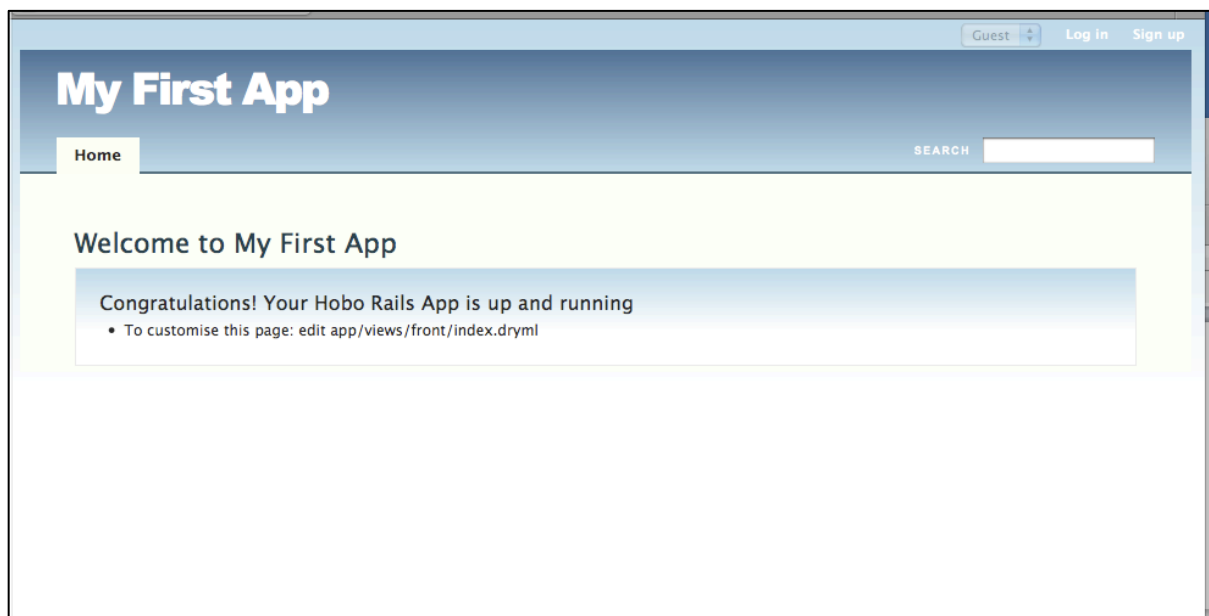


Figure 41: Home page for "My First App"

9. **Create some accounts.** Click the *Sign up* link above and create an account. The Hobo permissions system won't let you do anything until you do this.

Note: The first person to register is assigned the admin 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.

Create another account. We will call this and all other accounts you create *user* accounts. It is a good idea to have at least one admin account and one user account as you go through these tutorials. That way you can exercise the permission system and other features you will be learning about.

Log out of the *user* account you just created and login to the *admin* account for now.

Note: You will use the admin email address and password to login, not the name.

Note that in the upper left corner of your web page, there is a drop down box that lets you automatically sign in to any of your accounts without going through the login page.

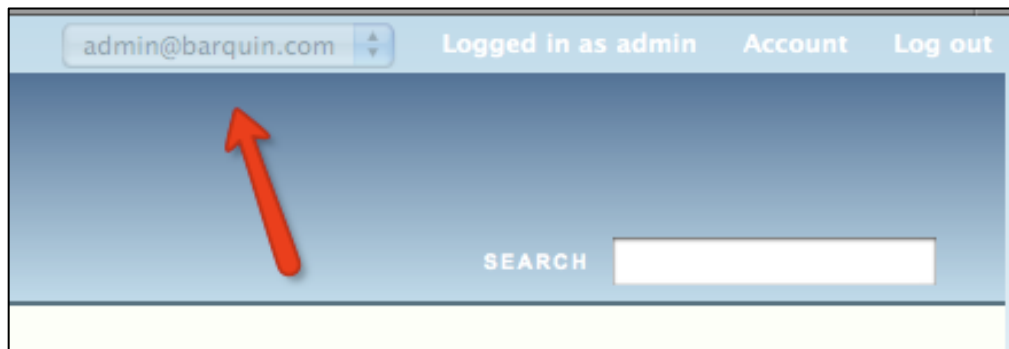


Figure 42: Drop down selector for the active user

Now realize that you have done nothing but run a couple of generators and you have a decent login capability.

10. **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.

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.

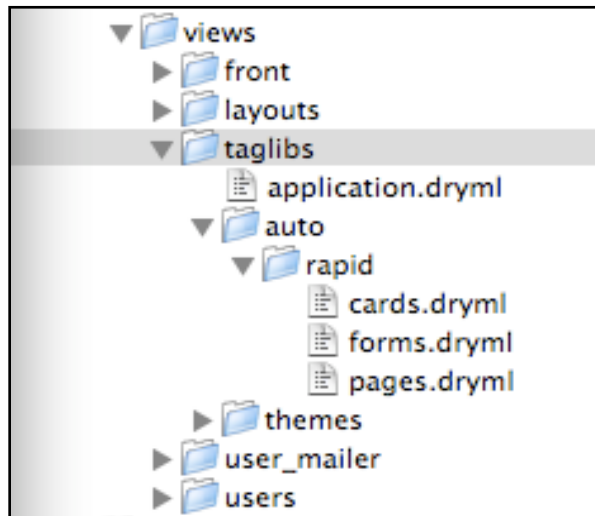


Figure 43: Location of the Rapid templates

Remember this. 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 `form.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 put the new tags in the `application.dryml` file. When you want them to be available only in a particular view template you put them in a template file under the 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 (**`hobo_model_resource`** or **`hobo_model`**) and running **`hobo_migration`**, that directories will be created named for your new models.

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

```
my-first-app> ruby script/generate hobo_model_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 `contacts_controller.rb` controller file, the `contact.rb` model file and the `views/contacts` template.

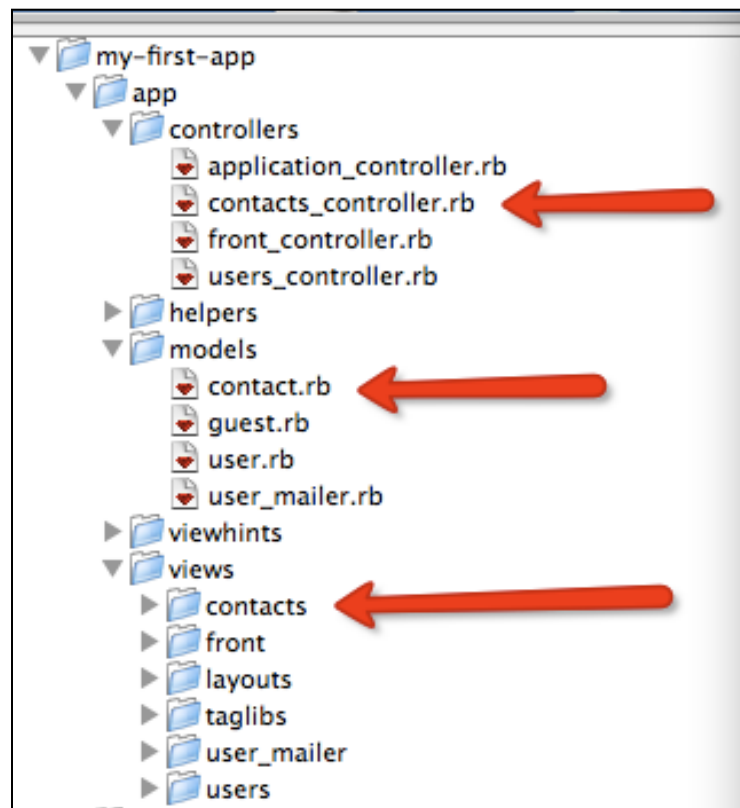


Figure 44: Folder location for Models and Views

12. **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.

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

```
end
```

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

When you run **hobo_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_migration** and observe what happens.

```
my-first-app> ruby script/generate hobo_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 views/db directory of your app.



Figure 45: 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
end
```

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

```
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.

`http://localhost:3000/`



Figure 46: Contacts tab on "My First App"

Now you have a new tab called “Contacts.”

13. **Create some contacts.** Now you should be able to create a new contact by clicking the ‘New Contact’ link in the Contacts tab. Display this link in the above screenshot. 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.

14. **Try out the search facility.** Type the name of one your contacts to exercise the search facility. [search only searches name and not other fields. Even for this field, partial word searches don’t work, need at least three characters for a search]
15. **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_migration** generator. Since we have already generated our model using **hobo_model_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:

```
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
  end
end
```



```
married :boolean
age      :integer
notes   :text
timestamps
end
```

Make sure you save your changes and run **hobo_migration**. Select the 'm' option and accept the default filename for the migration.

```
my-first-app> ruby script/generate hobo_migration
```

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

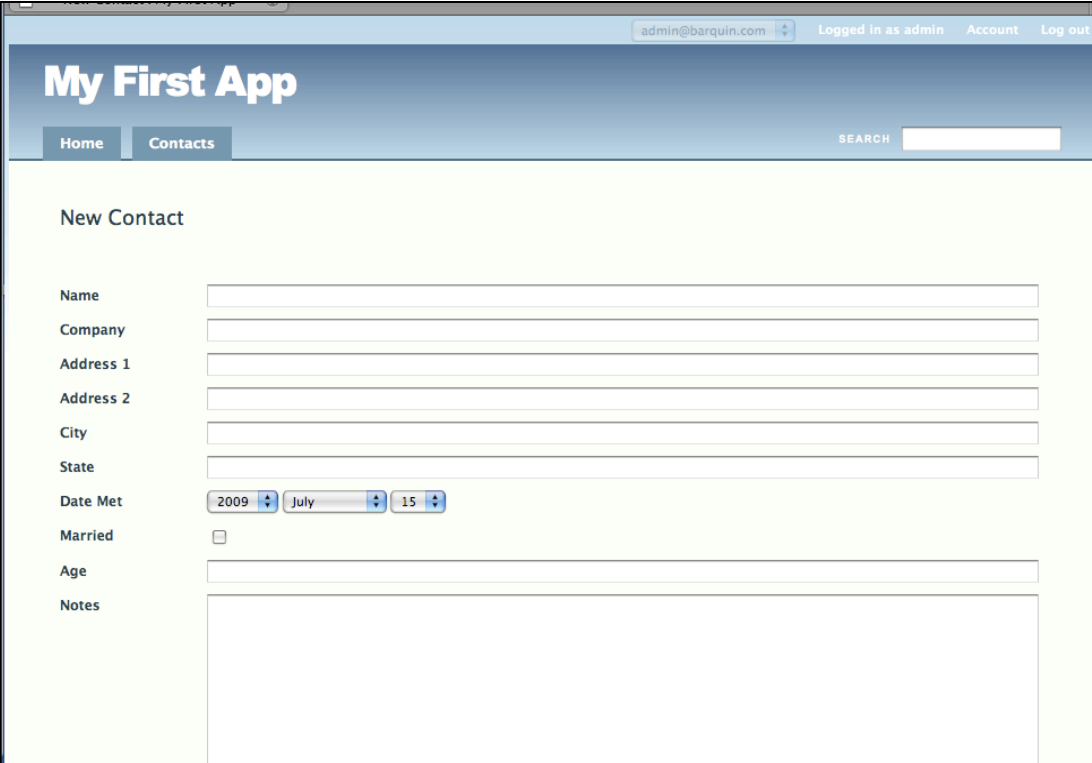
The screenshot shows a web browser window displaying the 'My First App' interface. The top navigation bar includes the app name, a 'Home' link, a 'Contacts' link, and a search bar. The 'Contacts' link is active. Below the navigation bar, the 'New Contact' form is displayed. The form contains several input fields: 'Name', 'Company', 'Address 1', 'Address 2', 'City', 'State', 'Date Met' (with a date picker set to 2009, July, 15), 'Married' (a checkbox), 'Age', and 'Notes' (a large text area). The form is styled with a light green background and white text labels.

Figure 47: 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.

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

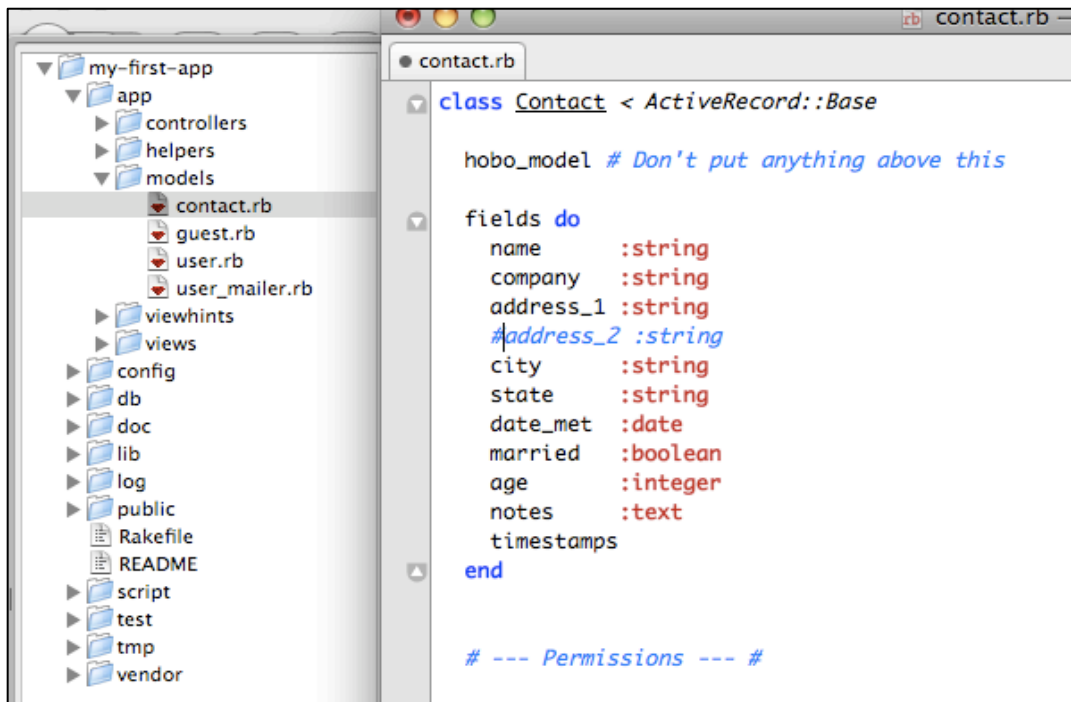


Figure 48: Remove field from contact model

```

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

end

```

Run **hobo_migration** again.

```
my-first-app> ruby script/generate hobo_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.

17. **Adding and removing database tables.** You can also use **hobo_migration** to remove a table. Simply delete the entire model file and run **hobo_migration**. As of Hobo version 1.0, only the table will be removed. You will have to manually remove the associated controller, helper and **viewhint** files, and the view template directory and files or you could create additional problems for yourself.
18. **Going back to earlier migrations.** Hobo does not provide this facility within **hobo_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 with View Hints

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.

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_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
    #gender    :string
    #married   :boolean
    married_now :Boolean
    age       :integer
    notes     :text
    timestamps
  end
```

```
my-first-app> ruby script/generate hobo_migration
```

Hobo should now respond:

```
DROP or RENAME?: column contacts.married
Rename choices: married_now
Enter either 'drop married' or one of the rename choices:
```

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.

Programming 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.’

3. **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 facility in its `viewhints` capability. Every model in a Hobo application has a corresponding `viewhints` file in the `viewhints` directory. Go to the `contact_hints.rb` file in the `viewhints` directory and enter the following code.



Figure 49: Creating a Hobo “ViewHints” definition for the Contact model

Refresh your browser and you should see the fields relabeled with your choices from above. Notice that a migration is not necessary when using viewhints.



Figure 50: View of field relabeled using the Hobo viewhints “field_names” method

4. **Using view hints to suggest field uses.** The viewhints file also provides the facility to provide a suggestion below the field on what to enter into it. Edit your `contact_hints.rb` file to look like this.

```
class ContactHints < Hobo::ViewHints
  field_names :name => "Friend", :address_1 => "Address"
  field_help :name => "Put your friend's name here.",
```

```
:company => "Where does your friend work?",  
:married_now => "Married or not?"  
  
end
```

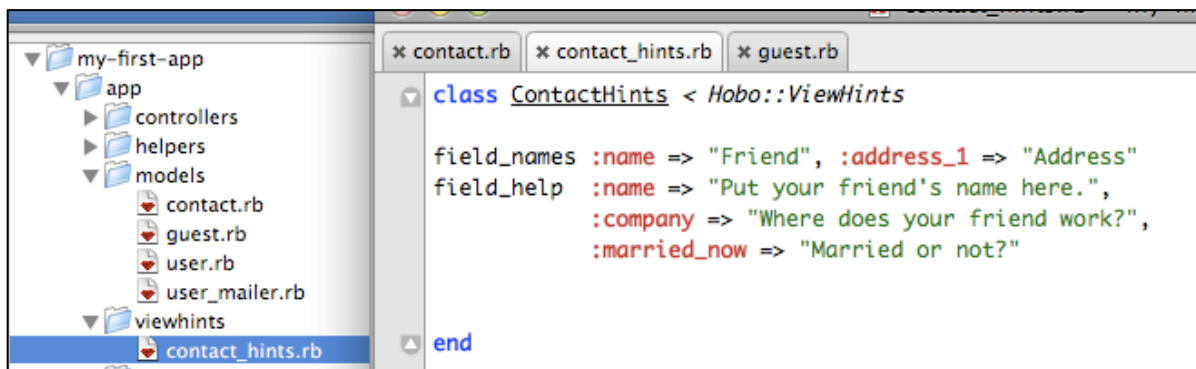


Figure 51: Adding help text using the Hobo viewhints "field_help" method

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

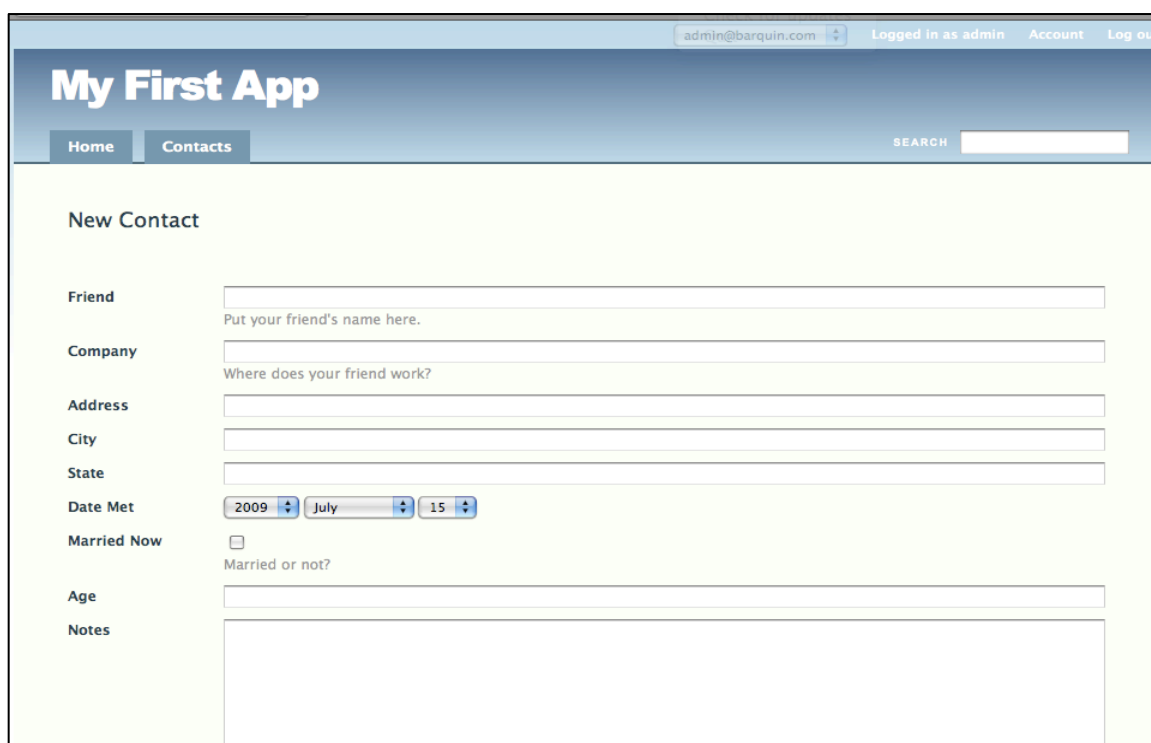


Figure 52: Contact entry page with ViewHints enabled

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.

5. **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
```

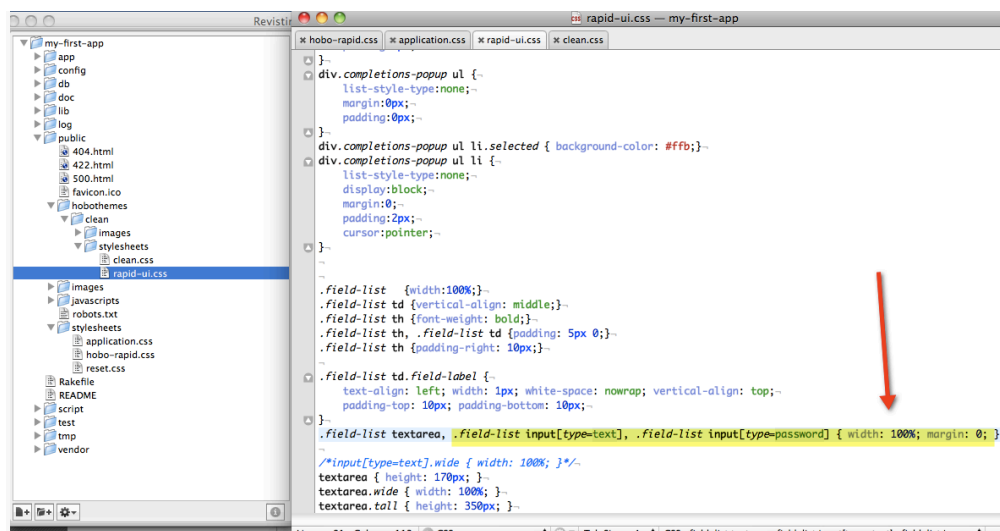


Figure 53: CSS definitions for the input text fields

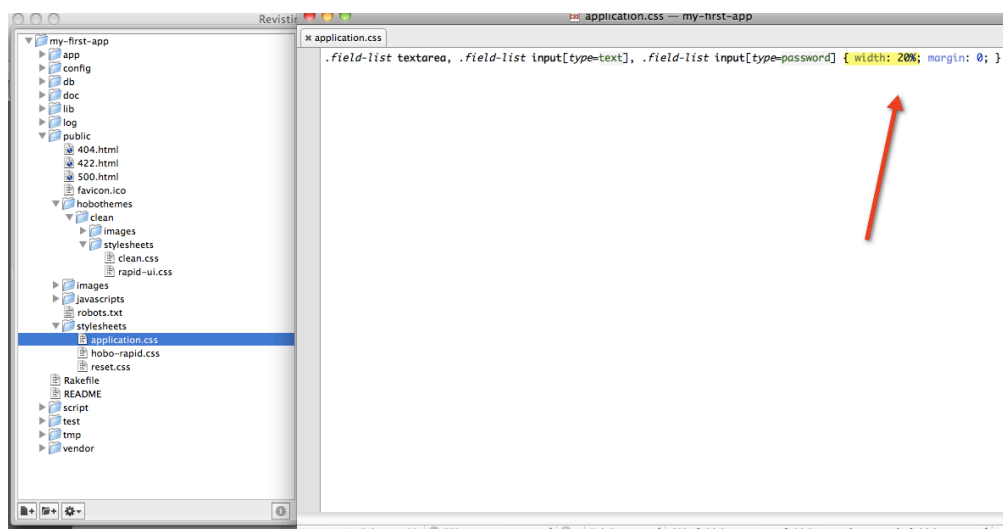


Figure 54: Modified entry in "application.css" to shorten text prompts

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
- 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*.

The screenshot shows the 'My First App' web interface. At the top, there's a navigation bar with 'Home' and 'Contacts' tabs. The 'Contacts' tab is active. Below the navigation bar, there's a 'New Contact' section. A red error message box states: 'To proceed please correct the following: Friend can't be blank'. Below this, there are two input fields: 'Friend' and 'Company'. The 'Friend' field is empty and has a red border, indicating it's required. The 'Company' field is also empty. The 'Address' field is partially visible at the bottom.

Figure 55: Page view of validating presence of name

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

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

Note that use must use the model field name, not the label name you used in the ViewHints file. Click the Contacts tab and then *New Contact*. Without entering anything in the form, click *Create Contact*.

The screenshot shows the 'My First App' web interface. At the top, there's a navigation bar with 'Home' and 'Contacts' tabs. The 'Contacts' tab is active. Below the navigation bar, there's a 'New Contact' section. A red error message box states: 'To proceed please correct the following: Friend can't be blank, Address can't be blank'. Below this, there are four input fields: 'Friend', 'Company', 'Address', and 'City'. The 'Friend' and 'Address' fields are empty and have red borders, indicating they are required. The 'Company' and 'City' fields are also empty.

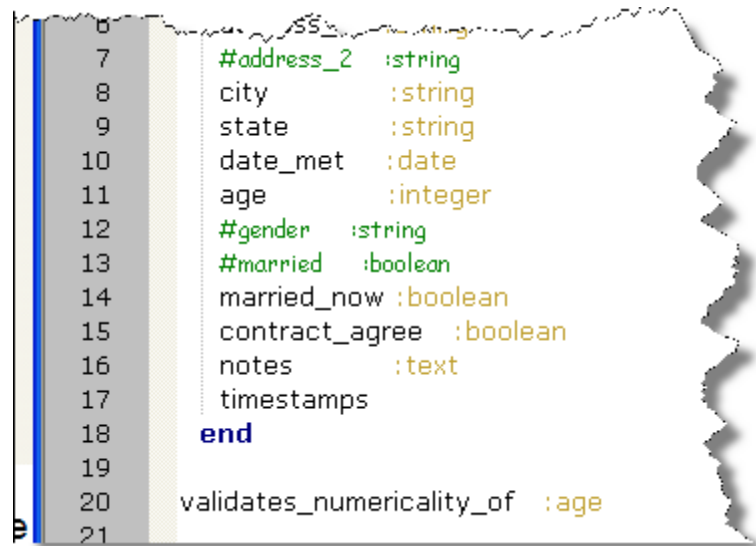
Figure 56: Page view of double validation error

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
```



```
6 # == Model ==
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 end
19
20 validates_numericality_of :age
21
```

Figure 57: Adding “validates_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 navigation bar containing "Home" and "Contacts" links, and a search bar. The main content area is titled "Edit Contact" and includes a "Remove This Contact" button. A red error message box states: "To proceed please correct the following: ■ Age is not a number". Below this, the form fields are: "Friend" (text input with "John" and a hint "Put your friend's name here."), "Company" (text input with a hint "Where does yhour friend work?"), "Address" (text input with "35 Elm street"), "City" (text input), "State" (text input), "Date Met" (three dropdown menus for year, month, and day, showing "2009", "November", and "20"), and "Age" (text input with "0").

Figure 58: 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 browser window with the address bar displaying 'odall@barquin.com'. The page title is 'My First App'. The navigation bar includes 'Home' and 'Contacts' tabs, and a search bar. The main content area is titled 'New Contact'. A red error message box states: 'To proceed please correct the following: Friend has already been taken'. Below this, the form fields are: 'Friend' (text input with 'Jeff' and a red border), 'Company' (text input with placeholder 'Where does your friend work?'), 'Address' (text input with '27 West street'), 'City' (text input), 'State' (text input), 'Date Met' (three dropdown menus showing '2009', 'November', and '20'), and 'Age' (text input with '33').

Figure 59: 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"
```

My First App

Home Contacts SEARCH

New Contact

To proceed please correct the following:

- Age Must be between 18 and 65

Friend
Put your friends's name here.

Company
Where does yhour friend work?

Address

City

State

Date Met

Age

Married Now ☐

Figure 60: 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:

My First App

Home Contacts SEARCH

Edit Contact

Remove This Contact

To proceed please correct the following:

- Friend - enter a longer one!

Friend
Put your friend's name here.

Company
Where does your friend work?

Address

City

State

Date Met

Figure 61: Page view of validation of text length error

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.

The screenshot shows a web application titled "My First App" with a navigation bar containing "Home" and "Contacts" tabs, and a search bar. The main content area is titled "New Contact". A red error message box at the top of the form states: "To proceed please correct the following: Contract agree must be accepted". The form fields are: Friend (text input with "John"), Company (text input), Address (text input with "123 Walnut"), City (text input), State (text input), Date Met (date picker with "2009", "July", "19"), Age (text input with "55"), Married Now (checkbox), and Contract Agree (checkbox). The "Contract Agree" checkbox is unchecked.

Figure 62: 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

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. Then change directory to the subdirectory `one_table`:

```
tutorials> hobo one_table
tutorials> cd one_table
one_table>
```

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

2. **Perform the migration.** Now perform the first Hobo migration to create the users table.

```
one_table> ruby script/generate hobo_migration
```

The **`hobo_migration`** generator creates the Rails migration file containing the users fields and from that the users table in one easy step.

Important Note: Look at the file `app\models\user.rb` and at the database schema `app\db\schema.rb`. Note that 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, the Mongrel server, by issuing the following command.

```
one_table> ruby script/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 63: 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 *signup* to create your first account. **Remember, by default, the user module assigns administrative privileges to the first account created.** We refer to it as the *admin* account. Logout and create a second account. We will refer to this 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_model_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> ruby script/generate hobo_model_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 which 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> ruby script/generate hobo_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.

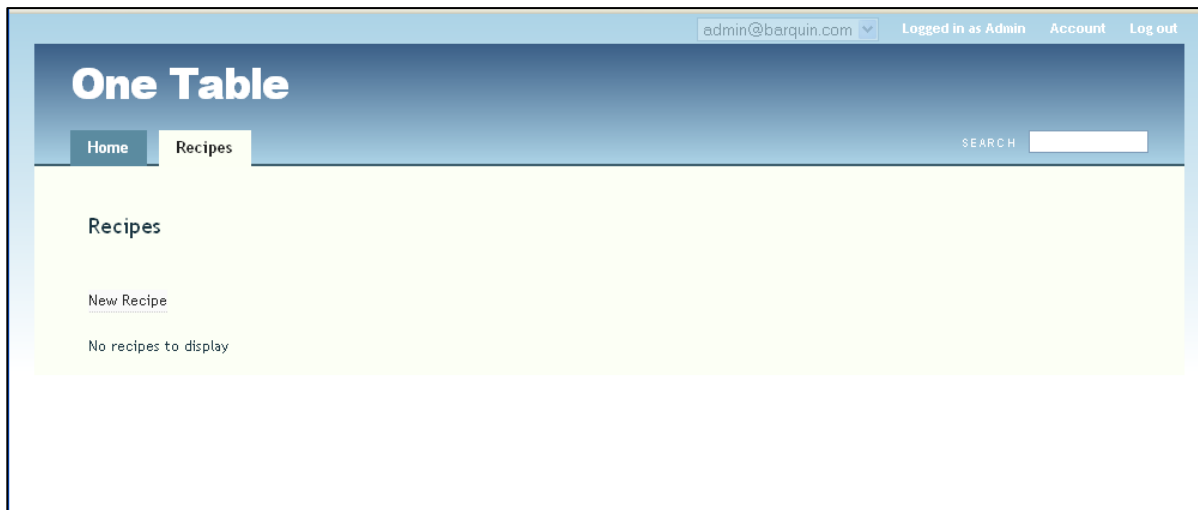


Figure 64: Recipes tab

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.

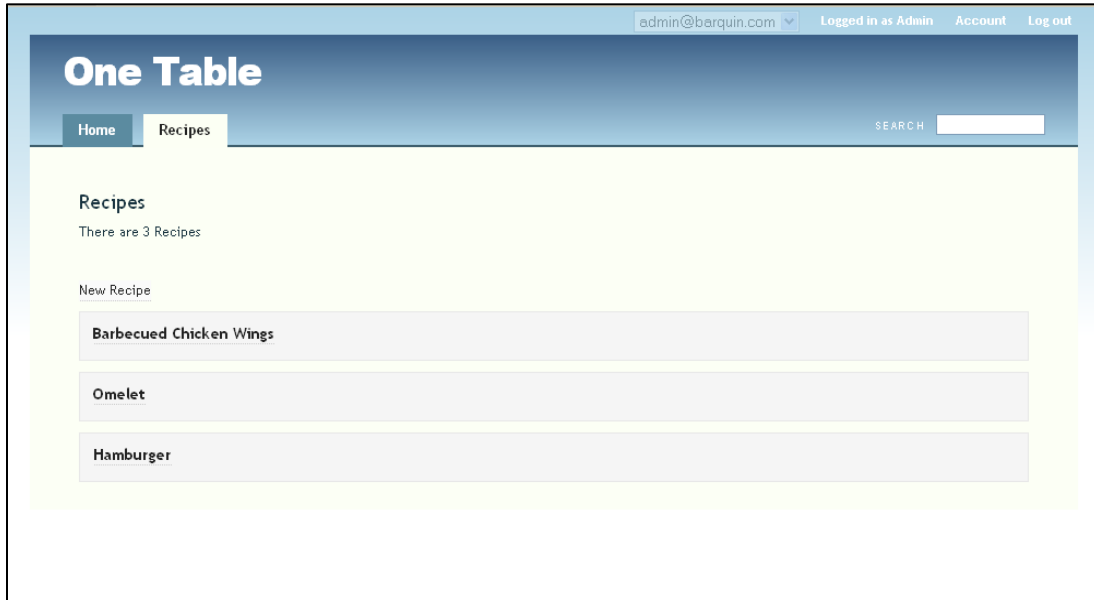


Figure 65: Page view of created recipes

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?  
  acting_user.administrator?  
end  
  
def destroy_permitted?  
  acting_user.administrator?  
end  
  
def view_permitted?(field)  
  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 |

Figure 66: Table of Hobo permission methods

For the code that is generated by the **hobo_model_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.

| acting_user method | Meaning |
|-----------------------------|---|
| <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 |

Figure 67: 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?` equals '1' if the user is the administrator and '0' 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.

Note: The '?' after signed up indicates the method is a Boolean method.

The meaning of the default permissions code can be summarized simply now. 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 in 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.

- 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 now.

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.

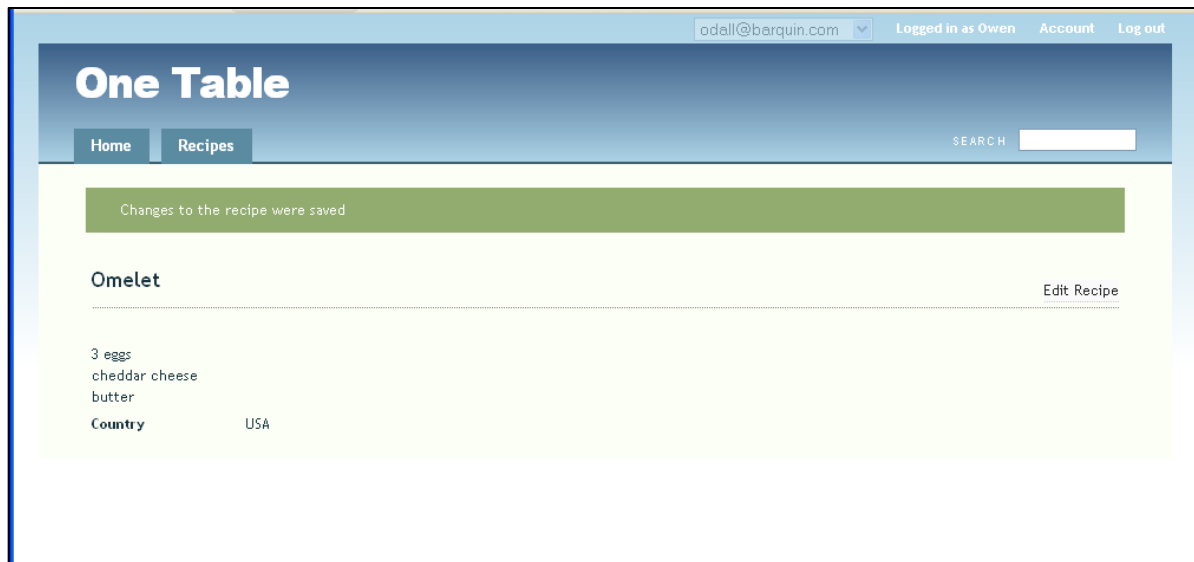


Figure 68: Page view of a Recipe

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.

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

1. **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 `controllers/recipes_controller.rb` file. Open up this file.

Programming 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-->`.

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:

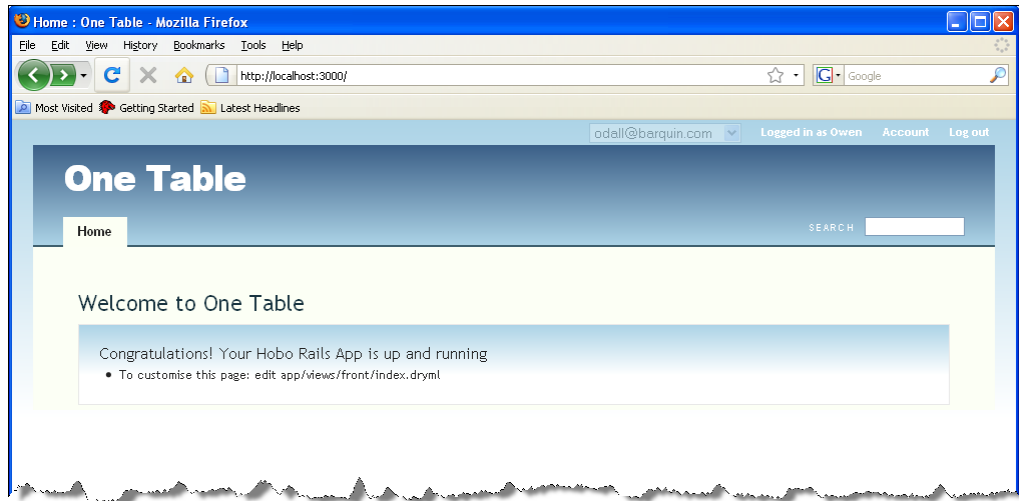


Figure 69: Making the Recipes tab disappear

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.

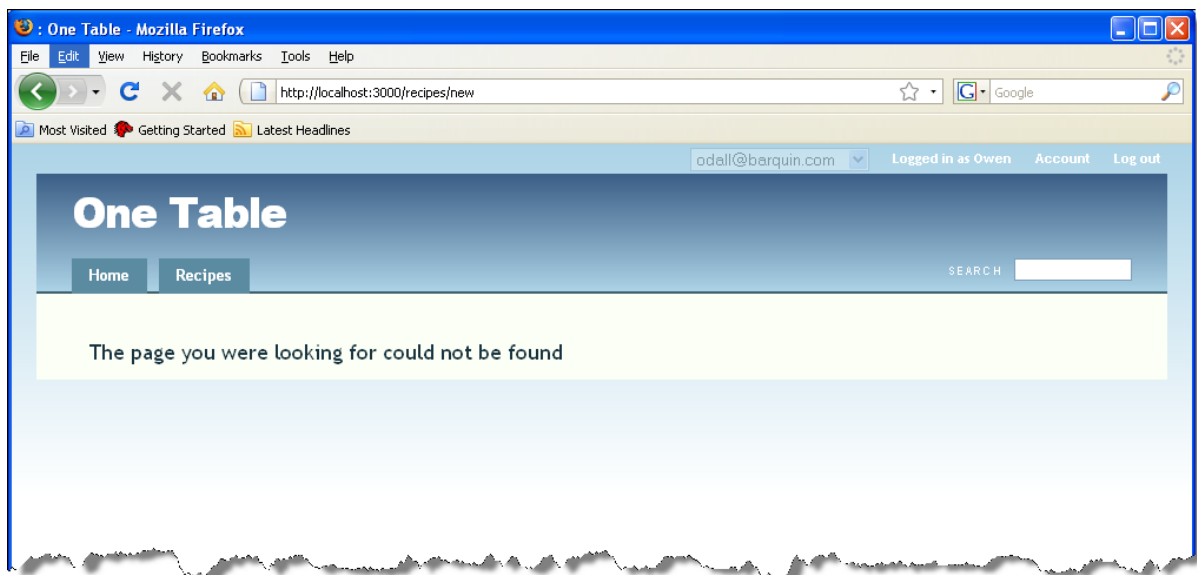
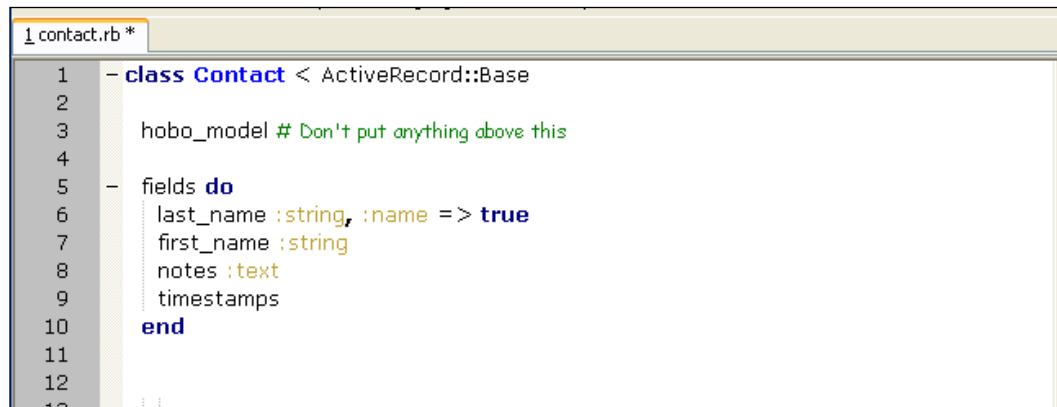


Figure 70: Error message “The page you were looking for could not be found”

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.

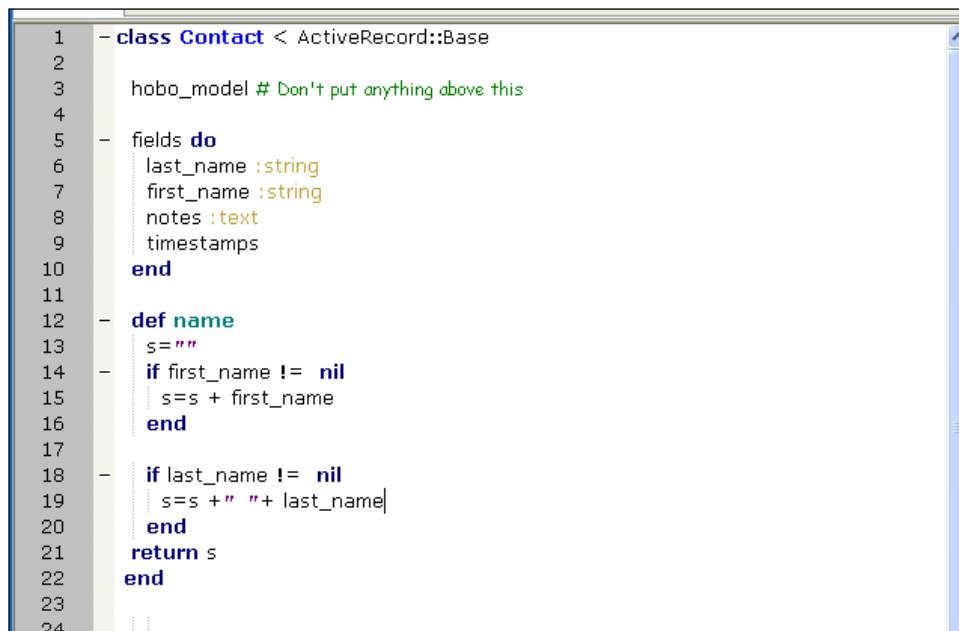
Note: Hobo assigns a name variable to the model equal to the value of the field it thinks is the most likely summary field. Hobo first looks for a field called `name`. Next it looks for the next most likely, which in this case it guesses is `title`. You can override the automatic name assignment by adding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



```
1 - class Contact < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   - fields do
6     last_name :string, :name => true
7     first_name :string
8     notes :text
9     timestamps
10  end
11
12
```

Figure 71: Setting the Hobo "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:



```
1 - class Contact < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   - fields do
6     last_name :string
7     first_name :string
8     notes :text
9     timestamps
10  end
11
12  - def name
13    s=""
14    - if first_name != nil
15      s=s + first_name
16    end
17
18    - if last_name != nil
19      s=s + " " + last_name
20    end
21    return s
22  end
23
24
```

Figure 72: Creating your own 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.

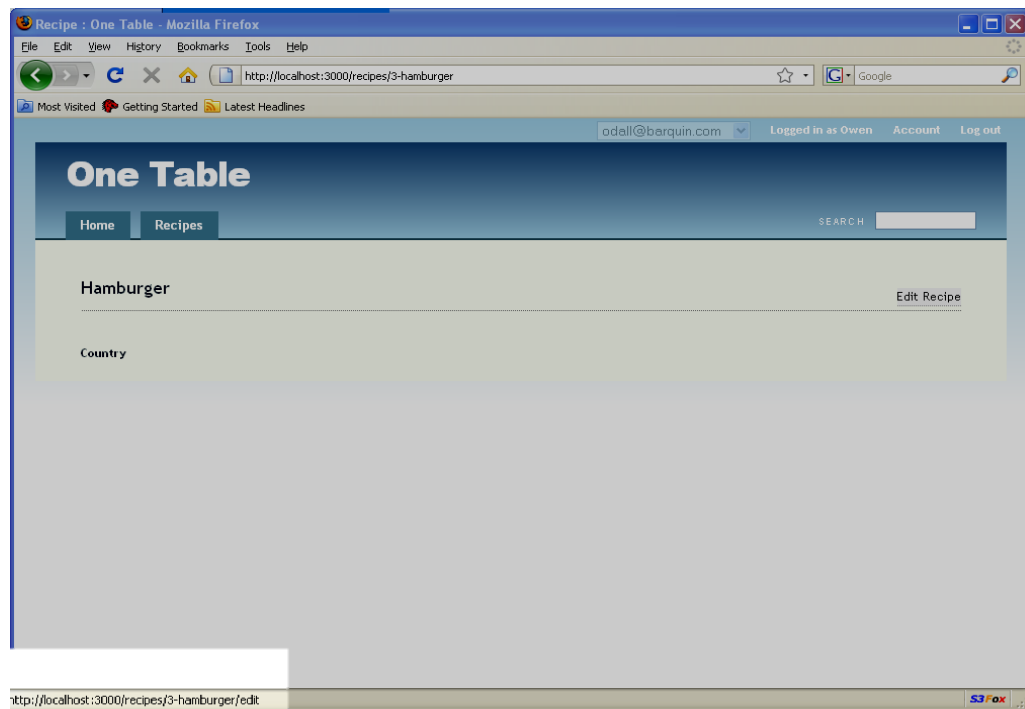


Figure 73: Viewing the edit URL

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

```
http://localhost:3000/recipes/6-hamburger/edit
```

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

```
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:

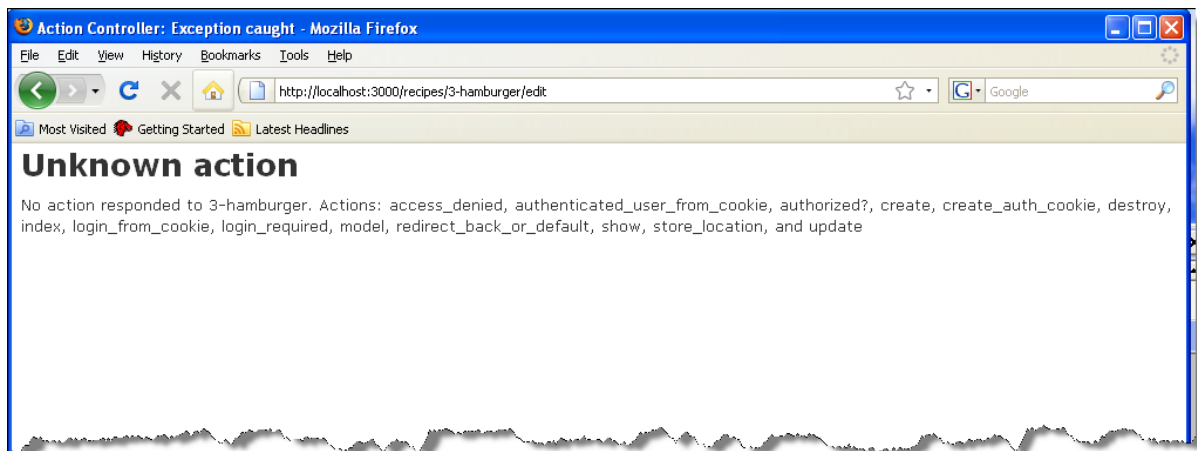


Figure 74: "Unknown action" error page

4. **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 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 (model - recipe) |
|---------|--|----------------------------------|-----------------------------|
| index | display list of records | /base/model(plural) | /base/recipes |
| show | display a single record | /base/model(plural)/ID-name | /base/recipes/2-omelette |
| 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/edit | /base/recipes |
| update | save the contents of an edited record | lands on show | /base/recipes |
| destroy | delete the record | lands on index | /base/recipes |

Figure 75: 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>  
    <nav-item href="{base_url}/">Home</nav-item>  
    <nav-item with="&Recipe">Recipes</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’. Just chain the content of the Recipe tab to ‘My Recipes’. Now your code should look like this:

```
<def tag="main-nav">
  <navigation class="main-nav" merge-attrs>
    <nav-item href="#{base_url}/">Home</nav-item>
    <nav-item with="&Recipe">My Recipes</nav-item>
  </navigation>
</def>
```

Refresh your browser and you will see a renamed tab:



Figure 76: Customizing the name of a tab

6. **Remove the Home Tab.** Instead of deleting the Home tab, just comment it out by surround 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.

```
<def tag="main-nav">
  <navigation class="main-nav" merge-attrs>
    <!--<nav-item href="#{base_url}/">Home</nav-item>-->
    <nav-item with="&Recipe">My Recipes</nav-item>
  </navigation>
</def>
```

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

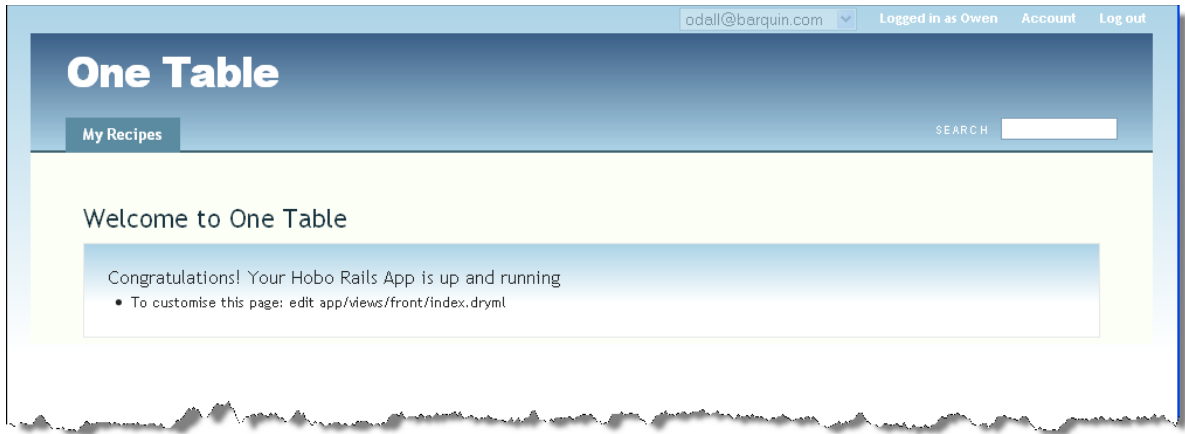


Figure 77: Removing the default Home tab

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">My Recipes</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

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 it the application and work on the new version. Go ahead and copy the entire application directory and paste it into a folder called `two_table` in your `tutorials` directory. Next, remove the piece we added to `application.dryml`, and change the `app-name` tag definition to “Two Table” from “One Table”

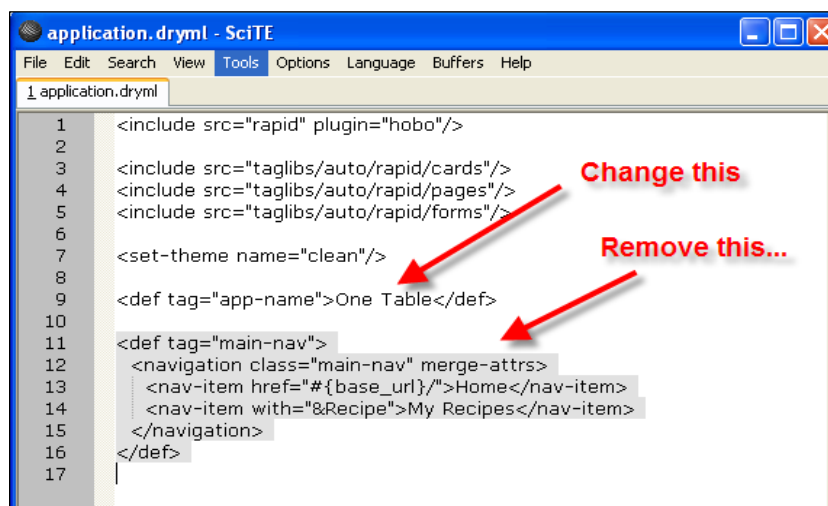


Figure 78: Renaming a copy of your application

Next shut down the web server by issuing a `<control-c>` in the command window where you issued the `ruby script/server` command.

Restart the web server and you are ready to go.

```
two_table> ruby script/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. In 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.

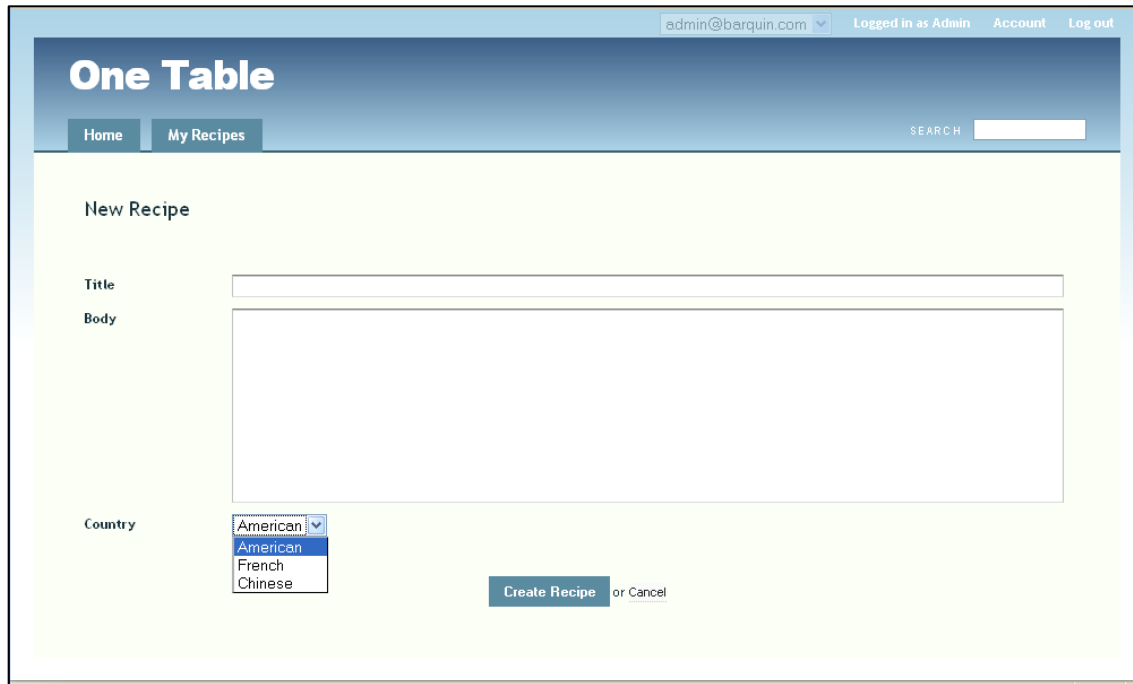
The screenshot shows a web browser window with the URL 'admin@barquin.com'. The page title is 'One Table'. There are navigation links for 'Home' and 'My Recipes', and a search bar. The main content area is titled 'New Recipe'. It contains three input fields: 'Title' (a single-line text box), 'Body' (a large multi-line text area), and 'Country' (a drop-down menu). The 'Country' menu is open, showing four options: 'American' (highlighted), 'American', 'French', and 'Chinese'. At the bottom right of the form are two buttons: 'Create Recipe' and 'Cancel'.

Figure 79: 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
end
```

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

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.** Using Hobo's `model_resource` generator 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> ruby script/generate hobo_model_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. Let's define our relationships now.

The `hobo_model_resource` generator also created some other directories and files. It created a controller file called `countries_controller.rb` and a view template directory called `views/countries`. 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.

Naming Convention 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 :string
  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 for every recipe there will be, at most, one country that it is associated with. Add the `belongs_to` declaration just before the `#permissions` comment.

```
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
```

Note: It is useful to read `belongs_to` as ‘refers to’ to remind your self 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* record, which contains the `name` field.

Note: 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"
```

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.

8. **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.

Now, 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` table that corresponds to a `country_id` in a country record.

```
two_table> ruby script/generate hobo_migration
```

You will get the following response:

```
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'.

Last it will prompt you to name the migration file:

```
Filename [hobo_migration_3]:
```

Just hit the 'enter' key and it will take the default name, `hobo_migration_3`.

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

```
ActiveRecord::Schema.define(:version => 20100119212115) 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    "name"
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    "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
```

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.

10. **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*.
11. **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 80: 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 you saw when you used the `enum_string` option for your attribute:

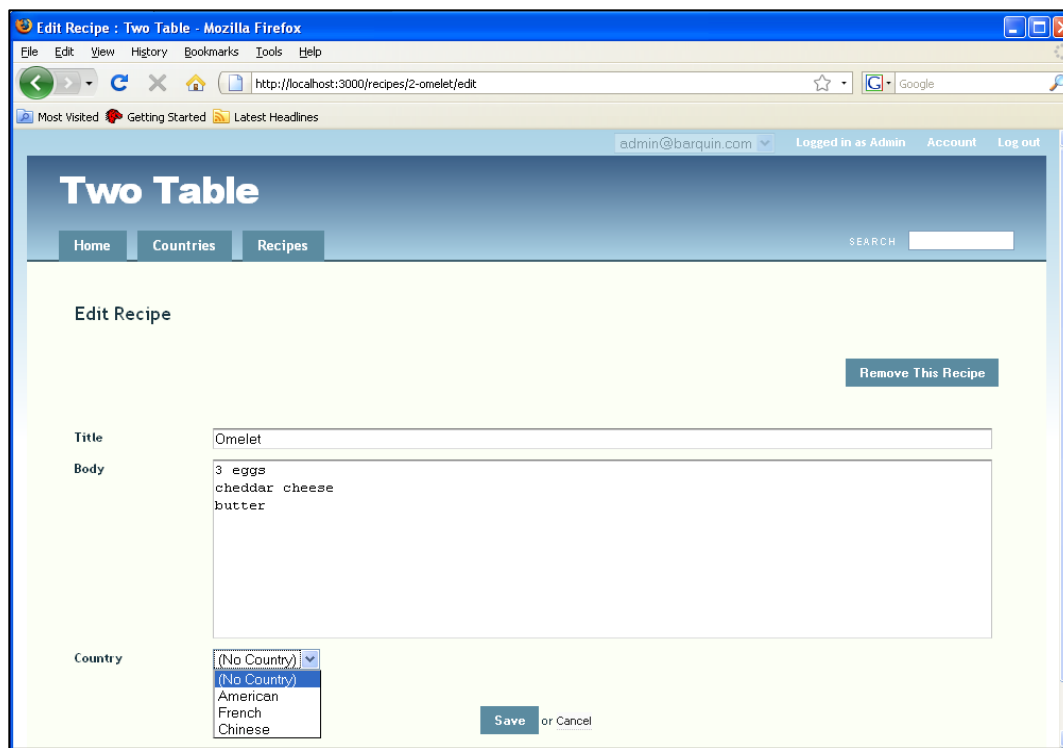


Figure 81: Selecting a Country for a Recipe

The difference is that you are now 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 maintains all of the necessary related keys automatically.

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

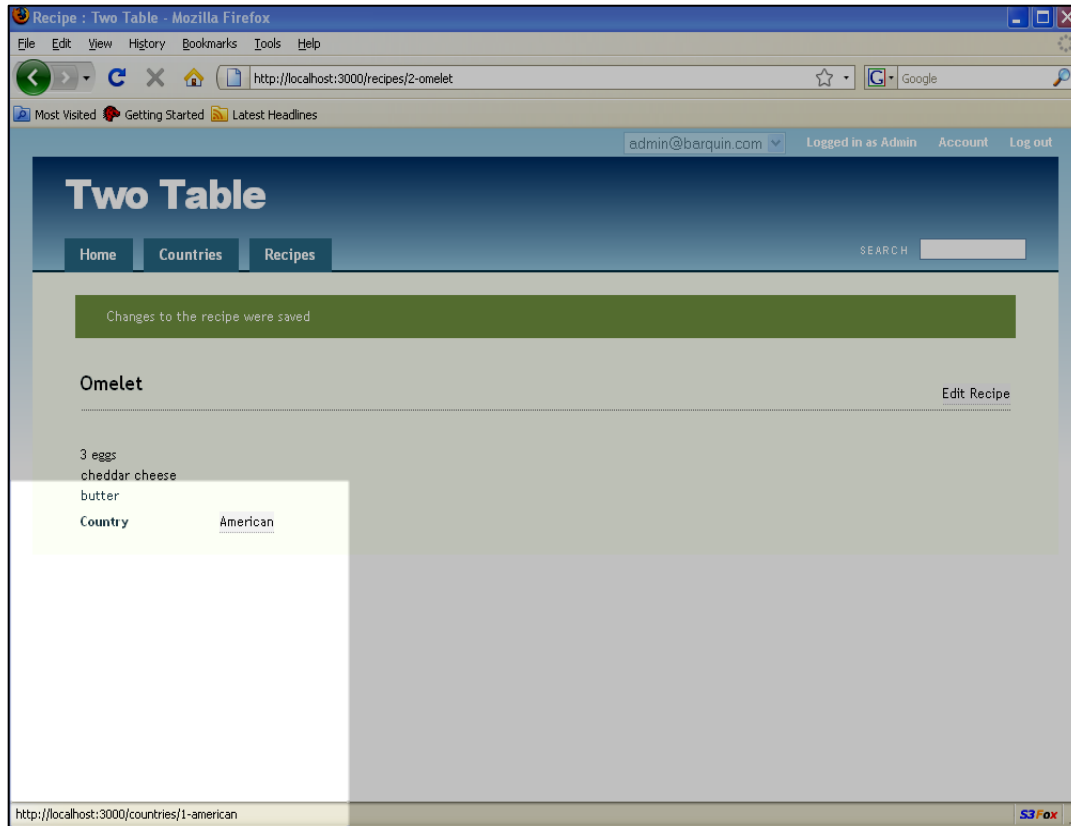


Figure 82: Active link on Country name in the Recipe show page

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

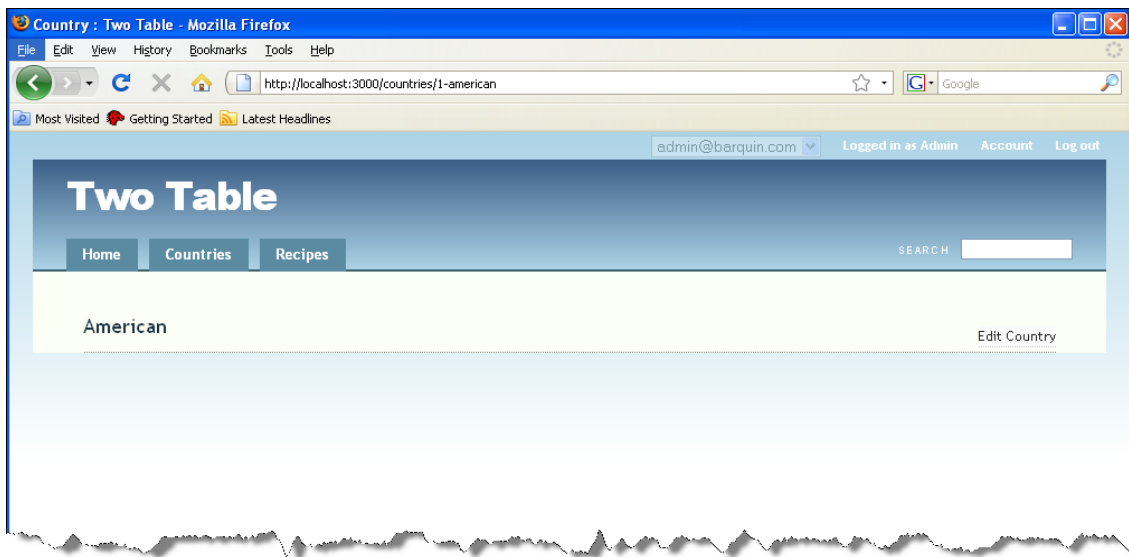


Figure 83: The Country show page accessed from the Recipe show page

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

  . . .
```

Now go to the Recipes tab, click on a recipe link and edit the recipe. Next click on the country name on the page. Now you see that the `Edit Country` link is no longer available.

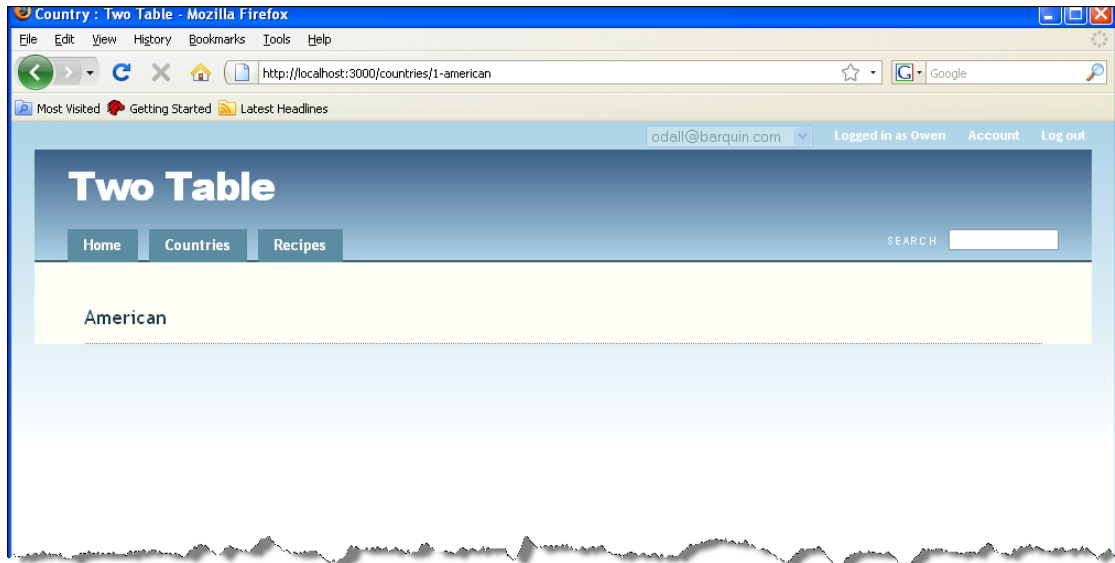


Figure 84: Editing Hobo Permissions to remove the Country Edit link

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 `ruby script/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> ruby script/server
```

You may wish to change the name of your application as displayed in the UI. Go to `views/taglibs/application.dryml`. Change the code `<app-name>` tag to read:

```
<def tag="app-name">Four Table</def>
```

Now 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. The first will be a *Category* model and the second will be 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 all for you.

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

```
four_table> ruby script/generate hobo_model_resource category name:string
four_table> ruby script/generate hobo_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

  has_many :categories, :through => :category_assignments, :accessible =>
true
  has_many :category_assignments, :dependent => :destroy
  belongs_to :country
```

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

end
```

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

end
```

4. **Discussion of model relationships.** Note above that 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.

Recall that 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 migration.** Go to your command prompt and run the following.

```
four_table> ruby script/generate hobo_migration
```


Remember to respond ‘m’ when prompted for migration and just ‘return’ 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> ruby script/server
```

6. **Populate the new table.** Open up your browser to <http://localhost:3000/> and you should see the following:

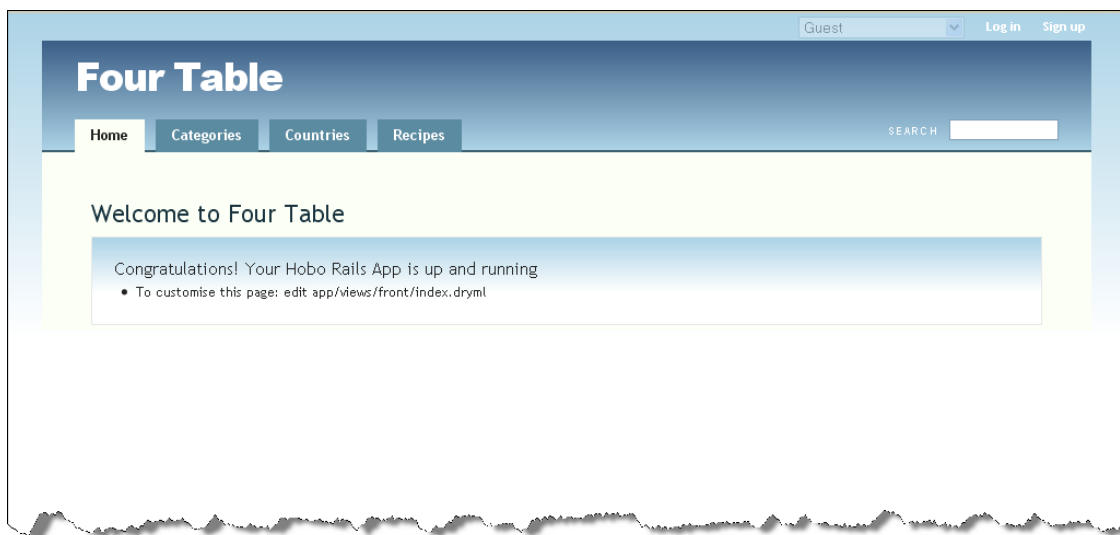


Figure 85: The Categories tab on the Four Table app

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

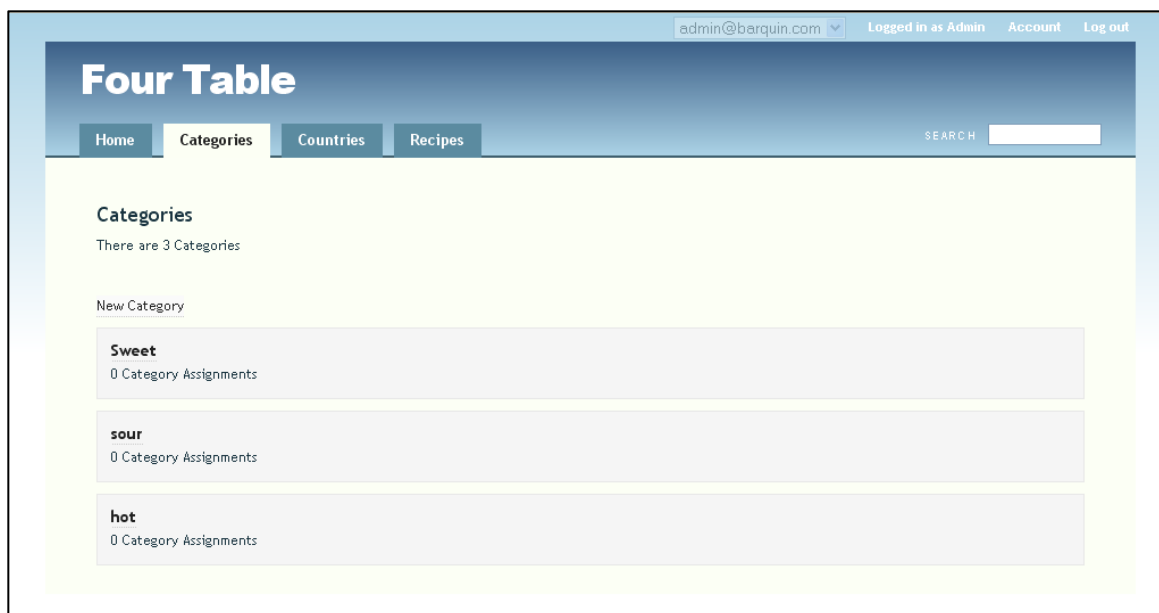


Figure 86: The Index page for Categories

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

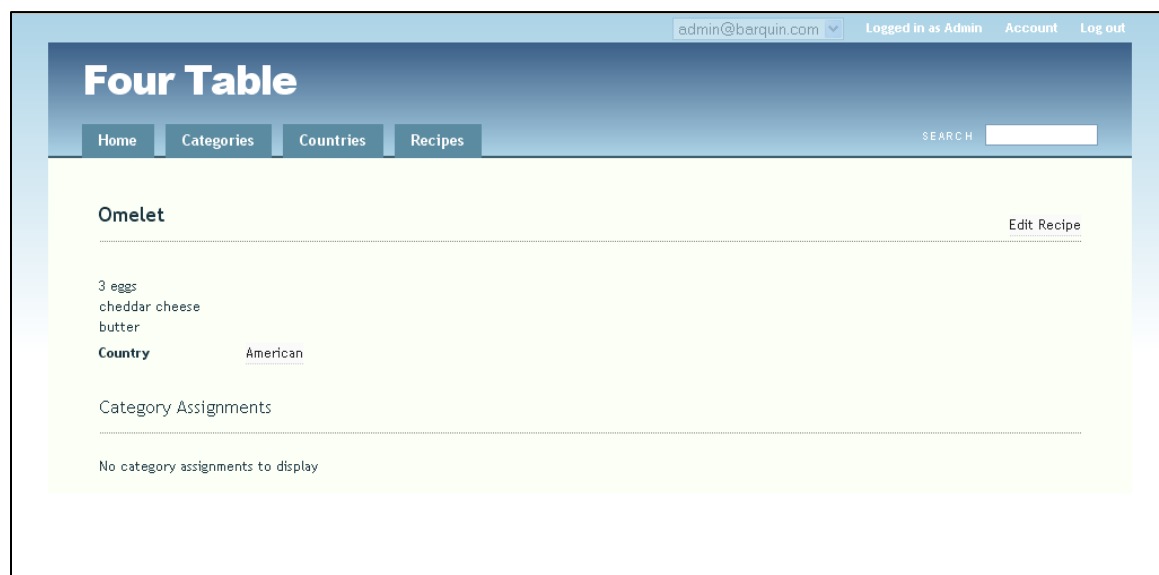


Figure 87: "Category Assignments" on the Recipe show page

Notice there is no category assignment.

Then click *Edit Recipe* on the right.

The screenshot shows a web application titled "Four Table". At the top, there is a navigation bar with links for "Home", "Categories", "Countries", and "Recipes". A search bar is also present. The user is logged in as "Admin" with the email "admin@barquin.com". The main content area is titled "Edit Recipe". It features a "Remove This Recipe" button in the top right. The form has four main sections: "Title" with a text input containing "Omelet"; "Body" with a text area containing "3 eggs", "cheddar cheese", and "butter"; "Categories" with a dropdown menu showing "Add Category"; and "Country" with a dropdown menu showing "American". At the bottom of the form are "Save" and "Cancel" buttons.

Figure 88: 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 a web application interface for editing a recipe. At the top, there's a navigation bar with 'Home', 'Categories', 'Countries', and 'Recipes' tabs. A search bar is on the right. The main heading is 'Edit Recipe'. A 'Remove This Recipe' button is in the top right. The form has several fields: 'Title' with the value 'Omelet', 'Body' with the text '3 eggs', 'cheddar cheese', and 'butter', 'Categories' with 'hot' and 'sour' (each with a 'Remove' button), an 'Add Category' dropdown, and 'Country' with a dropdown set to 'American'.

Figure 89: 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. **Display the associations.** After you save the recipe record with its associations you should obtain something like the screen pictured below. This is not really what you want. You would probably prefer to see Category 1 and Category 2 explicitly shown. Hobo tries to guess which table's values to display but, in this case, it has not chosen the ones that you prefer.

It is showing the index numeric value instead of the Category name.

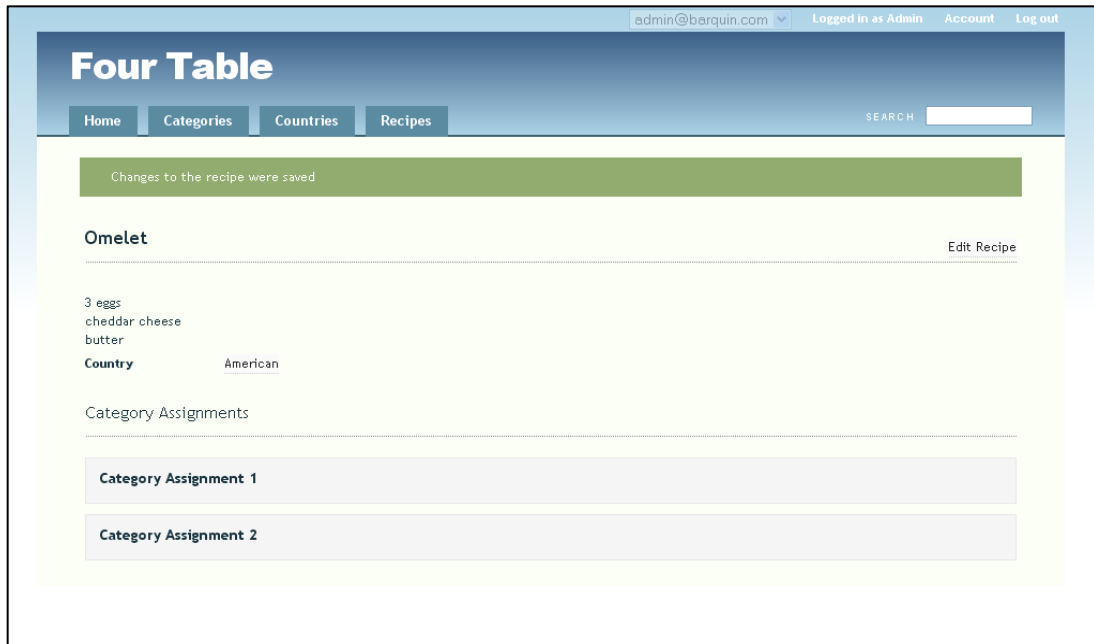


Figure 90: Show page view of Categories assigned to a recipe

In Chapter 4 you will learn the fancy ways to deal with this problem by editing the auto-generated tags in the Rapid directories. Luckily there is a much easier way to deal with this problem using Hobo ViewHints.

Go to the `app/viewhints/recipe` directory.

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

```
class RecipeHints < Hobo::ViewHints
  children :categories
end
```

Now refresh your browser.

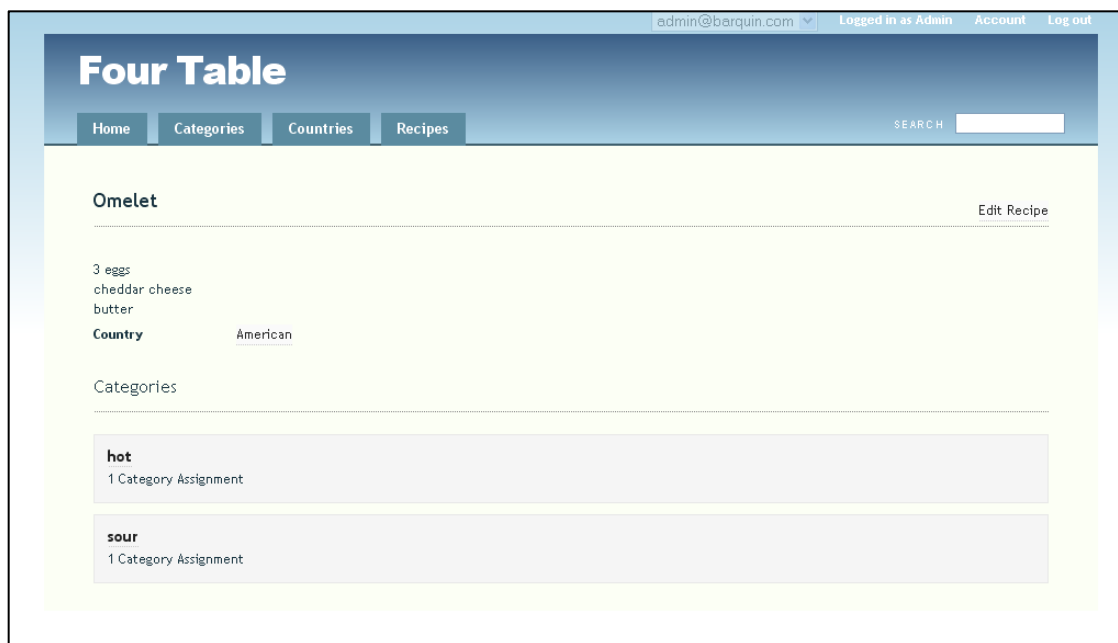


Figure 91: Using Hobo ViewHints to enhance the view of related records

Now instead of generic values, Category Assignment 1 and Category Assignment 2, you get the actual categories, hot and sour.

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'.

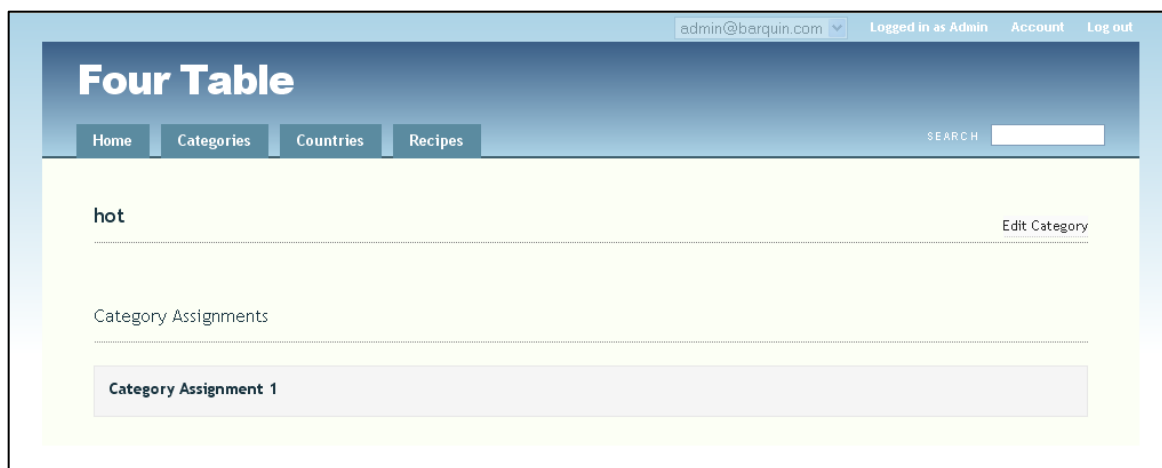


Figure 92: Show page for a Category before using ViewHints

Either way you get a non-specific Category Assignment.

Now let's fix this in the same way as we fixed the categories belonging to a specific recipe.

Go to the `app/viewhints/category` directory.

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

```
class CategoryHints < Hobo::ViewHints
  children :recipes
end
```

Refresh your browser.

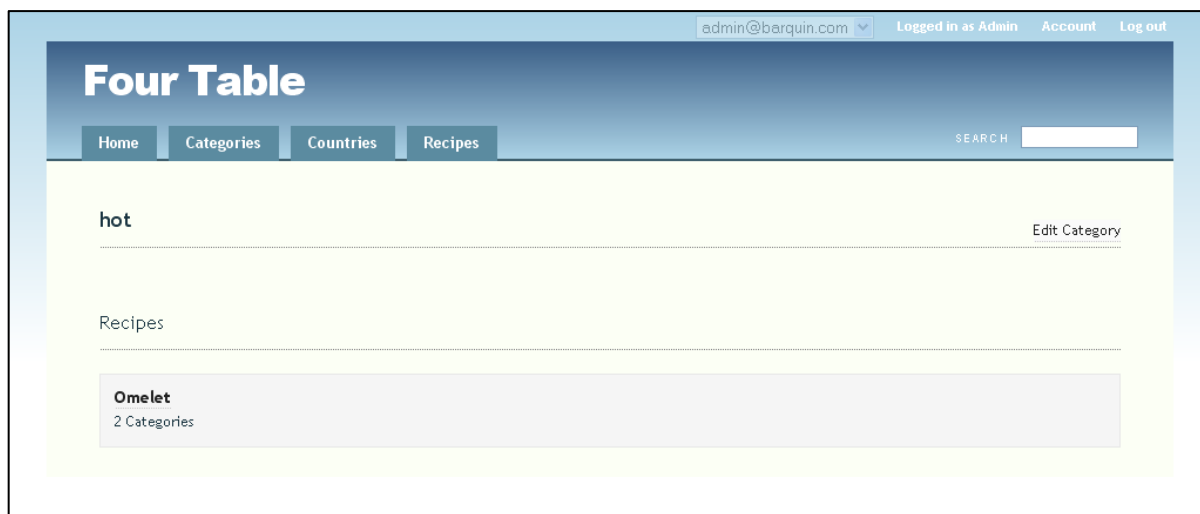


Figure 93: Category page view after adding ViewHints "children :recipes" declaration
n

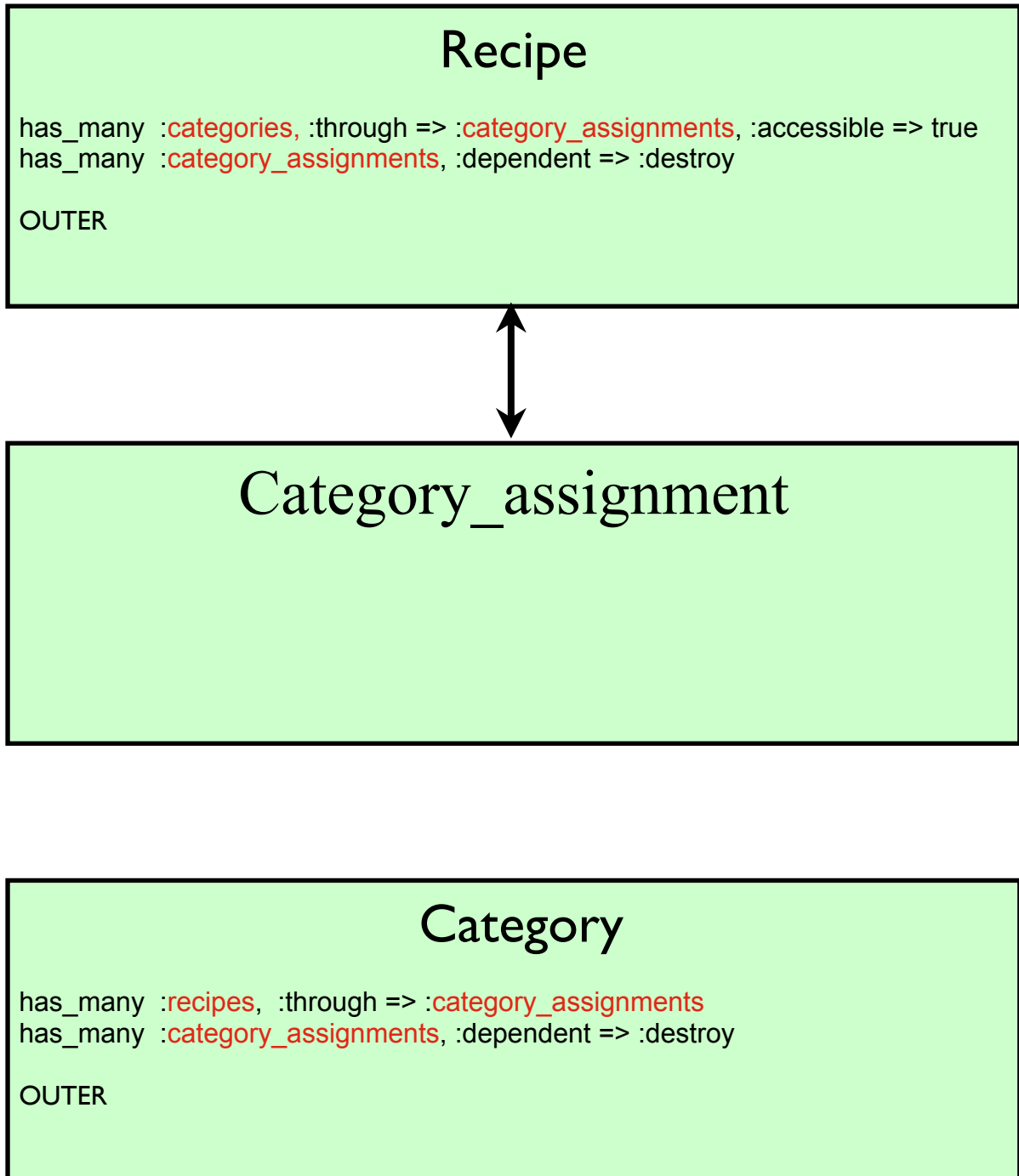
Now it is all fixed.

9. **Comments on the many-to-many relationship.** Now 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.



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

Tutorial 13 - Listing Data in Table Form

Tutorial 14 - Working with the Show Page

Tutorial 15 - New and Edit Pages With The Form Tag

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 magician worth his salt will reveal his tricks to an apprentice all at once. There is only so much we can absorb at one time. 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. And after that, you learn to make more patterns that you and others can use again and again.

So, in this chapter we will start revealing how “Rapid” (Hobo’s process of automatically rendering forms, views and routing) works in way we think it can 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, invisibly.

So 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:

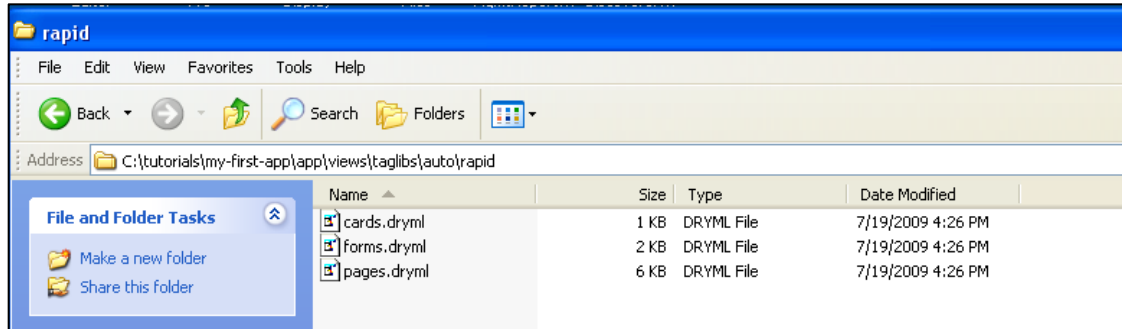


Figure 94: Folder view of \taglibs\auto\rapid

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 when needed,

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, without the necessity of specifying the granular level of detail that other frameworks require, such as the hybrid of Ruby and HTML in its views as Rails does with its eRB (embedded Ruby) pages.

In this chapter we will explore:

- (4) The Hobo Rapid library of tags
- (5) The auto-generated DRYML files that expose the Rapid process
- (6) 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> ruby script/server
```

You should now have a UI that looks like this:

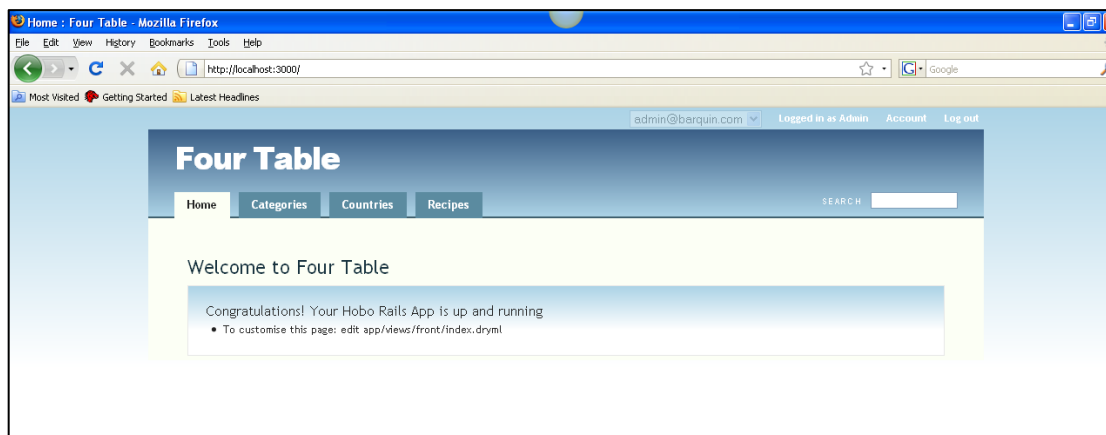


Figure 95: Front page view of the Four Table application

Now open your editor and navigate to the `views/taglibs` directory:

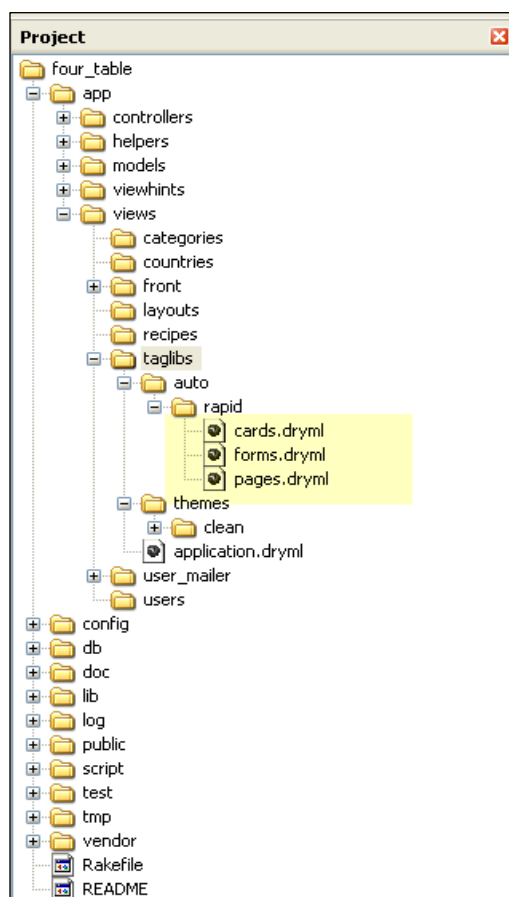


Figure 96: Folder view of the rapid DRYML files

Take a look at this directory structure. Focus on the files in the `views/taglibs/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_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.

Now 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>
```

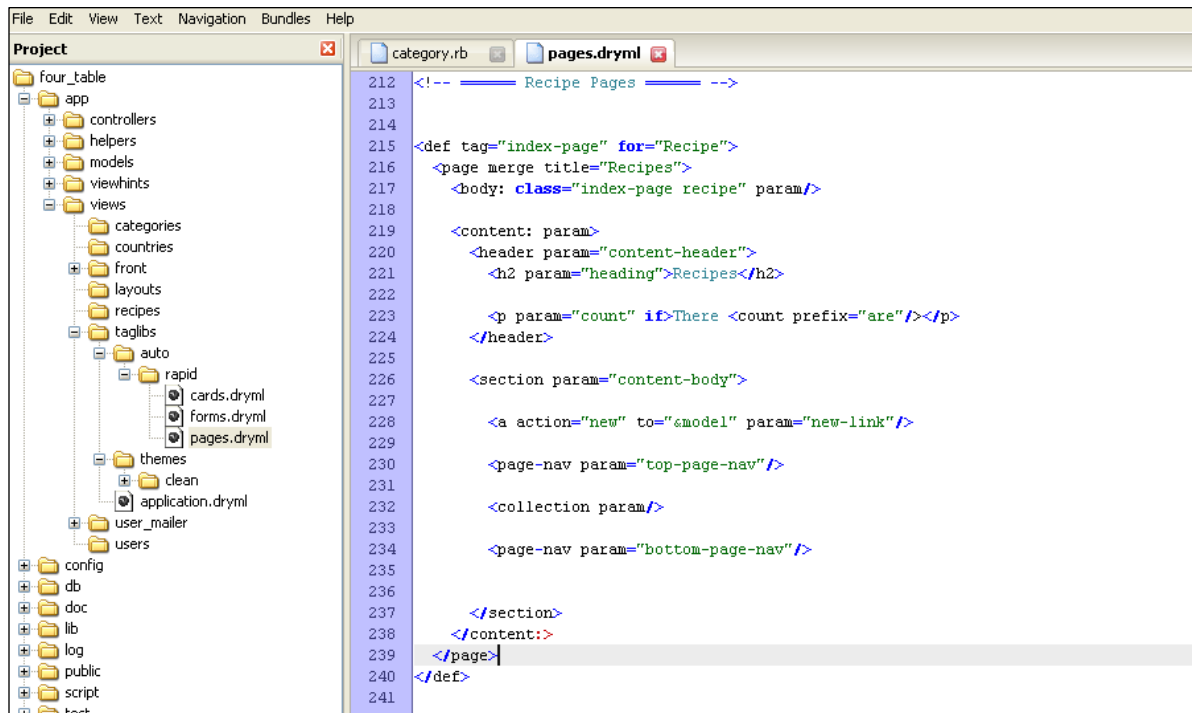


Figure 97: Content of the "pages.dryml" file

. 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 page for a new record. | Forms | new | */model_name/new |
| <show-page> | renders a single record. | None | show | */model_name/ID-record_name |
| <edit-page> | renders a data entry page for an existing record. | Forms | edit | */model_name/edit/ID-record_name |

Figure 98: 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.

The last important point to realize is that 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 noted below. The comments above are summarized in the following table.

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:

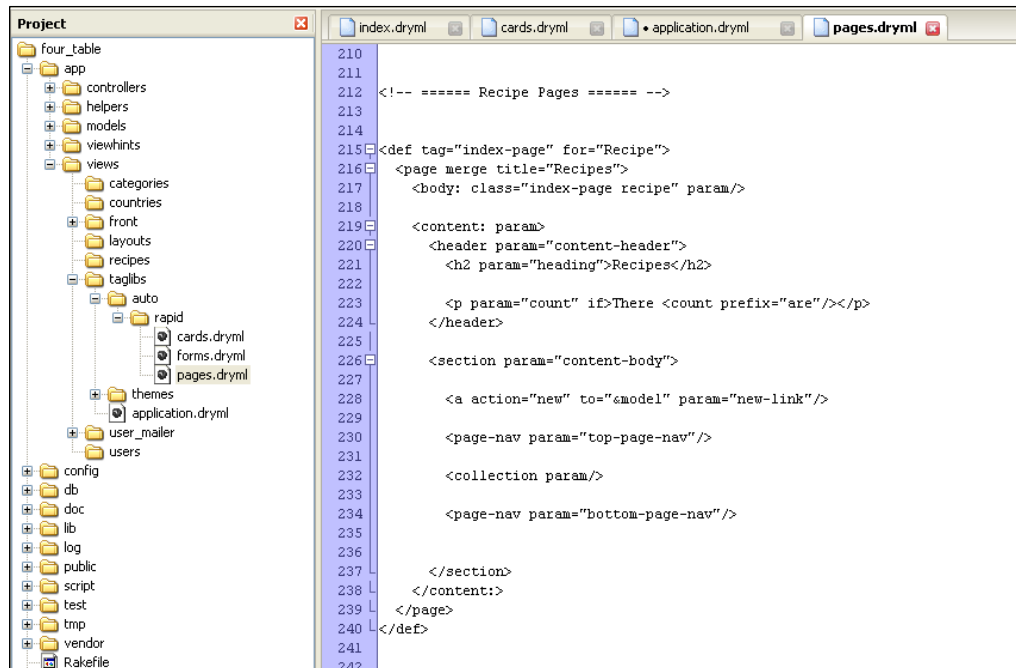


Figure 99: The Hobo Rapid `<index-page>` tag definition in the `pages.dryml` file

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 16 of Chapter 3

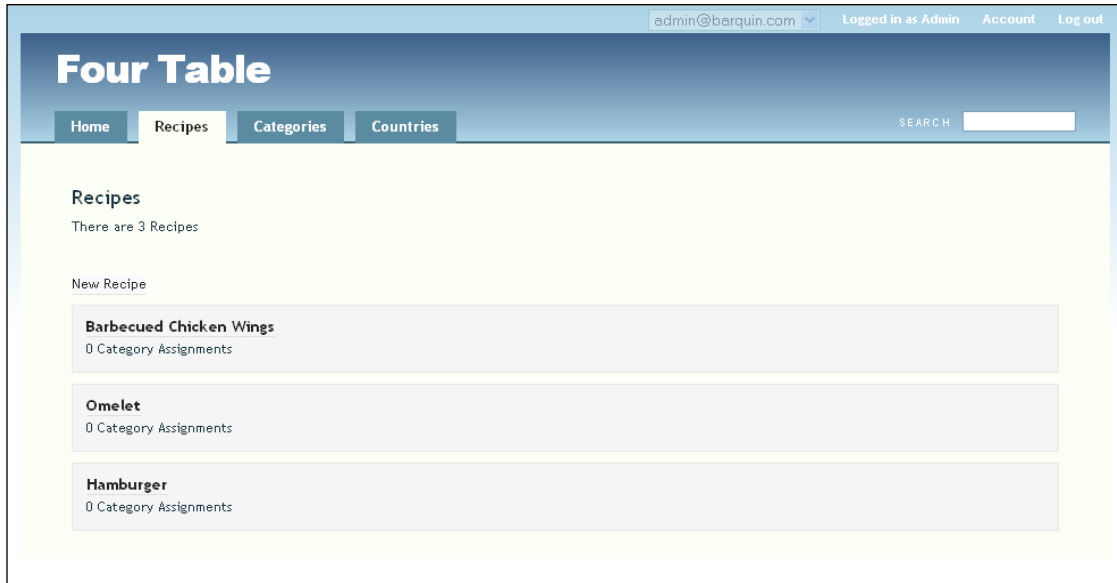


Figure 100: The Recipes Index page

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

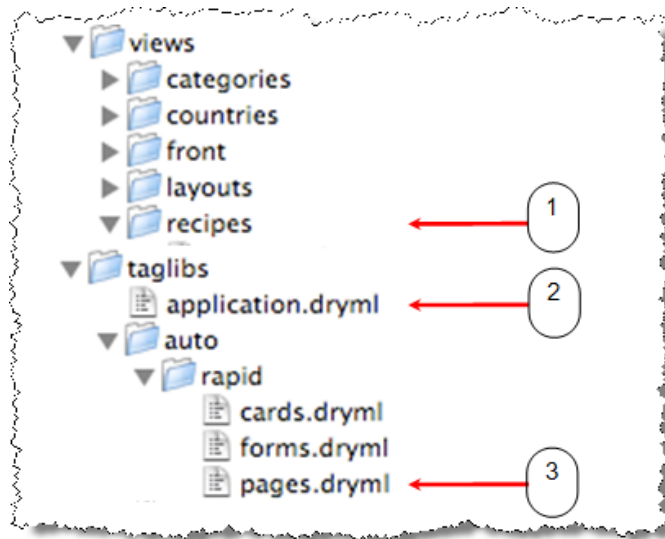


Figure 101 : View of the taglibs/auto/rapid folder

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="app-name">Four Table</def>
<def tag="index-page" for="Recipe">
. . .
```

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

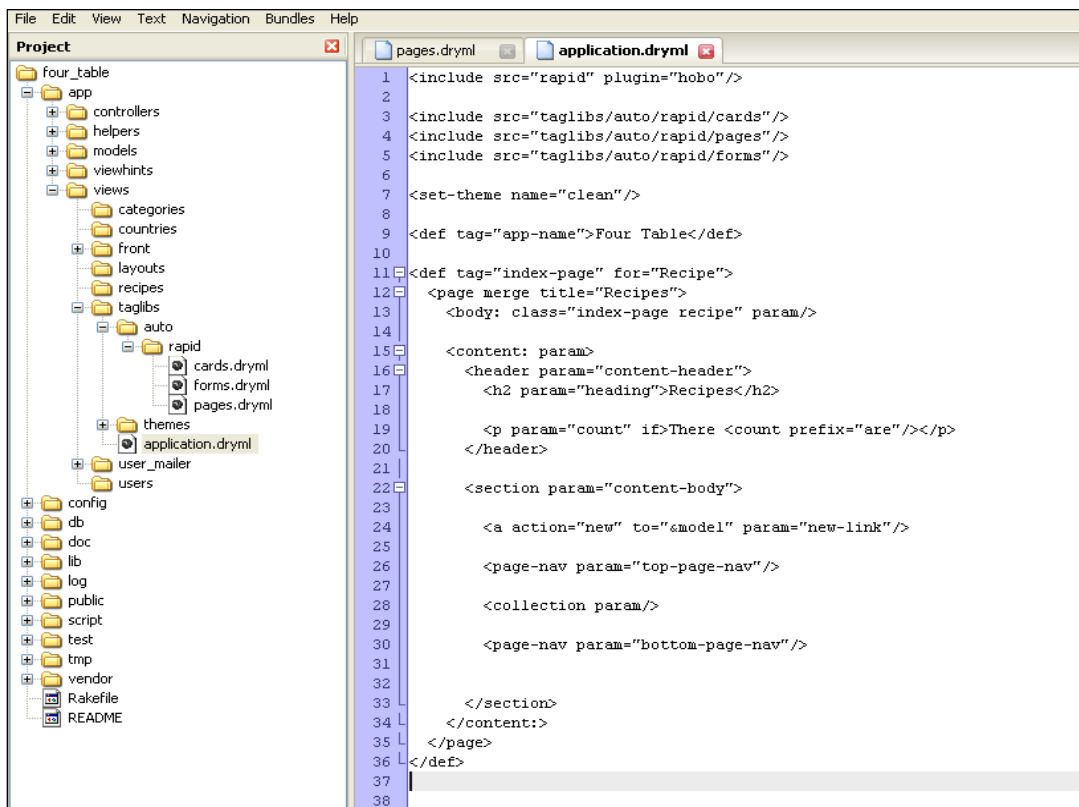


Figure 102: Adding the definition of index-page into the application.dryml file

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`, 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 ***bold italics*** below is what has changed.

```
<def tag="index-page" for="Recipe">
  <page merge title="Recipes">
    <body: class="index-page recipe" param/>

    <content: param>
      <header param="content-header">
        <h2 param="heading">My Recipes</h2>

        <p param="count" if>There <count prefix="are"/></p>
      </header>

      <section param="content-body">

        <a action="new" to="&model" param="new-link"/>

        <page-nav param="top-page-nav"/>

        <collection param/>

        <page-nav param="bottom-page-nav"/>
      </section>
    </content:>
  </page>
</def>
```

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



Figure 104: 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”.

Let us describe what happened.

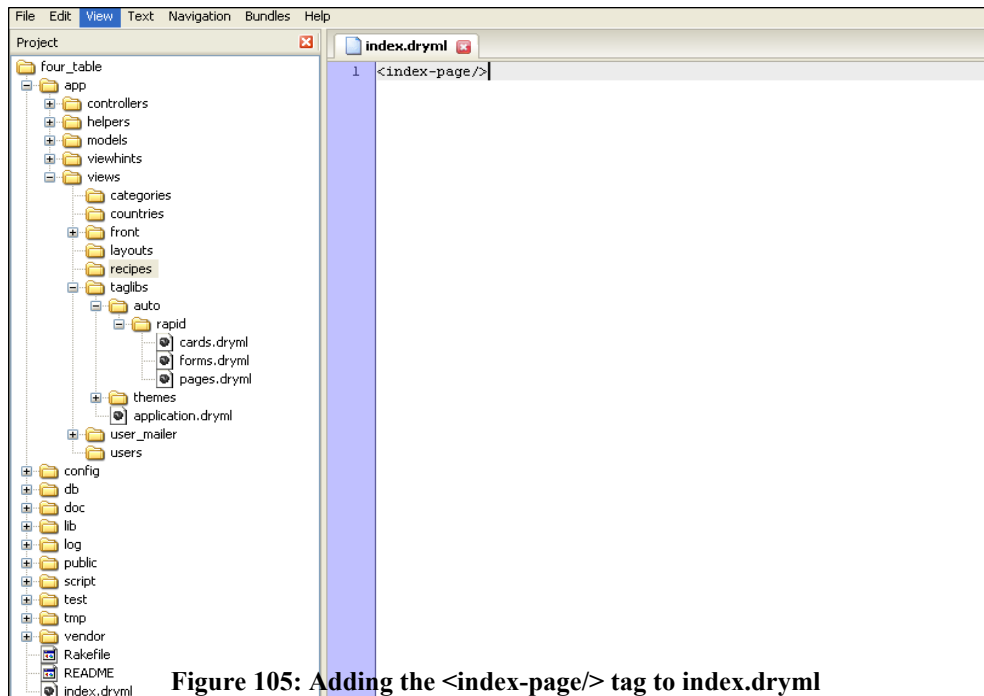
- Step 1: Hobo looked for a template in the `views/recipes/` directory called `index.dryml`.
 - Step 2: Since `views/recipes/index.dryml` did not exist, Hobo next looked in `views/taglib/application.dryml` where it found the tag definition for the index page.
 - Step 3: Hobo used this tag definition 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 105: 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.

Programming 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>
    <count:categories param/>
  </body:>
</card>
</def>
```

Again, we will not explain the detailed syntax of this tag yet. Let's just make a simple change (in ***bold 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/>...test</a></h4>
    </header:>
    <body: param>
      <count:categories param/>
    </body:>
  </card>
</def>
```

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

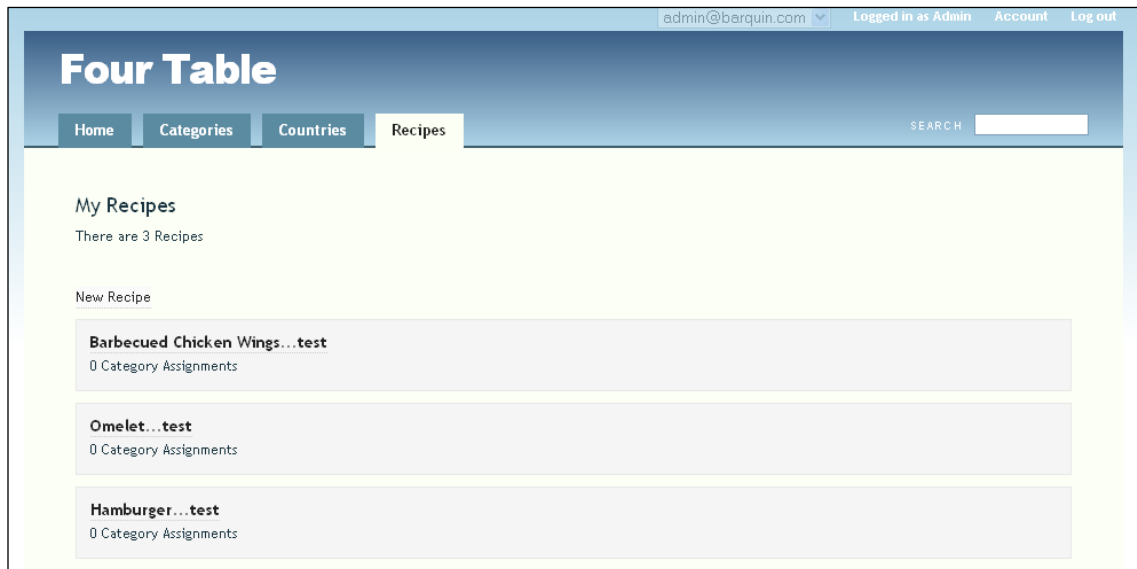


Figure 106: How a change to the `<index-page>` tag affects a collection

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>
    <nav-item href="#{base_url}/">Home</nav-item>
    <nav-item with="%Category">Categories</nav-item>
    <nav-item with="%Country">Countries</nav-item>
    <nav-item with="%Recipe">Recipes</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 ***bold italics***.

```
<def tag="main-nav">
  <navigation class="main-nav" merge-attrs>
    <nav-item href="#{base_url}/">Home</nav-item>
    <nav-item with="%Recipe">Recipes</nav-item>
    <nav-item with="%Category">Categories</nav-item>
    <nav-item with="%Country">Countries</nav-item>
  </navigation>
</def>
```

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



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 `<app-name>` tag definition is found near the top of the `application.dryml` file and is automatically generated from the name when you originally generated the application. Just change the content of the `<def>` tag to what you want.

```
<def tag="app-name">Four Tables, No Waiting</def>
```

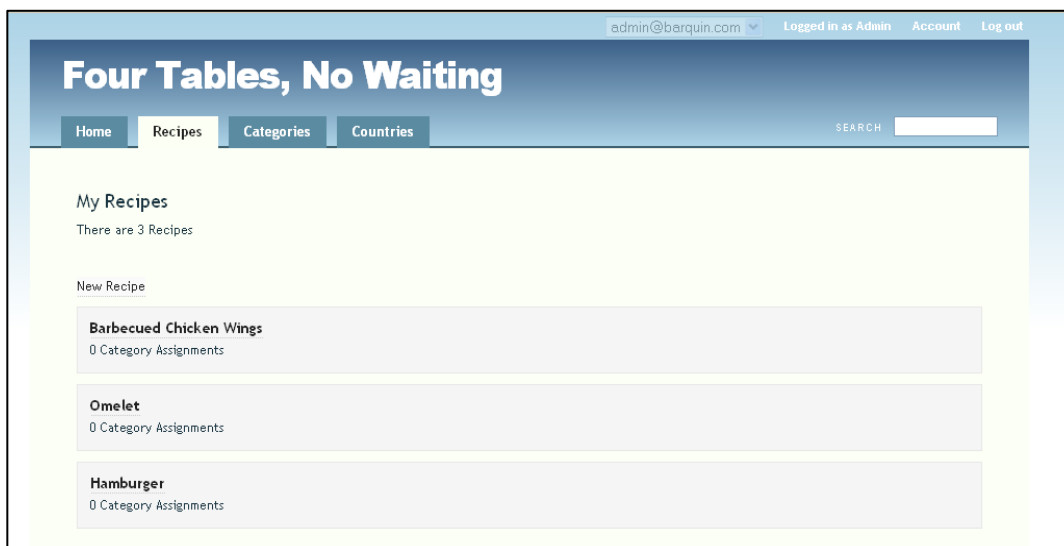


Figure 108: Changing the application name with the `app-name` tag

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).

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`.

Vocabulary Note: Notice the double meaning of parameter in the former sentence. Also, be sure not to confuse the DRYML `param` with the Rails `params` object, which you might know about if you are a Rails programmer.

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/>
```

Programming 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.

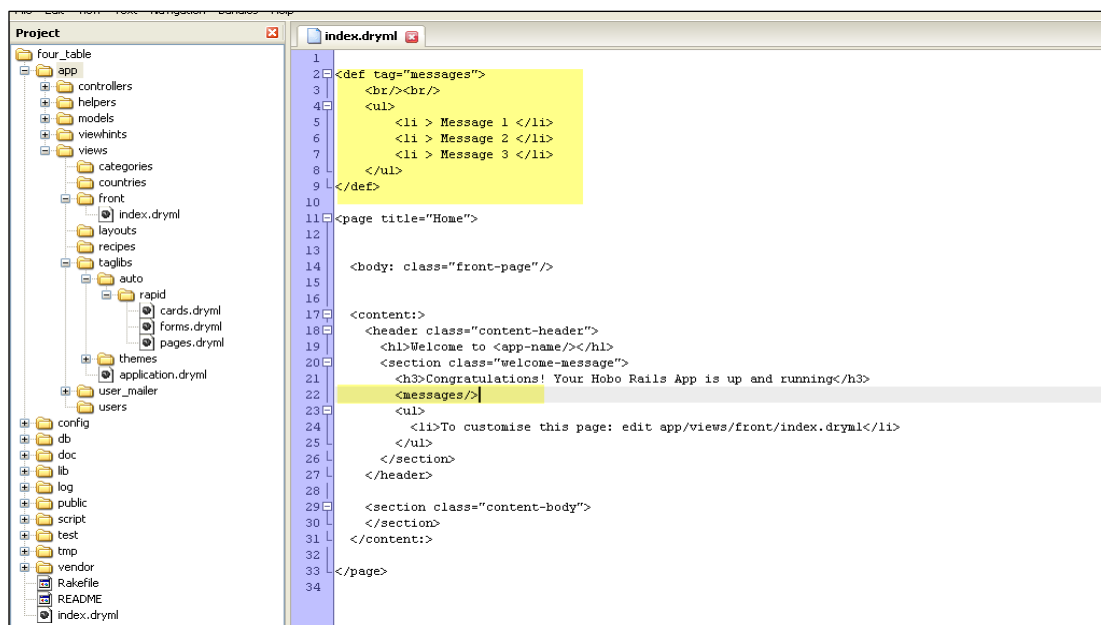


Figure 109: The \views\front\index.dryml file after the first modification

Then refresh your browser:

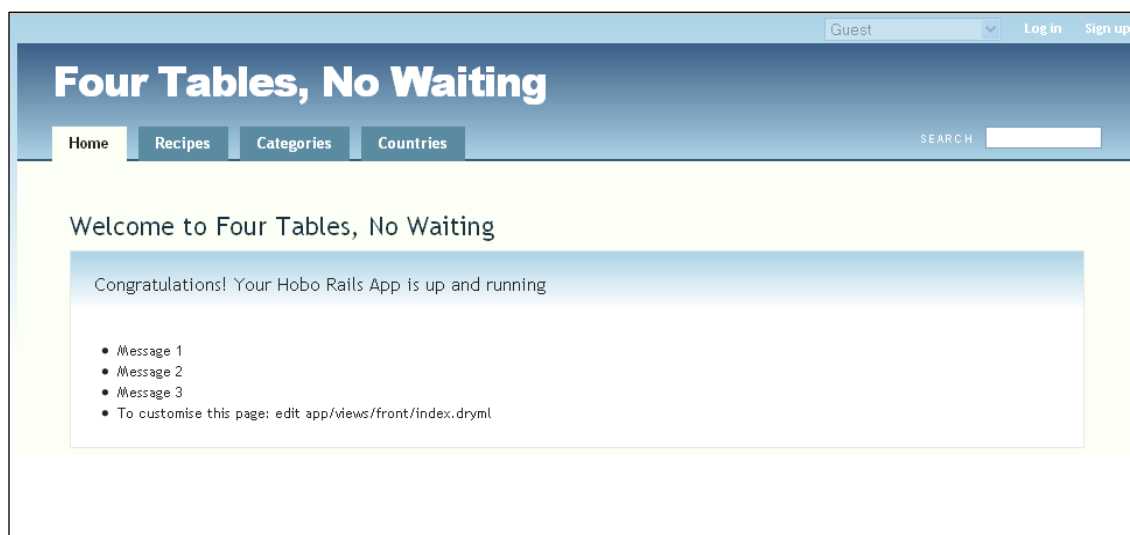


Figure 110: The Home page with the first set of custom messages

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 invoke a tag such as `<index-page>` explicitly.

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 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 invoke them explicitly.

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.

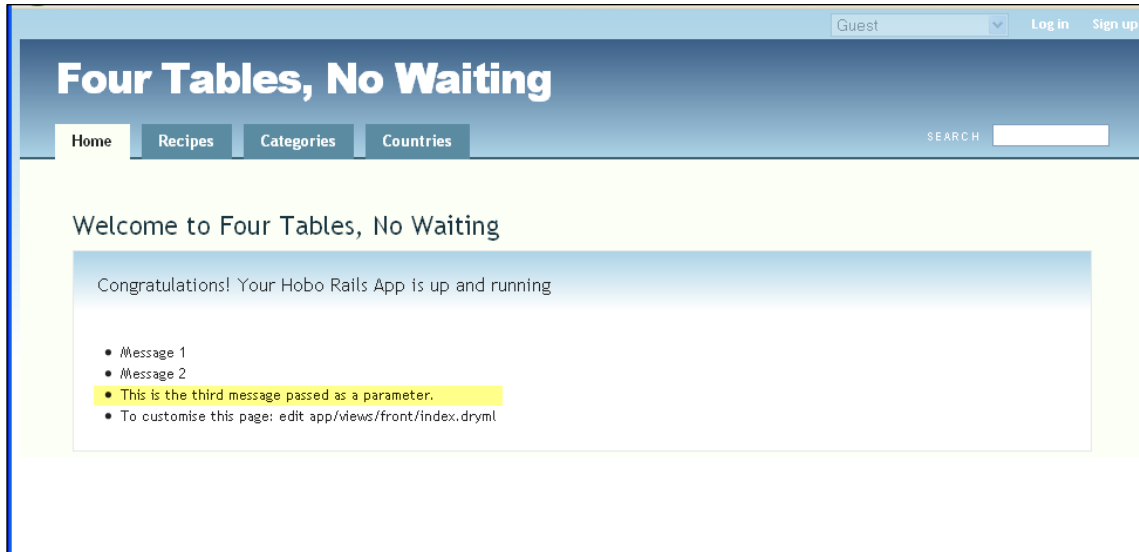


Figure 112: How the passed parameter displays on the page

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:

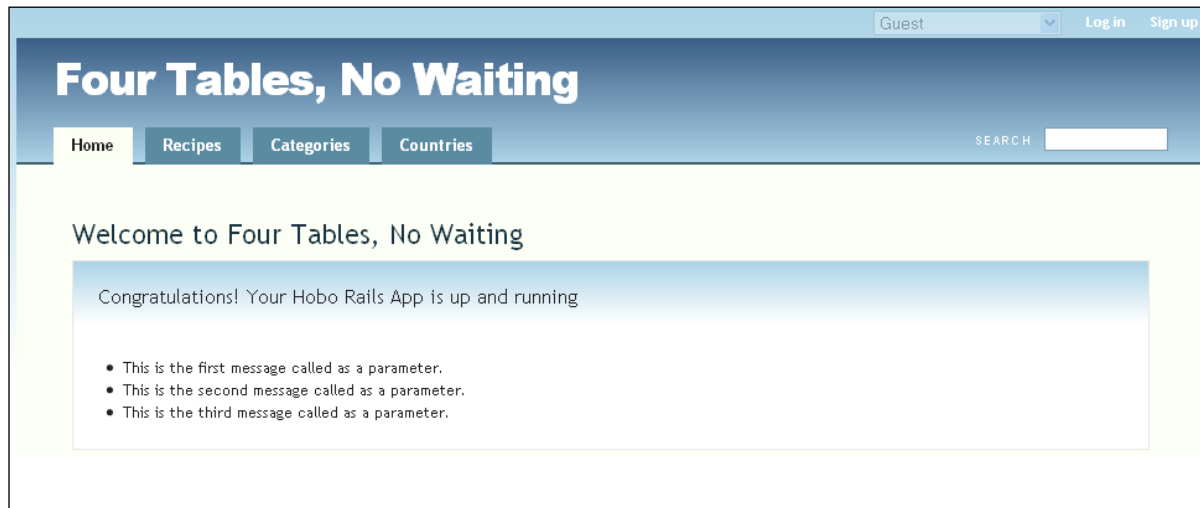


Figure 113: Passing three parameters to your <messages> tag

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. Next 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.

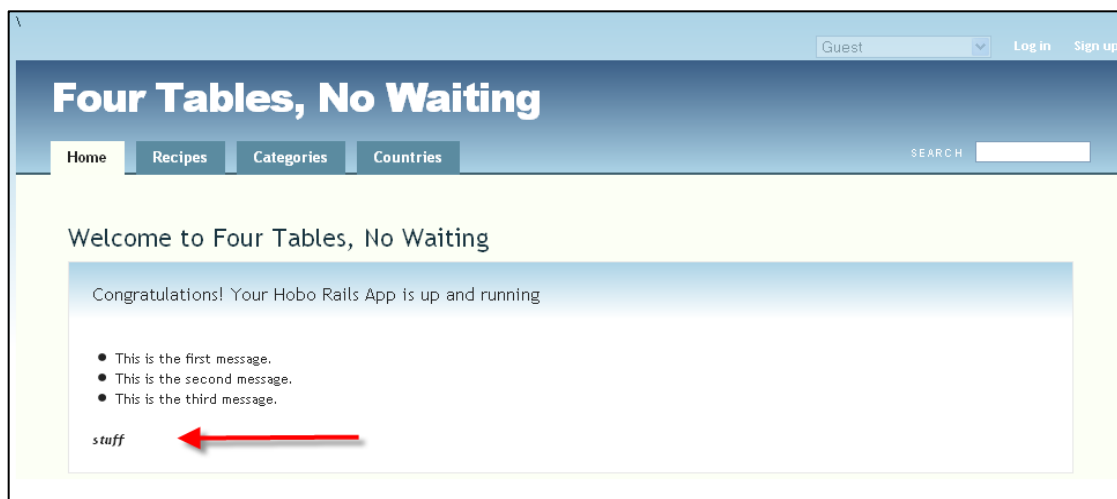
Go back to the top of the `views/front/index.dryml` file and enter the following code after the first `<def>` . . . `</def>` tags.

```
<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’ content with something you might want to format.

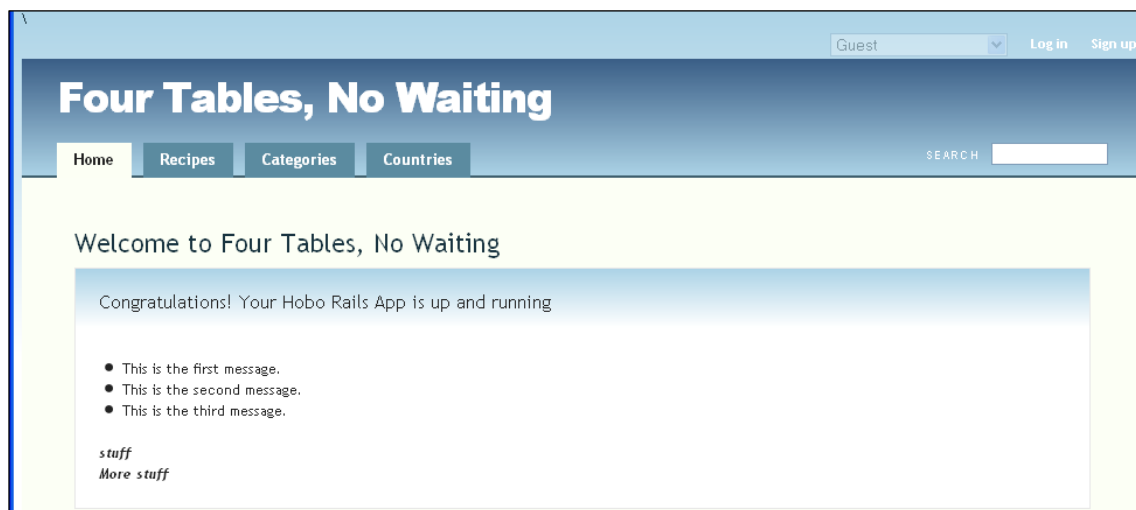
Call the `<bd-it>` tag right after the closing `</messages>` tag without using the `<span:>` parameter. This will demonstrate that the tag will just emit the formatted default word *stuff*.

```
<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/>
```

Figure 114: Page display using your custom `<bd-it>` tag

If you use the `` parameter tag, you will format your content.

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

Figure 115: Calling `<span:>` explicitly within to your `<bd-it>` tag

But the second line is a kind of clumsy looking 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>
```

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 call it. Now 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 and only emit the default content if you call 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 second message.</msg2>
          <msg1:>This is the first message.</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

You will go 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

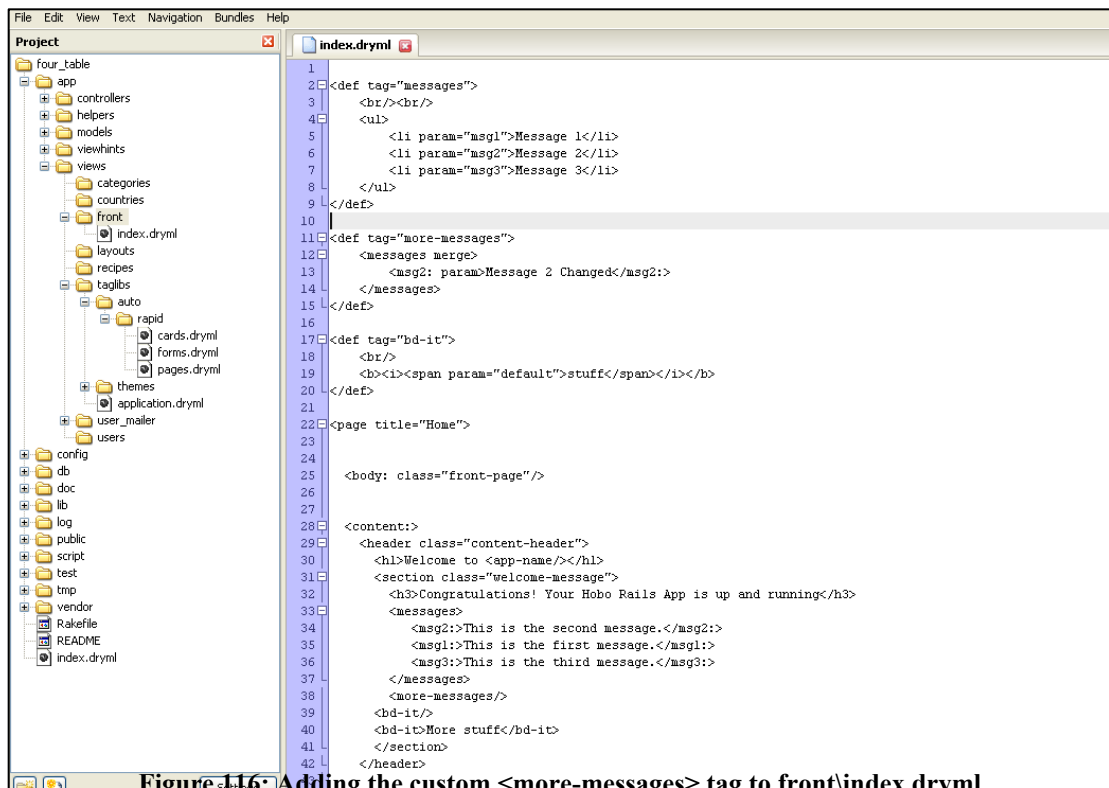
- Defining tags from tags using the `merge` tag
- Defining tags from tags using the `extend` tag
- Replacing tag parameters (not tag content)

1. **Define a tag based on another tag: Method 1.** 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.)

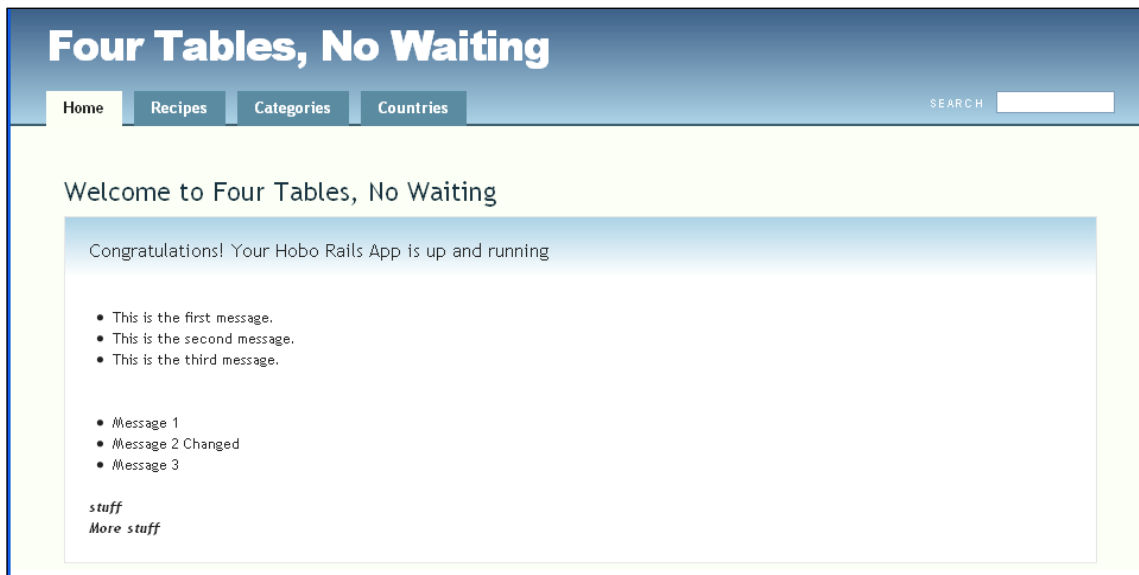
```
<def tag="more-messages">
  <messages merge>
    <msg2: param>Message 2 Changed</msg2:>
  </messages>
</def>
```

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. Now let's invoke this tag. Place the following code below your last code from the previous tutorial.

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

Figure 116: Adding the custom `<more-messages>` tag to `front/index.dryml`

Refresh your browser to see the change the below.

Figure 117: 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: Method 2.** 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.

Now let’s 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. But 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-mesagex" alias-of="messagex"/>
```

And then extend “new-mesagex” with the functionality you need.

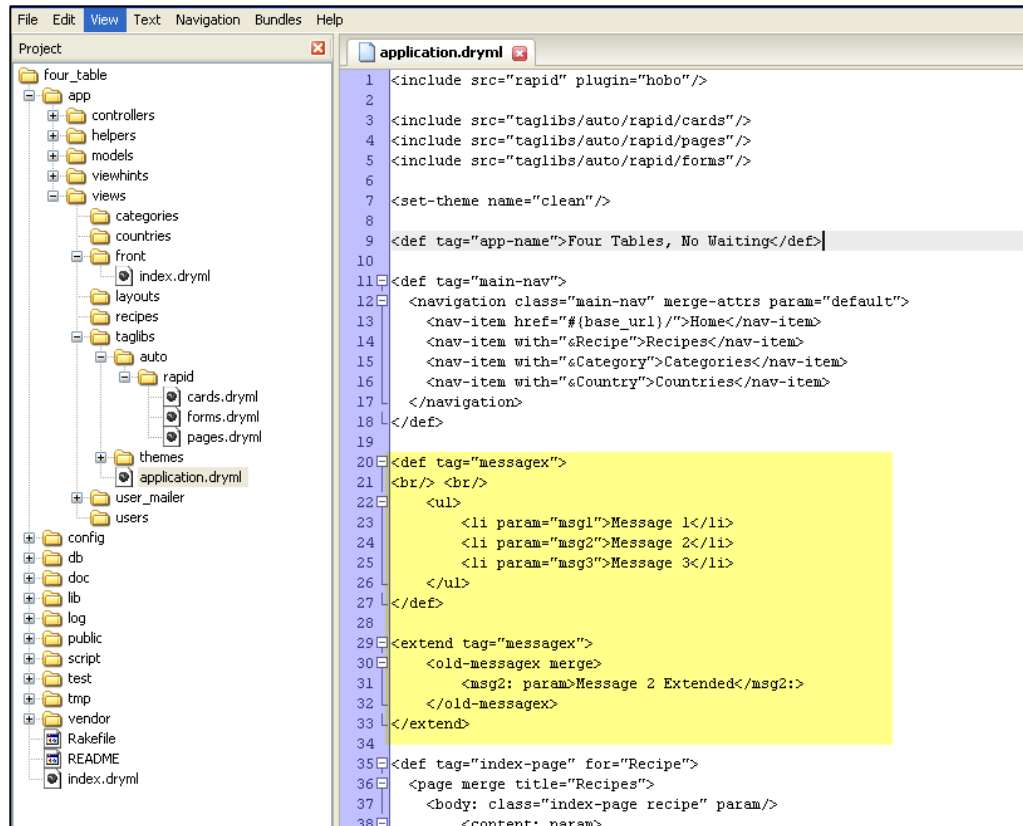


Figure 118: Extending the tag <messagex> in application.dryml

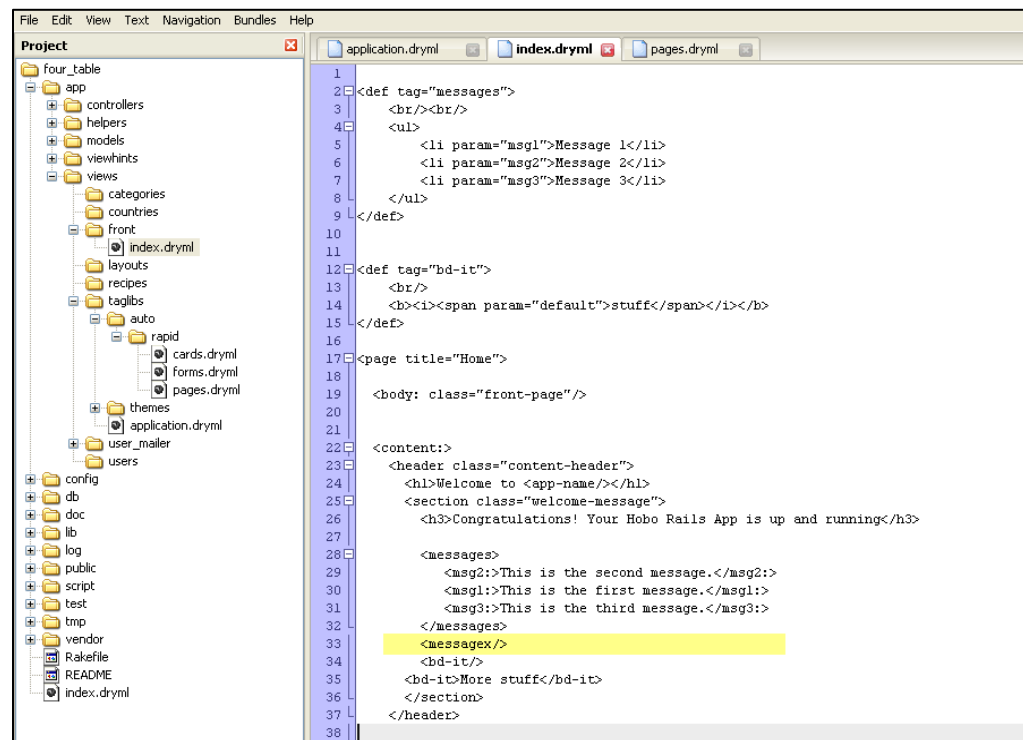


Figure 119: Using the extended <messagex> tag

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 `<message>` just like the old `<messages>` tag. We then extended it so that it would look just like the

`<more-messages>` tag from Step 1.

Now call the `<message>` tag in `front/index.dryml` to confirm that it yields output like the `<more-messages>` tag.

```
<message/>
```

You should see the following rendering:

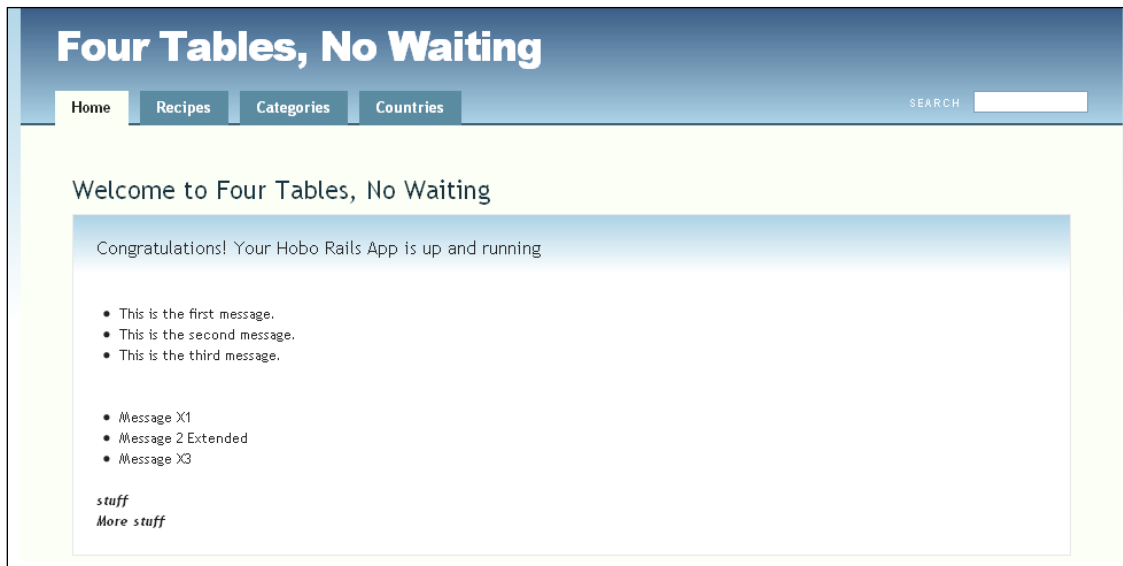


Figure 120: Page view of the next additions to `<message>`

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 `<message>` 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. First we will 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.

Next we will show you that 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.

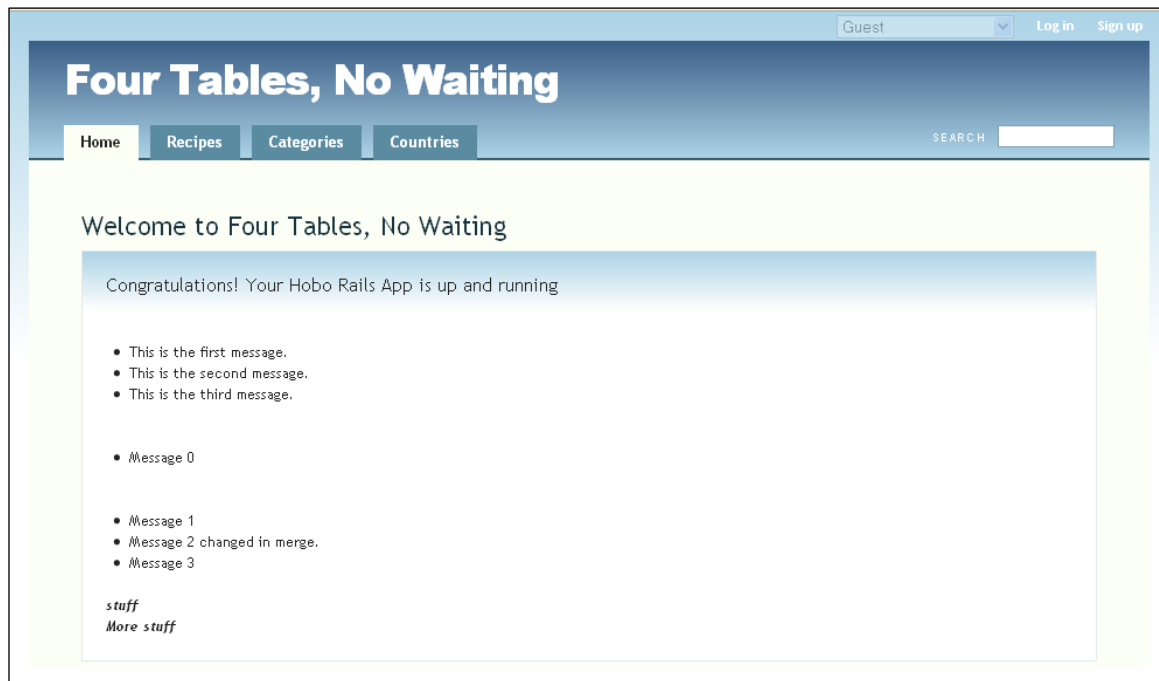


Figure 121: Page view of the `<more-messages>` tag usage

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>
```

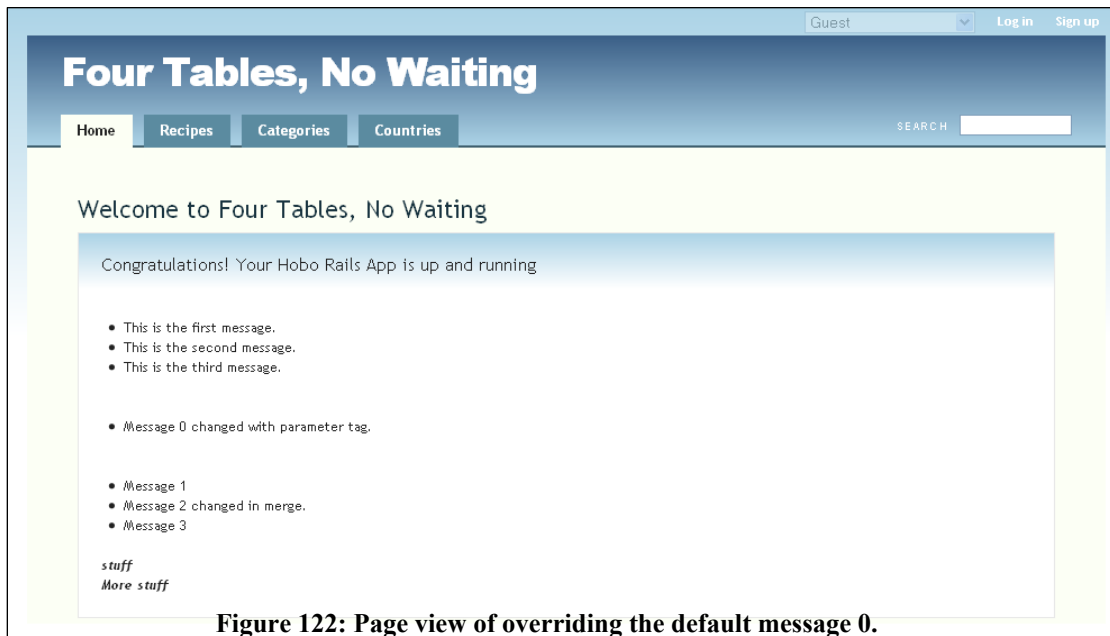


Figure 122: Page view of overriding the default message 0.

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.

Now let's 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>
```

Now let's 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>
```

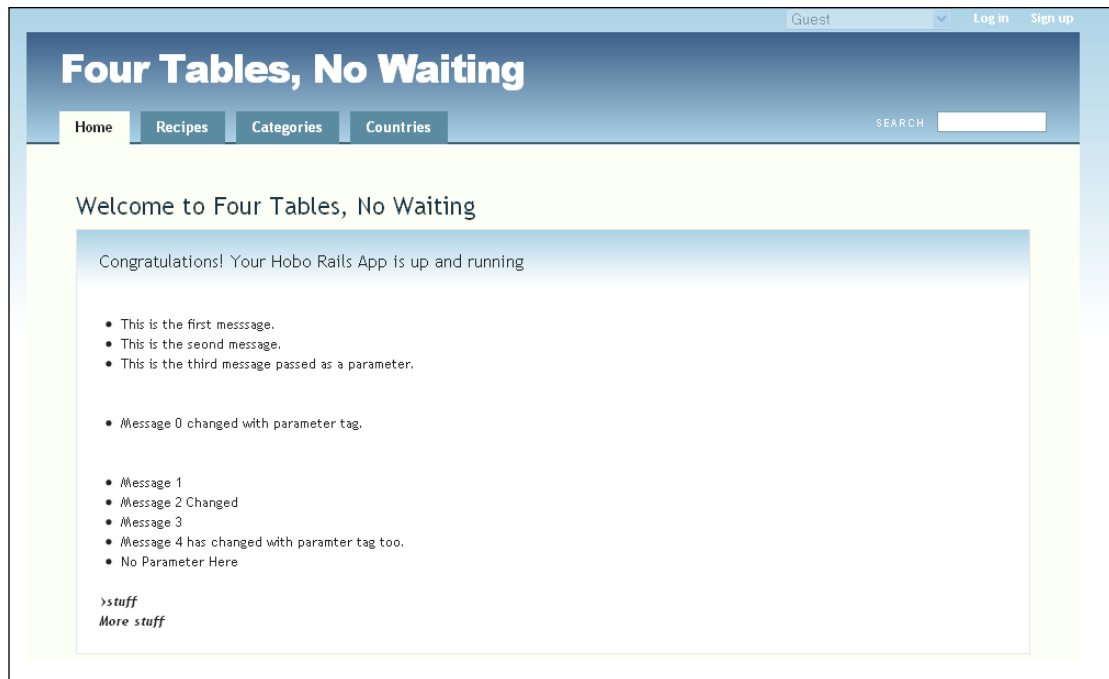


Figure 123: More parameter magic

Tutorial 12 – Rapid, DRYML and Record Collections

You will learn how to create a new index page that will replace the default index page that Hobo generates on the fly, and 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).

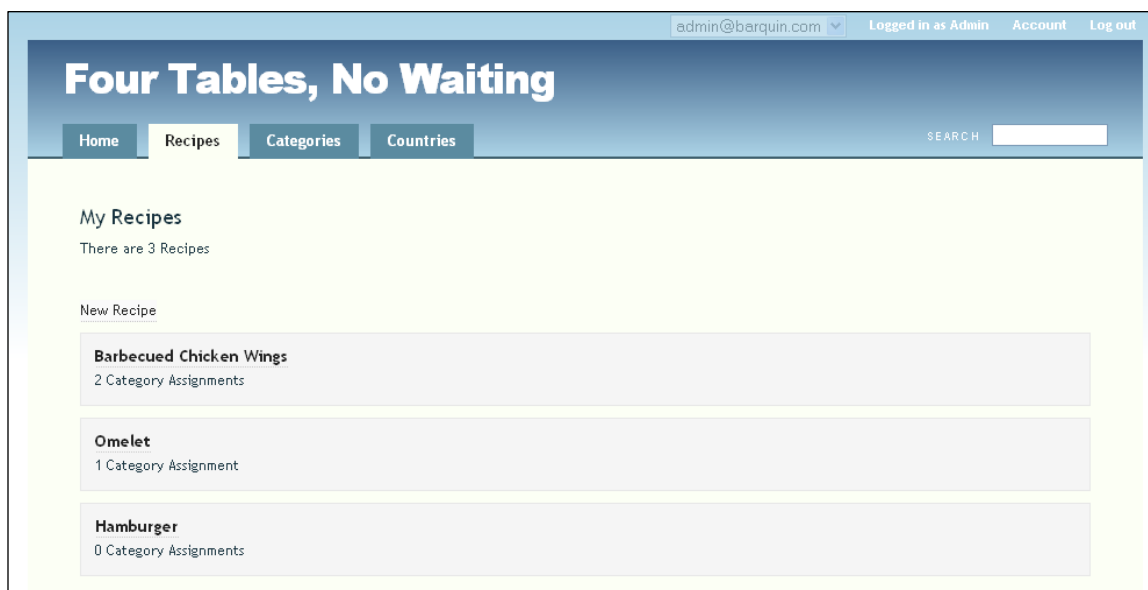


Figure 124: The Four Tables application as we left it

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_migration_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/>
```

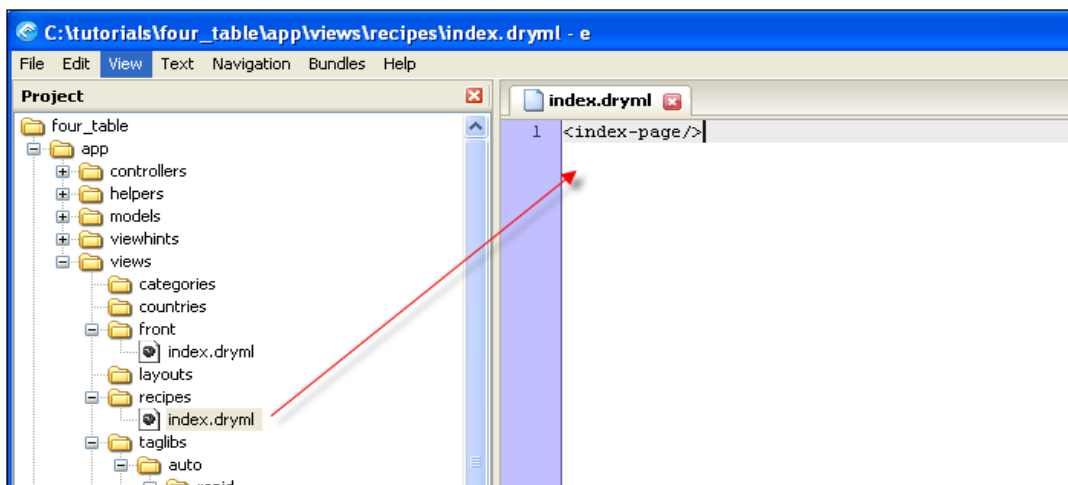


Figure 125: Creating the `/views/recipes/index.dryml` file

Refresh your browser. It should look just like it did in Step 1. This is because `<index-page>` is exactly the 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 it 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 bold* below.

```
<def tag="index-page" for="Recipe">
  <page merge title="Recipes">
    <body: class="index-page recipe" param/>

    <content: param>
      <header param="content-header">
        <h2 param="heading">Recipes</h2>

        <p param="count" if>There <count prefix="are"/></p>
      </header>

      <section param="content-body">

        <a action="new" to="&model" param="new-link"/><br/>

        <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.

Now let's 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.

OK, let's learn a little more about the `<collection>` tag. 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, that is, nothing.

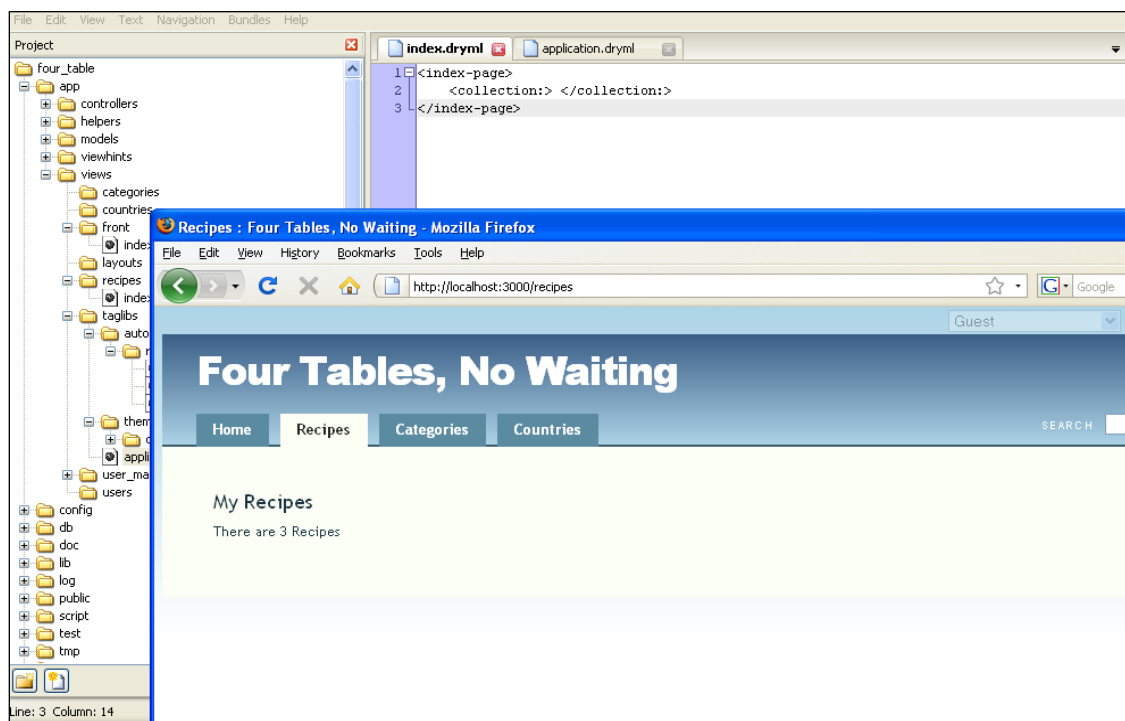


Figure 126: page view of using a blank "<collection:></collection:>" tag

Now 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.

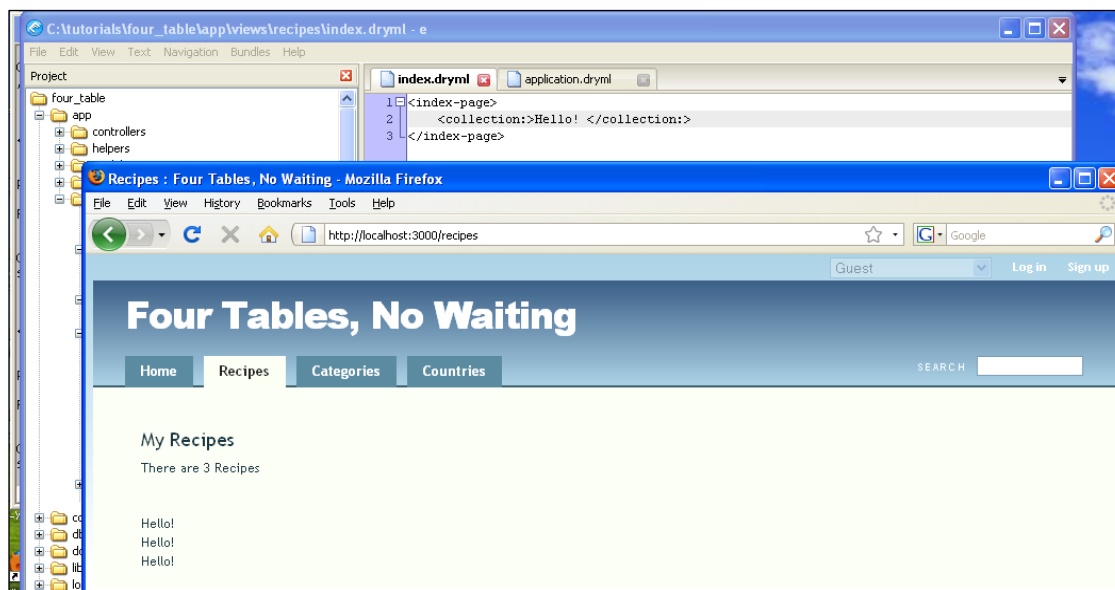


Figure 127: How the <collection> tag iterates

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.

Now 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:>
```

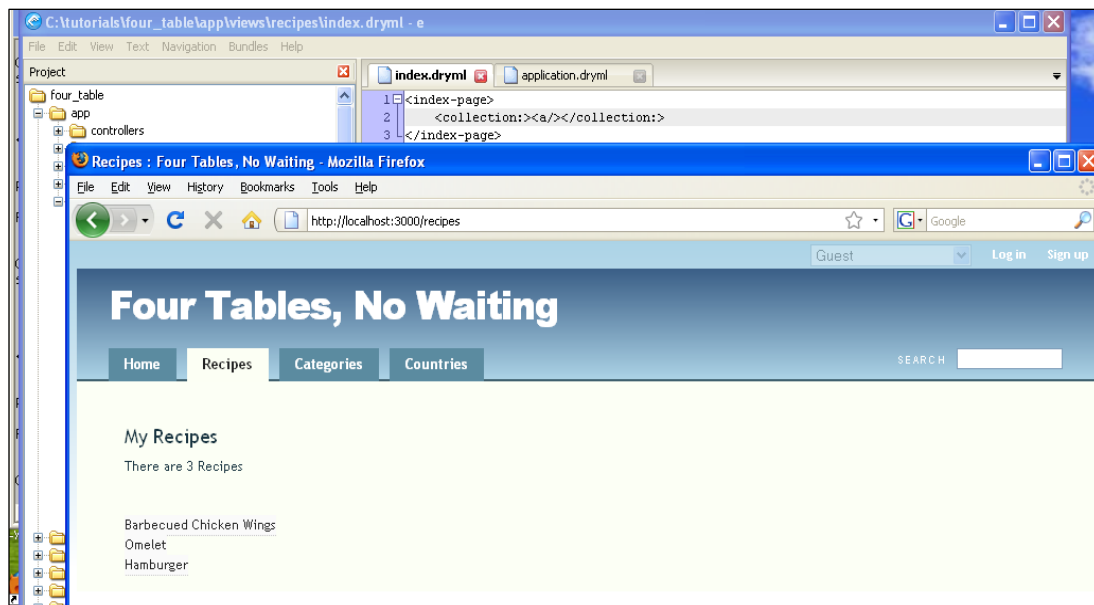


Figure 128: Using the `<a/>` hyperlink tag within a collection

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>
      <count:categories param/>
    </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. Right now 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:

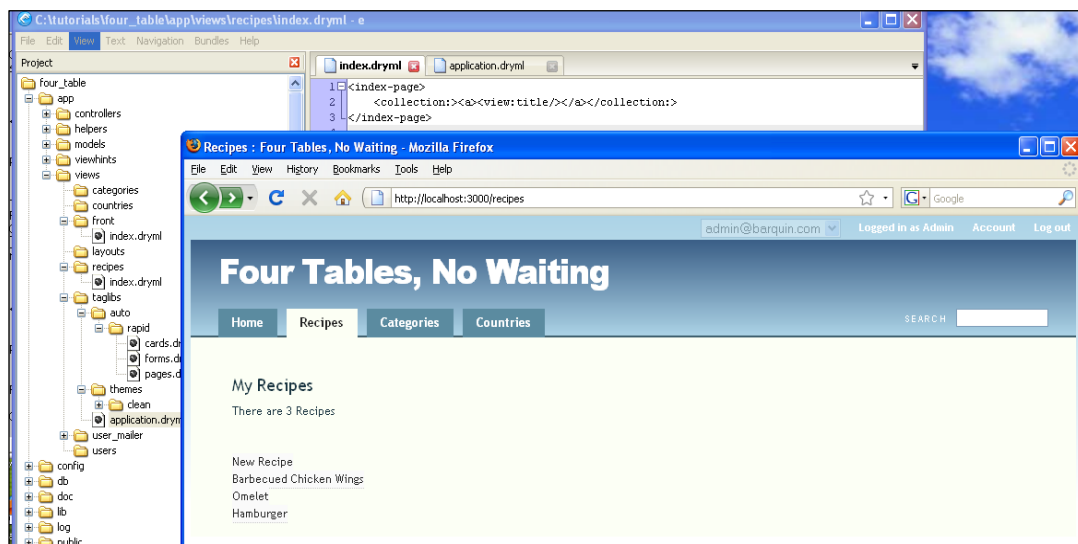


Figure 129: Specifying what `<collection>` tag will display

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.

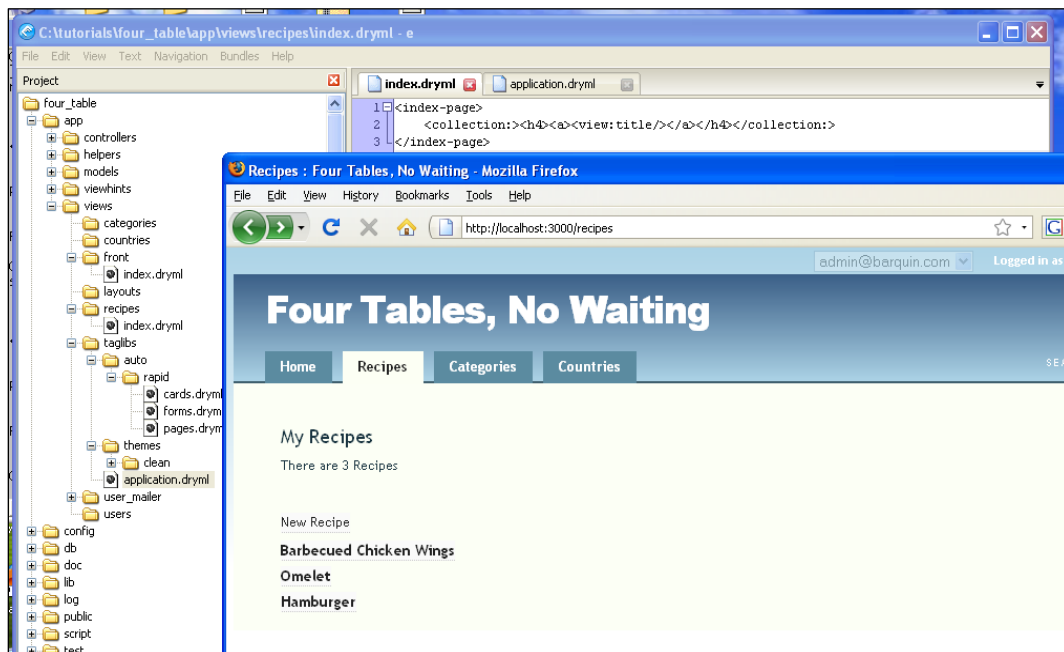


Figure 130: Changing the display style within `<collection>`

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. It would be nice to see this relationship without having to click through to 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>
```

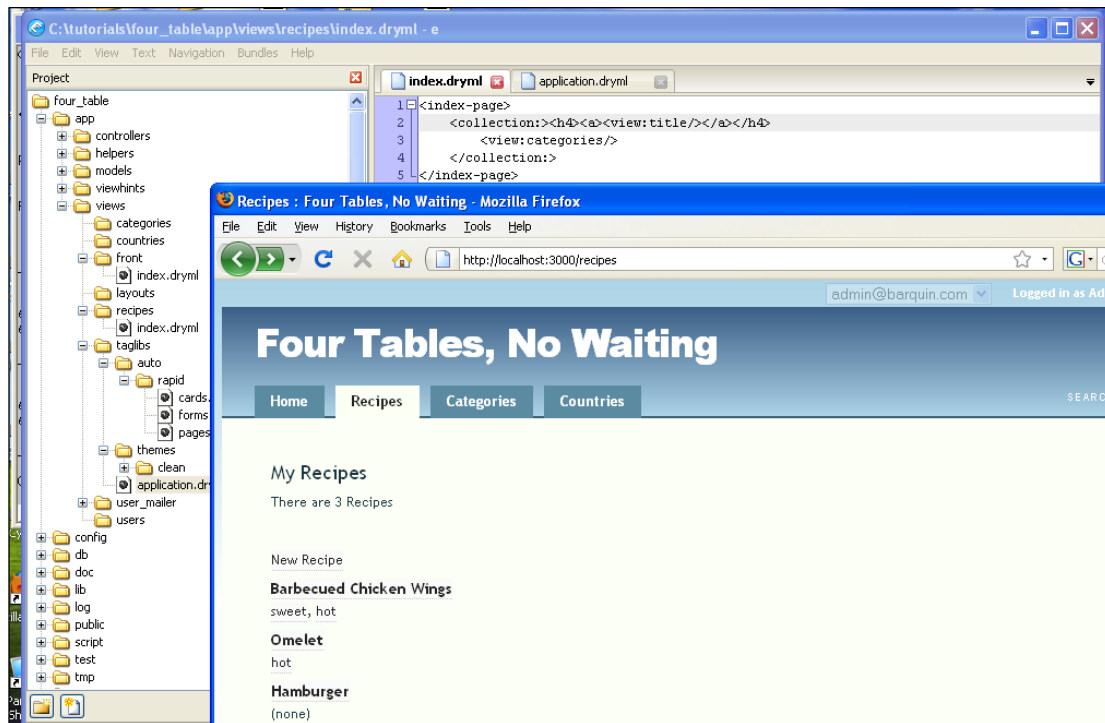



Figure 131: Changing the implicit context within `<collection>`

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 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:

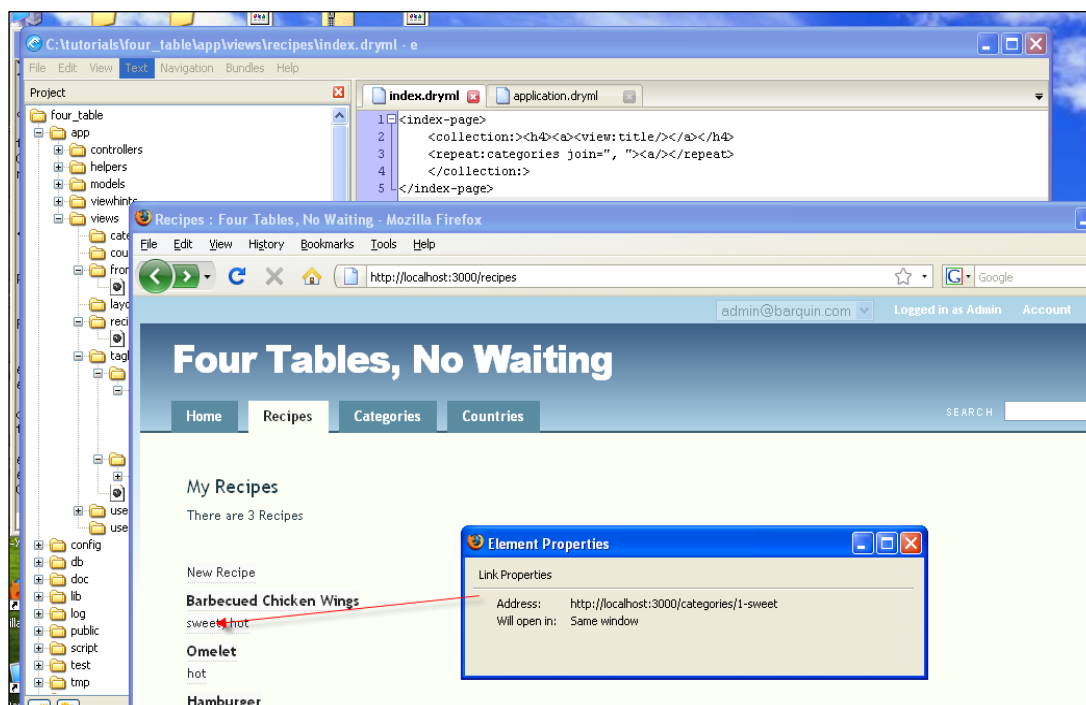


Figure 132: Creating comma-delimited multi-valued lists in a <collection>

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/>
```

Now 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 below; note 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>
      <count:categories param/>
      <br/><view:categories/>
    </body:>
  </card>
</def>
```

Refresh your browser.

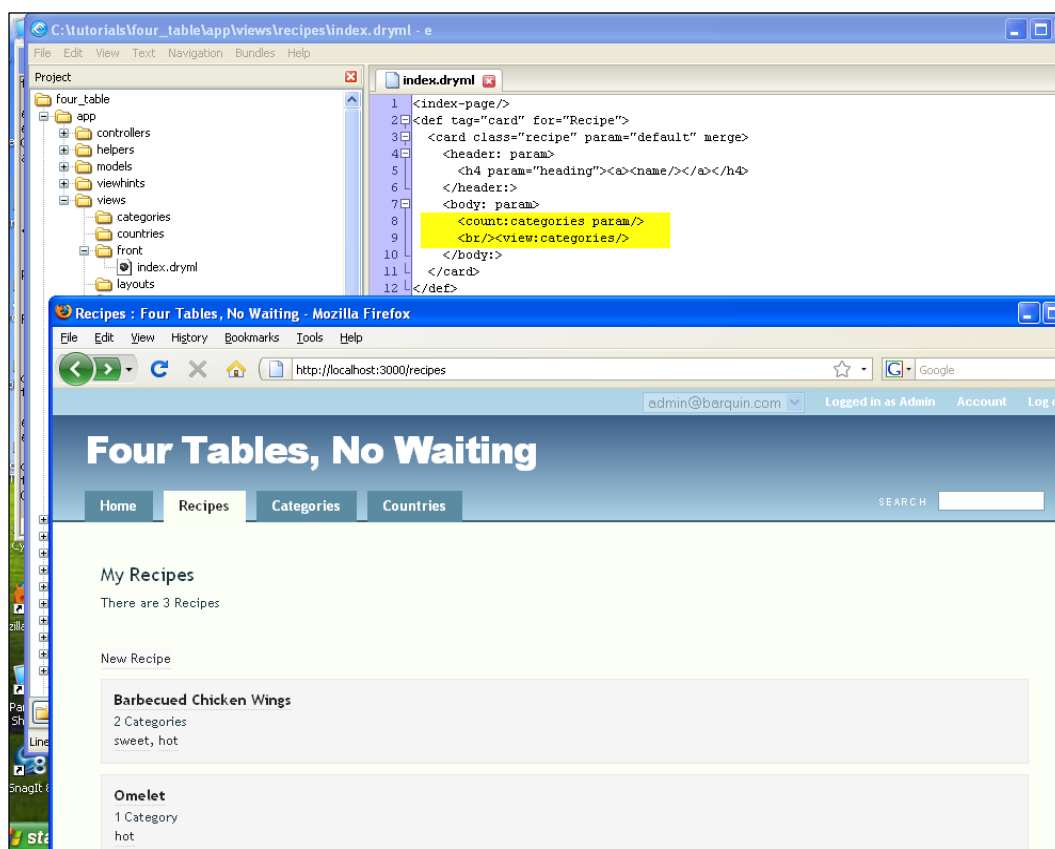


Figure 133: Adding the count of values in the `<card>` tag

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>
```

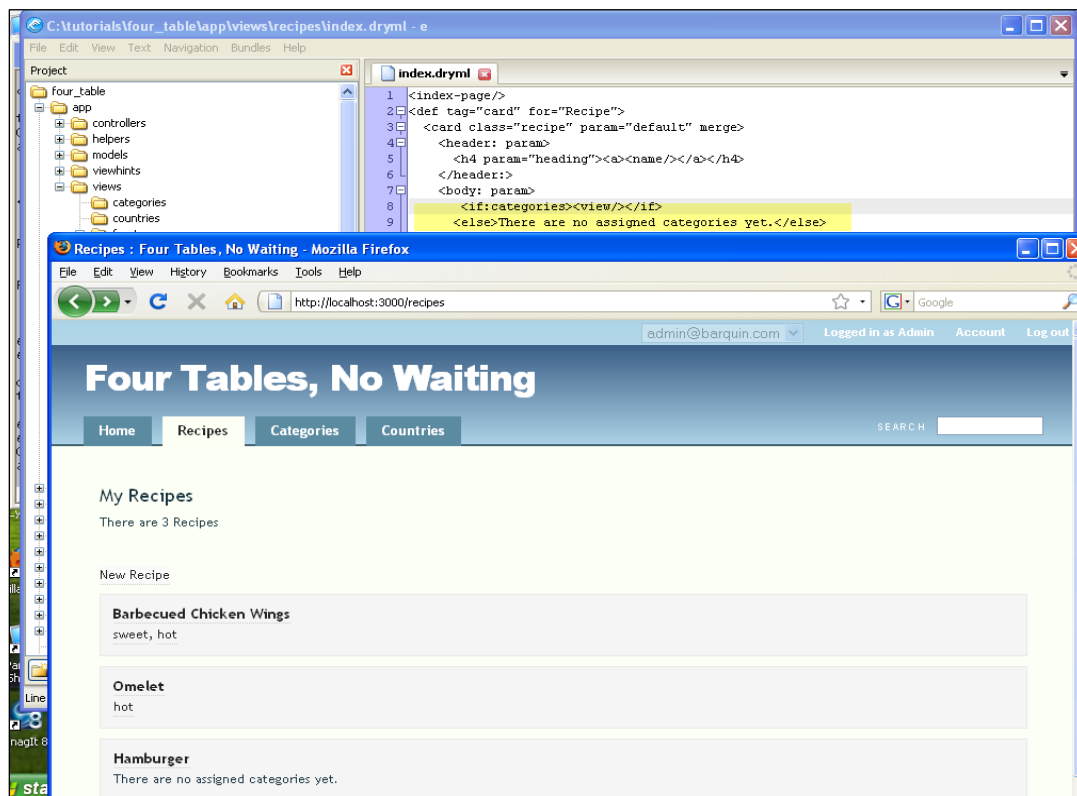


Figure 134: Using "if---else" within a tag to display a custom message

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

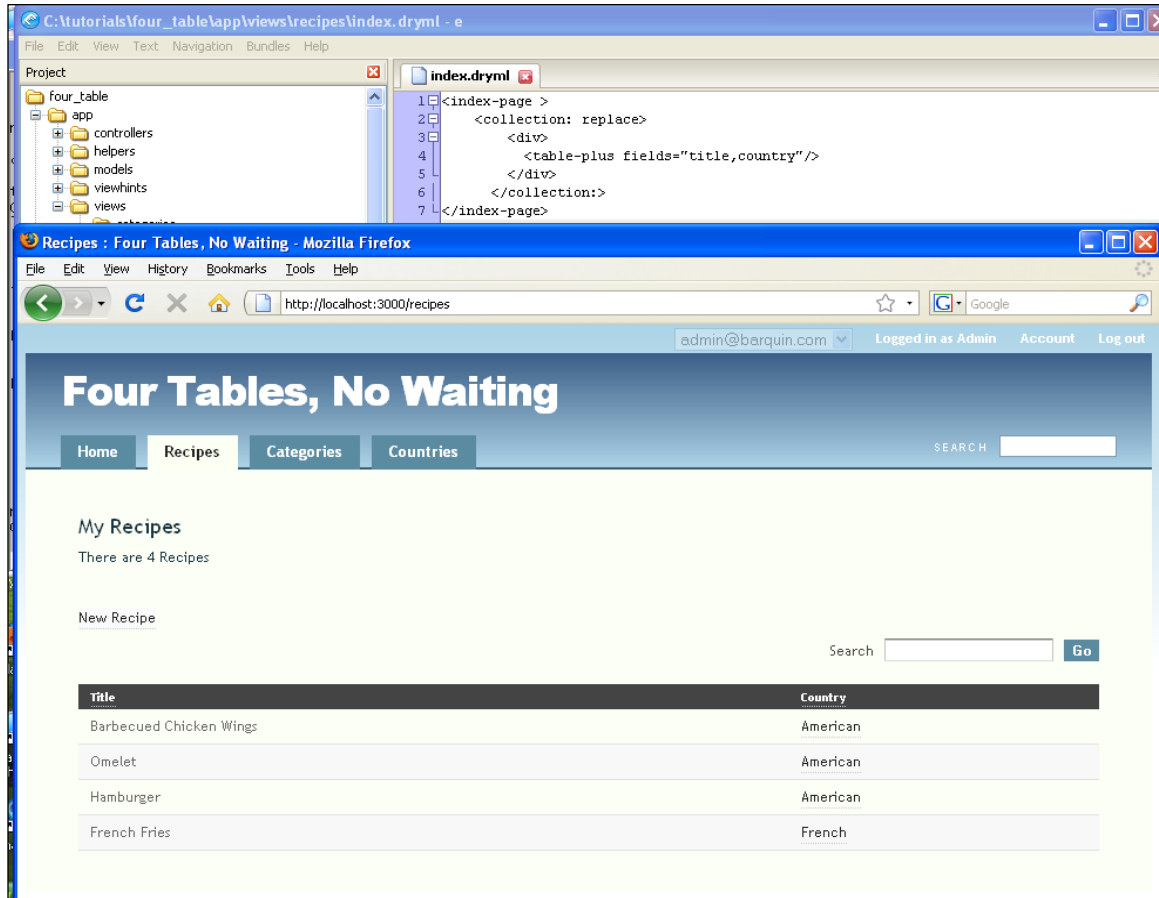
- Display model data in table form.
- Use the *replace* attribute to change the content of a parameter tag.
- Display associated record counts in the table
- Add search and sort to the table.

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 135: 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>
```

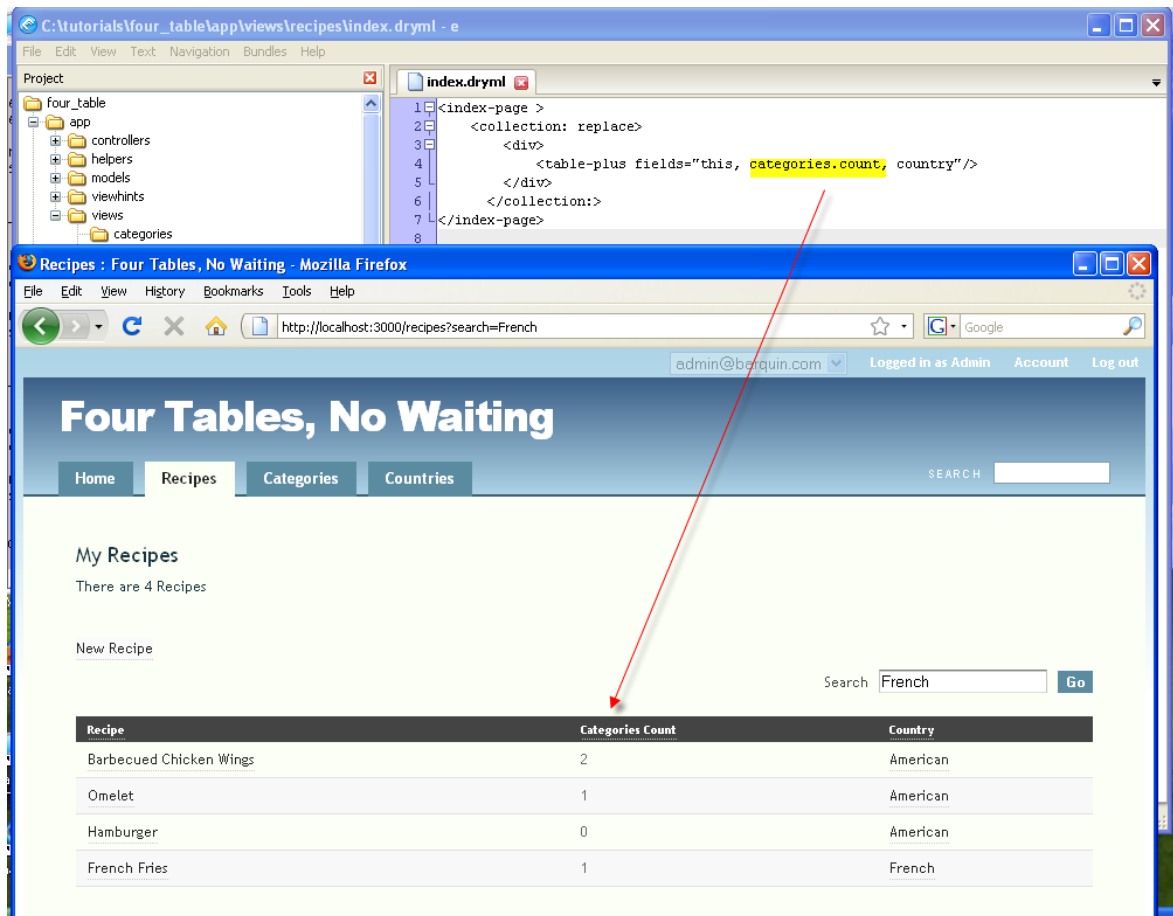


Figure 136: Adding a "Categories Count" to <table-plus>

That was pretty straightforward. Before we refresh our browser again, let's also 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>
```

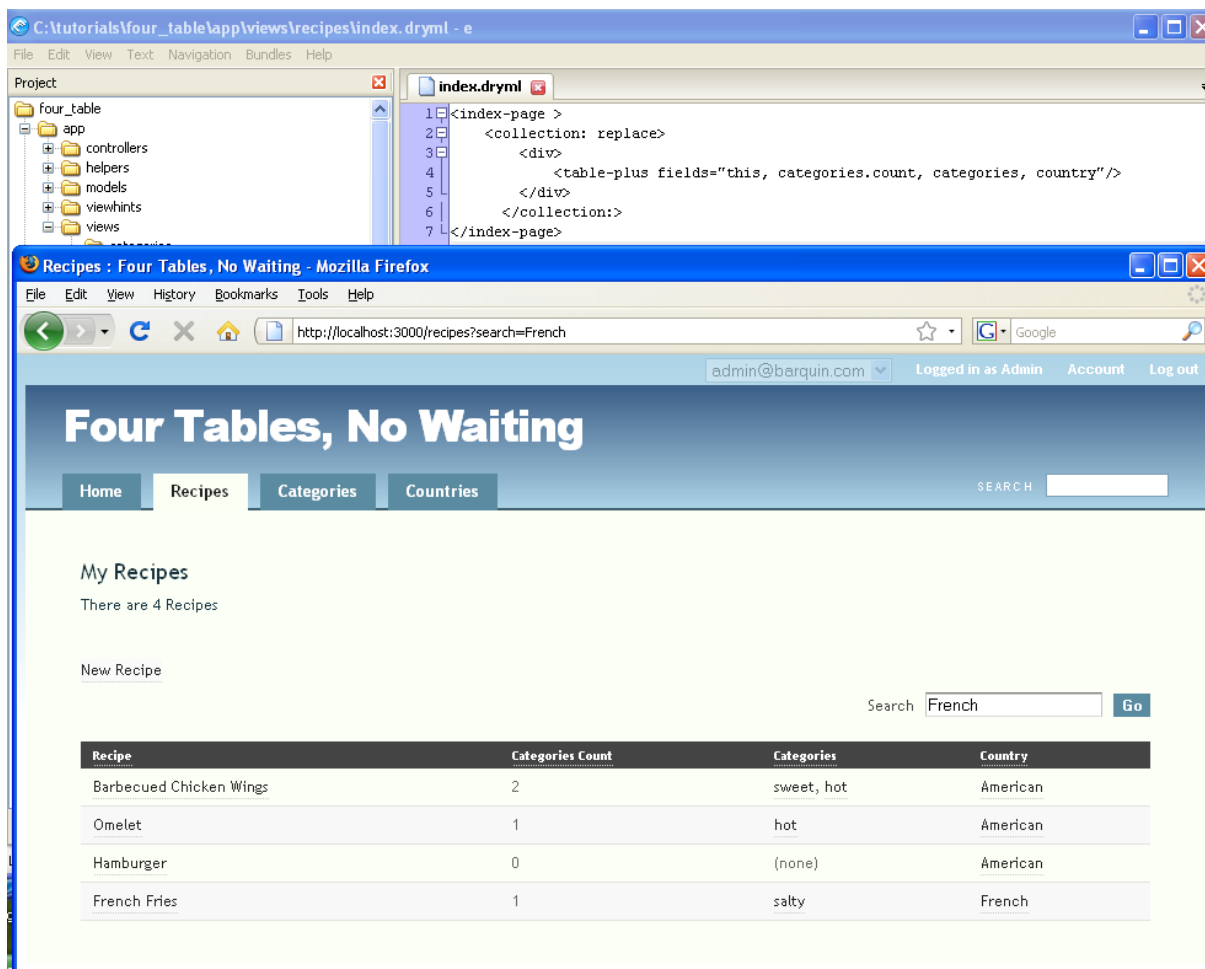


Figure 137: Adding a comma-delimited list within a `<table-plus>` column

4. **Add search and sort capability to the table.** Until now we have worked with controllers relatively little. If you think about it a bit, you will quickly realize that to add search and sort, we will have to make a change in the *recipe* controller. You can 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.

Programming 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),
    :order_by => parse_sort_param(:title, :country, :count)
end
```

Note that 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:

The screenshot shows a web application interface with a navigation bar at the top containing 'Home', 'Recipes', 'Categories', and 'Countries' tabs. A search bar is located on the right of the navigation bar. Below the navigation bar, the 'My Recipes' section displays 'There are 4 Recipes' and a 'New Recipe' link. A table of recipes is shown, sorted by country in descending order. A red arrow points to the 'Country' header of the table, with the text 'Sorted by Country, descending...' above it. A search bar with a 'Go' button is also visible.

| 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 138: adding a search facility to <table-plus> using Hobo’s `apply_scopes` method

Now search/filter by “French” in the title or body:

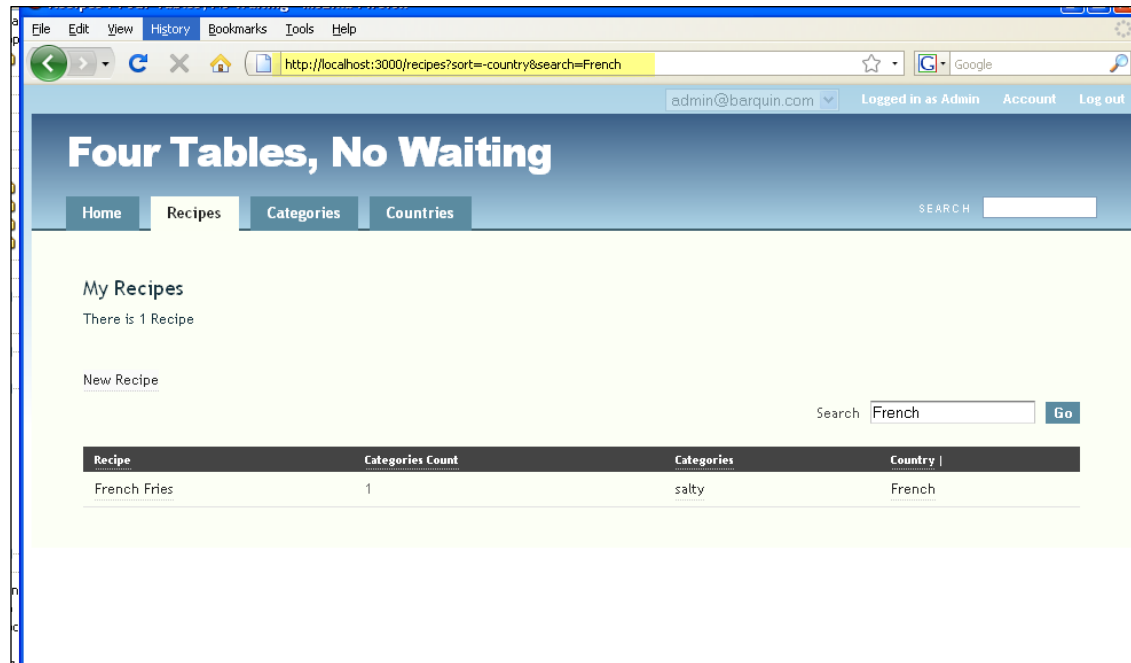


Figure 139: Found Recipes searching for "French"

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

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="Recipe">

    <body: class="show-page recipe" param/>

    <content: param>
      <header param="content-header">
        <h2 param="heading"><name/></h2>

        <field-names-where-true fields="" param/>

        <a action="edit" if="&can_edit?" param="edit-link">Edit Recipe</a>
      </header>

      <section param="content-body">
        <view:body param="description"/>
        <field-list fields="country" param/>
        <section param="collection-section">
          <h3 param="collection-heading">Categories</h3>

          <collection:categories param/>
        </section>
      </section>
    </content:>

  </page>
</def>
```

We are going to focus in on three display components of this tag, noted in bold italics above, to help you understand how to change the display of individual records.

Click on the Recipes tab and then click on an individual recipe.

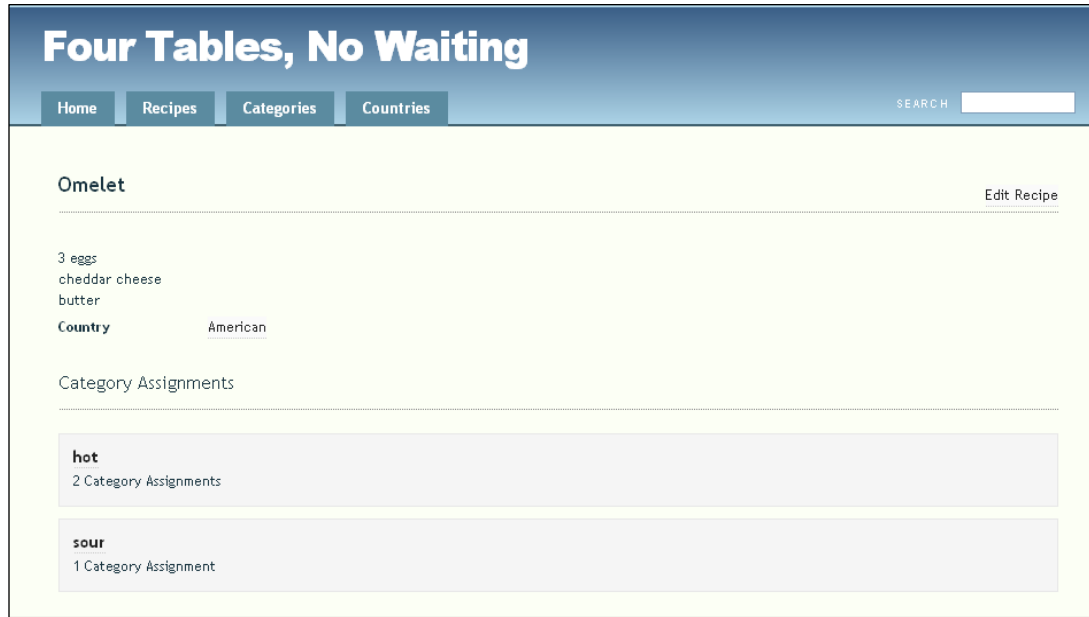


Figure 140: The Recipe show page before modification

Now comment out the three lines above in bold italics using `<!-- ... -->`, and confirm that you have removed the display of the individual recipe record.

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:



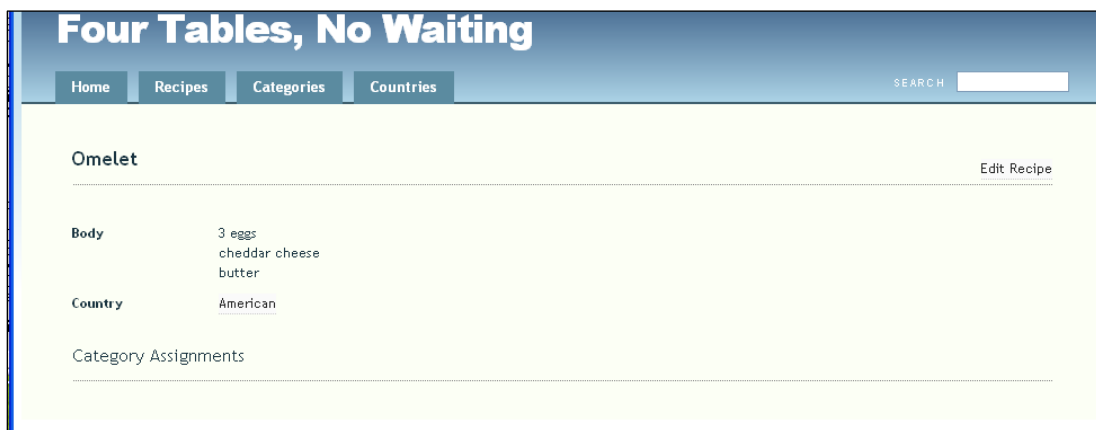
Figure 141: Recipe show page after removing three critical lines of code

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 142: 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.

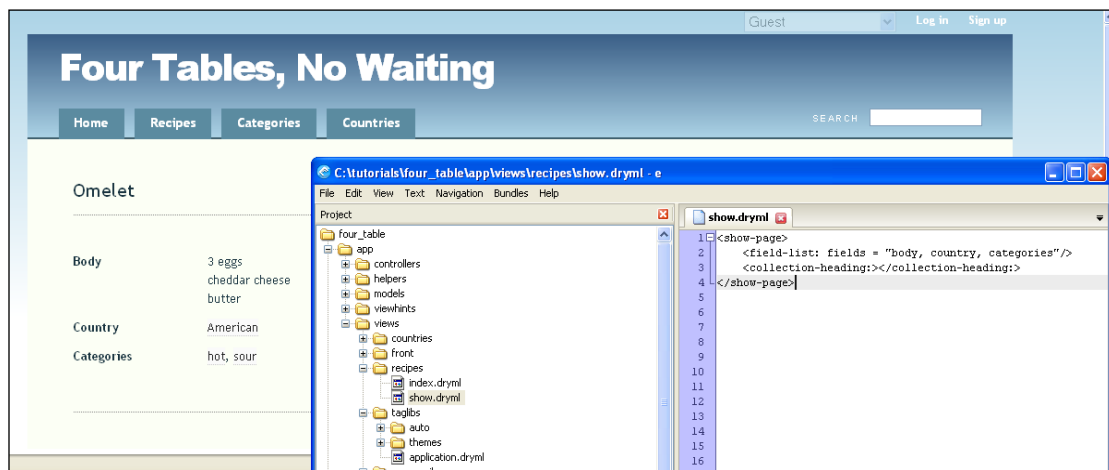


Figure 143: Using the `<collection-heading:>` tag

- 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>
```

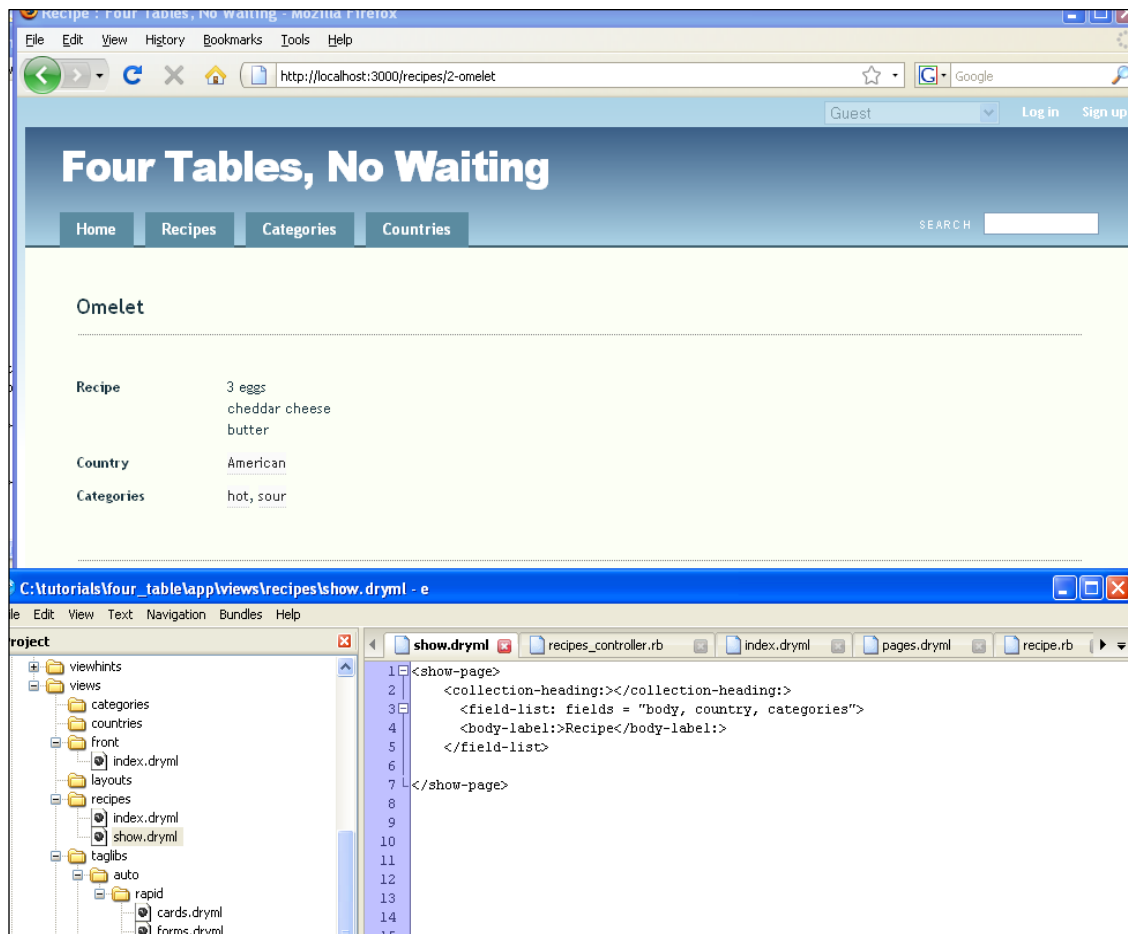


Figure 144: Using the <body-label:> parameter tag

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. Note that 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.

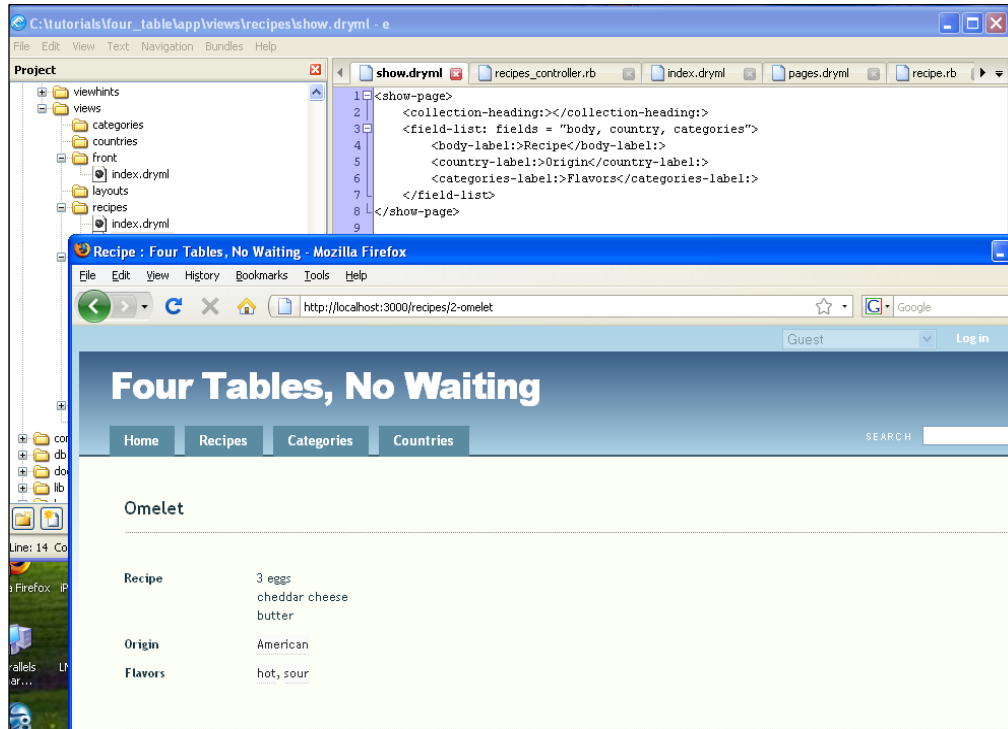


Figure 145: Using the `<country-label:>` parameter to change the label on the page

7. **Using the `<view>` tag to display a record.** There is still 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 as you probably have gathered that correspond to the Rapid tags.

If, for example, 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 see confirm that the formatting will be removed.

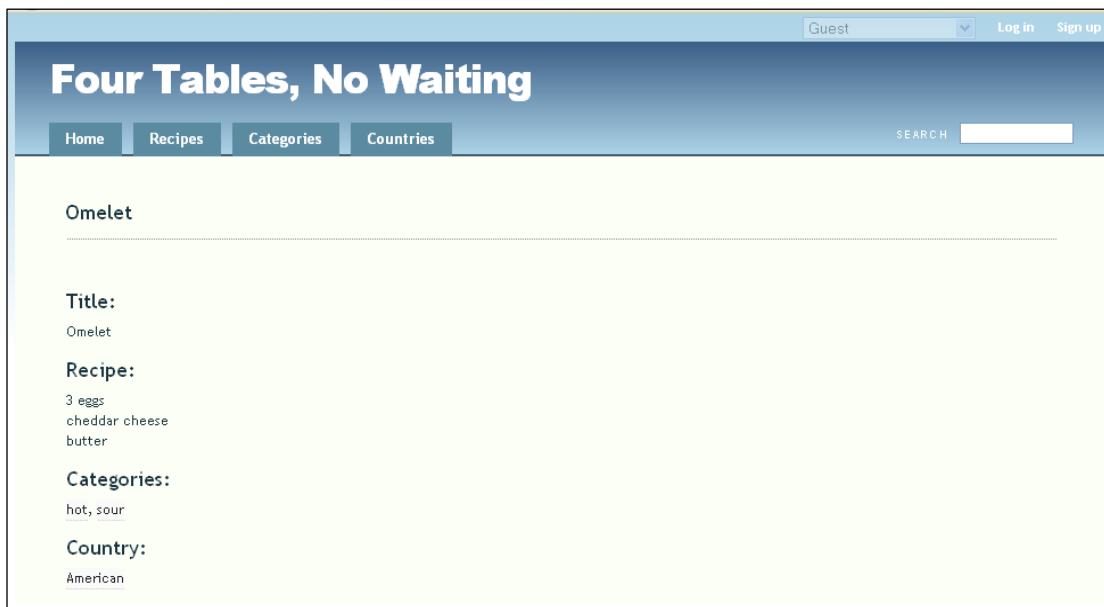


Figure 146: A new show page for Recipes

Here is what happens when you add the `replace` attribute.

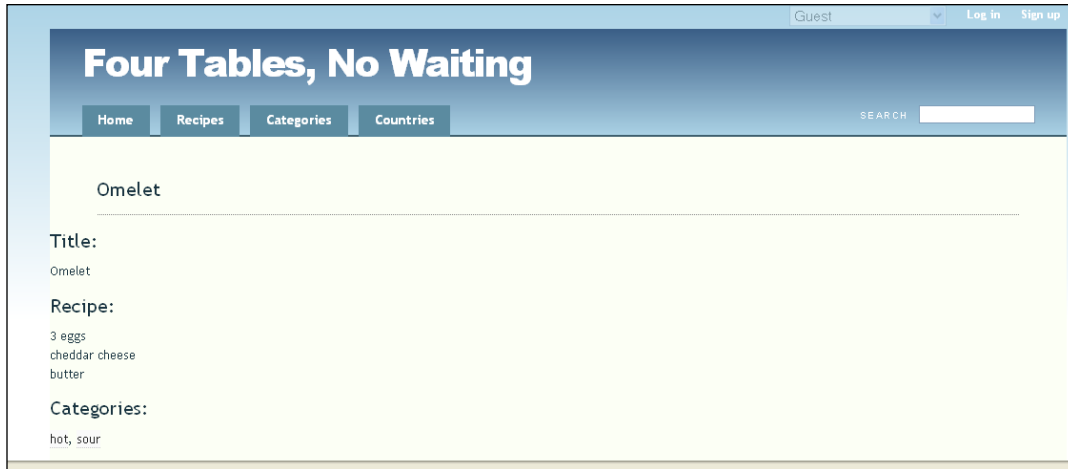


Figure 147: Page view of using the replace attribute in the `<content-body:>` parameter tag

Now take out the `replace` attribute before proceeding.

6. **Summary.** You have now learned to create a new template called `show.dryml` in the `views/recipes` directory that 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

1. **Get introduced to the `<new-page>` and `<edit-page>` tags.** Go into `pages.dryml` and take a look at the code for both of these tags. Here is the `<new-page>` definition:

```
<def tag="new-page" for="Recipe">
  <page merge title="New Recipe">
    <body: class="new-page recipe" param/>

    <content: param>
      <section param="content-header">
        <h2 param="heading">New Recipe</h2>
      </section>

      <section param="content-body">
        <form param>
          <submit: label="Create Recipe"/>
        </form>
      </section>
    </content:>
  </page>
</def>
```

And here is the `<edit-page>` definition:

```
<def tag="edit-page" for="Recipe">
  <page merge title="Edit Recipe">

    <body: class="edit-page recipe" param/>

    <content:>
      <section param="content-header">
        <h2 param="heading">Edit <type-name/></h2>
        <delete-button label="Remove This Recipe" param/>
      </section>

      <section param="content-body">
        <form param/>
      </section>
    </content:>
  </page>
</def>
```

```
    </section>
  </content:>

</page>
</def>
```

The components that we are going to focus on are shown in *bold italics*. Let's 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 no doubt are noticing that the `<field-list>` tag is doing something different here. 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 that was 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 from inspecting table relationships. The *recipe* model is related to both the *country* model and the *category* model. This is a pretty powerful capability for just 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 web application titled "Four Tables, No Waiting". The navigation bar includes links for Home, Recipes, Categories, and Countries, along with a search bar. The user is logged in as Admin. The main content area is titled "New Recipe" and contains the following form elements:

- Title:** A single-line text input field.
- Body:** A large multi-line text area.
- Categories:** A dropdown menu with an "Add Category" button next to it.
- Country:** A dropdown menu currently showing "(No Country)".
- Buttons:** "Create Recipe" and "or Cancel" buttons at the bottom.

Figure 148: 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 by 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. But there is still 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 just relies on the default for both of these tags so there are no calls to them in the `<edit-page>` tag definition.

2. **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/>
```

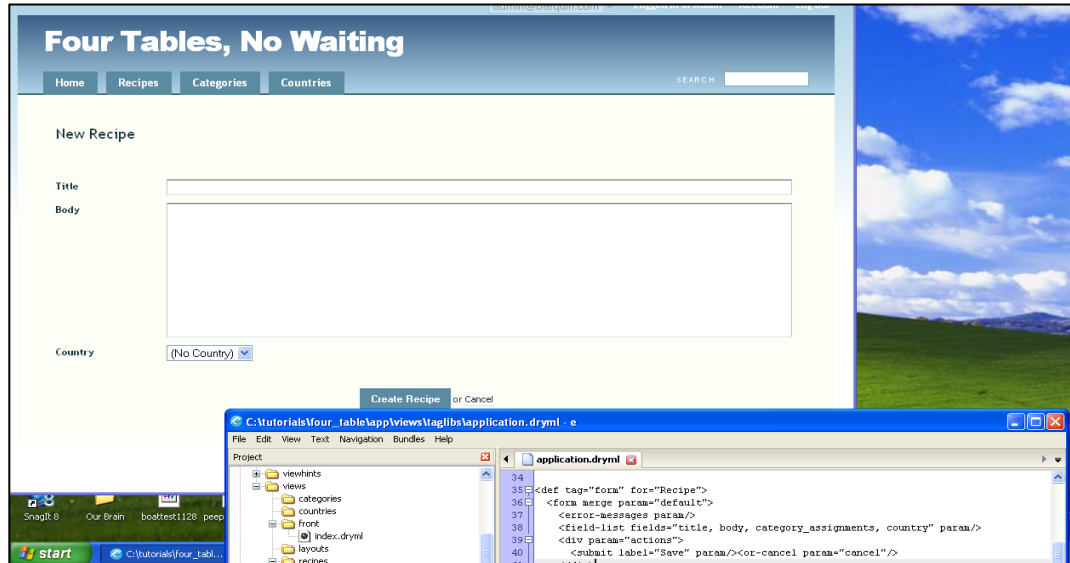


Figure 149: Modifying the `<field-list>` tag to remove fields on a page

Now your *categories* drop-down box is gone.

You may be wondering why we did not remove the `category_assignments` attribute also or for that matter 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.

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, `<field-list>` calls the `<input>` tag when it is creating an empty form to enter a record or populating a form for editing a record. 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, which in this case is set by working within the *Recipe* form. Further, 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.

First, 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.

```
<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.

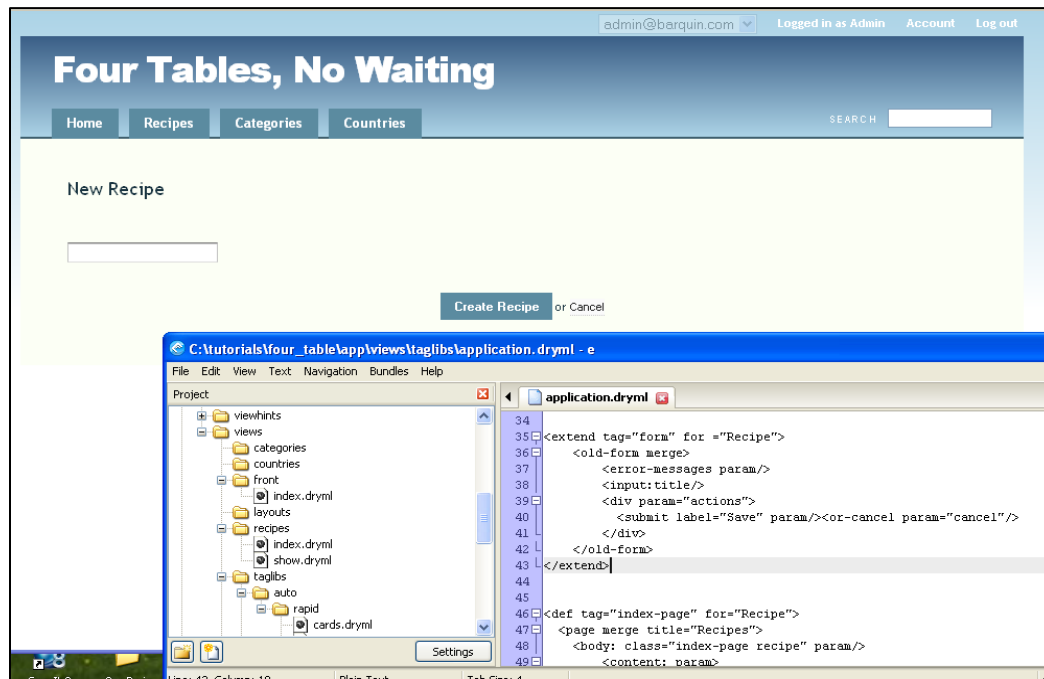


Figure 150: First step using the `<input>` tag

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:

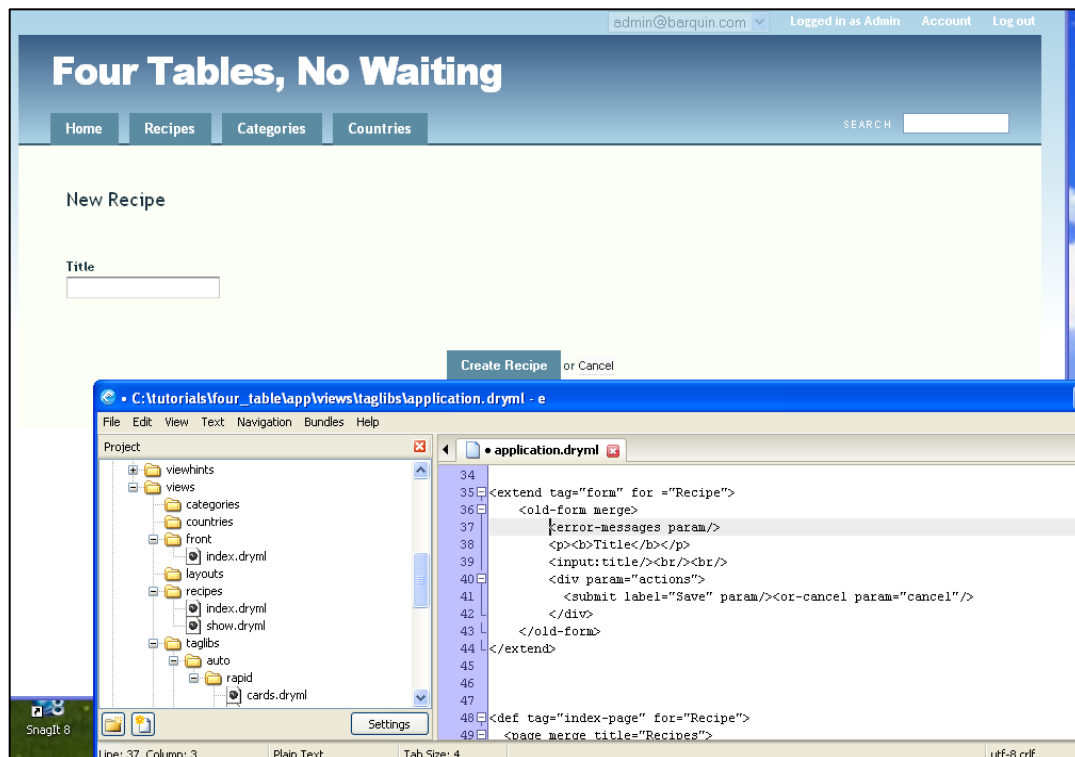


Figure 151: Adding the label for the filed "Title"

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>
```

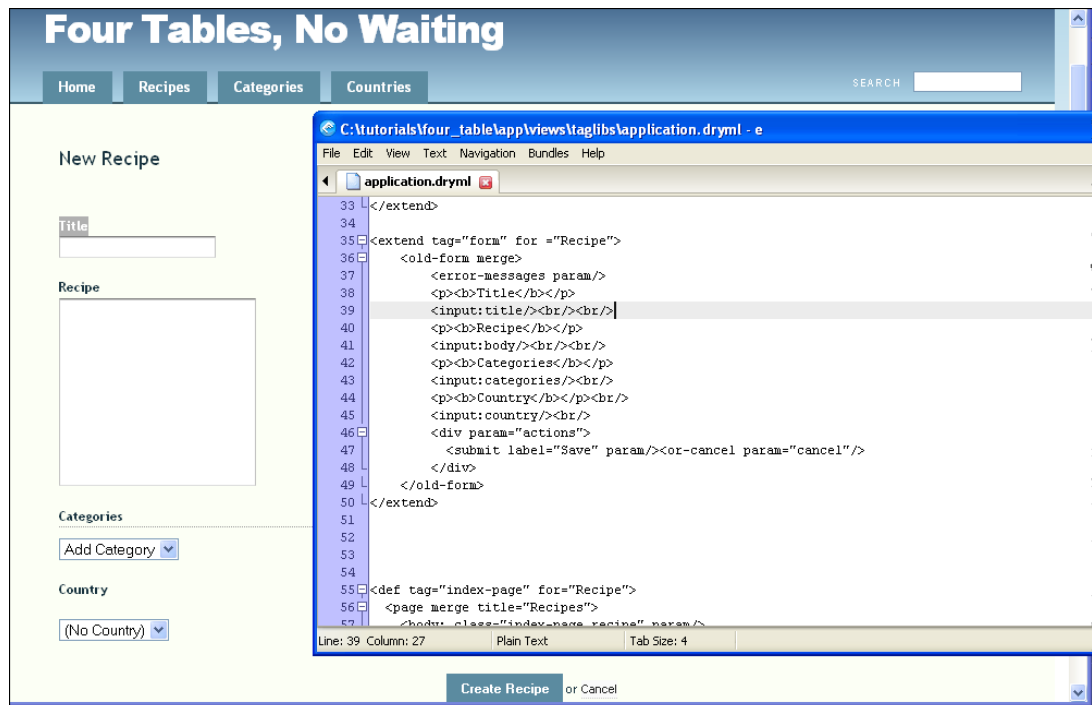


Figure 152: Adding the rest of the input fields

Now you have succeeded in reconstructing a form with the `<input>` tag and a little bit of additional HTML formatting.

Summary. Hobo provides some 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

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`.

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.

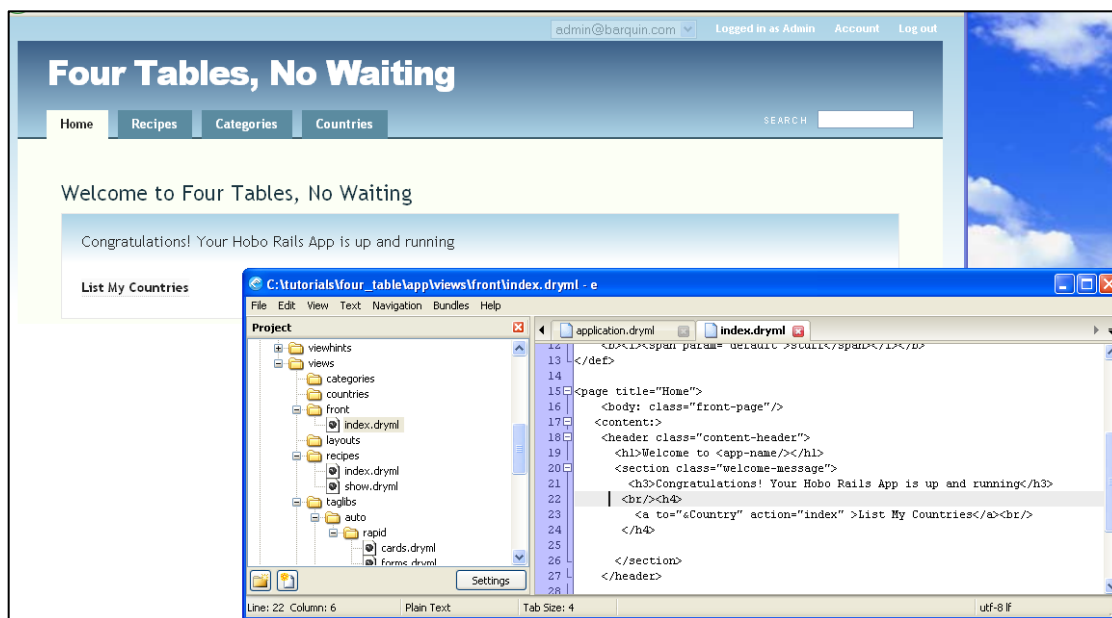


Figure 153: Generating an active link to a list of Countries

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 be referred to by the *action* attribute too.

Of course, if you click on the ‘List My Countries’ link, you will now see a listing of countries.

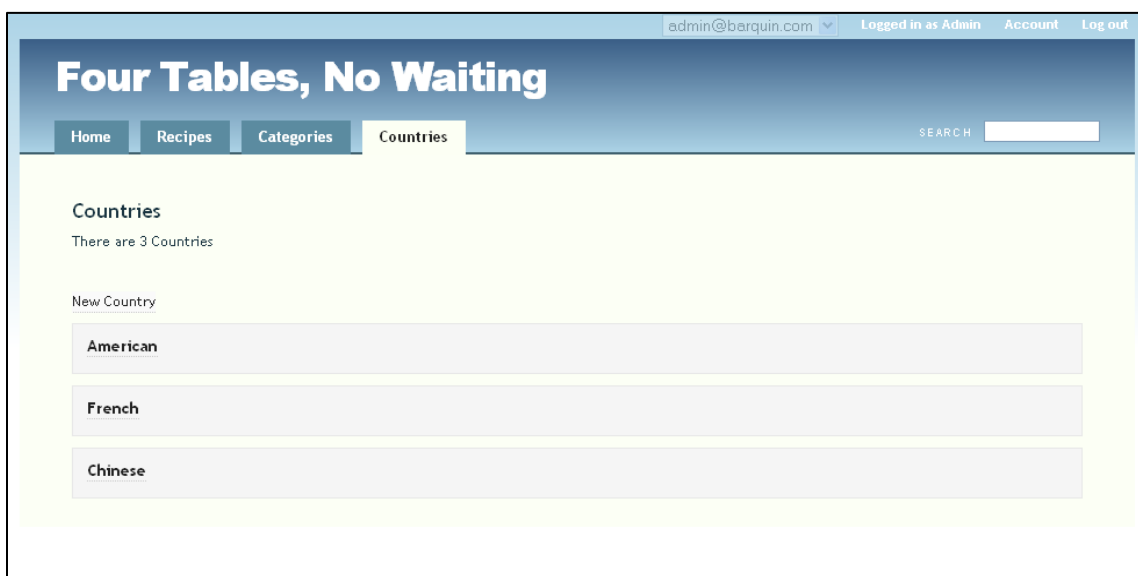


Figure 154: The Countries index page activated by your custom link

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/>
```

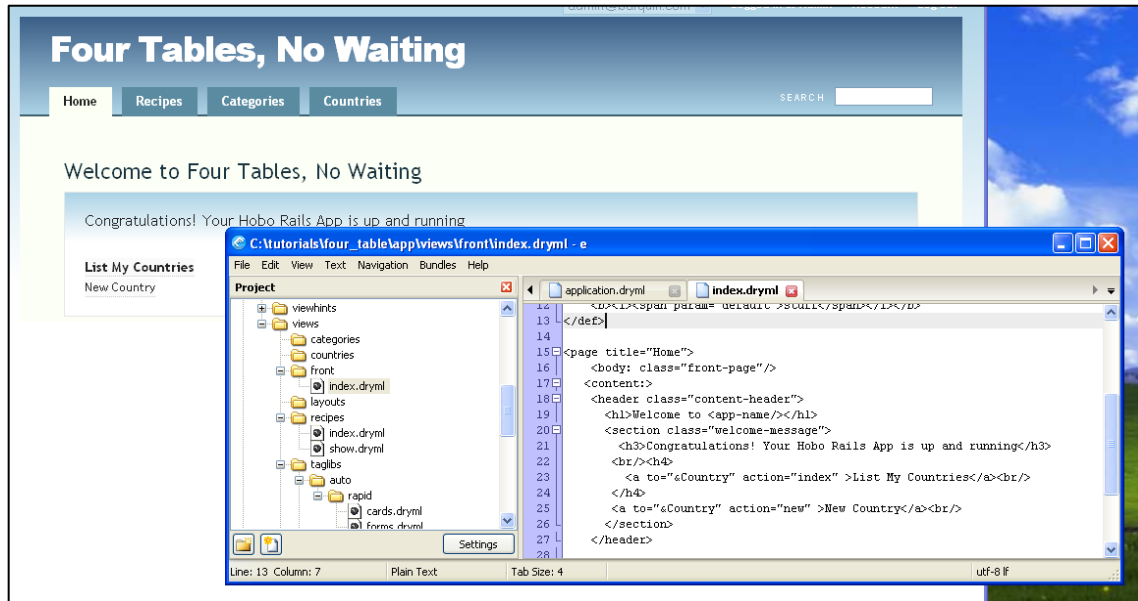


Figure 155: Constructing a custom link to the "New Country" page

Now you've got another link to try out.

4. **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. 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.

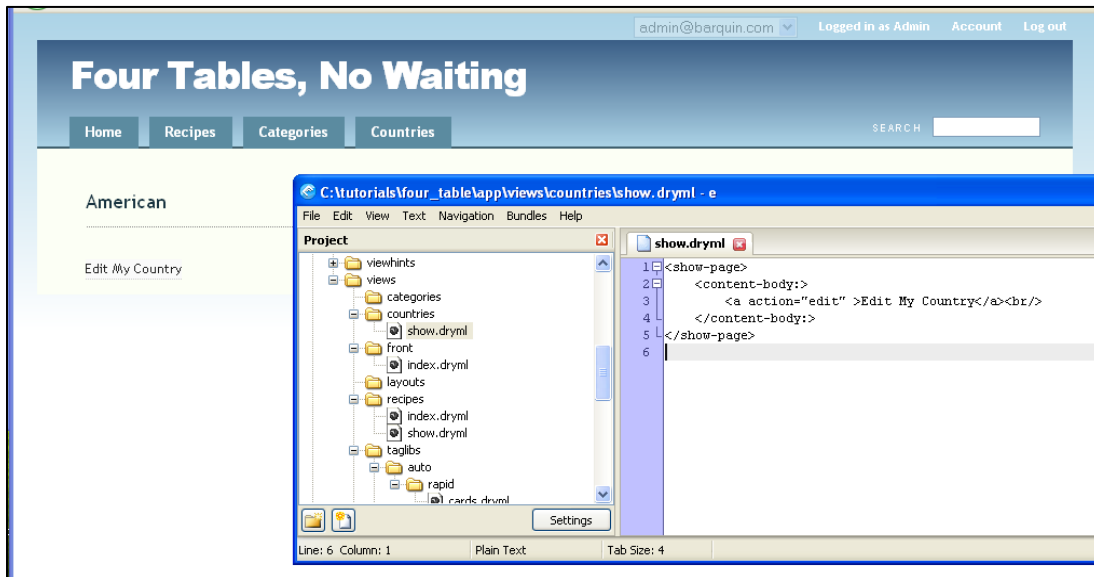


Figure 156: Page view of custom <show-page> tag

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”.

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.

Here is a quick summary of our 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

Create the application like you have for the other tutorials:

```
> hobo projects
```

Now look again about 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)

Now we need to create the models outlined above using the Hobo generator:

```
> ruby script/generate hobo_model_resource project name:string  
> ruby script/generate hobo_model_resource requirement title:string  
body:text status:string  
> ruby script/generate hobo_model_resource task name:string
```

Remember that the **hobo_model_resource** generator builds the entire MVC (Model/Controller/View) infrastructure needed for any model requiring a web-front end. The “task assignments” model is simply the table required to support many-to-many relationships behind the scenes. So a view or controller is not needed, so we only need the hobo model generator:

```
> ruby script/generate hobo_model task_assignment
```

Note that 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:

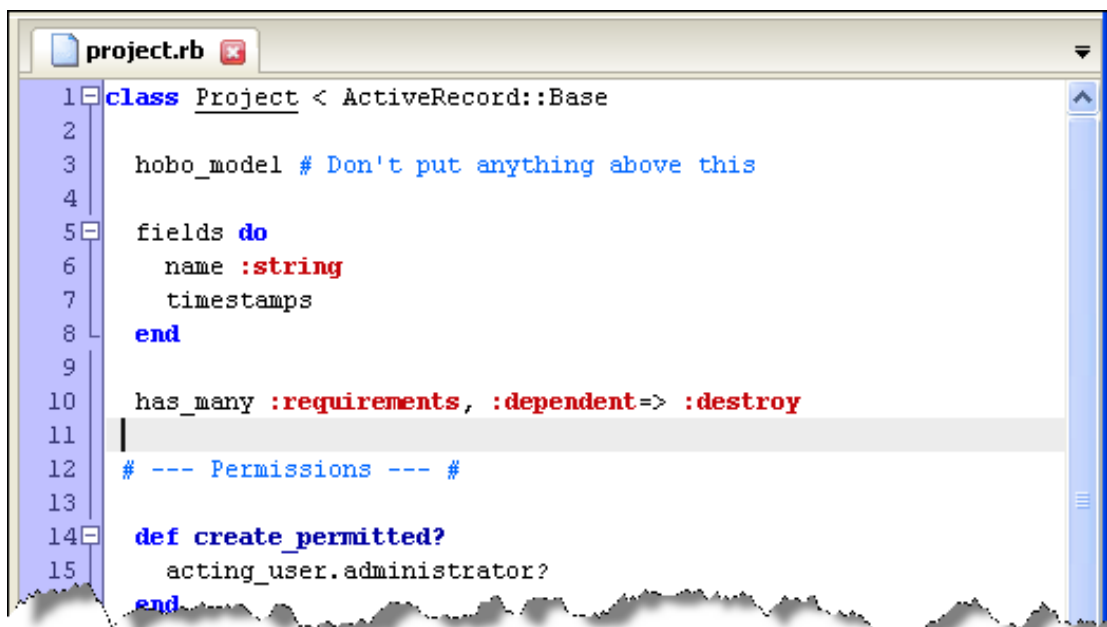
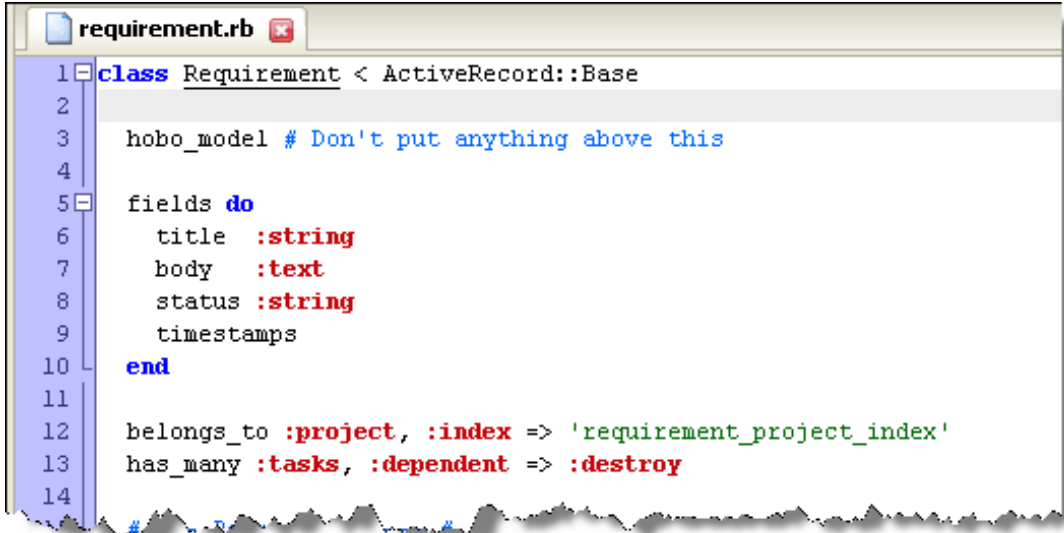


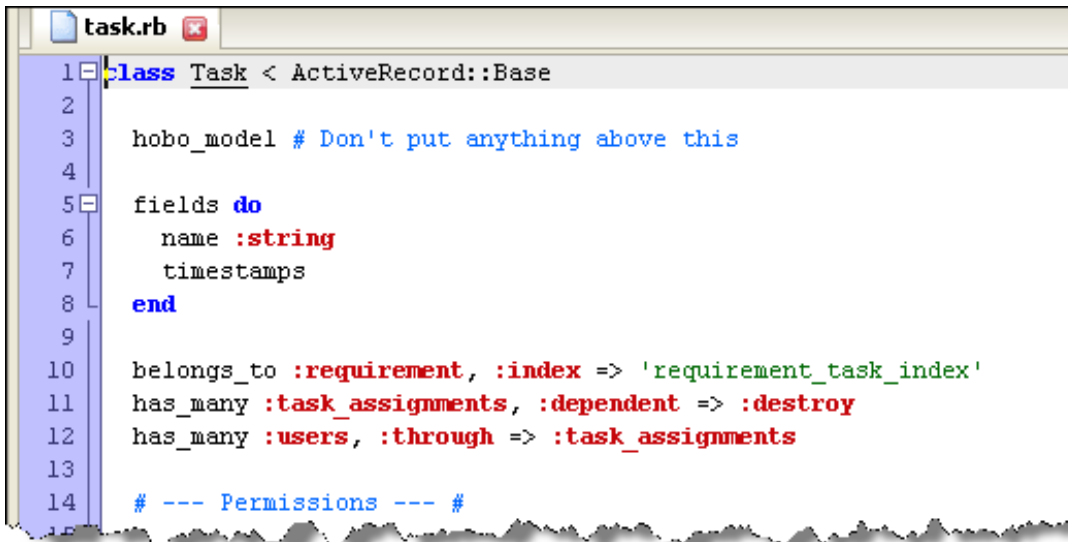
Figure 157: Adding "has_many :requirements" to the Project class



```
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
15  # --- Permissions --- #
```

Figure 158: 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, and Oracle has this irritating limitation of 30 characters for table, column, and index names. If we had not specified the index name, Rails would chose 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 --- #
15
```

Figure 159: Adding the “belongs_to” and “has_many” declarations to the Task model

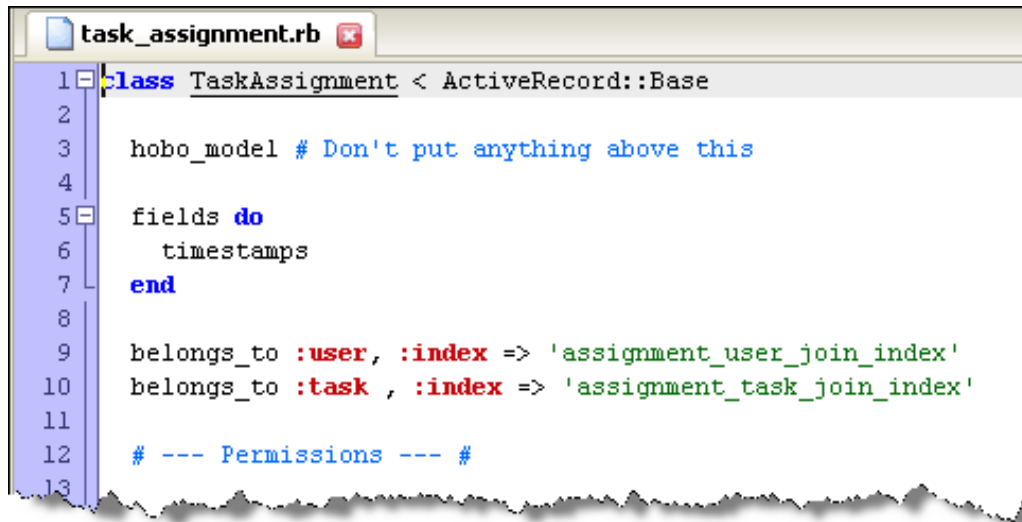


Figure 160: Adding the two "belongs_to" definitions to the TaskAssignment model

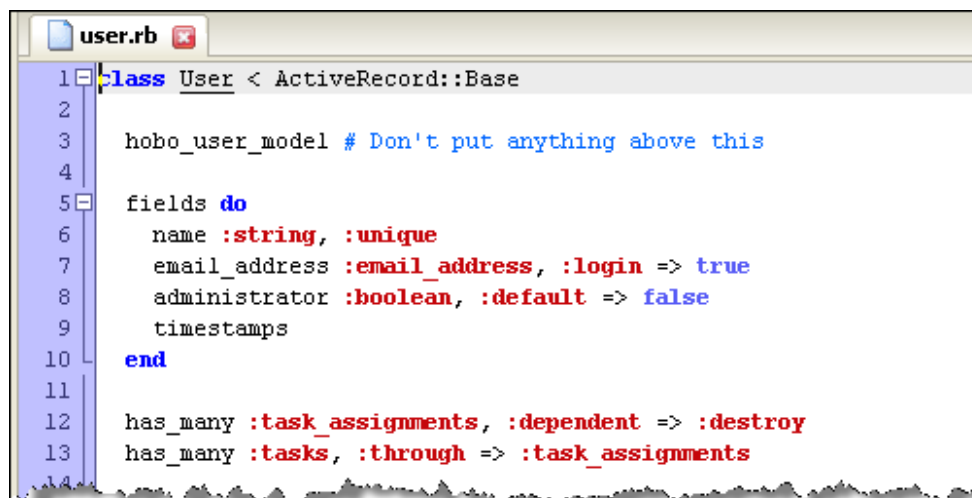


Figure 161: Adding the "has_many" declarations to the User model

Now Hobo will create a single migration for all of these changes:

```
> ruby script/generate hobo_migration
```

Load the migration file in your text editor to see what was generated:

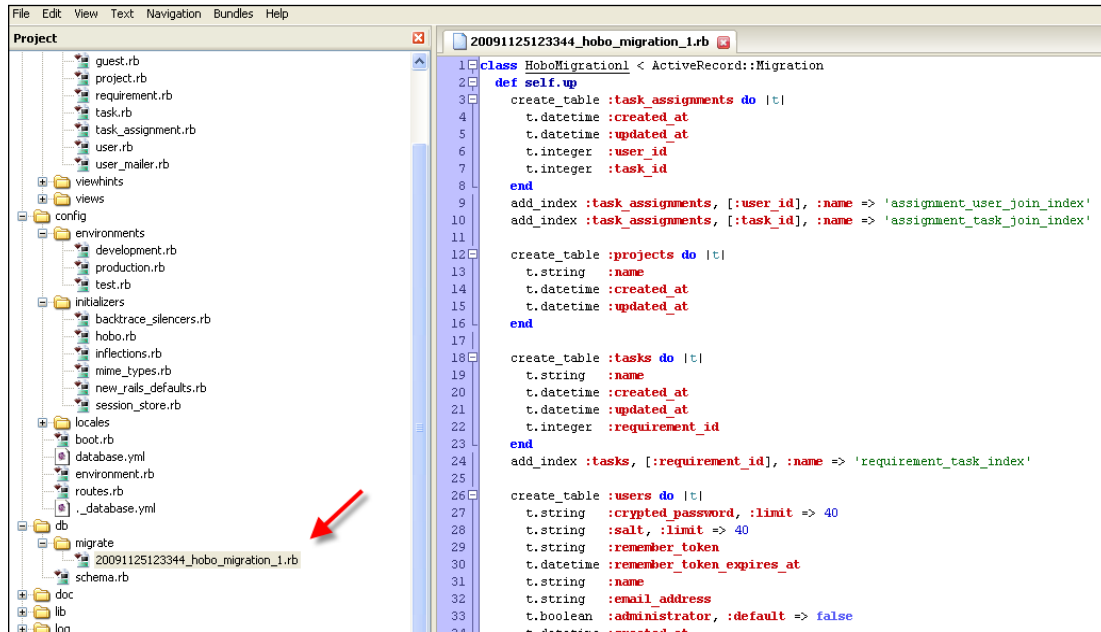


Figure 162: First Hobo migration for Projects

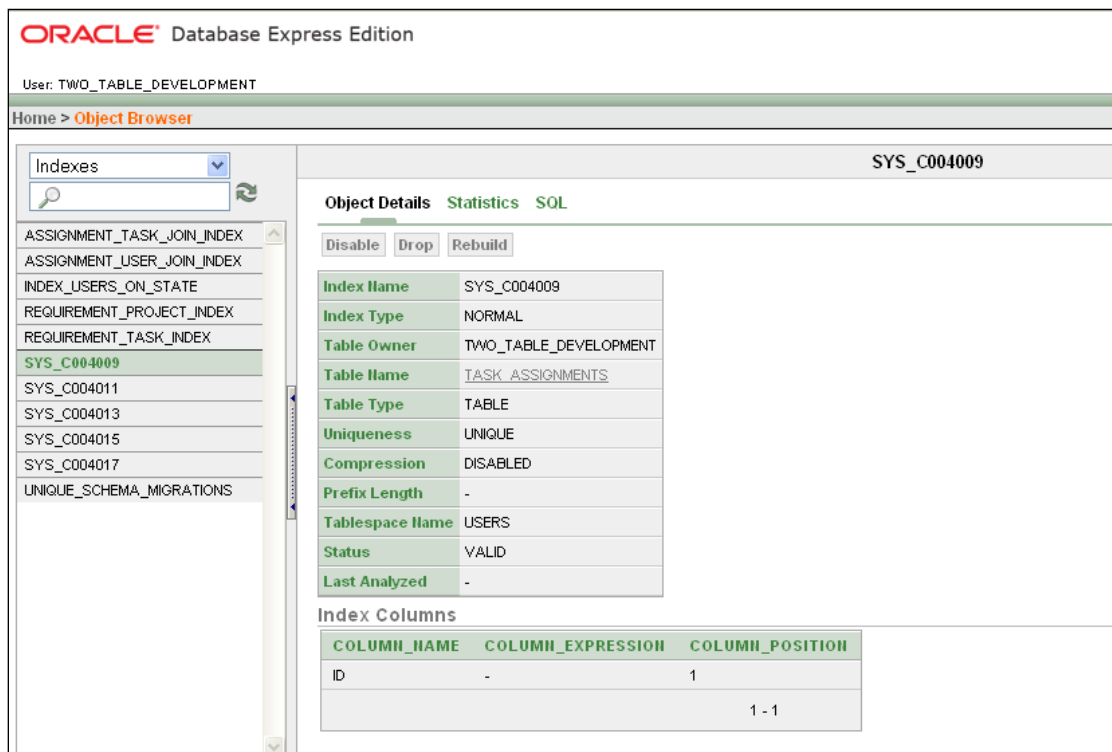


Figure 163: View of indexes created by the migration

In the figure above 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.

After you run the migration fire up the app:

```
> ruby script/server
```

Here is what you app should look like now:

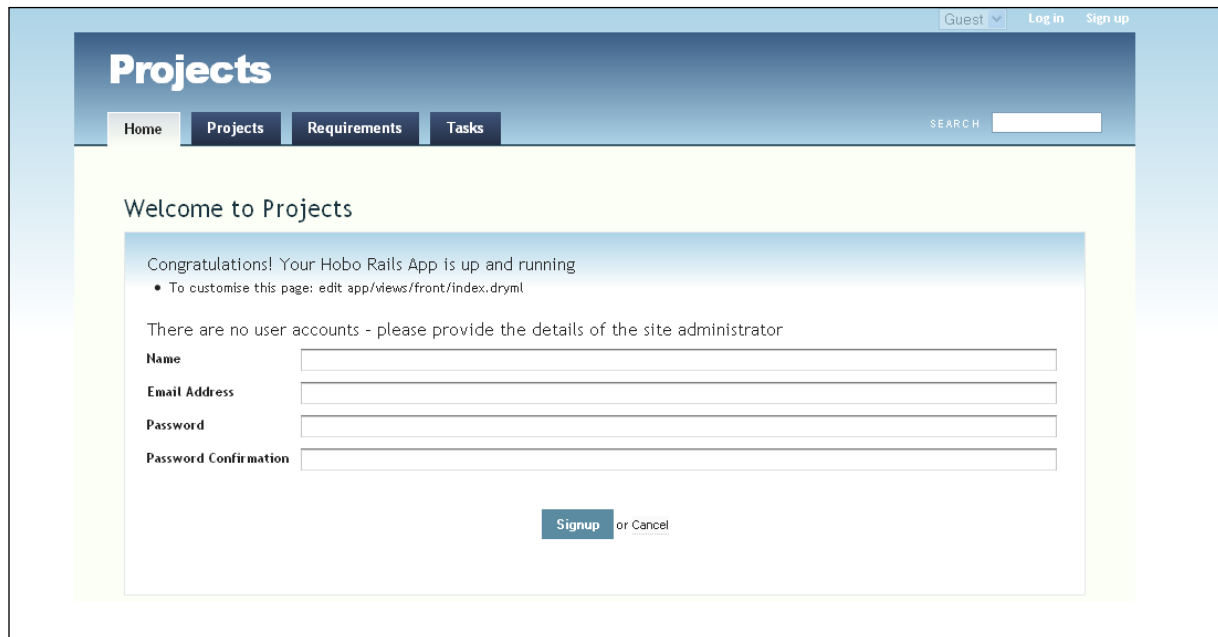


Figure 164: The default Home page for the Projects application

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.

Now enter a few projects like this:

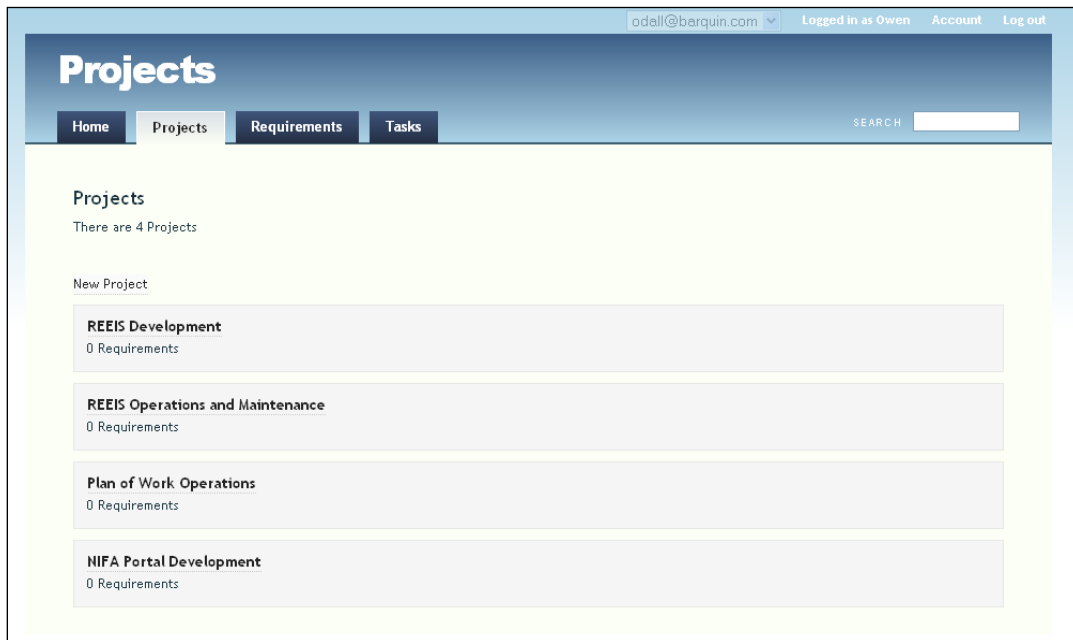


Figure 165: The Projects index page

Enter a couple of requirements for one of your projects:

The screenshot shows the 'New Requirement' page. It features a form with the following fields: 'Title' (containing 'New nightly project ETL for projects'), 'Body' (containing 'switch to using Talend from PL/SQL'), 'Status' (an empty text field), and 'Project' (a dropdown menu with 'REEIS Development' selected). At the bottom, there are two buttons: 'Create Requirement' and 'or Cancel'.

Figure 166: New Requirement page



Figure 167: Index view for Requirements

And enter some tasks for one of the requirements:

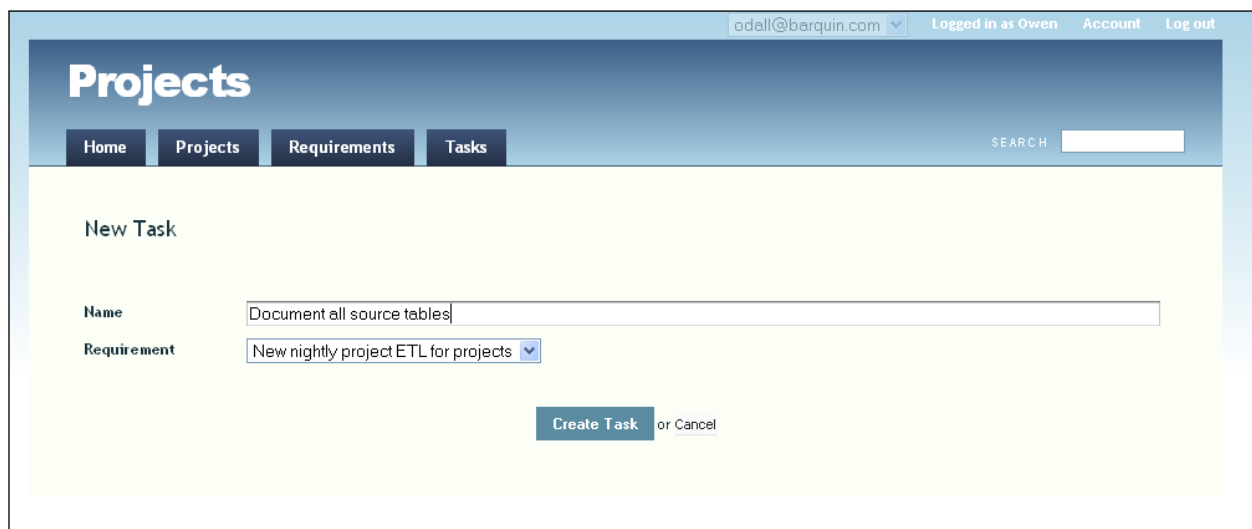


Figure 168: New Task page

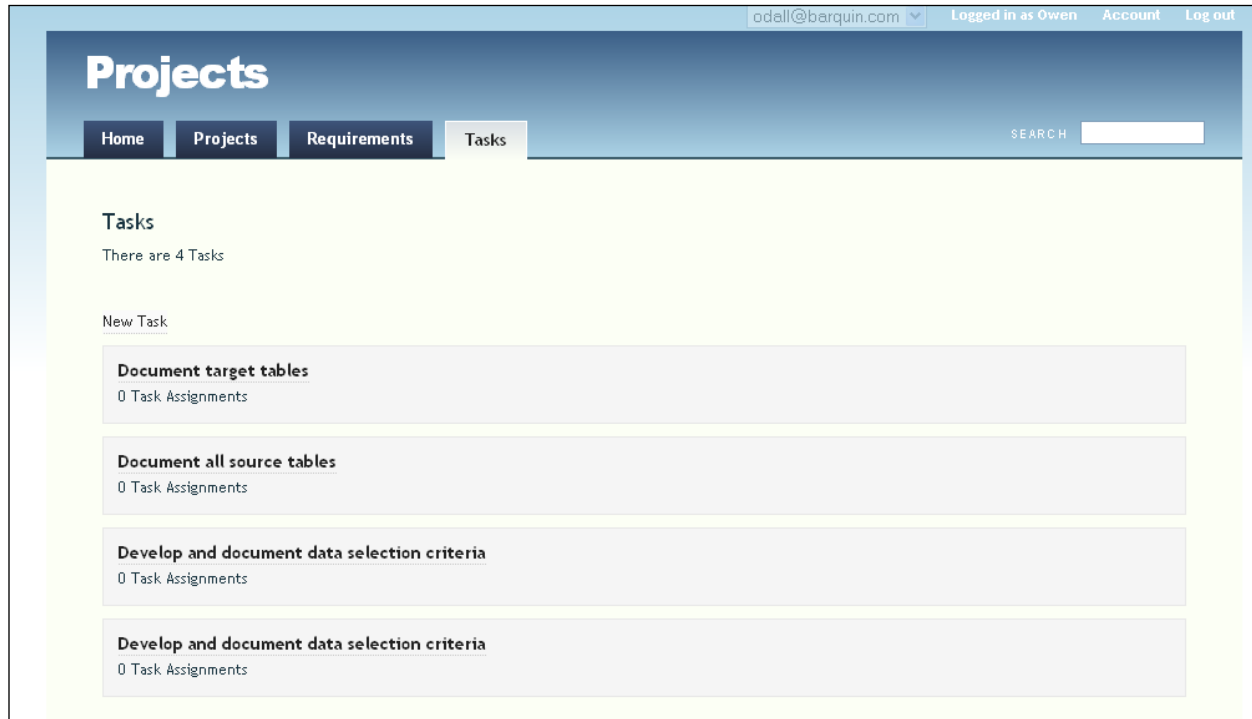


Figure 169: Index view for Tasks

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/front/summary`

This summary provides you quick access to information on:

- Application Name
- Application Location
- Rails Version/Location
- Change Control (e.g., Git)
- Gems
- Plugins
- Environments
- Models/Tables
- Model Associations

The following are screen shots of the Projects application so far. Notice that the development environment we have been using is Oracle.

Note: The Application Summary is refreshed each time a **hobo_migration** is executed.

Projects

Home Projects SEARCH

Application Summary

| | |
|----------------------|-----------------------|
| Application Name | Projects |
| Application Location | C:/tutorials/projects |
| Rails Version | 2.3.2 |
| Rails Location | gem |
| Mode | development |

Change Control

Method: other

Gems

| | Required | Installed | Status | Dependencies |
|----------------------|----------|-----------|-----------|---|
| hobo | >=0 | 0.9.0 | installed | rails >=2.2.2 mislav-will_paginate >=2.2.1 hobosupport =0.9.0 hobofields =0.9.0 |
| rails | >=2.2.2 | 2.3.2 | installed | |
| mislav-will_paginate | >=2.2.1 | 2.3.11 | installed | |
| hobosupport | =0.9.0 | 0.9.0 | installed | |
| hobofields | =0.9.0 | 0.9.0 | installed | rails >=2.2.2 hobosupport =0.9.0 |

Figure 170: Part 1 of the Application Summary page

Plugins

| Location | Method | Clean? | Version |
|--|--------|--------|---------|
| hobo c:/ruby/lib/ruby/gems/1.8/gems/hobo-0.9.0 | other | | |

Environments

| | database |
|-------------|----------------------------|
| development | oracle XE |
| production | oracle projects_production |
| test | oracle projects_test |

Models

| Class | Table |
|----------------|------------------|
| Guest | |
| Project | projects |
| Requirement | requirements |
| Task | tasks |
| TaskAssignment | task_assignments |
| User | users |
| UserMailer | |

Figure 171: Part 2 of the Application Summary page

| | | |
|--------------|----------|-------------|
| 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 |

Figure 172: Part 3 of the Application Summary page

| | | |
|----------------|------------|-------|
| TaskAssignment | | |
| Column | Type | |
| created_at | datetime | |
| updated_at | datetime | |
| Association | Macro | Class |
| user | belongs_to | User |
| task | belongs_to | Task |

| | | |
|---------------------------|-------------------|----------------|
| User | | |
| Column | Type | |
| crypted_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 |
| tasks | has_many :through | Task |
| task_assignments | has_many | TaskAssignment |

Figure 173: 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. But 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.

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.

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
```

Next, refresh the browser and you’ll notice that “Tasks” has been removed from the main navbar.

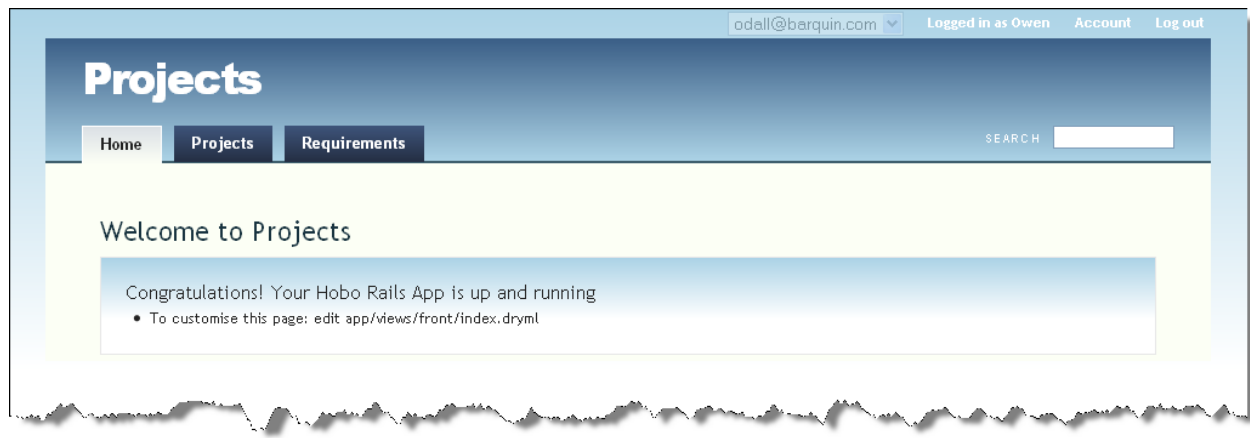


Figure 174: Effect of removing the "index" action from the Tasks controller

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 175: 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.

Add the following declaration to the requirements controller:

```
auto_actions_for :project, [:new, :create]
```

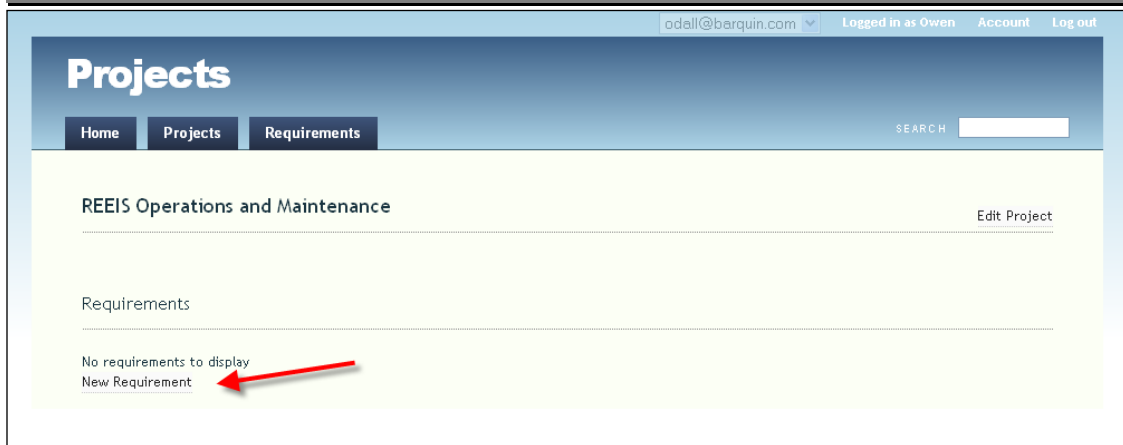
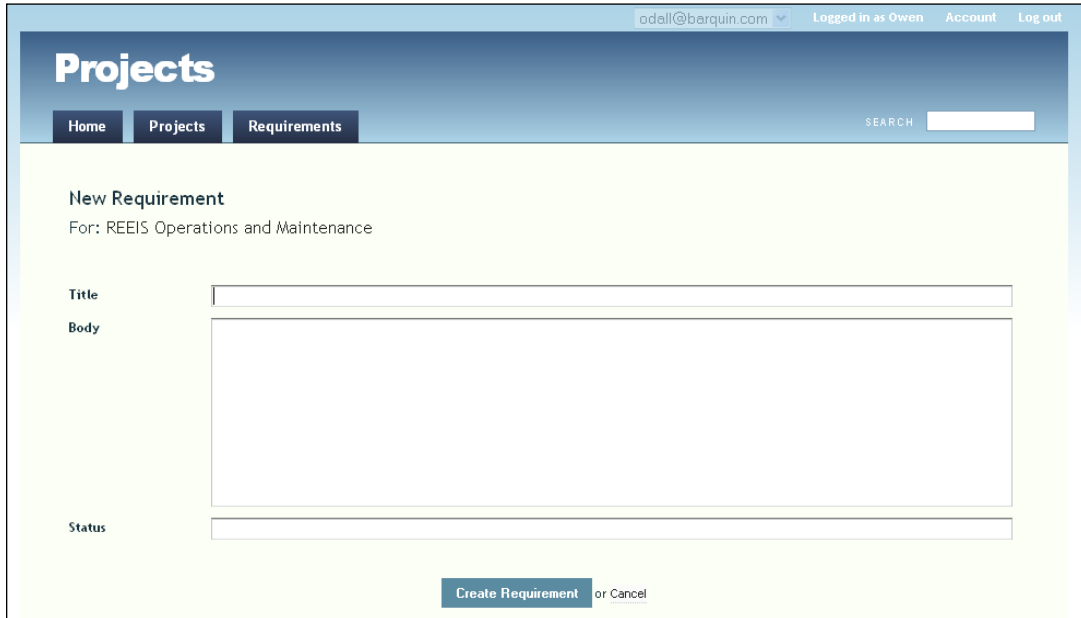


Figure 176: The "New Requirement" link now appears

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:



The screenshot shows a web application interface for 'Projects'. At the top, there's a navigation bar with 'Home', 'Projects', and 'Requirements' tabs. A search bar is on the right. The main content area is titled 'New Requirement' and specifies 'For: REEIS Operations and Maintenance'. It contains three input fields: 'Title', 'Body' (a larger text area), and 'Status'. At the bottom, there are two buttons: 'Create Requirement' and 'or Cancel'.

Figure 177: View of the "New Requirement" page

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!

The screenshot shows a web application interface for 'Projects'. At the top, there's a navigation bar with 'Home', 'Projects', and 'Requirements' tabs. A search bar is on the right. Below the navigation bar, a message states 'The requirement was created successfully'. The main content area is titled 'Yearly load of State Plan data' and includes a link to 'Edit Requirement'. A section titled 'Tasks' shows 'No tasks to display'. At the bottom, there's an 'Add a Task' form with a 'Name' input field and an 'Add' button.

Figure 178: View of the in-line "Add a Task" form

Now 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
```

There's also "white-list" style where you list 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 is handy for things that are manipulated by AJAX, but never viewed directly:

```
auto_actions :write_only # short for -- :create, :update, :destroy
```

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:

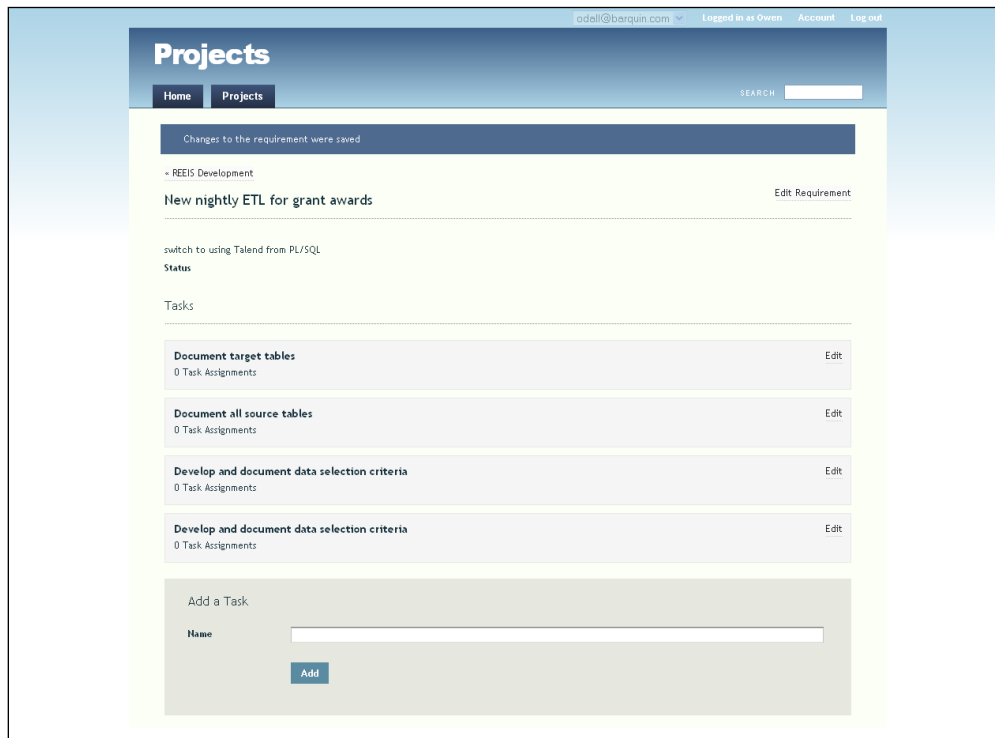


Figure 179: Requirement page after modifying controller definitions

Permissions

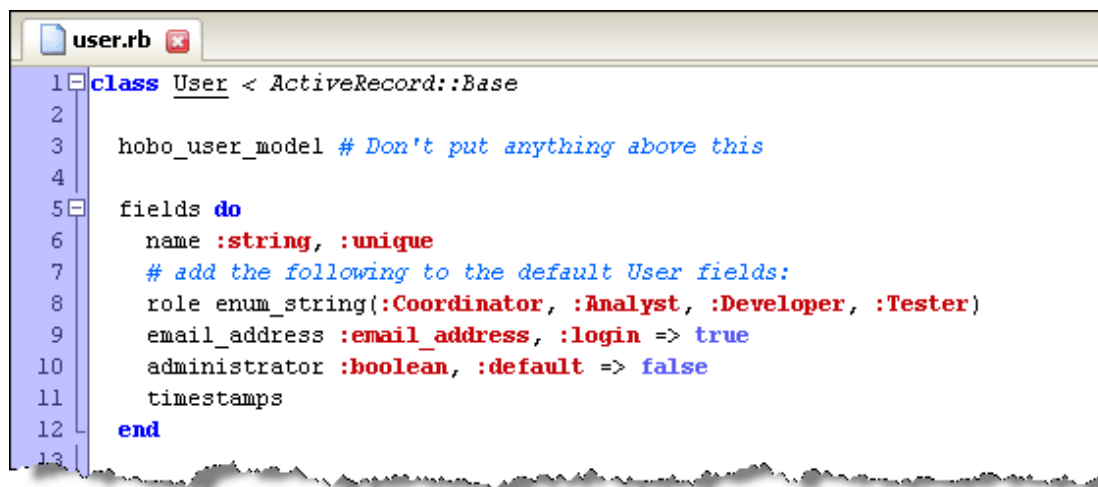
So far we've done two major things with our app:

- Created models and specified associations
- Modified controllers to specify which actions are available

There's one more thing we typically do when creating a new Hobo app, before we even touch the view layer. We modify permissions in the model layer.

Adding Roles

Let's do a simple addition to the User model. Here 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 180: Defining available roles using "enum_string"

Run a **hobo_migration** to add this field to the database.

Modify the create permission to allow an Administrator to create a new user:

```
43
44 # --- Permissions --- #
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, :crypted_password,
53                                           :current_password, :password, :password_confirmation))
54   # Note: crypted_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
```

Figure 181: Modifying the "create_permitted" method to the User model

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 182: Users Controller with "auto actions :all:"

Run the server again and then refresh your browser:



Figure 183: The Users tab is now active

Now we can edit a user and add a role:



The screenshot shows the 'Edit User' page in the Agile Project Manager application. The page has a blue header with the 'Projects' logo and navigation links for 'Home', 'Projects', and 'Users'. A search bar is located in the top right. The main content area is light green and contains the 'Edit User' form. The form includes fields for 'Name' (Owen), 'Role' (Analyst), 'Email Address' (odall@barquin.com), and 'Administrator' (checked). A 'Remove This User' button is in the top right of the form area. At the bottom, there are 'Save' and 'Cancel' buttons.

Figure 184: 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.

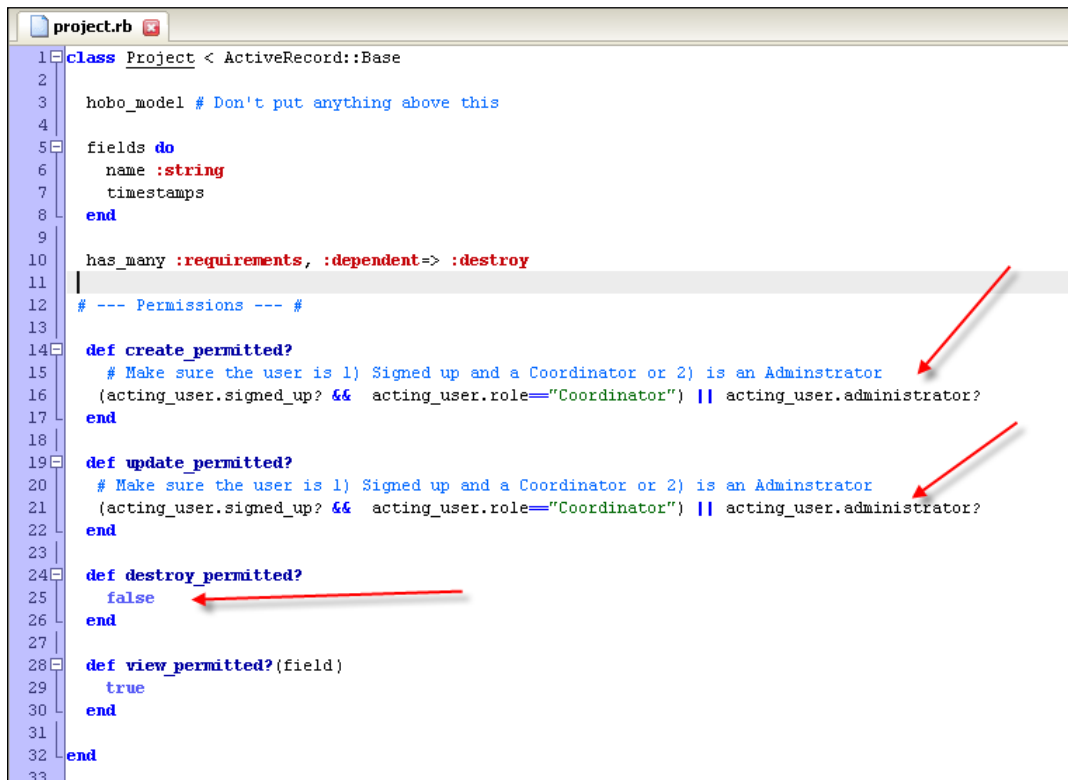
Now let’s see how to use this information.

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:



```
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
```

Figure 185: Adding the use of Role in Permissions

Notice that to create a project, the active user must be an administrator OR:

- The user must be signed up (not a guest)
- 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 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
    Administrator
    (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
    Administrator
    (acting_user.signed_up? && acting_user.role=="Coordinator") ||
    acting_user.administrator?
  end

  def destroy_permitted?
    false
  end

  def view_permitted?(field)
    true
  end

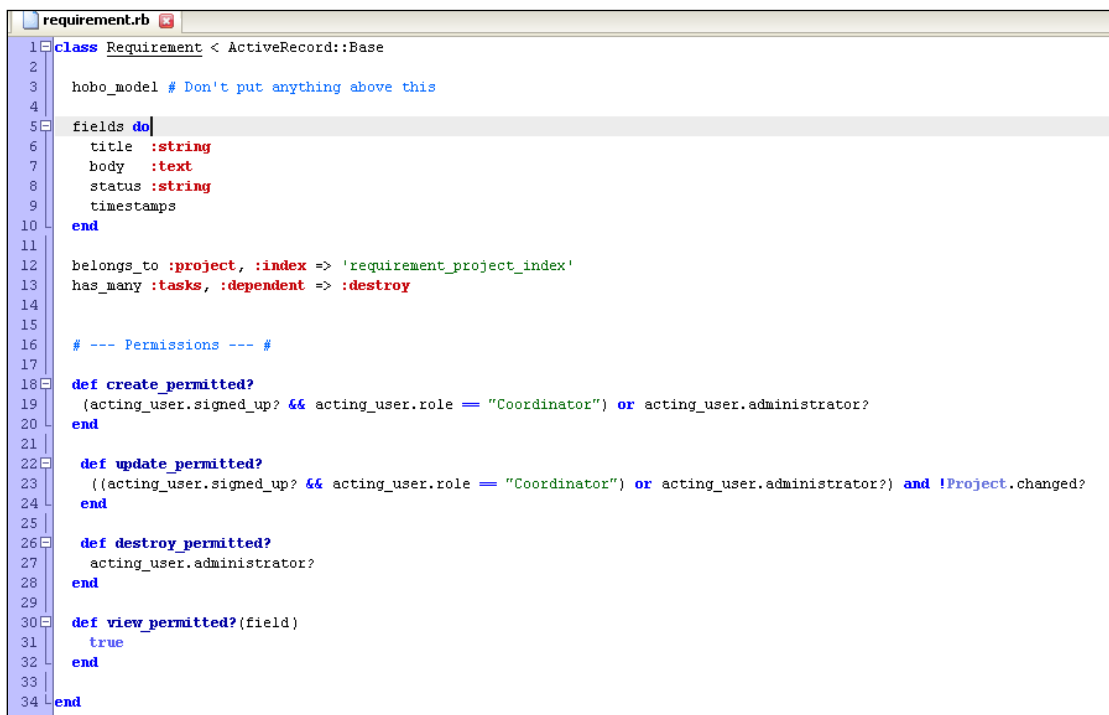
end
```

Permissions for data integrity

The permissions system is not just for providing operations to some users but not to 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.

So, if you want to stop this from happening, change the “`update_permitted?`” method in `requirement.rb`:



```

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
15  # --- Permissions --- #
16
17  def create_permitted?
18    (acting_user.signed_up? && acting_user.role == "Coordinator") or acting_user.administrator?
19  end
20
21  def update_permitted?
22    ((acting_user.signed_up? && acting_user.role == "Coordinator") or acting_user.administrator?) and !Project.changed?
23  end
24
25  def destroy_permitted?
26    acting_user.administrator?
27  end
28
29  def view_permitted?(field)
30    true
31  end
32
33  end
34

```

Figure 186: 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?) and !Project.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, the permission system was reporting that this 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 now 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:

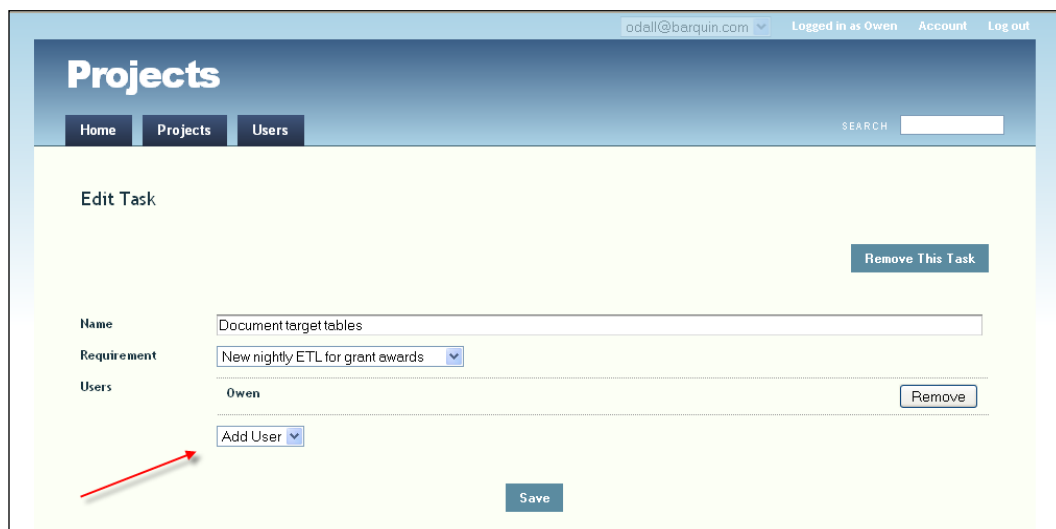


Figure 187: Assigning multiple Users to a Task in the Edit Task page

View Hints

Hobo has a great facility that makes it easy to modify the display of a field name, (without changing the model,) and field help that is displayed in the edit form, and to let Hobo know when you want to include “child” entities on a page.

Hobo creates a ViewHints template file for each model you generate in the format “{model_name}_hints”

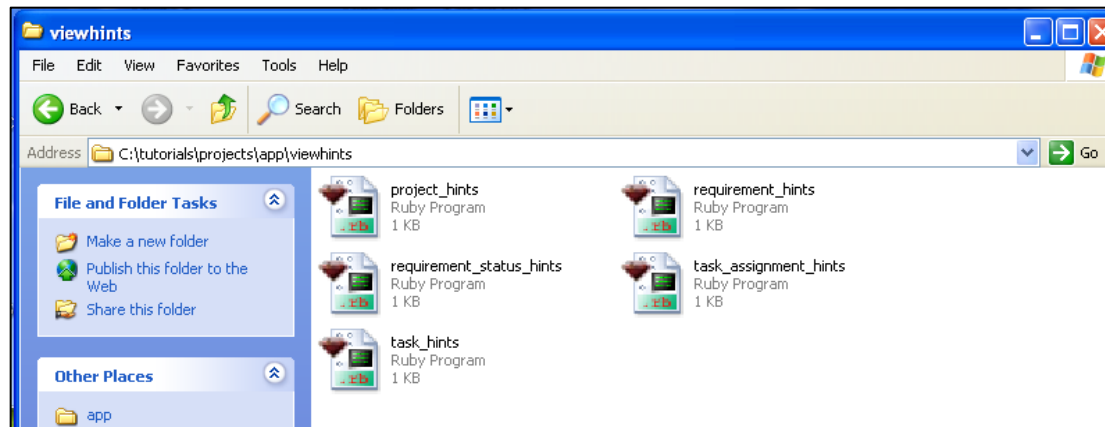


Figure 188: Contents of the \apps\viewhints folder

Here is what a blank one looks like:

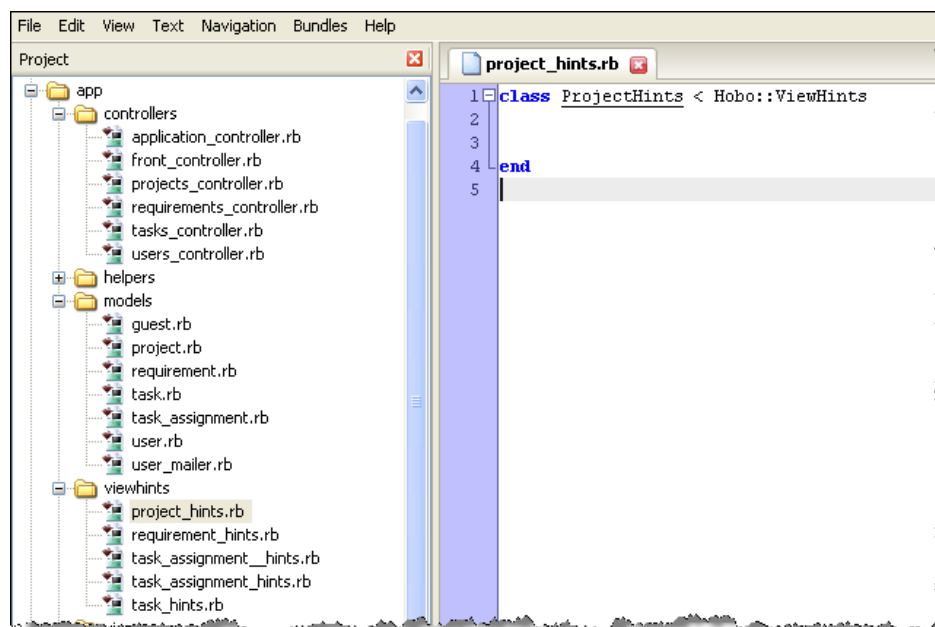
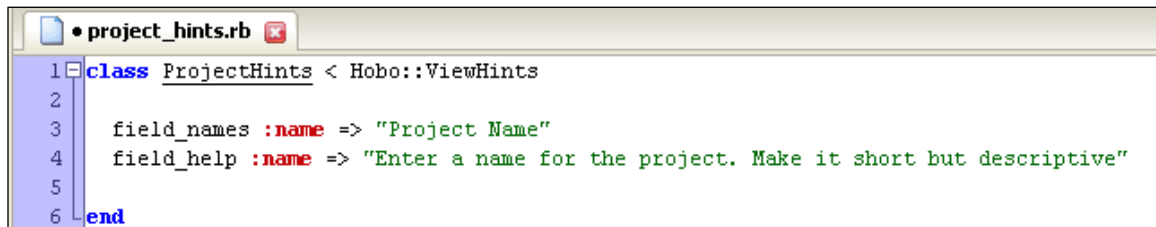


Figure 189: The default blank “project_hints.rb” file for the “ProjectHints” class

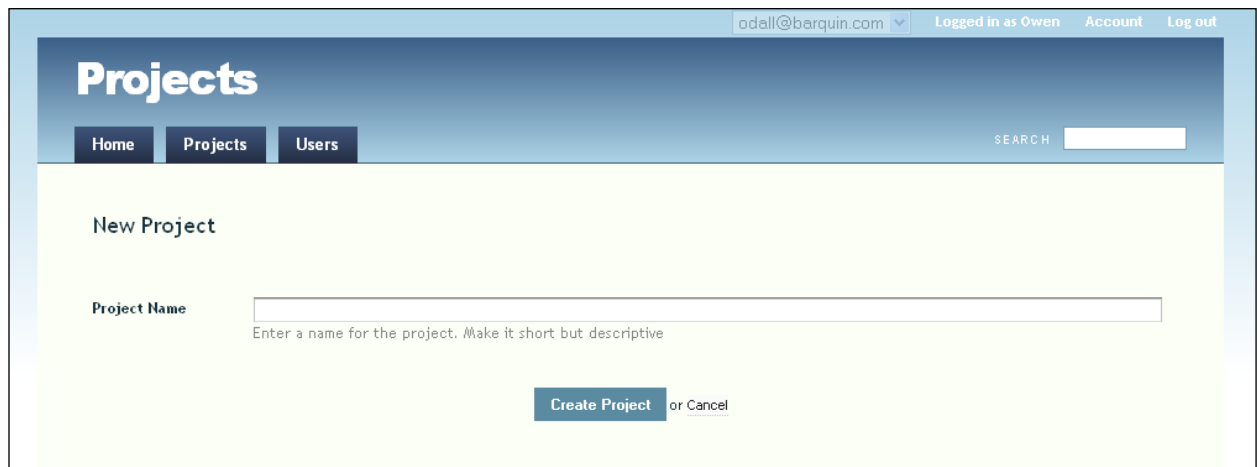
Now add two additional lines like the following:



```
1 class ProjectHints < Hobo::ViewHints
2
3   field_names :name => "Project Name"
4   field_help :name => "Enter a name for the project. Make it short but descriptive"
5
6 end
```

Figure 190: Defining “field_names” and “field_help” in ProjectHints

Refresh your browser and enter a new project:



The screenshot shows a web browser window displaying the 'New Project' page. The page has a blue header with the title 'Projects' and navigation links for 'Home', 'Projects', and 'Users'. A search bar is located in the top right corner. The main content area is yellow and contains a form for creating a new project. The form has a label 'Project Name' and a text input field. Below the input field is a hint: 'Enter a name for the project. Make it short but descriptive'. At the bottom of the form are two buttons: 'Create Project' and 'or Cancel'.

Figure 191: The New Project page using “ProjectHints”

Customizing views

It's pretty surprising how far you can get without even 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 just those 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, thanks 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 because it allows you to make very focused changes if a given piece of pre-packaged HTML is not quite 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:

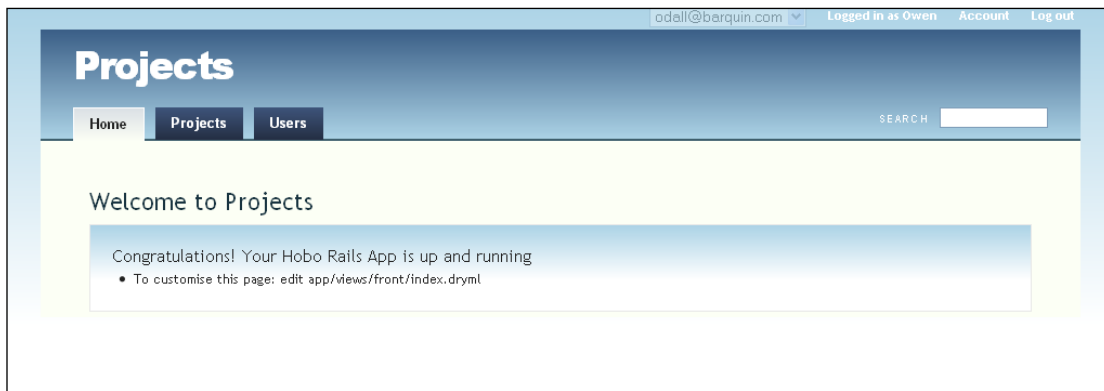


Figure 192: The default application name and welcome message

Edit `/app/views/taglibs/application.dryml`:

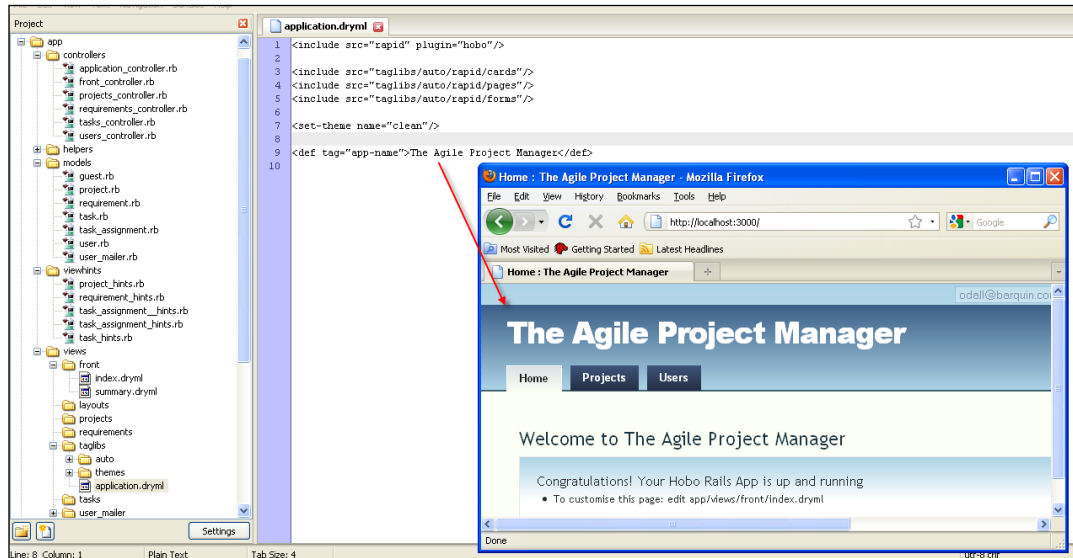


Figure 193: Changing the application name in "application.dryml"

Changing the 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:

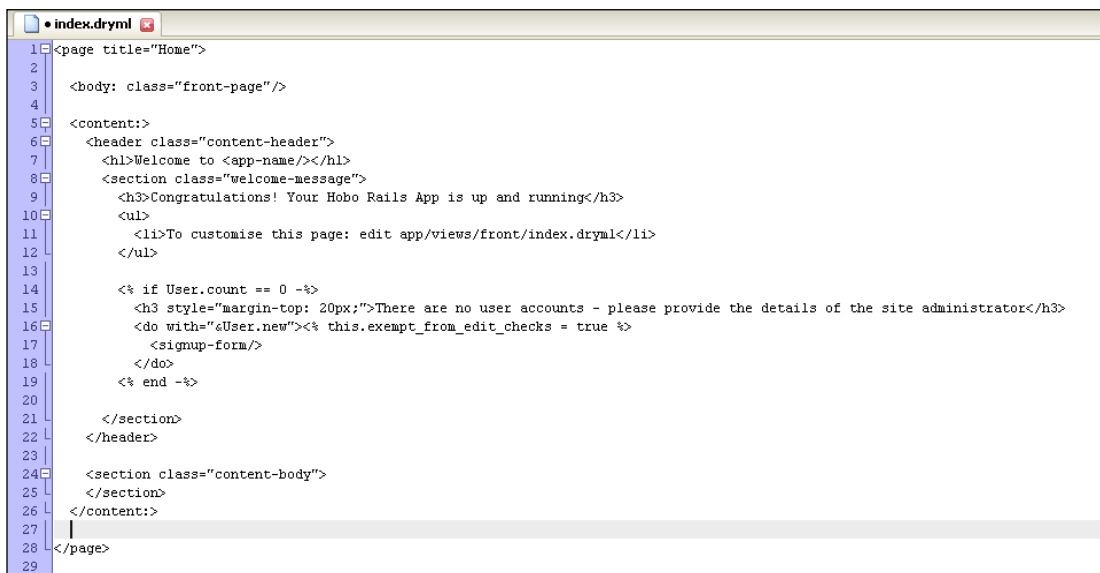


Figure 194: Modifying "front/index.dryml"

This is what it looks like before you change it. Now 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>
```

Now refresh your browser:



Figure 195: 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 `<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/></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 <def tag="app-name">The Agile Project Manager</def>
10
11 <extend tag="card" for="Task">
12   <old-card merge>
13     <append-body:>
14       <div class="users">
15         Assigned users: <repeat:users join=", "><a/></repeat><else>None</else>
16       </div>
17     </append-body:>
18   </old-card>
19 </extend>
```

Figure 197: Extending the card tag for Task in "application.dryml"

Now 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).

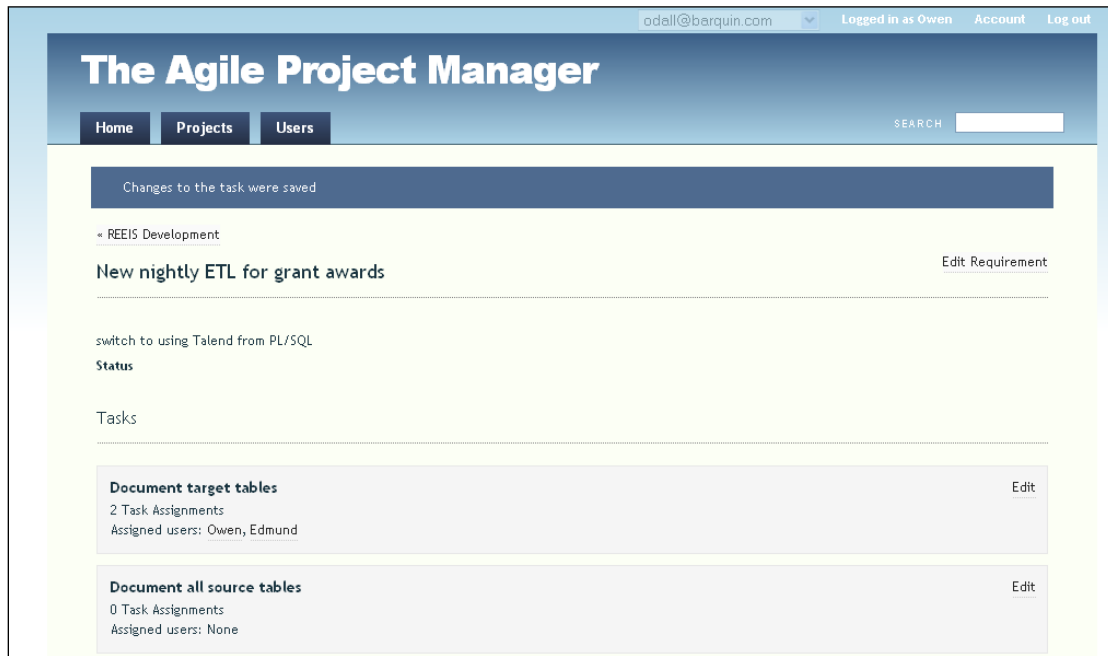


Figure 198: Viewing assigned users on a the Task card

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

Now that 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 there 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`.

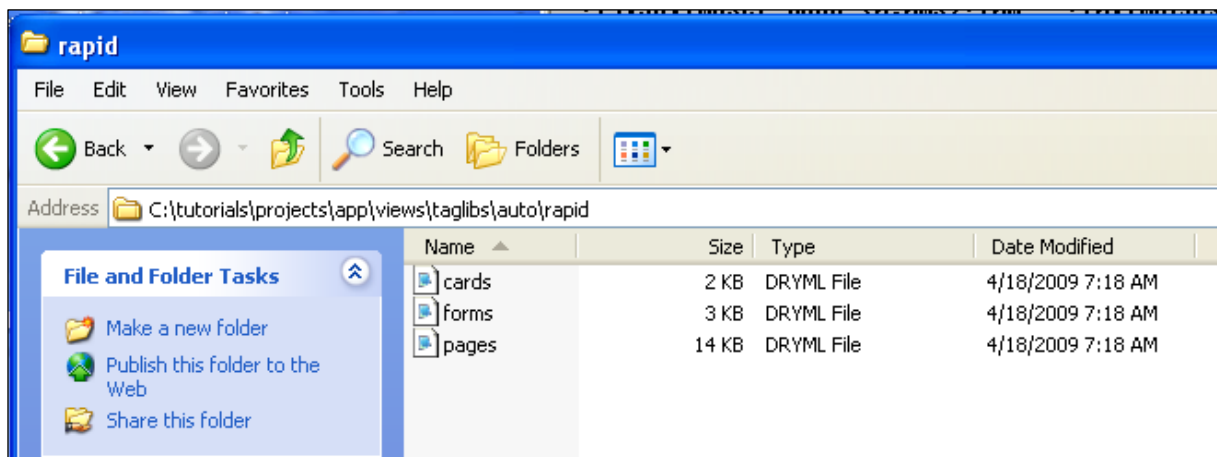


Figure 199: Listing the contents for the "\views\taglibs\auto\rapid" folder

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:

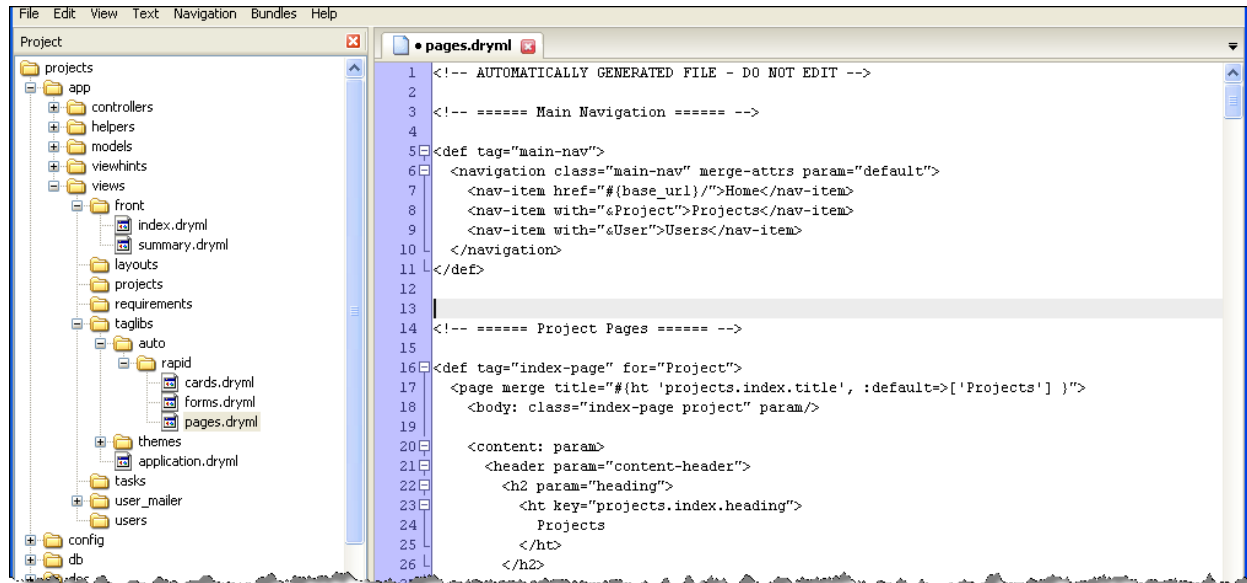


Figure 200: contents of the pages.dryml file

Now find the “show-page” tag for User:

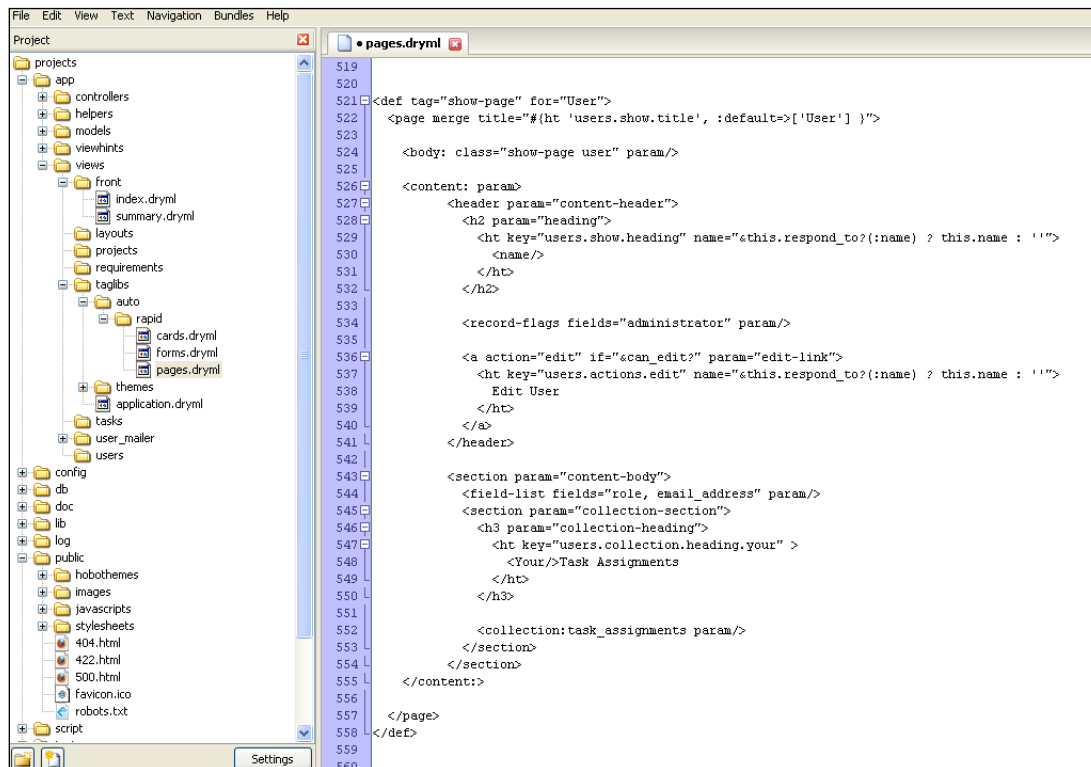
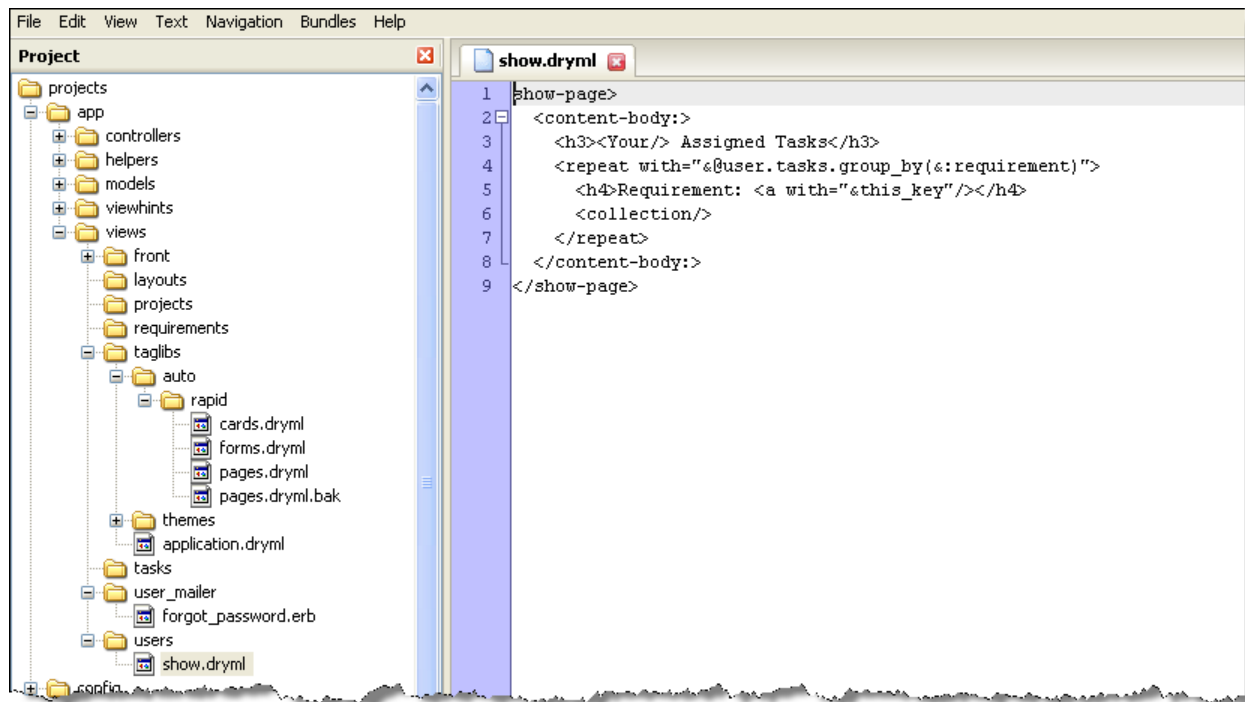


Figure 201: The auto-generated "show-page" tag for User in "pages.dryml"

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:

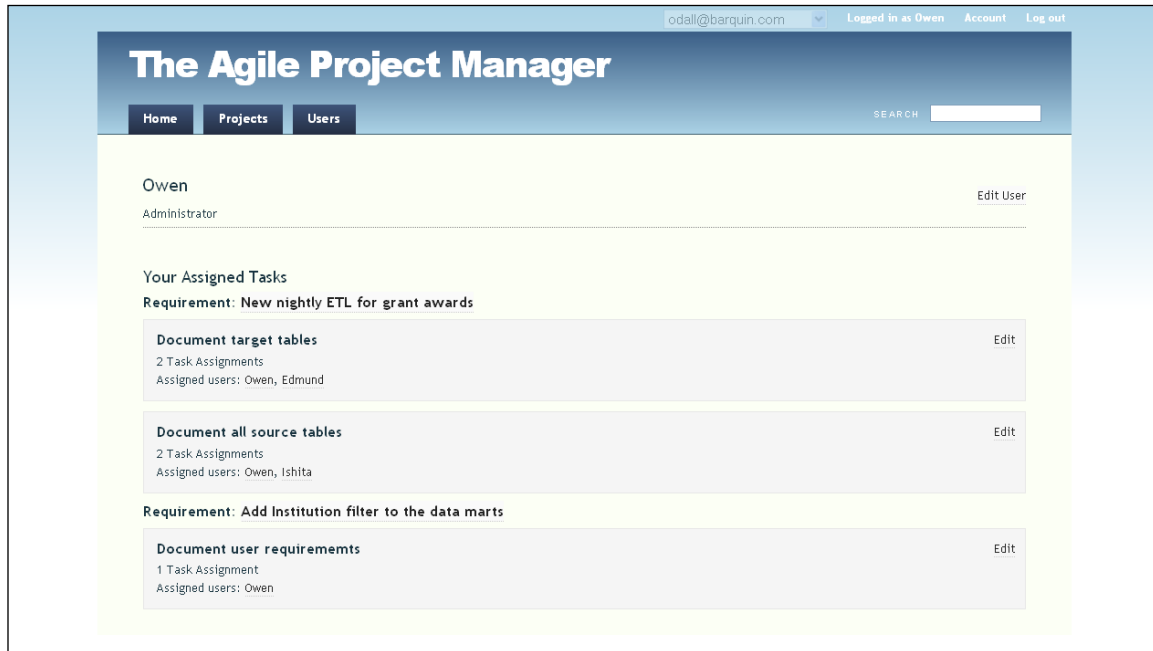


Figure 202: View of the enhanced User "show-page"

The `<Your>` tag is a handy little 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. Now click on the Users tab to see a summary of tasks for all users:

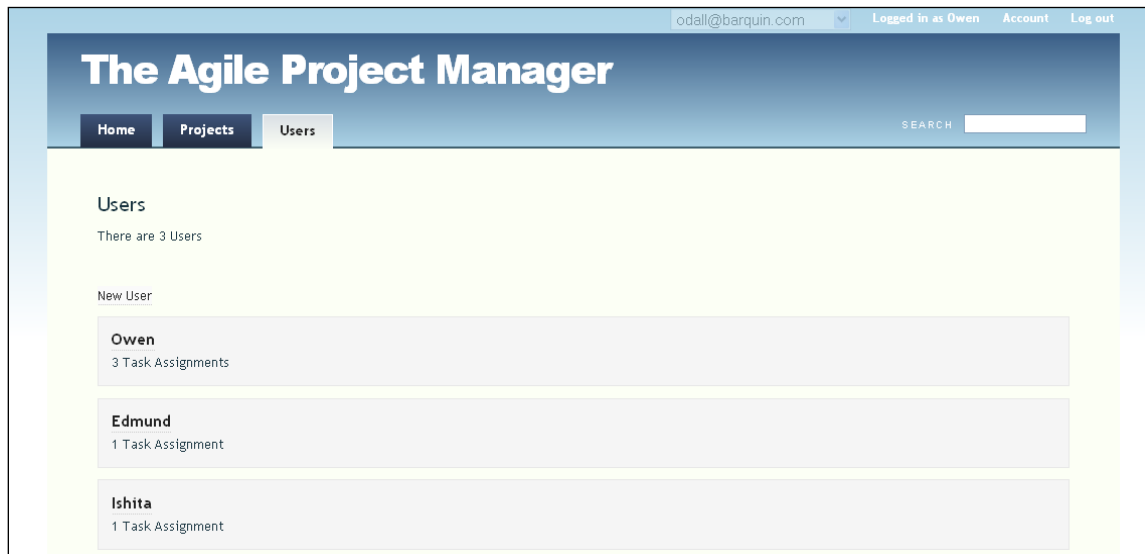


Figure 203: The Users tab showing all assignments

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 improve it a lot. 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 now 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”.

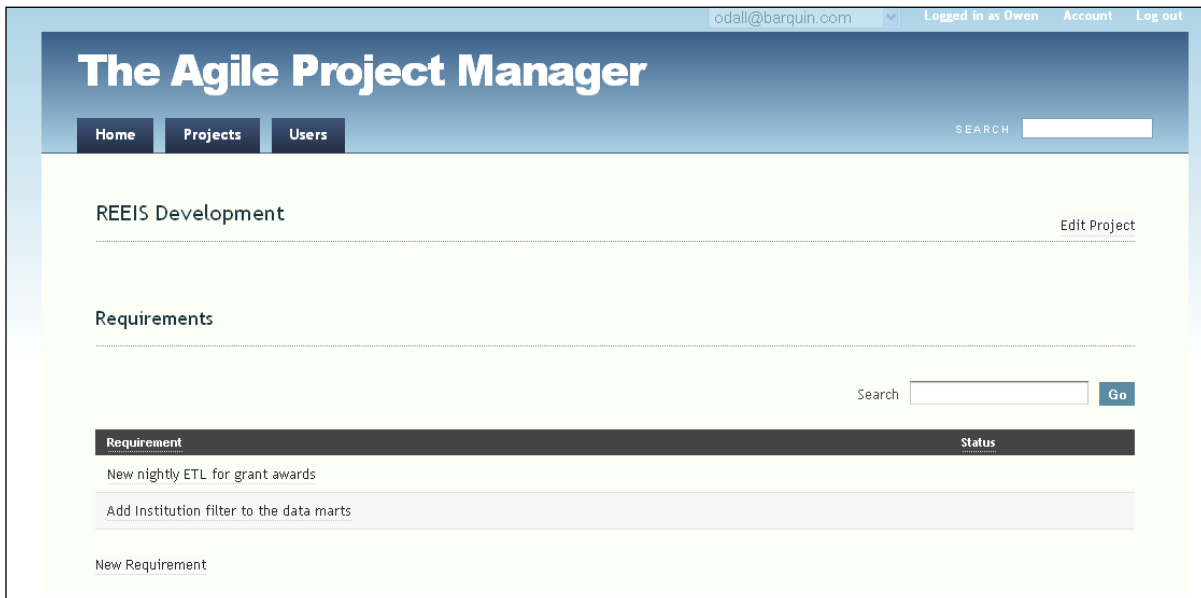


Figure 204: Using the Hobo “`<table-plus>`” feature to enhance the Requirements listing

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.apply_scopes(:search => [params[:search],
:status], :order_by => parse_sort_param(:title, :status))
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">
```

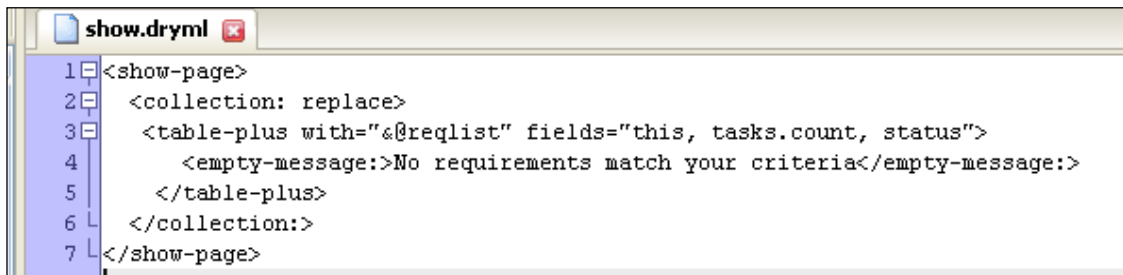


Figure 205: Enhancing the `<table-plus>` listing

Now enter a word in the Search box and see how the requirement list is filtered:

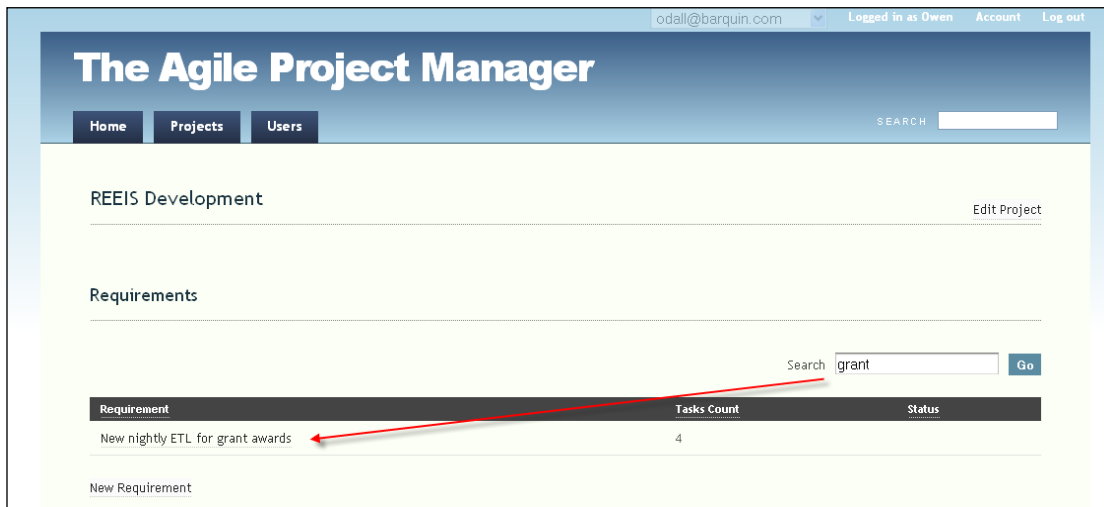


Figure 206: Using a search within the Requirements listing

Other Enhancements

We're now going to work through some more easy but very valuable enhancements to the application. We will add:

- A menu for requirement statuses. We'll do this first 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:

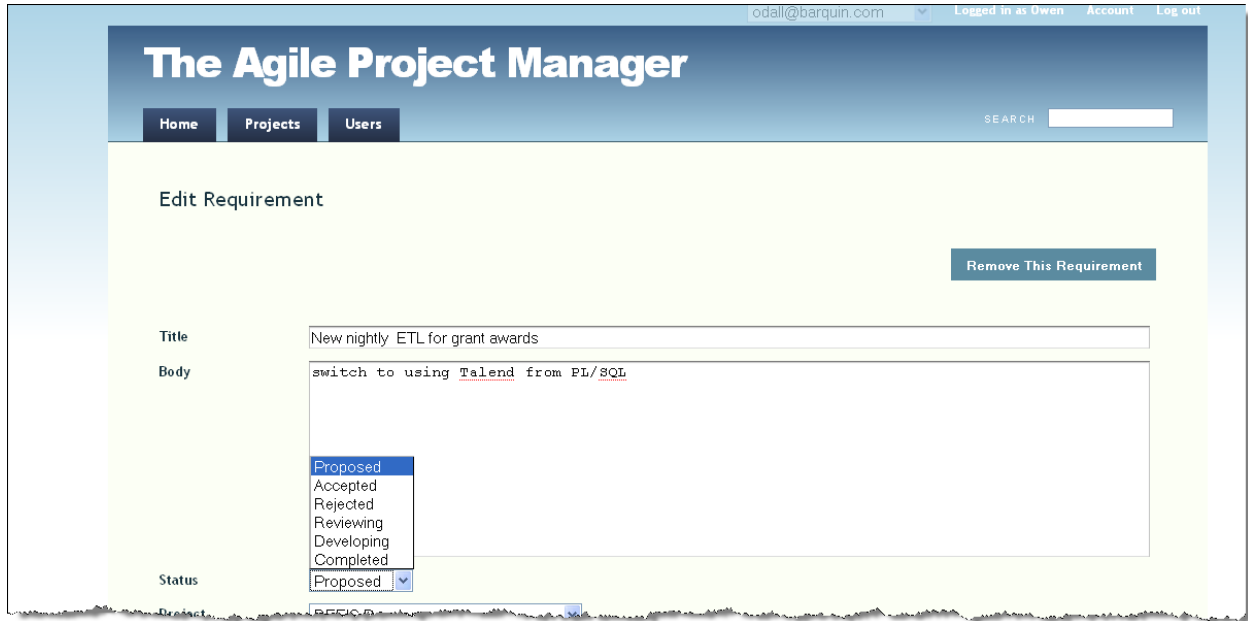


Figure 207: The Edit Requirement form with selectable status codes

The menu is working in the edit requirement page now. It would be nice though if we had an “AJAX-ified” editor right 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.

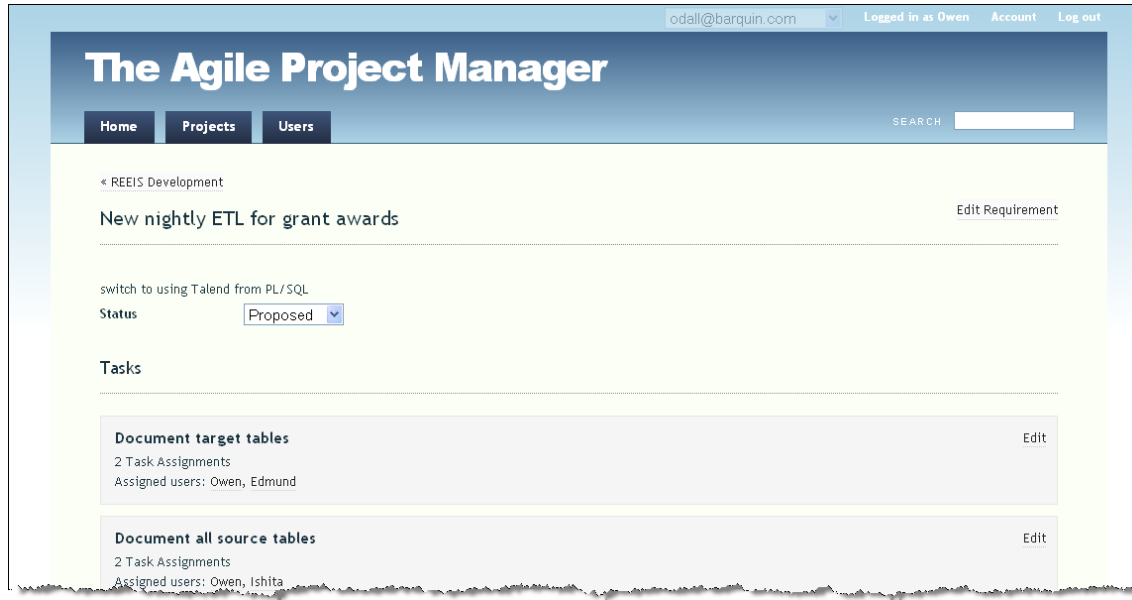


Figure 208: Creating an AJAX status update for Requirements

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:

```
> ruby script/generate hobo_model_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
```

Next, 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'
```

Now run the migration generator

```
> ruby script/generate hobo_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 lose with precious data.

So, for the case at hand, 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

Home Projects Requirement Statuses Users

SEARCH

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 |

Now that we've got more structured statuses, let's do something with them...

Reordering Tasks

We're now 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:

```
> ruby script/plugin install acts_as_list
```

Now 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, so you can just run it and you'll get the new position column on Task which is needed to keep track of ordering for you.

```
> ruby script/generate hobo_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="description, 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 it's 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:

```
> gem install validates_timeliness
```

Next update your `config\environment.rb` file by adding the following line:

```
config.gem 'validates_timeliness'
```

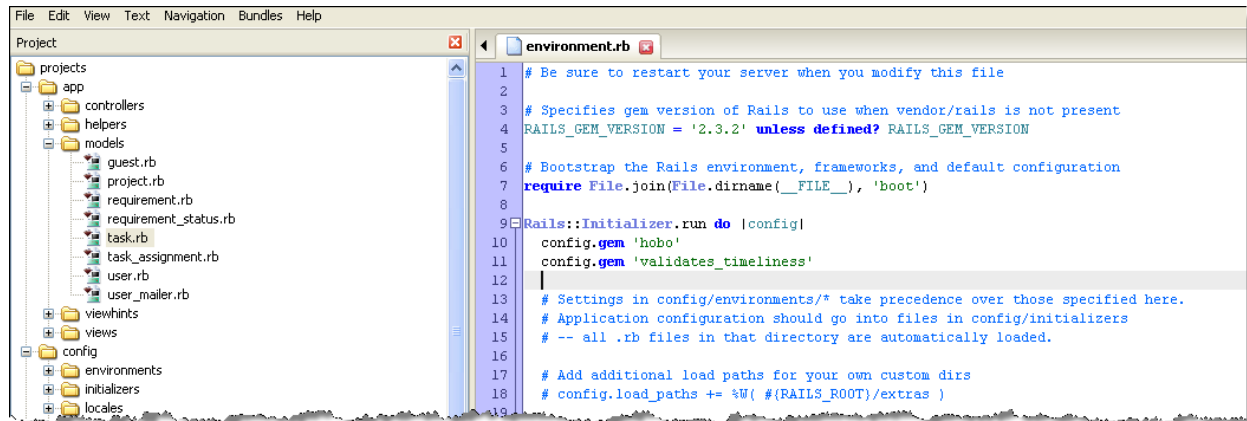
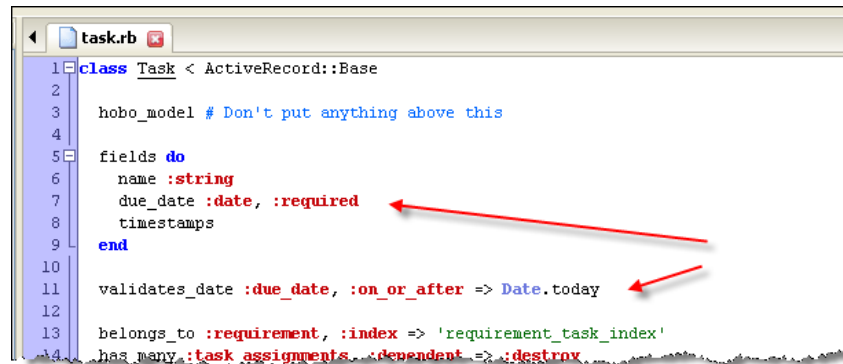


Figure 209: adding the "validates_timeliness" gem to "environment.rb"

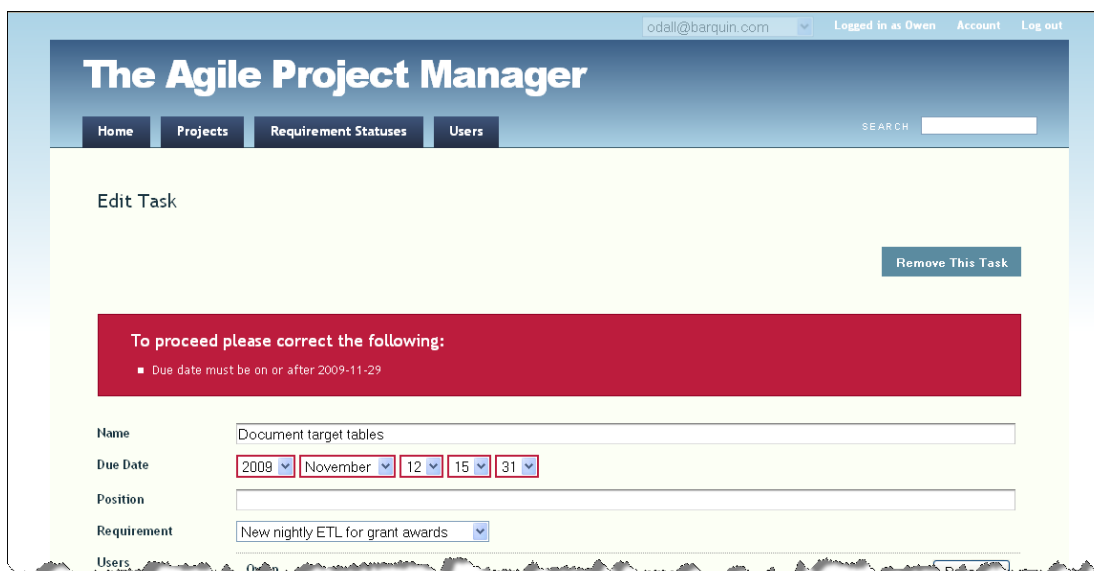
Now update your Task model with a due date, and add this validation for that date field:

```
validates_date :due_date, :on_or_after => Date.today
```



```
1 class Task < ActiveRecord::Base
2
3   hobo_model # Don't put anything above this
4
5   fields do
6     name :string
7     due_date :date, :required
8     timestamps
9   end
10
11   validates_date :due_date, :on_or_after => Date.today
12
13   belongs_to :requirement, :index => 'requirement_task_index'
14   has_many :task_assignments, :dependent => :destroy
```

Figure 210: Task model with "due_date" and a validation for the date



odall@barquin.com Logged in as Owen Account Log out

The Agile Project Manager

Home Projects Requirement Statuses Users SEARCH

Edit Task

Remove This Task

To proceed please correct the following:

- Due date must be on or after 2009-11-29

Name: Document target tables

Due Date: 2009 November 12 15 31

Position:

Requirement: New nightly ETL for grant awards

Users:

Figure 211: Error message from trying to enter a date earlier than today

Tutorial 18 – Using CKEditor (Rich Text) with Hobo

By Tola Awofolu

CKEditor is the new rich text editor that replaces the popular FCKeditor used by many web developers for years.

To use **CKEditor (3.x)**:

Download **CKEditor** from the download website: <http://www.ckeditor.com>

Extract the download from Step 1 to a new directory, `public/javascripts/ckeditor` in your Hobo application from the website:

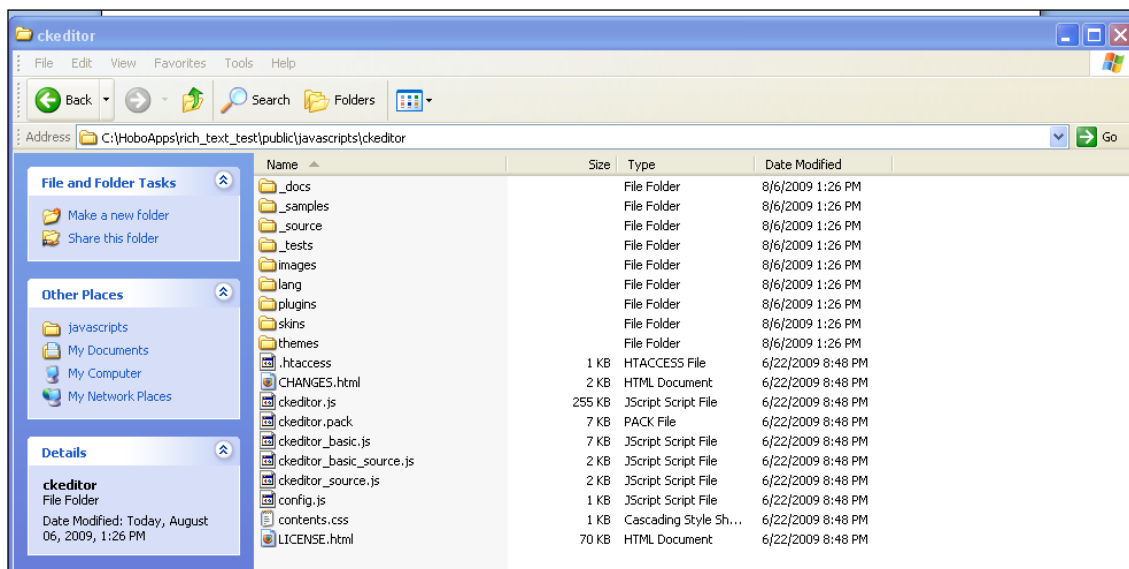


Figure 212: CKEditor source folder listing

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

```

Notice that the “standardToolBarConfig” portion of this JavaScript customizes the CKEditor toolbar options. Read the CKEditor documentation for more options you may wish to add.

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: class= ‘large’

Here’s an example of HTML markup that would be replaced:

```
<textarea id= "contact[notes]" class= "contact large"/>
```

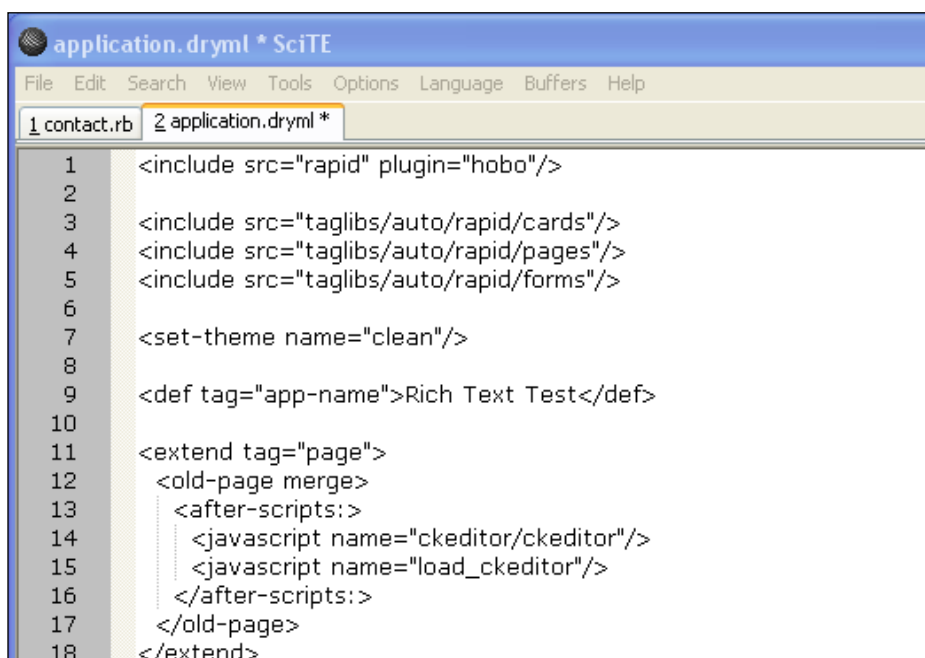
This HTML markup is automatically generated by Hobo for fields defined with the :html symbol in the model:

```
1 class Contact < ActiveRecord::Base
2   hobo_model # Don't put anything above this
3
4   fields do
5     name :string
6     notes :html
7     timestamps
8   end
9
10  #Permissions
```

Figure 213: 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>
```



The screenshot shows the SciTE editor with the file `application.dryml` open. 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 <def tag="app-name">Rich Text Test</def>
10
11 <extend tag="page">
12   <old-page merge>
13     <after-scripts:>
14       <javascript name="ckeditor/ckeditor"/>
15       <javascript name="load_ckeditor"/>
16     </after-scripts:>
17   </old-page>
18 </extend>
```

Figure 214: Adding the required CKEditor references in `application.dryml`

Now refresh your browser. Any field with the type “html” will now be displayed with an editor similar to the following:

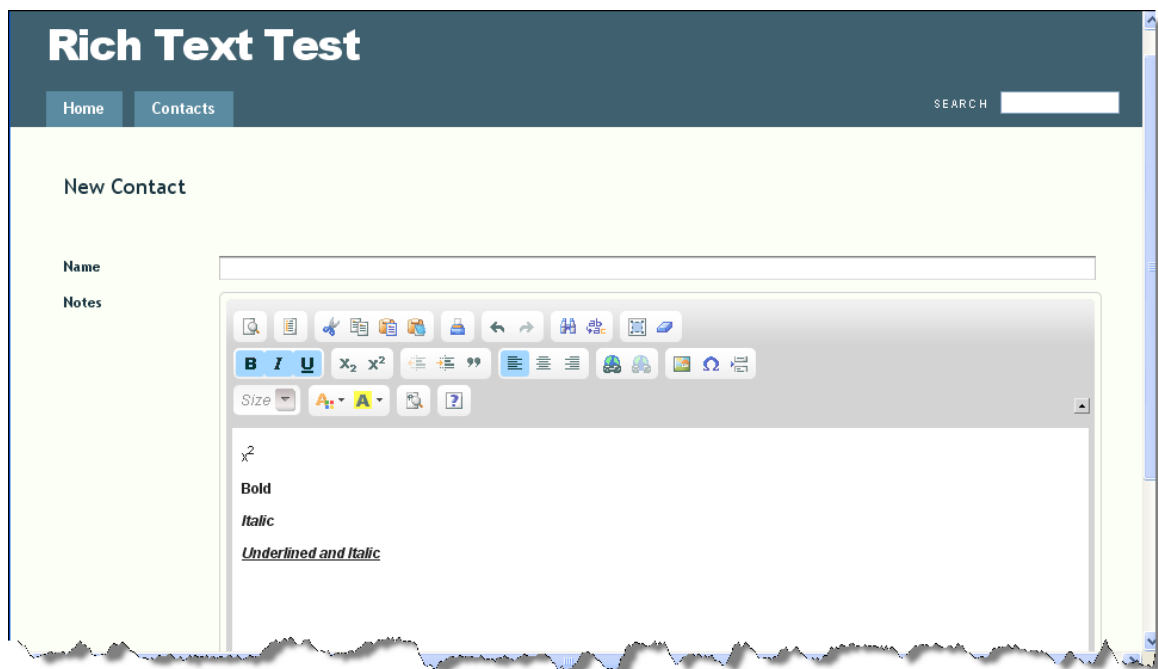


Figure 215: Sample Hobo form using CKEditor

Tutorial 19 – Using FusionCharts with Hobo

By Marcelo Giorgi

Many times we need to present to the user a nice visual presentation of some measure about our data that we need to check. We can achieve this, by using charts or graphs, and FusionCharts (<http://www.fusioncharts.com/>) are an excellent way to go!

Basically, FusionCharts offers a wide range of flash components that renders Charts of various types. The way we have to feed those flash components with our data is to create an XML file (with an specific format and semantics understood by FusionCharts) and then setting the URL for that file so that the Flash component (running on the client browser) can reach it.

In this tutorial, we start assuming you completed the last tutorial with the project called `four_table`. This way, we can leverage the existing models, and just focus on the Chart functionality. We'll be adding two charts to the project: i) Recipes By Country (which counts the number of recipes for each country) and ii) Recipes by Category (counts the amount of recipes grouped by categories).

Setting-Up FusionCharts in our Hobo application

First thing we need in order to use FusionCharts, is to submit a form within <http://www.fusioncharts.com/Download.asp>, as shown below:

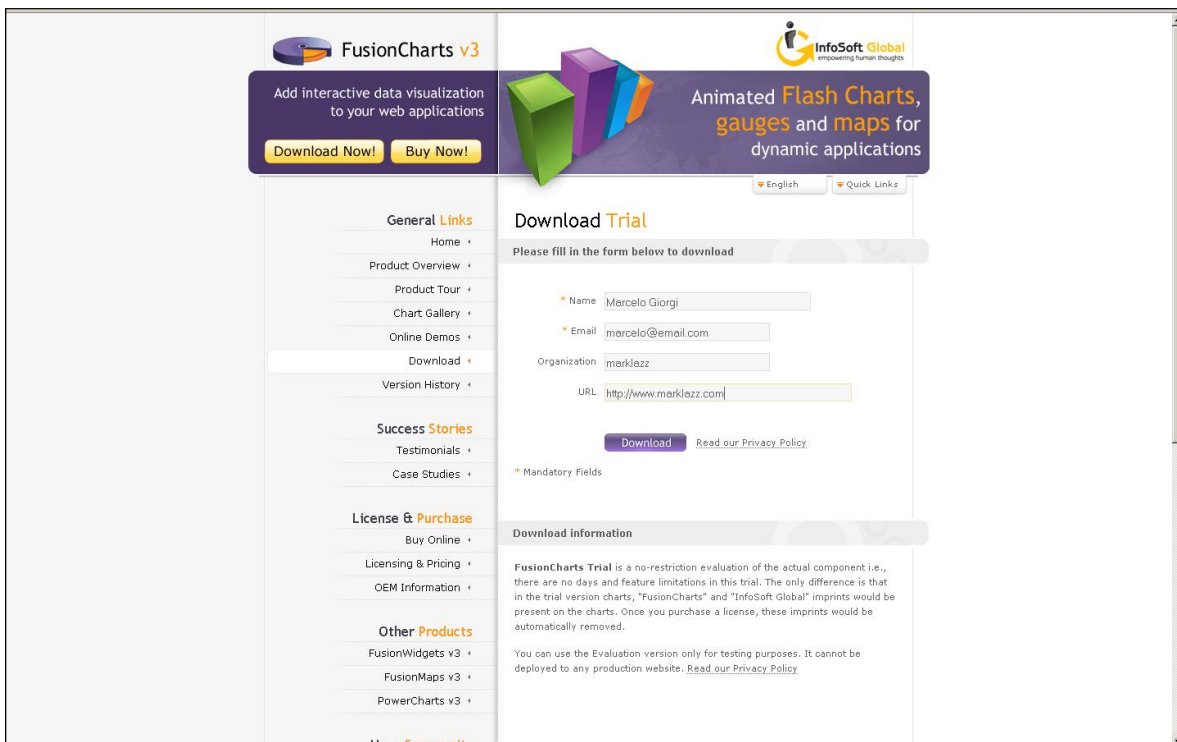
The image shows a screenshot of the FusionCharts v3 website's 'Download Trial' form. The page has a purple header with the FusionCharts logo and a navigation menu on the left. The main content area contains a form titled 'Download Trial' with fields for Name, Email, Organization, and URL. A 'Download' button is at the bottom of the form. Below the form, there is a section titled 'Download information' with text explaining the trial version's limitations. The footer of the page includes a 'User Community' link.

Figure 216: Registration form to request FusionCharts

After clicking on Download, you can download an evaluation copy.

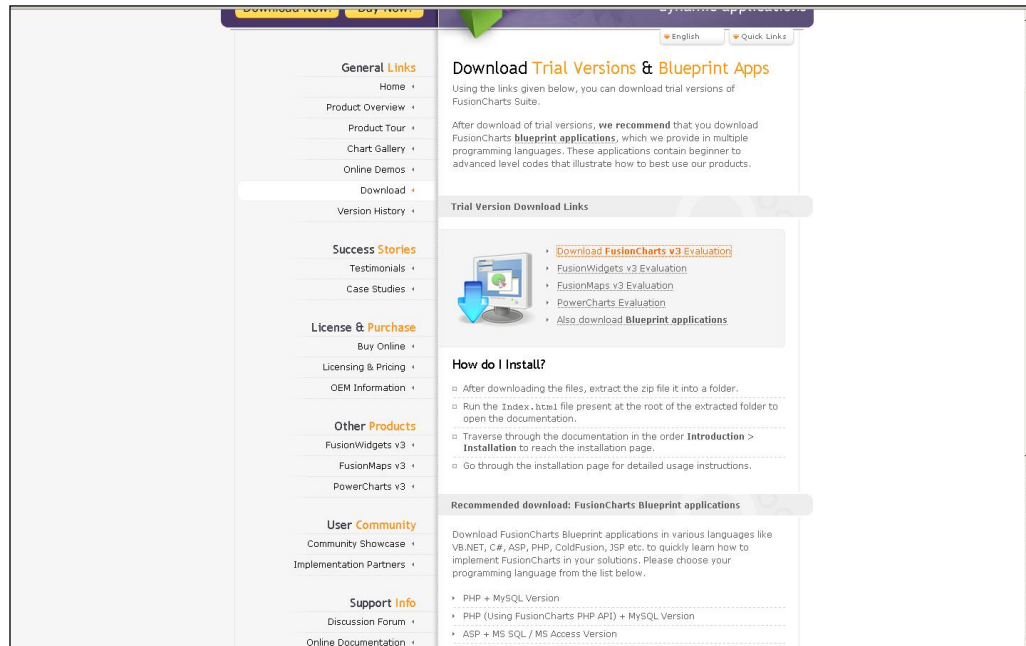


Figure 217: Download page for FusionCharts

After downloading (and decompressing it under, say in `c:\FusionChartsDistribution`) the trial version provided by the site (or a purchased one), there 2 steps that we must follow in order to complete the FusionCharts installation within our Hobo application:

1. Copy all `swf` files contained in the directory `c:\FusionChartsDistribution\Charts` to the public directory of our Hobo application, for simplicity we will paste them under `app/public/FusionCharts` directory.
2. Next, we should copy the file under `c:\FusionChartsDistribution\JSClass\FusionCharts.js` to `app\public\javascripts`

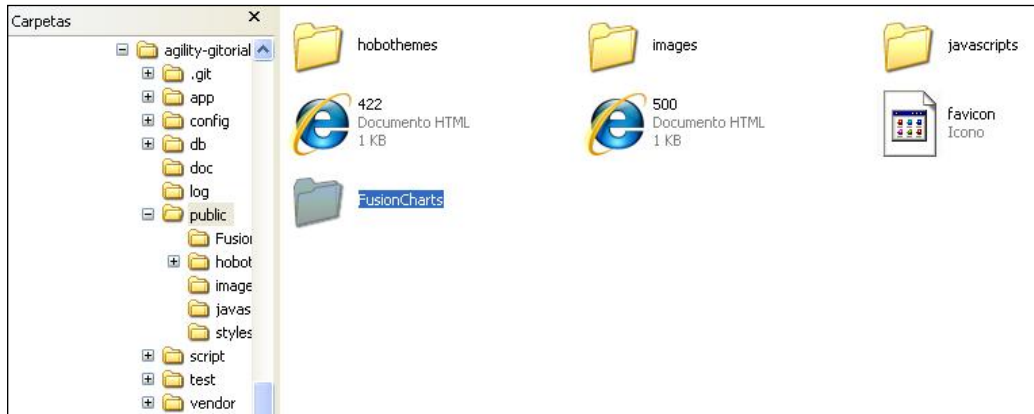


Figure 218: Target location for the FusionCharts SWF files

3. Finally, we are ready to reference the JavaScript file (copied in Step 2) in our `application.dryml` file, 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  <def tag="app-name">Four Tables, No Waiting</def>
10
11 <extend tag='page'>
12   <old-page merge>
13     <before-scripts:>
14       <javascript name='FusionCharts' />
15     </before-scripts:>
16   </old-page>
17 </extend>

```

Figure 219: Adding the required `<extend tag='page'>` definition in `application.dryml`

As you can see from the code of `application.dryml`, we extend the ‘page’ view so that we always include the JavaScript file `FusionCharts.js`. We could include this JavaScript at a page level, but for the purposes of this tutorial seemed more practical to do it this way.

Adding sample data

Before implementing the chart functionality, I’ve just created a random set of data to feed our charts, as you can see from the following picture:

| 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 220: Screen shot of sample recipe data for the tutorial

It is probably better to use the data presented here to make sure your charts would look the same as the ones for this tutorial. But, after that, feel free to change the values so you can see different chart types and options!

Recipes By Country

Mainly, in order to implement this chart, we would need to complete two steps:

1. Save the data to an XML file
2. Configure the Flash Component to retrieve the generated data.

1. Save the data to an XML file

For our first chart we need to modify the `RecipesController.rb`, in order to save the data (XML file) needed by the FusionCharts Flash component. In our particular case, we will put both Charts within the `index.dryml` view (`/recipes` path) as they both reflect information concerned with the collection of Recipes.

recipes_controller.rb

```
class RecipesController < ApplicationController

  hobo_model_controller
  before_filter :save_fusion_chart_data, :only => [:index]
  auto_actions :index, :show, :new, :edit, :create, :update, :destroy
  belongs_to :category
  belongs_to :recipe
  ...
end
```

As you can see (modifications are highlighted in bold italics), we add a new filter to store the XML file only when we receive a request for the index page.

Now we must define the function `save_fusion_chart_data` for this controller, in particular, we implement it like this:

```
1  private
2  def save_fusion_chart_data
3    @recipies_count_by_countries = Recipe.find(:all, :select => 'country_id,
count(*) as counter', :group => 'country_id')
4    filename = "#{RAILS_ROOT}/public/recipies_count_by_countries.xml"
5    xml_string = render_to_string(:partial =>
'chart_data_generator_for_count_by_country')
6    save_xml_file(filename, xml_string)
7  end
8  def save_xml_file(filename, data)
9    FileUtils.rm(filename, :force => true)
10   f = File.new(filename, 'w')
11   f.write(data)
12   f.close
13 end
```

Let's go through this code:

- In line #3, we define an instance variable that resolves the query of how many recipes are for each country. (You may be thinking that it would be better to encapsulate that behavior within Recipe's model, perhaps using a `named_scope`. I agree with you! But again, I'm still focusing on the implementation of FusionCharts functionality for this tutorial, so refactoring is your homework!)
- In line #4 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, and that's necessary because the file must be available so that the FusionCharts Flash component (on the client side) can load it.
- Line #5 is critical for this tutorial. Using the `render_as_string` method (included in Rails), it generates an XML string (based on the instance variable defined in line #3) with the appropriate semantics that FusionCharts needs.
- The final step, line #6, saves the string stored in the variable `xml_string` (which represent an XML file) into the path received as parameter.

Now, it's time to review the implementation of the partial that generates the XML string. Let's look at the code below.

`recipes/_chart_data_generator_for_count_by_country.builder`

```
1 xml.instruct!
2 xml.chart :caption => 'Recipies Count by Country' do
3   @recipes_count_by_countries.each do |recipe|
4     xml.set(:label => recipe.country.name, :value => recipe['counter'])
5   end
6 end
```

This builder extension instructs Rails to use the XML Builder component, so that we have the `xml` object, which is an instance of `XmlBuilder`, available that we use to build the XML structure. 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_bount_by_countries` it adds (within XML chart element) a *set* XML elements that contains both the name of the Country and a counter for the number of recipes for that entry.

Below we can check out a sample generated with that builder:

```
<?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>
```

2. Configure the Flash Component to retrieve the generated data

At this point, we have the data needed by our FusionCharts Flash Component ready to be used. We just need to instruct our FusionCharts Flash Component, by means of the JavaScript API available (thanks to the included file `FusionCharts.js`), to load it.

Here we show a simple snippet of `recipes/index.dryml` that exemplifies how we can accomplish that:

```

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 <script>
11 var chart_recipes_by_countries = new FusionCharts('http://localhost:3000/FusionCharts/Bar2D.swf', 'Recipes_Countries_Chart', '1000', '400');
12 chart_recipes_by_countries.setDataURL('http://localhost:3000/recipes_count_by_countries.xml');
13 chart_recipes_by_countries.render('recipes_count_by_countries');
14 </script>
15 </after-content:>
16 </index-page>
17
18
19

```

Figure 221: Content of recipes/index.dryml used to render the FusionChart

- First thing to notice is that 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 JavaScript API of FusionCharts 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, 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 equal to `recipes_count_by_countries`.

Ant 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:

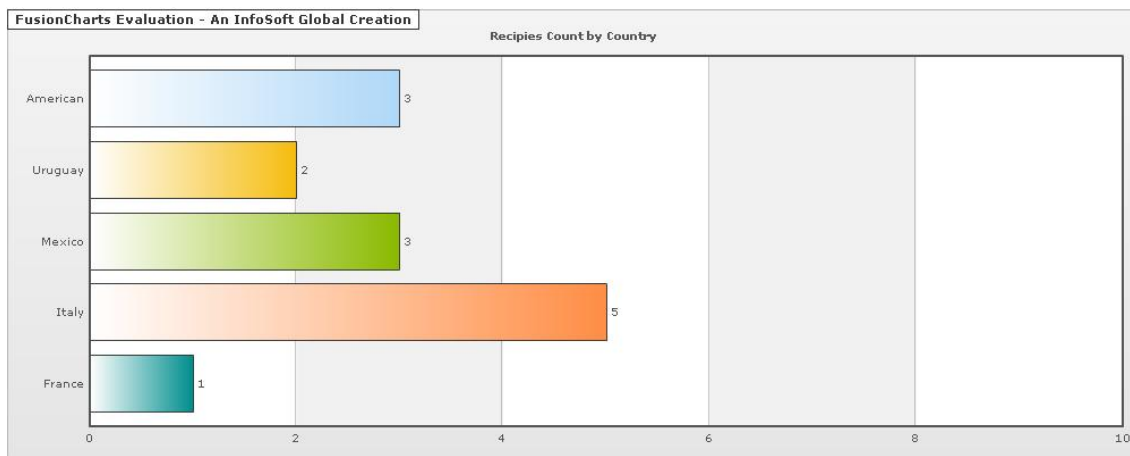


Figure 222: Screen shot of rendered FusionCharts bar chart

Recipes By Category

Now, it would be interesting to render 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 for this new chart.

1. 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:

`recipes_controller.rb`

```
1 private
2 def save_fusion_chart_data
3   @recipes_count_by_countries = Recipe.find(:all, :select => 'country_id,
count(*) as counter', :group => 'country_id')
4   filename = "#{RAILS_ROOT}/public/recipes_count_by_countries.xml"
5   xml_string = render_to_string(:partial =>
'chart_data_generator_for_count_by_country')
6   save_xml_file(filename, xml_string)
7   @recipes_count_by_categories = CategoryAssignment.find(:all, :select =>
'category_id, count(*) as counter', :group => 'category_id')
8   filename = "#{RAILS_ROOT}/public/recipes_count_by_categories.xml"
9   xml_string = render_to_string(:partial =>
'chart_data_generator_for_count_by_category')
10  save_xml_file(filename, xml_string)
11 end
12 def save_xml_file(filename, data)
13   FileUtils.rm(filename, :force => true)
14   f = File.new(filename, 'w')
15   f.write(data)
16   f.close
17 end
```

Again, statements marked with bold italics represent the modifications to the previous code. As you can see, these new lines just implement the same functionality as before, but 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, :value => 5
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 were different.

After adding this, we'll be generating both XML data files, each time a request to `Recipes#index` arrives.

2. Configure the Flash Component to retrieve the generated data

The only thing missing now to render this second chart is to add a placeholder for the flash and invoke the proper JavaScript to do the job for us. Below we show the last piece of the puzzle:



```
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 223: `recipe/index.dryml` to render a FusionCharts pie chart and bar chart

And then, we're done!! Here is the final result:

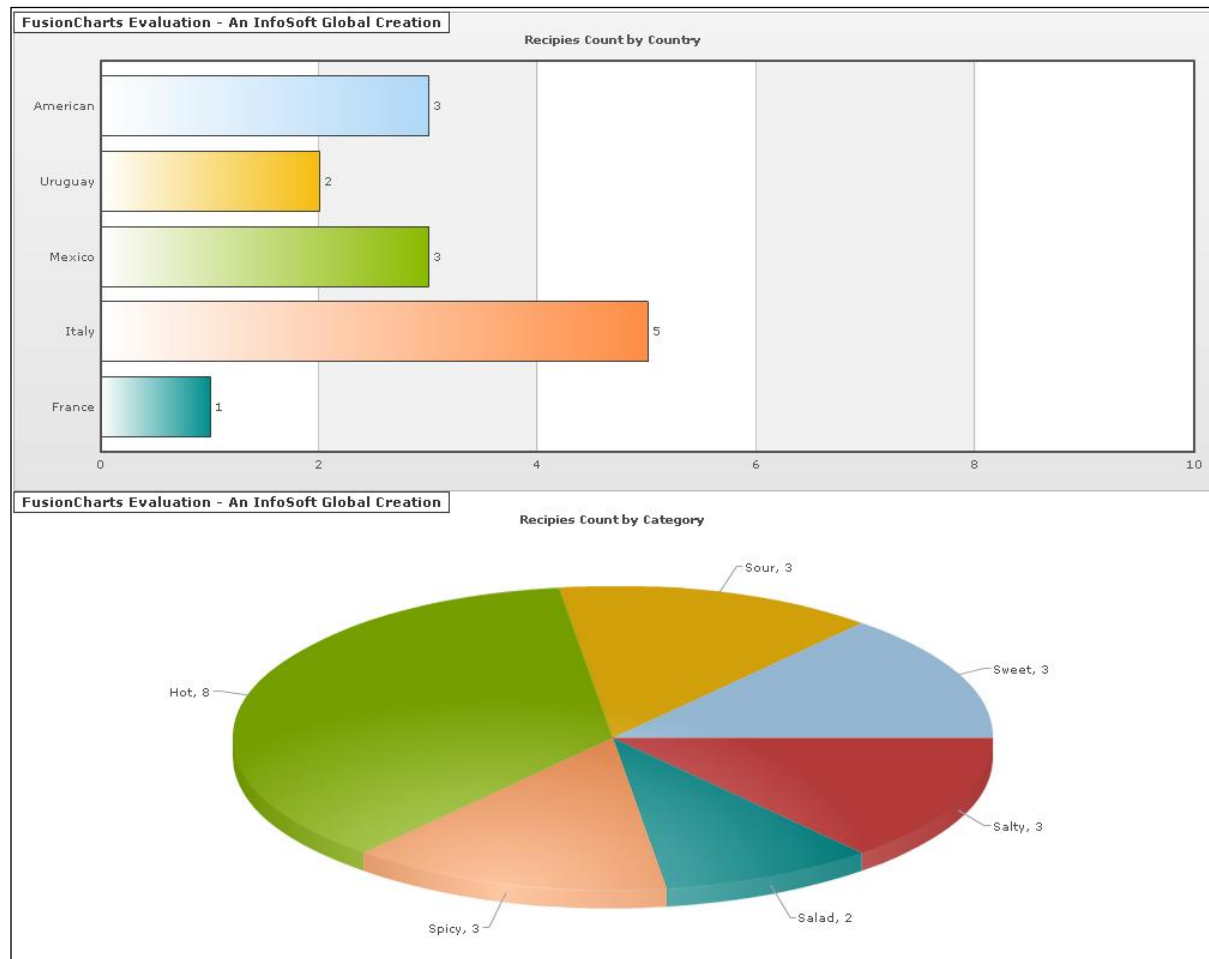


Figure 224: Screen shot of the rendered FusionCharts bar and pie charts

Have fun with FusionCharts!! And explore the different options here:

<http://www.fusioncharts.com/OnlineDocs.asp>

Tutorial 20 – Adding User Comments to Models

By Tiago Franco

Almost every application on the web allows users to post comments and provide feedback to almost every item (books, blog posts, other users, etc). This recipe will show you how to support user comments on Hobo.

Sometimes we want users to post comments to more than one table object. For example, suppose that we are developing a social network where users can enroll in basketball games and search for courts to play. We also want to allow users to post comments to games (e.g., users that didn't win sometimes like to blame the referee) or provide feedback about the court (e.g., if it was suitable or not). In this recipe we will be adding comments to both games and courts. Because we are focused on the comments, we will ignore the attributes of games and courts.

First, create an Hobo application named “comments-recipe”:

```
> hobo comments-recipe
```

Now, edit the file `application.dryml` (`app\views>taglibs`) and change the app-name to “comments' recipe”. We need to add an apostrophe to correct the spelling error, as shown below:

```
8
9   <def tag="app-name">Comments' Recipe</def>
10
```

Figure 225: Editing the application name for the Comments Recipe

We will now add a model class to support the management of basketball games. This can be done with the following command:

```
> ruby script/generate hobo_model_resource game
```

Don't forget to generate and run the migration. This can be done with:

```
> ruby script/generate hobo_migration --migrate --default-name create_games
> rake db:migrate
```

Let's run the application to perform a sanity check. We expect to see an image similar to the figure below.



Figure 226: Home page for the Comments Recipe

Notice the games entry on the menu. If it is there, it means that the games controller is working fine.

To add comments support to the application, we need follow similar steps. First, we need to create the model with:

```
> ruby script/generate hobo_model_resource comment
```

We will add the body attribute to hold the text of the user's comment. Edit the file *comment.rb* (app/models) and add the line number 6 as shown by the following figure:

```
5   fields do
6     body      :html, :required, :primary_content => true
7     timestamps
8   end
9
10  belongs_to :user, :creator => true
11  belongs_to :game, :accessible => true
12
13  # --- Permissions --- #
```

Figure 227: Adding Body and Game to Comments

Additionally, add line 10 and 11 from the same figure. Line 10 is used to keep track of the user that created the comment, while line 11 records the game that is being commented.

Some applications allow users to edit or delete their comments. But they never let a user change comments made by someone else. So we need to update the permissions

of our comment model. Just edit the `comment.rb` (`app/model`) and make sure the permissions are like the ones shown on the figure below:

```
13 # --- Permissions --- #
14
15 def create_permitted?
16   acting_user.signed_up? && user == acting_user
17 end
18
19 def update_permitted?
20   acting_user.administrator? || (acting_user == user && !user_changed?)
21 end
22
23 def destroy_permitted?
24   acting_user.administrator? || acting_user == user
25 end
26
27 def view_permitted?(attribute)
28   true
29 end
```

Figure 228: Permissions for the Comment model

Now, we only want users to create, edit or browse comments if a game is being shown (i.e. in `game/show` view). So we need to update line 5 of `comments_controller.rb` (`app/controllers`) from:

```
auto_actions :all
```

To:

```
auto_actions :destroy
```

The result is shown on the figure below:

```
1 class CommentsController < ApplicationController
2
3   hobo_model_controller
4
5   auto_actions :destroy
6
7   auto_actions_for :game, [:create]
8
9 end
```

Figure 229: The `auto_actions` for the `comments_controller`

Line 7 also needs to be added, to allow comments to be created from the `game/show` view. Without this line the user won't be able to comment a game when it is being displayed. Add the line to `app/controllers/comments_controller.rb`.

We now need to deal with the game/comment relation on the other end. Edit the file `app/models/game.rb` (and add line 10):

```
5      fields do
6
7          timestamps
8      end
9
10     has_many :comments, :dependent => :destroy
11
```

Figure 230: Adding comments to the Game model

We're just two steps away from testing our new feature: create and run the migration. But we already know how to do that. We need to execute the following commands in the command line:

```
> ruby script\generate hobo_migration --migrate --default-name create_comments
> rake db:migrate
```

And we should be ready for a test drive. Create a user account (if you haven't already done it), create a game and add two comments. The result should be something similar to:

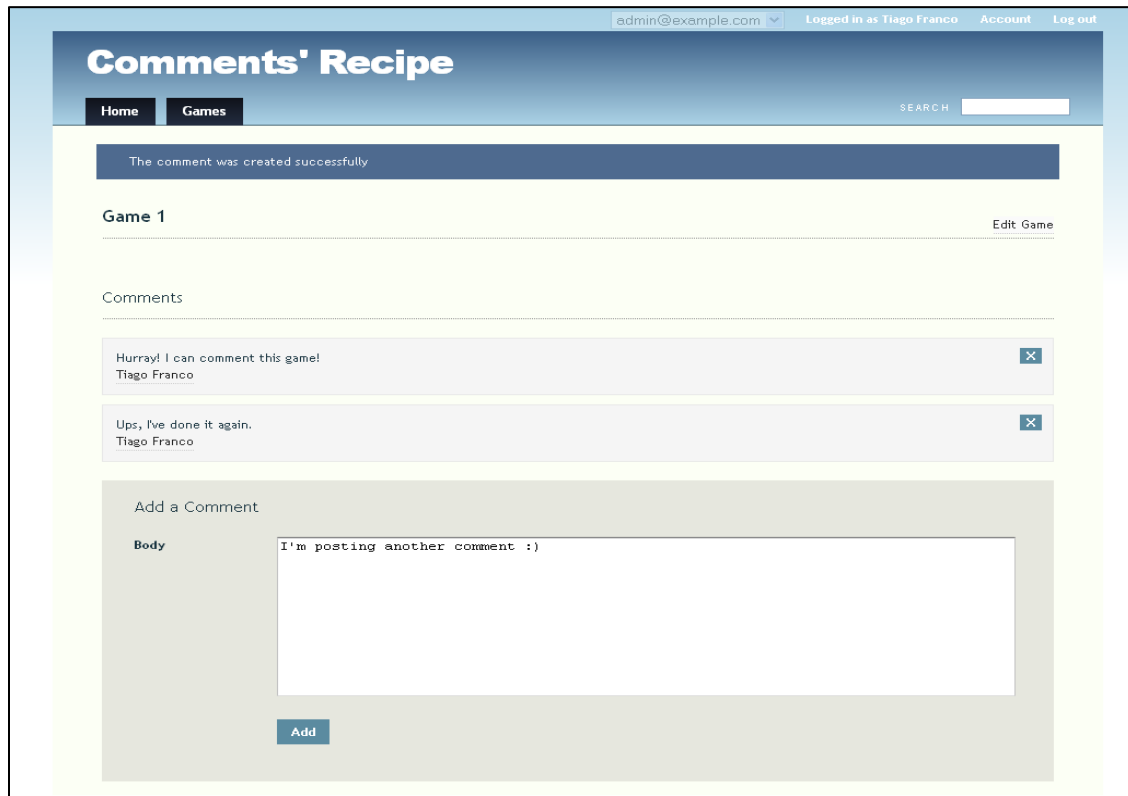


Figure 231: Posting comments about a game

Easy isn't it? So let's not waste time and start working on the courts!

Let's create a model to store the courts on our database.

```
> ruby script/generate hobo_model_resource court
```

Because we are not interested in the details of the courts, let's just create and run the migration:

```
> ruby script/generate hobo_migration --migrate --default-name create_courts
> rake db:migrate
```

Et voila! As we can see in the figure below the application can now store courts.

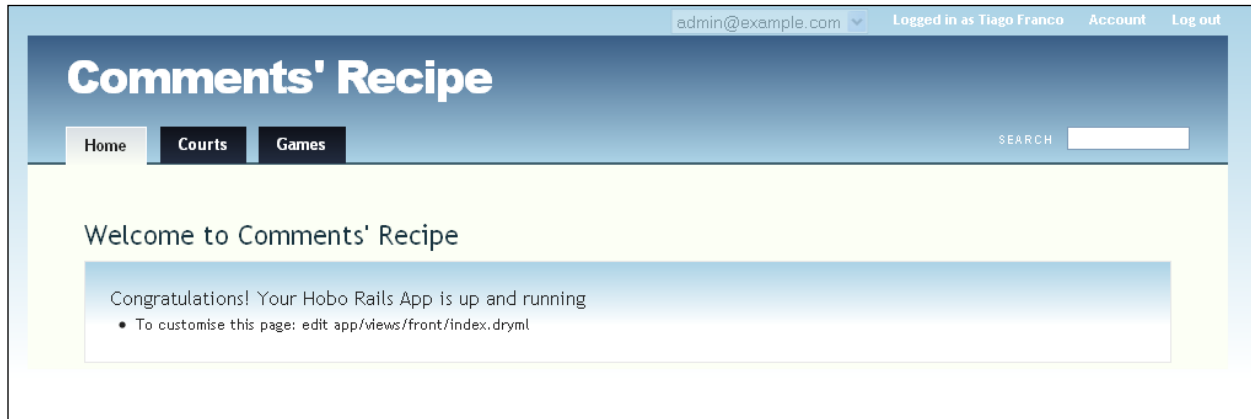


Figure 232: Comments' Recipe with support for courts

Now we need to update the existing infrastructure to allow users to comment the courts. Since we already have a comment model, let's just make a few updates so that it can also be related with a court.

First, we need to update the existing comment model. Add the contents of line 12 on figure below to the file `comment.rb` (in `app/models`). This will allow a comment to be related with a court.

```
9
10     belongs_to :user, :creator => true
11     belongs_to :game, :accessible => true
12     belongs_to :court, :accessible => true
13
14     never_show :game, :court
```

Figure 233: Adding courts to comments

Then update the court model, file `court.rb` (in `app/models`), to deal with the other end of the one-to-many relationship. Update the file with the contents of line 10:

```
9
10     has_many :comments, :dependent => :destroy
11
12     # --- Permissions --- #
```

Figure 234: Adding comments to courts

We now need to update the `comments_controller` to allow the creation of comments in the court/show page. Add line 8 as seen in the figure below to the file `comments_controller.rb` (in `app/controllers`).

```
1  class CommentsController < ApplicationController
2
3      hobo_model_controller
4
5      auto_actions :destroy
6
7      auto_actions_for :game, [:create]
8      auto_actions_for :court, [:create]
9
10 end
```

Figure 235: Modifying `auto_actions` for the `comments_controller` (allow court)

Finally, create and run the migrations using the following commands:

```
> ruby script/generate hobo_migration --migrate --default-name
add_comments_to_courts
> rake db:migrate
```

Now, create a court and insert a new comment. Hmm... it seems that the application is asking to add a game to the comment. By default Hobo auto-generates forms to fill every attribute on the model. We need to tell the framework not to show the game list-box on the new comment form.

This can be performed by adding line 14 below `comment.rb` (`app/models`).

```
12      belongs_to :court, :accessible => true
13
14      never_show :game, :court
15
16      # --- Permissions --- #
```

Figure 236: Hiding court and game in the comment's form

Now you will be able to see something like the following:

The screenshot shows a web application interface. At the top, there's a header with the title "Comments' Recipe" and navigation links: Home, Games, and Pitches. A search bar is also present. Below the header, the main content area displays "Pitch 2" with an "Edit Pitch" link. Underneath, a section titled "Comments" shows two existing comments by "Tiago Franco". The first comment says "The best field in Lisbon." and the second says "Want to find a great place to play in Lisbon? Pick this one, I've been there several times and it has a great floor. The lockers and showers are very clean. The staff is nice and helpful and the price is below average. Thumbs up!". Below the comments, there's a form to "Add a Comment" with a "Body" label and a large text input area, and an "Add" button.

Figure 237: View of the in-line "Add a Comment" form

In this recipe we have learned how to support comments to the application models. The example was performed with games and courts, but can easily be mapped to any Hobo based application in the wild.

Tutorial 21 – Replicating the Look and Feel of a Site

By Tom Locke

Introduction

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 this look and feel happen to our new app almost ‘automatically’. We want to be able to develop at “Hobo speed”, and have the look and feel “just happen”. This is not trivial to set up, but once it is, the pay-back in terms of development agility will be more than worth it. That is the topic of this chapter.

We’ll use the example of 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 at least, this recipe will document how to create a *close approximation* to this theme. In particular, we’re going to skip some of the details that cannot be implemented without resorting to images. This is just to keep the recipe getting too long and complicated.

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 something 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 this one idea can be understood and applied, you're well on the 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 this principle. So this is probably just a recap, but to remind ourselves 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 so on. In a Hobo app the presentation is handled essentially the same way as with any app, with CSS stylesheets and image assets.

Having said that, we need to inject a note of pragmatism:

- Humans being 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

We'll start with a look at the elements of the existing site that we'll need to replicate. The main ones are:

A banner image:

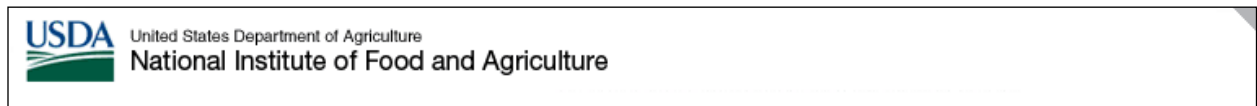


Figure 240: The NIFA banner image

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:



Figure 243: NIFA navigation panels

And more navigation in the page footer:



Figure 244: NIFA footer navigation

One of the important things to notice at this stage, is that this 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 content elements and presentation.

That means we're going to be creating three things to capture this 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, and the various images in the page are present in the markup as `` tags. In other words, the existing markup is very *presentational*.

So 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 makes it a great deal easier to jump from one Hobo app to the next.

Building the new app

Let's do this properly and actually 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, you should use Firefox and the Firebug extension you can find at <http://getfirebug.com>



```
> hobo nifa-demo  
> cd nifa-demo  
> ruby script/generate hobo_migration
```

If you fire up the server, you'll see the default Hobo app of course:

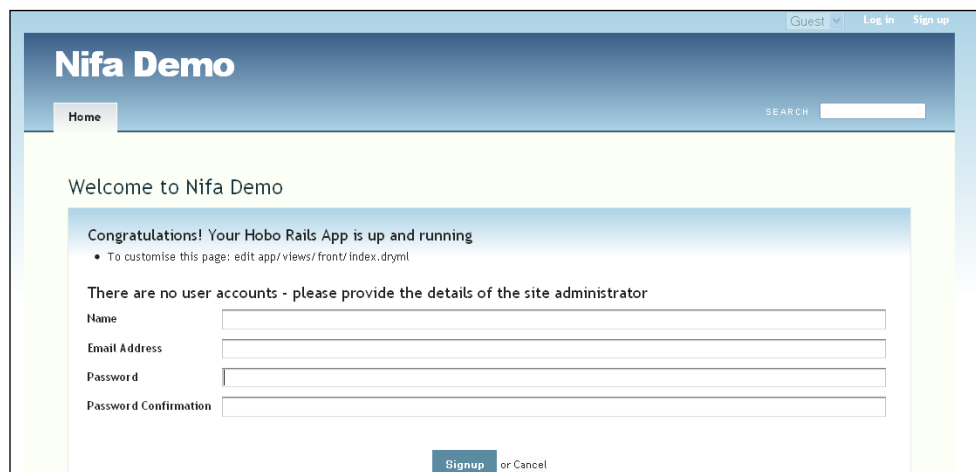


Figure 245: The NIFA Demo default home page

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:

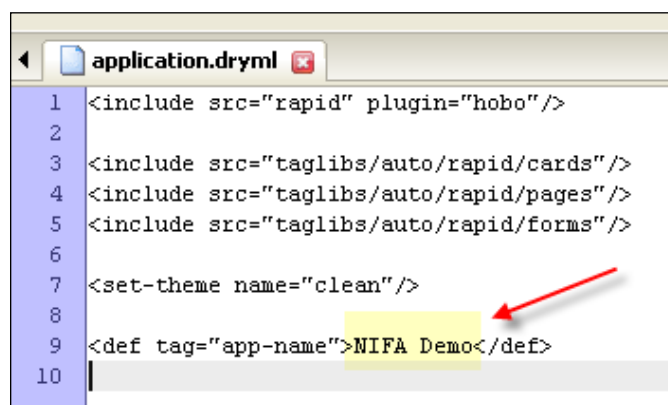


Figure 246: Using the "app-name" tag to change the default application name

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:

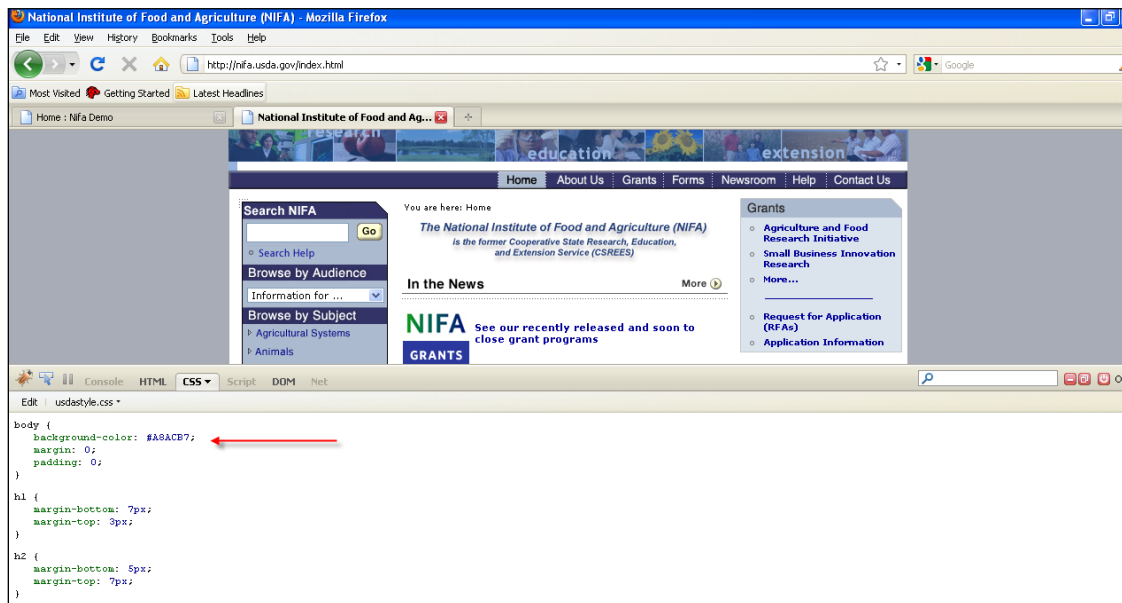


Figure 247: Using Firebug to locate the background color

Now 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:

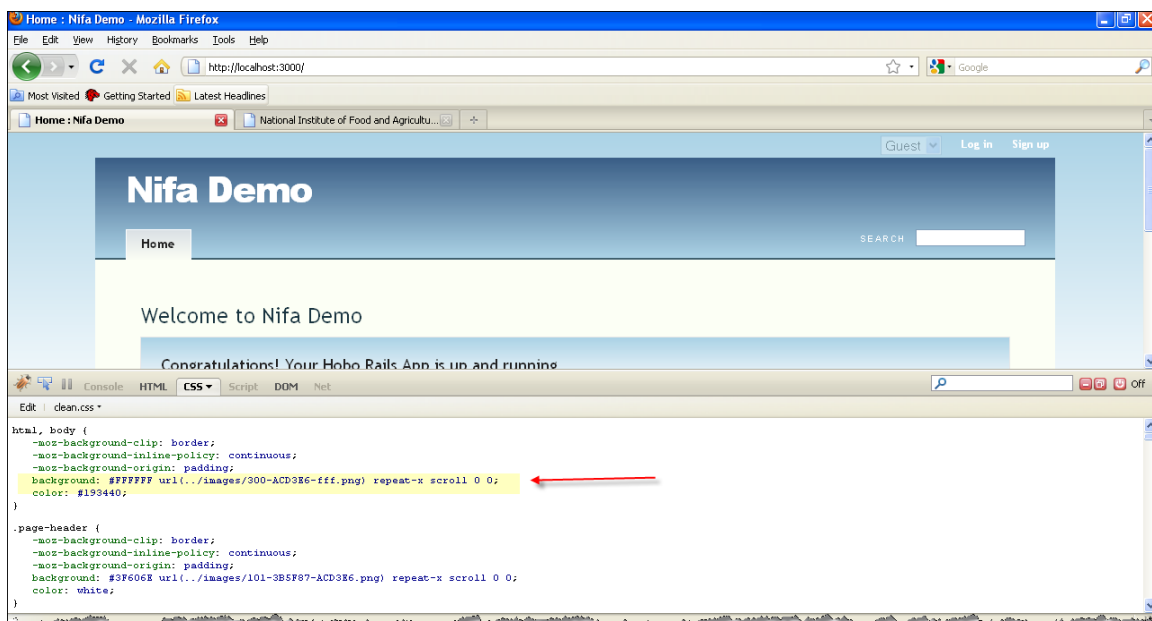


Figure 248: Using Firebug to find the images used by Hobo for the default background

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 }
```

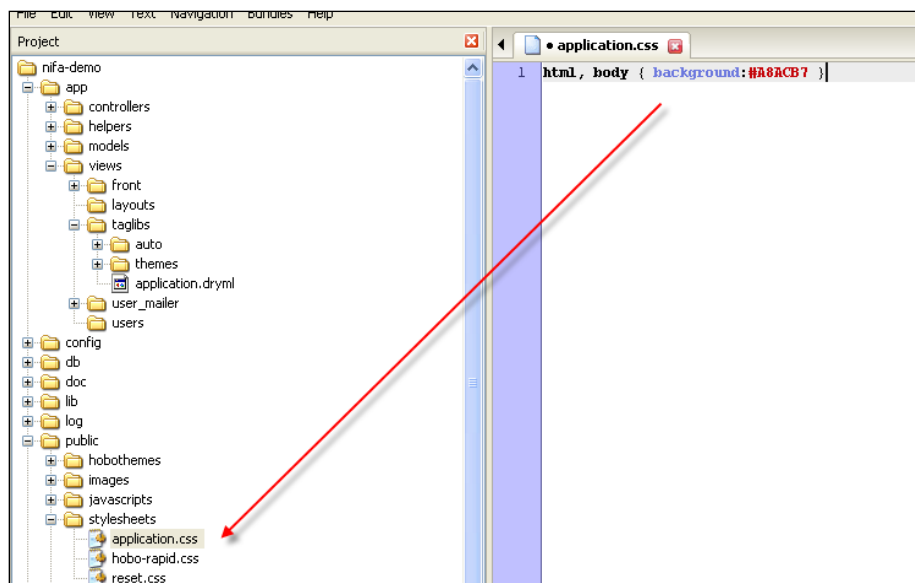


Figure 249: Adding the new background color to "application.css"

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. So in `application.css` I add

```
body { width: 766px; }
```

Note we've not changed any markup yet - that's how we like it.

Account navigation

These are the log-in and sign-up links in the top right. They are not on the NIFA site, but if 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`:

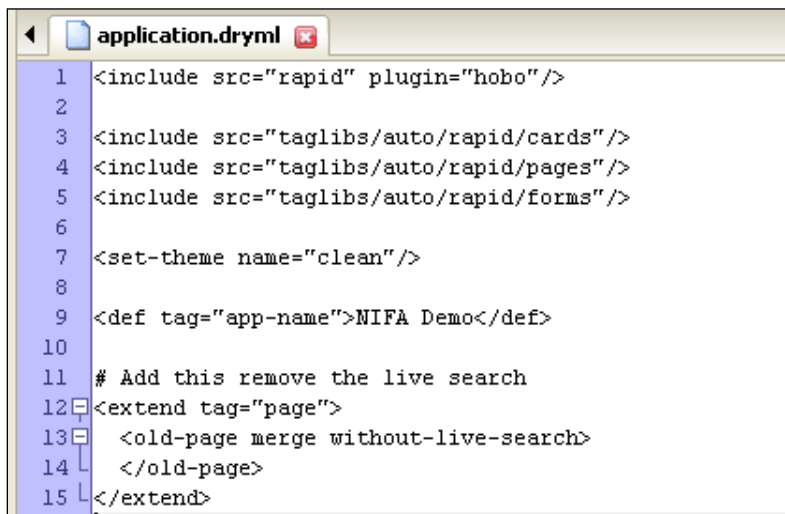


Figure 250: First pass at modifying "application.dryml"

```

<extend tag="page">
  <old-page merge without-live-search>
  </old-page>
</extend>

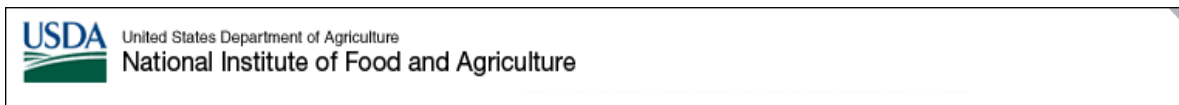
```

So 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.

This one:



And this one:

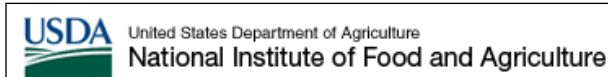


Figure 251: The two images used in NIFA's top banner

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, but to understand our approach, first take a look at a simplified view of the page markup that we're working with:

```
<html>
  <head>...</head>
  <body>
    <div class="page-header">
      <h1 class="app-name">NIFA Demo</h1>
    </div>
    ...
  </body>
</html>
```

Notice that this image:



Is essentially a graphical version of that `<h1>` tag, so we'll use CSS to make that same `<h1>` be rendered as an image. The existing text will 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;
}
```

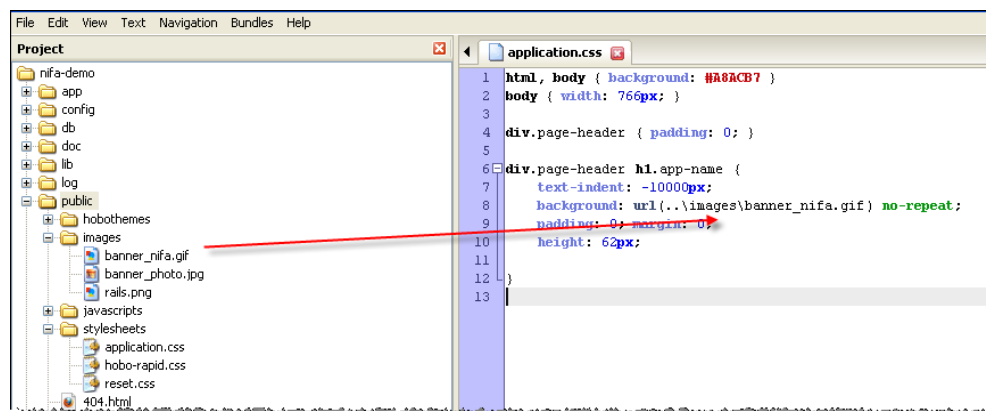


Figure 252: How to reference the banner gif in "application.css"

OK that was a bit of a leap. Why `padding: 0px` for the page-header, for example? The fact of the matter is that working with CSS is all about trial and error. Using Firebug to figure out what rules are currently in effect, flipping back and forth between the stylesheet in your editor and the browser. Try experimenting by taking some of those rules out and you'll see why each 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;
}
```

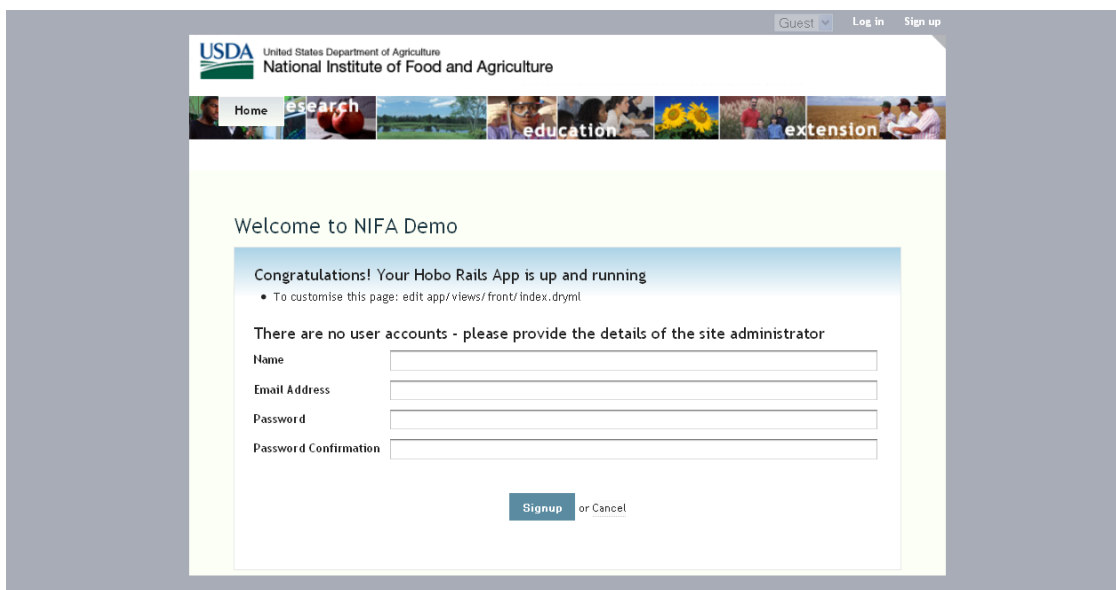


Figure 253: View of the NIFA Demo login page

Taking shape now, except the main navigation panel (“Home” tag) is hovering on top of the photos:

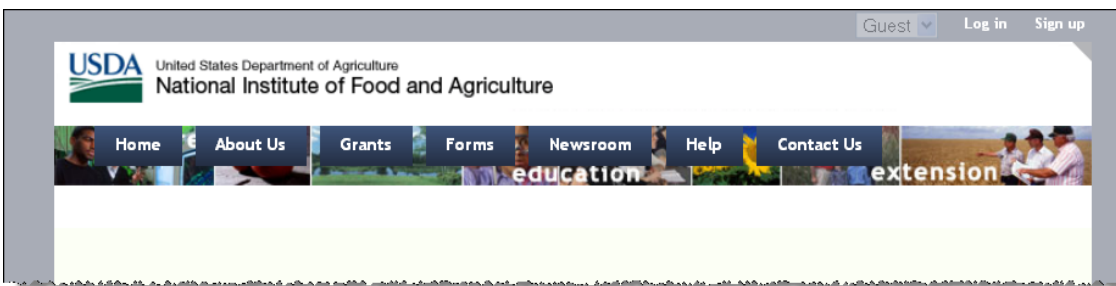


Figure 254: The Navigation Panel before refactoring

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. So 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 better - cleaner, clearer and more professional looking.

Our app only has a home page right now, so first let’s define a fake navigation bar to work with. 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>
```

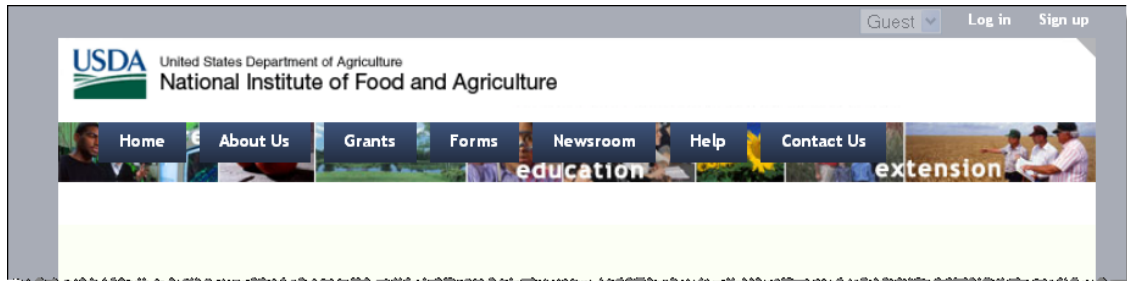


Figure 255: View of our first pass at the main navigation menu

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 at this point:

- 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 needs fixing
- The font needs to be smaller, and not bold
- The background color needs to change, as do the colors when you mouse-over a link

Now this is not a CSS tutorial, so we're not going to explain every last detail, but 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;
}
```

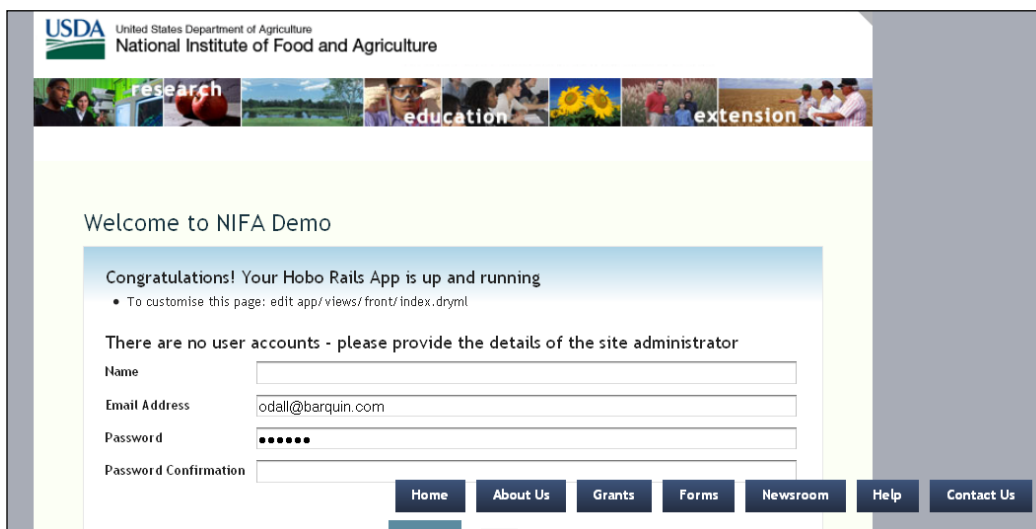


Figure 256: Still need more to fix the top navigation menu...

The nav-bar still looks wrong. We'll now 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:

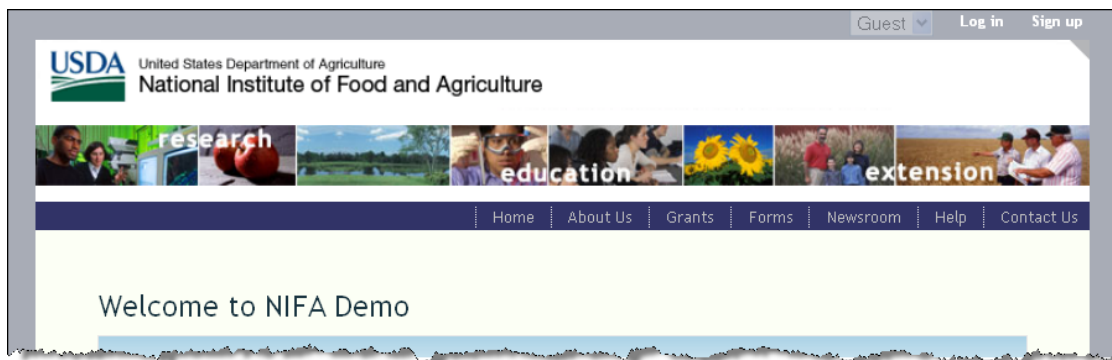


Figure 257: The fixed NIFA man navigation bar

The sidebars

The existing site has both left and right sidebars. We’ll add those now. The 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, we’ll 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:

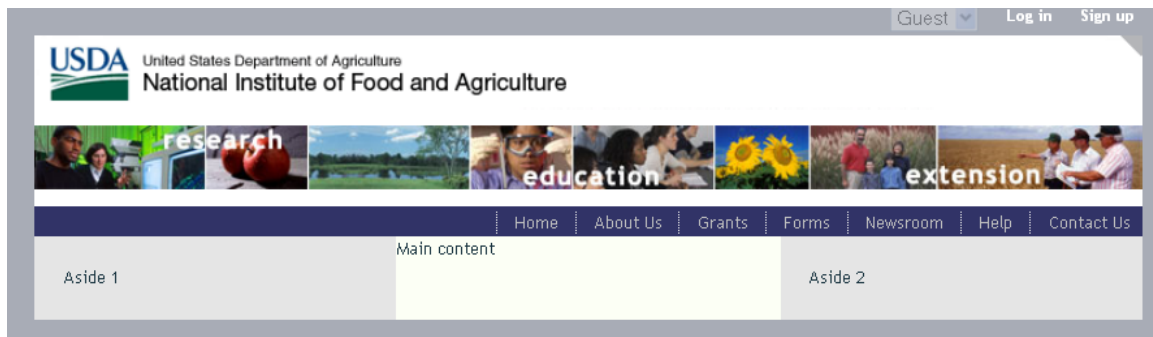


Figure 258: View of the default three-column formatting

Obviously we’ve got 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, so 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>
    </div>
  </div>
</def>

```

```

    <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>
```

Then remove the `<aside1:>Aside 1</aside1:>` parameter from `front/index.dryml`.

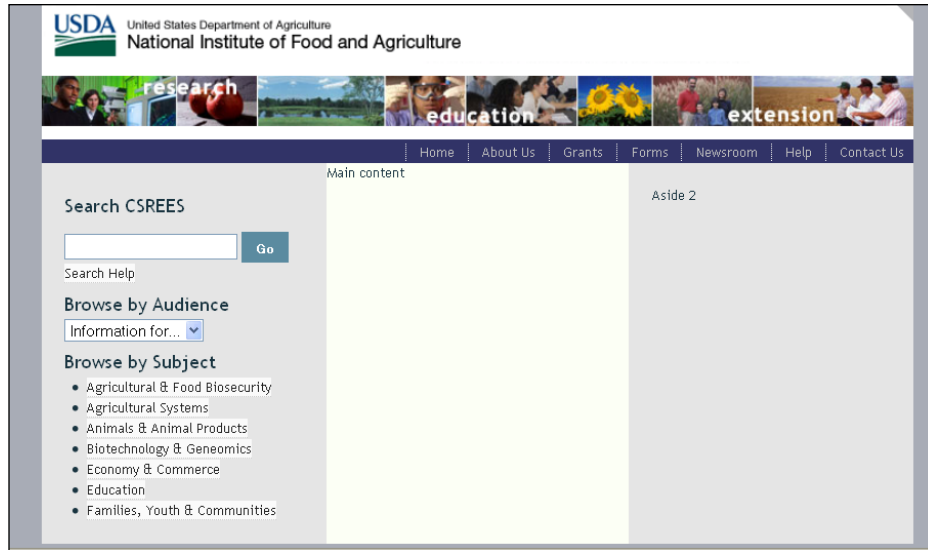


Figure 259: View of the left panel contact without styling

Now 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:

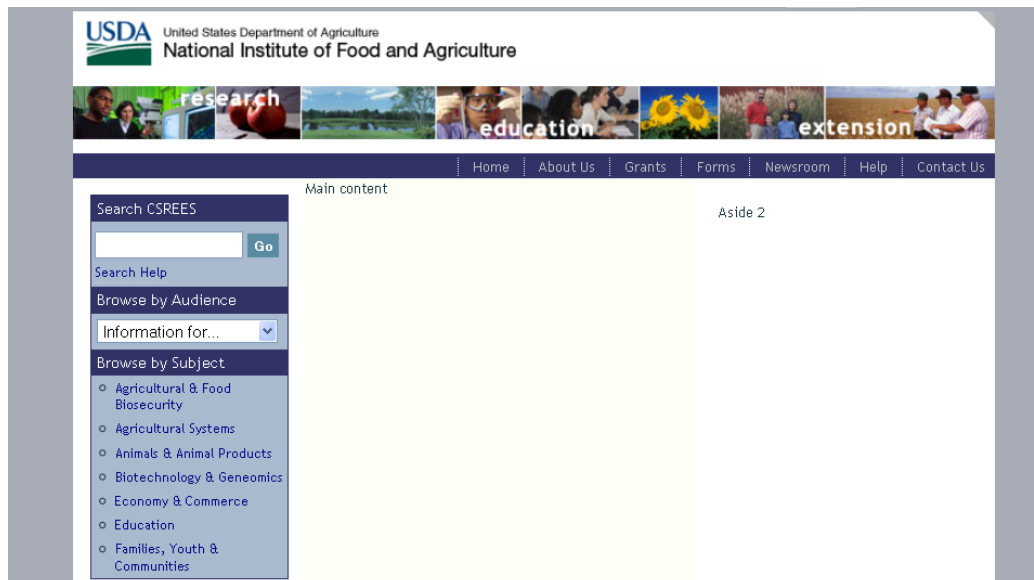


Figure 260: View of the left panel content with correct styling

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, say:

```
<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 here.

Now 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;}
```

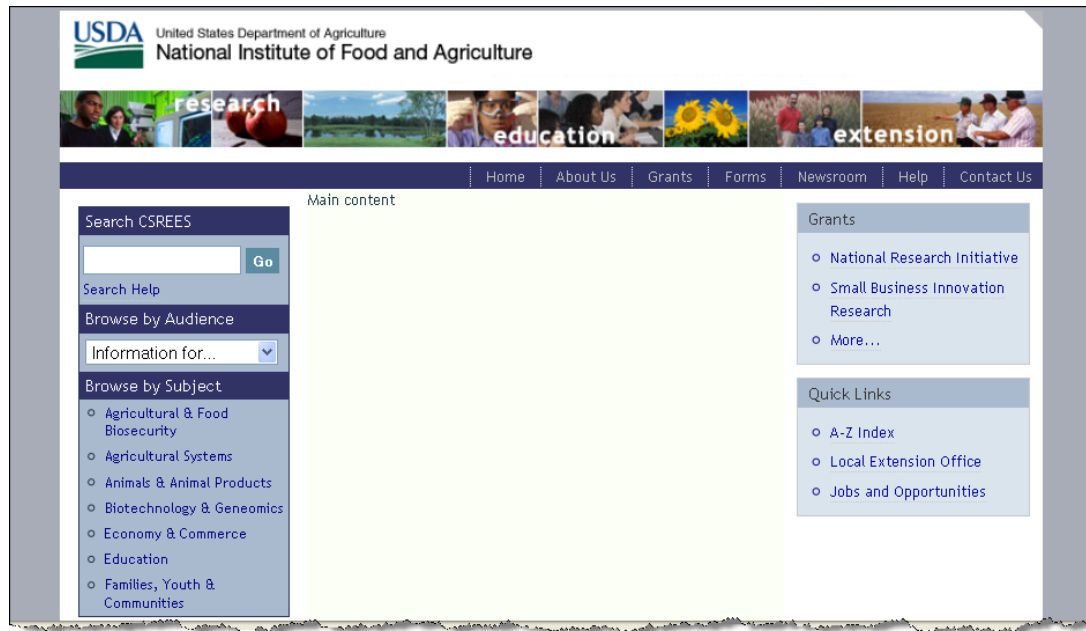


Figure 261: View of the right panel content with styling

Main content

The main content varies a lot from page to page, so let's just make sure that the margins are OK, and leave it at that. First we need some 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:

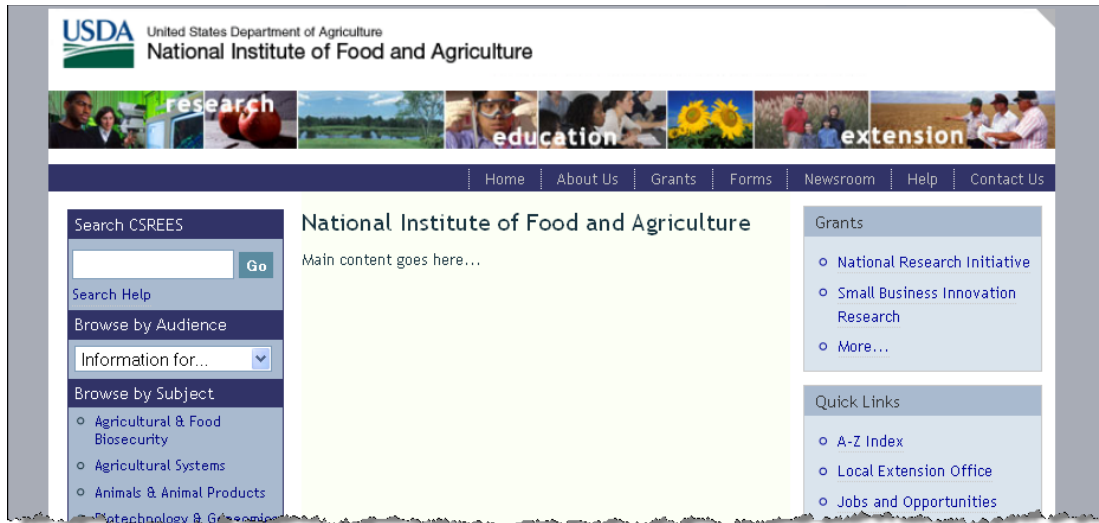


Figure 262: View of the main content panel

The footer

The footer is the same throughout the site. Let's 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>
```

And 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.

And finally, the CSS. To get the corner graphic that we’ve used here, 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 there that is worth a mention. In the `.page-footer` section, we’ve specified:

```
overflow: hidden; height: 100%;
```

This is the famous “self clearing” trick. Because all the content in the footer is floated, without this trick the footer loses its height.



Figure 263: NIFA Demo with final footer styling

That pretty much brings us to the end of the work of reproducing the 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 here and there 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 this code? That is the subject of our next tutorial.

Tutorial 22 - Creating a “Look and Feel” Plugin for Hobo

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

Somehow the idea of “creating a plugin” seems like a big deal, but it’s there’s really nothing to it. Pretty much 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:



```
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\nifa\stylesheets\nifa.css
11 copy public\images\* vendor\plugins\nifa\public\nifa\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\nifa\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, but 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:

```
# nifa.dryml

<append-stylesheets:>
  <stylesheet name="\nifa\stylesheets\nifa.css"/>
</append-stylesheets:>

# 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 that we have our “Four Table” application working the way we want, let’s add an approval process so that new recipes need to be approved 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

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 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. So 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 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:

```
> script/generate hobo_migration
```

Select ‘m’ when prompted to migrate now, and then specify a name for this migration.

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. HOB0 makes this very easy by giving us a predefined dryml tag called `<transition-buttons/>` and 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.

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, just 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.

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| {:conditions =>
  "#{acting_user.signed_up??1:0}=1 or lifecycle_state='published'" }}
```

...which returns 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 highlighted 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))
```

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:

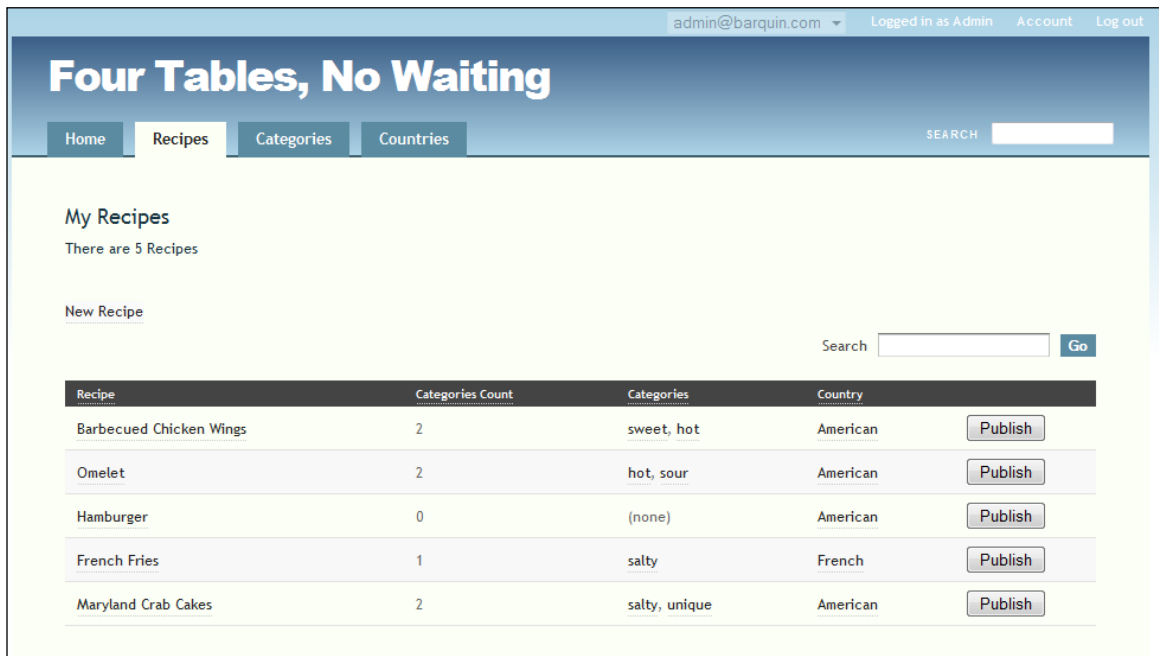


Figure 266: Recipes ready to Publish.

To publish a Recipe just click on the ‘Publish’ button. For this example, I’ll publish the Omelet recipe. After clicking on the button, I’ll get the show page for the Omelet.

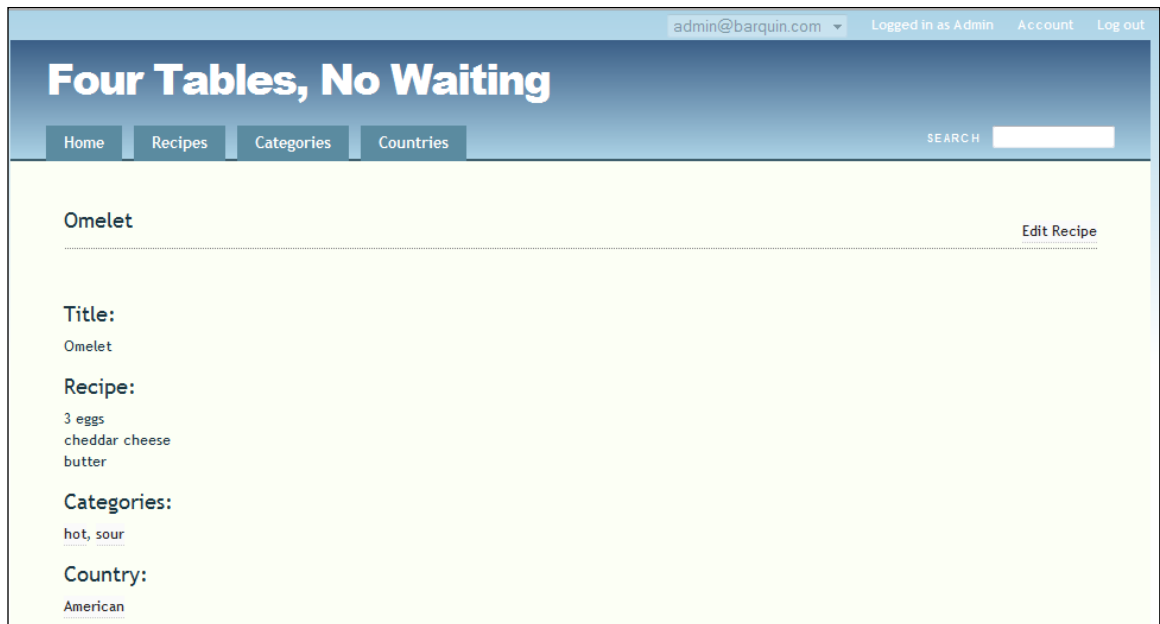


Figure 267: Omelet recipe after being placed in the "Published" state

And if I go back to my Recipe listing page I see:

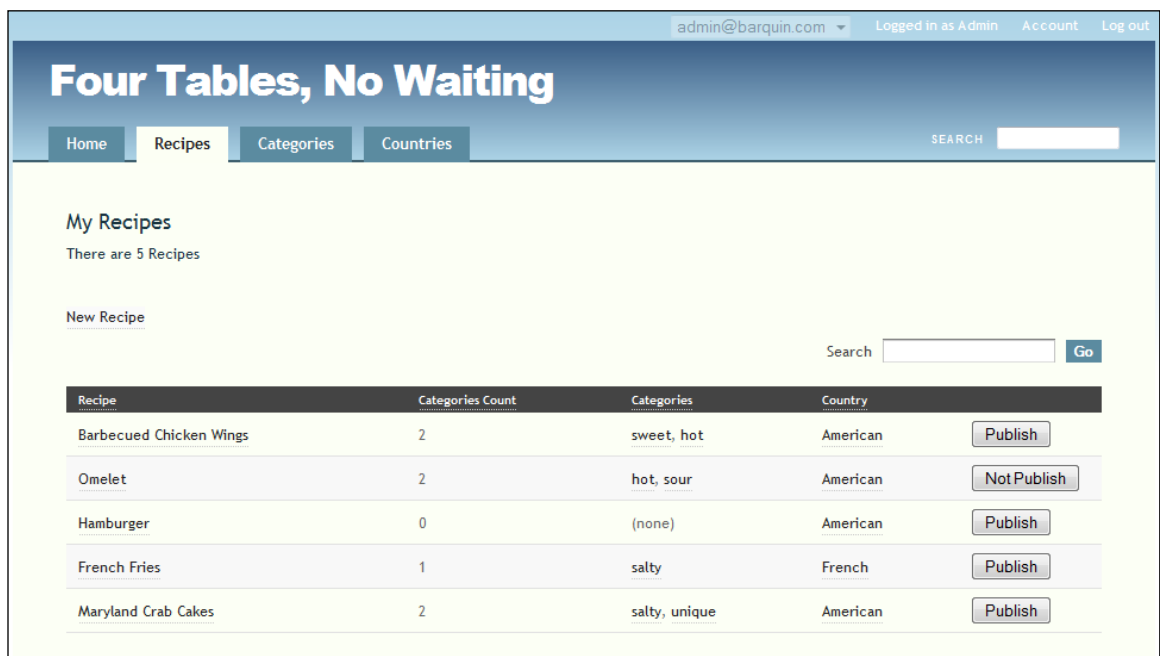


Figure 268: Recipe index with buttons for "Publish" and "Not Publish"

Since my Omelet recipe has been published, the only available action for it is to ‘Not Publish’ it.

If I go to the Recipe listing page as a Guest user, I should now see my Omelet recipe:

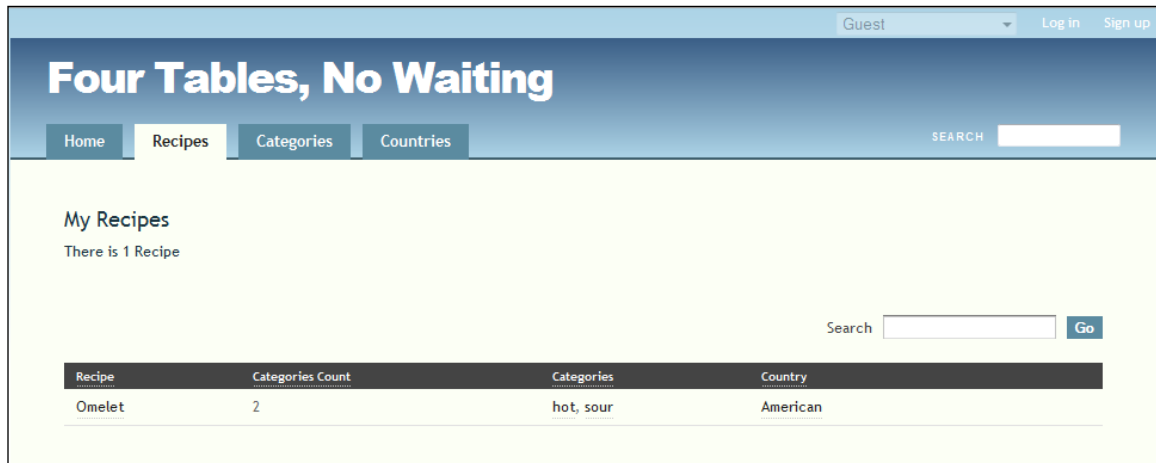


Figure 269: Guest user can only see the published Recipe

- Improve the navigation.** So at this point we are able to Publish and Not Publish our recipes, so our workflow is behaving as we expect. But the navigation can be improved and would be cleaner if after we clicked 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 transition we define, hobo creates 2 controller actions, 1 for a GET request and 1 for a PUT request. So, 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 we wanted to show a form before executing the transition action, i.e. if we wanted to collect comments from the user before he/she Publishes or Not Publishes, we could show a form with a comments box and a Publish/Not Publish submit button. But in this example, we just want to configure the application so that after a Recipe is Published or Not Published, the browser should redirect back to the Recipe listing page. To do this we'll add the following 2 actions to our 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 we can specify the page redirect after the transition has been executed. Once you have added these actions, if you access the Recipe index page and click on a Publish or Not Publish button, you'll just see the page get refreshed.

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, and 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

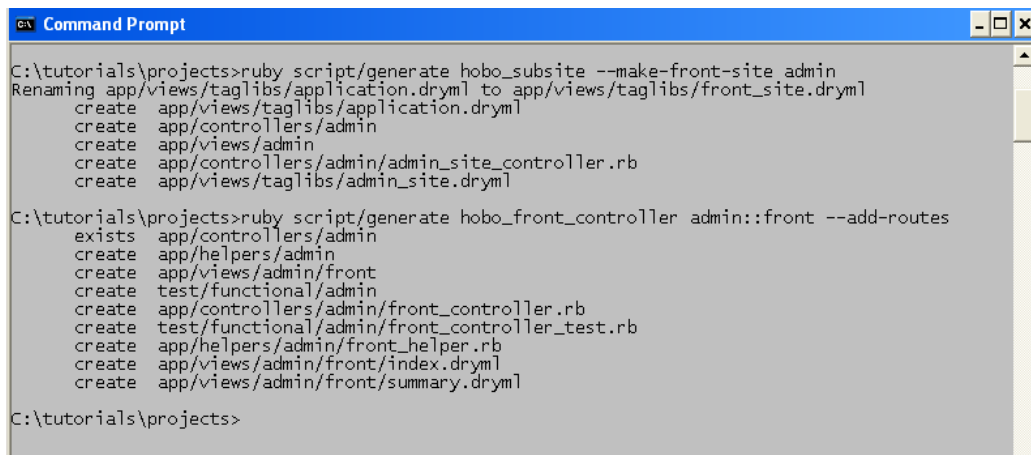
By Bryan Larsen

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.

Generator steps

Let's add an admin sub-site to the project we created in the “Agile Project Manager” tutorial.

```
\projects> ruby script/generate hobo_subsite --make-front-site admin  
\projects> ruby script/generate hobo_front_controller admin::front --add-routes
```



```
Command Prompt  
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`).

Let's first change all of the permissions for this model to “true”, as only an administrator will be able to access this sub-site:

```

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
11   # --- Permissions --- #
12
13   def create_permitted?
14     true
15     # acting_user.administrator?
16   end
17
18   def update_permitted?
19     true
20     # acting_user.administrator?
21   end
22
23   def destroy_permitted?
24     true
25     # acting_user.administrator?
26   end
27
28   def view_permitted?(field)
29     true
30   end
31
32 end

```

Controller Modifications

We need to move the controller for RequirementStatus to the admin folder and modify it to be:

```

Class Admin::RequirementStatusesController < Admin::AdminSiteController
  hobo_model_controller RequirementStatus
  auto_actions :all
end

```

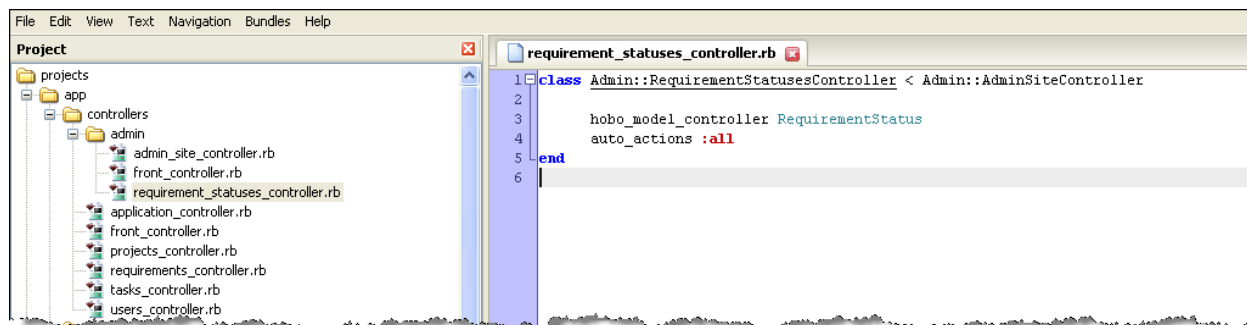


Figure 271: View of the Admin folder contents

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:

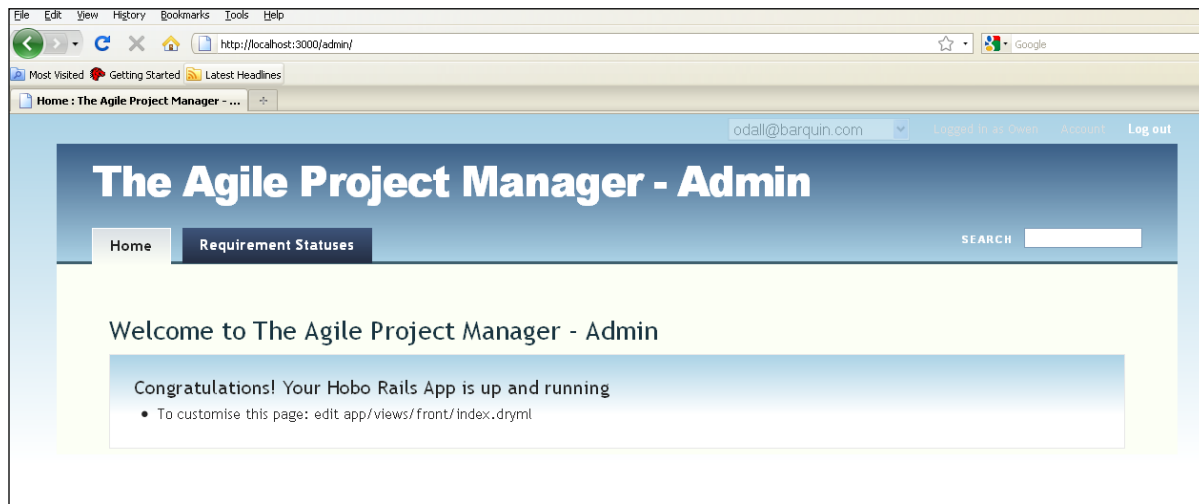


Figure 272: View of the Admin Sub-Site

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, and provides shorthand for 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.

A Note For Oracle Users. 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 yet, I encourage you to do so. This is the wave of the future.

Tutorial 26 – 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:

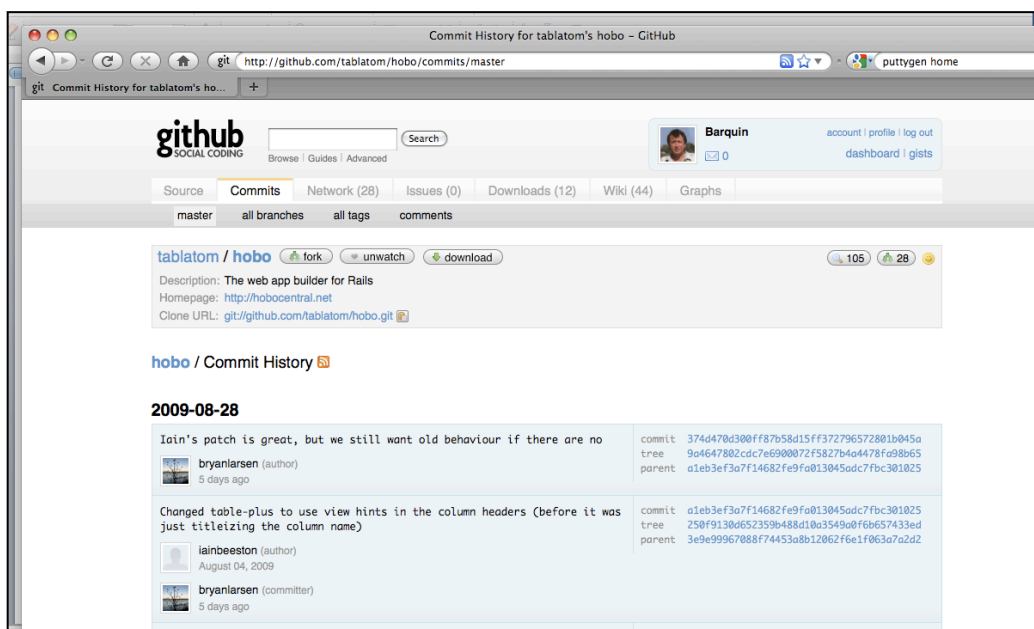


Figure 273: Hobo source code on github.com

It is also where the Hobo gems are stored:

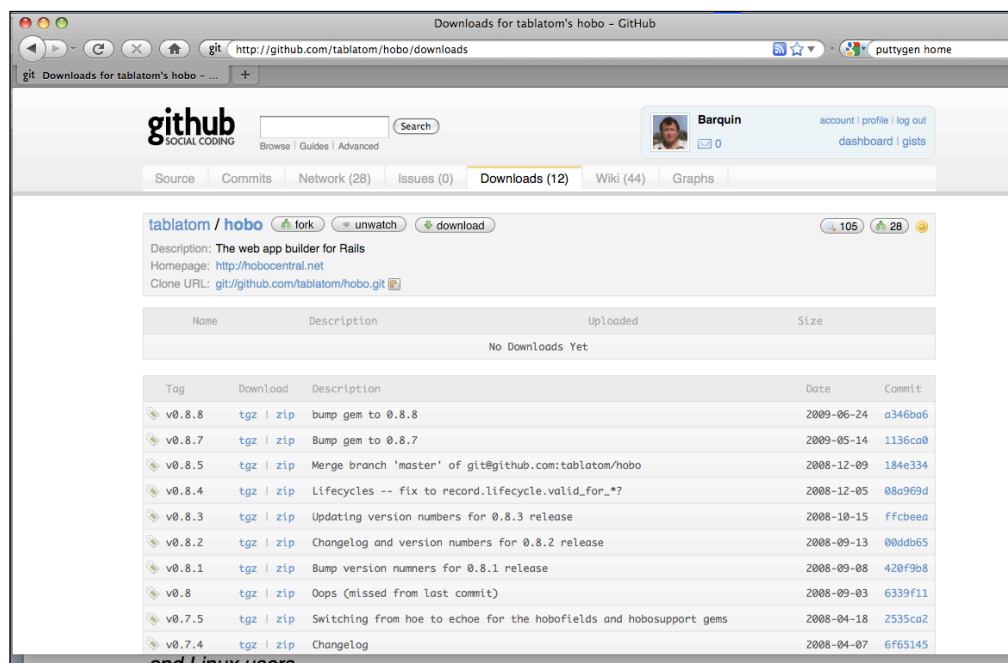


Figure 274: Hobo gems are also available on github.com

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>

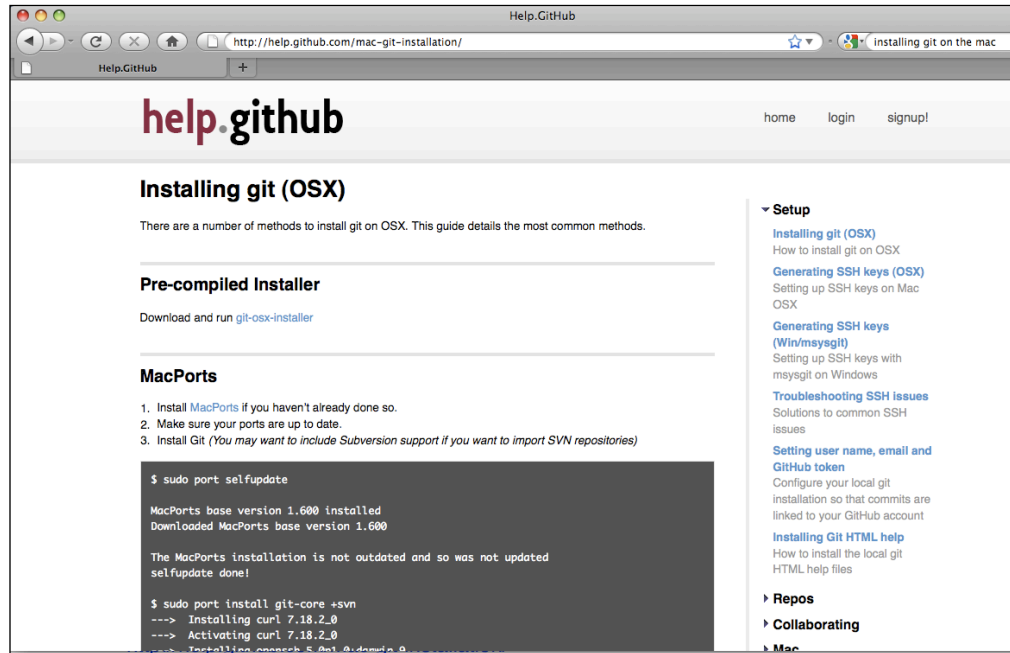


Figure 275: Installing Git for Mac OSX

OK. So let's get the software we need for Git:

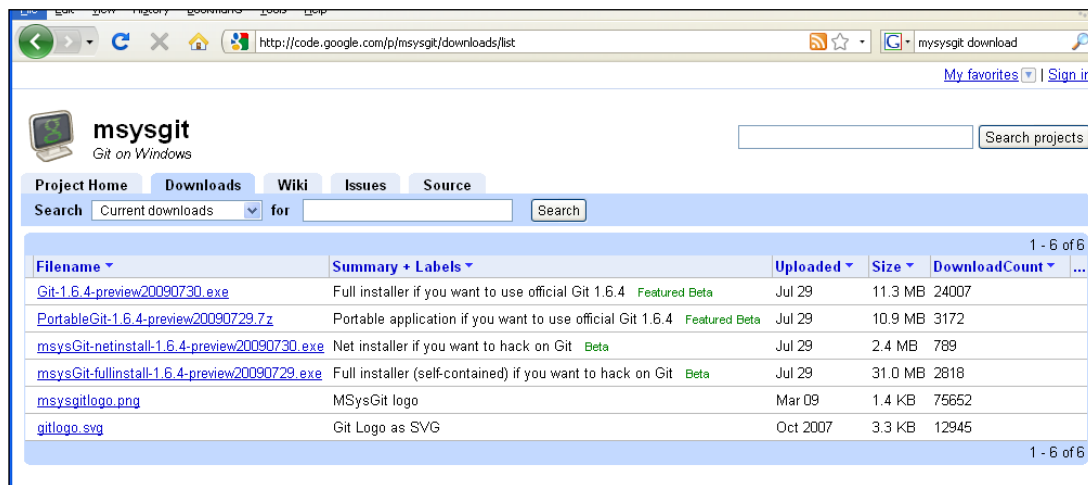


Figure 276: Download the msysgit installer for Windows

Download and run the git installer for windows:

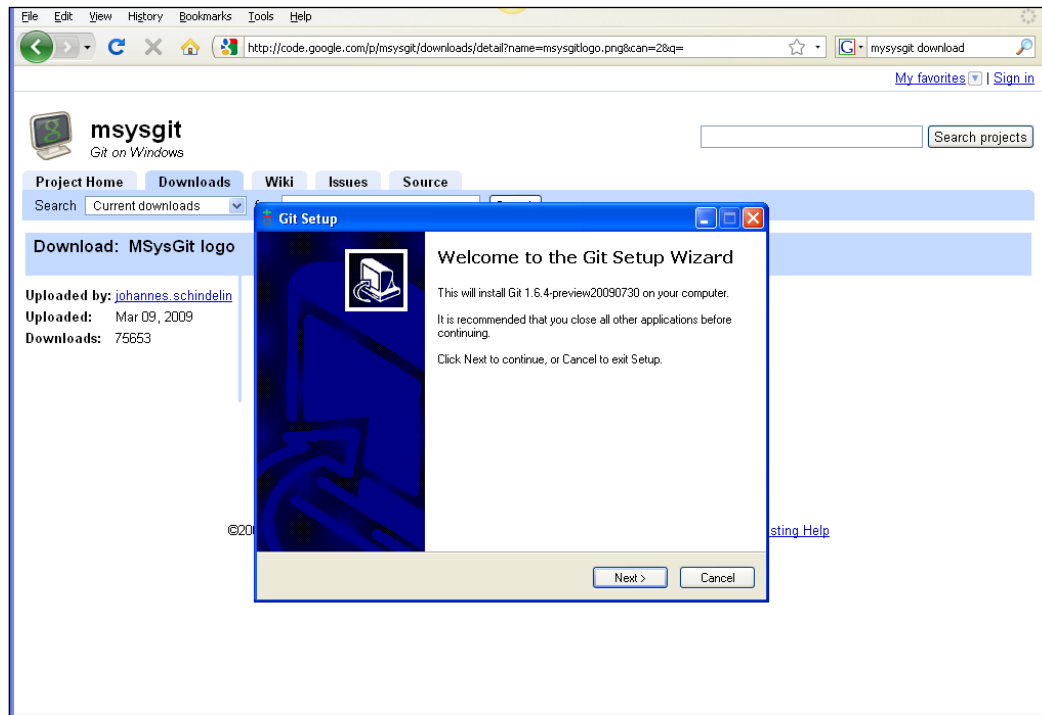


Figure 277: Running the Git Setup Wizard

Select the following options:

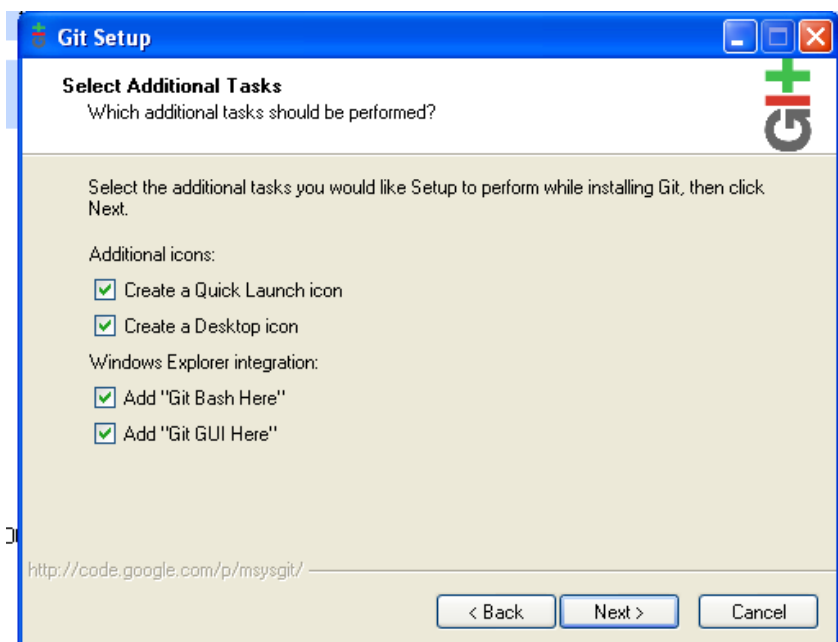


Figure 278: Git setup options

Select the “Use OpenSSH” option:

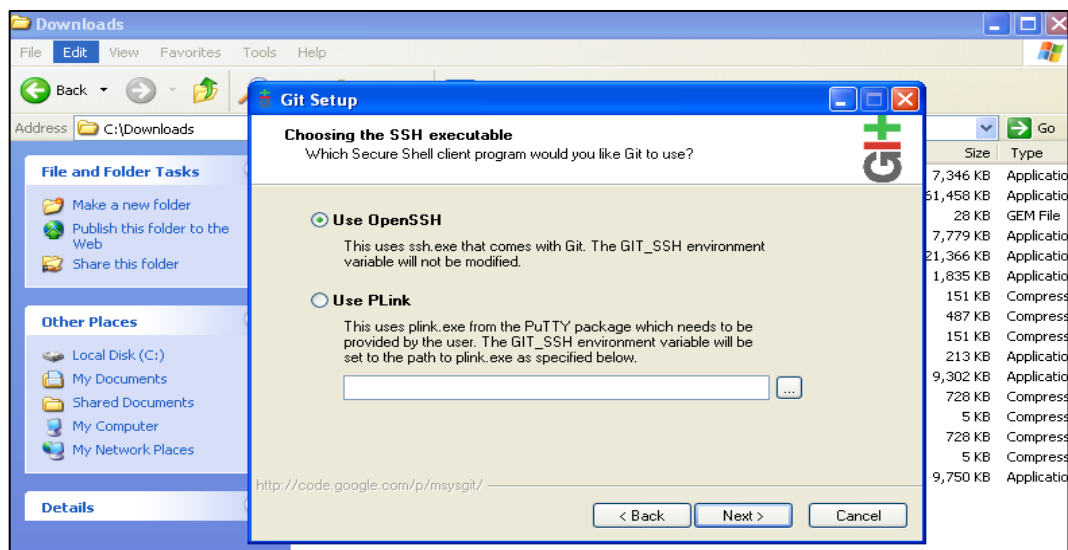


Figure 279: Select the OpenSSH option

Allow the installer to configure running git from the Windows command prompt:

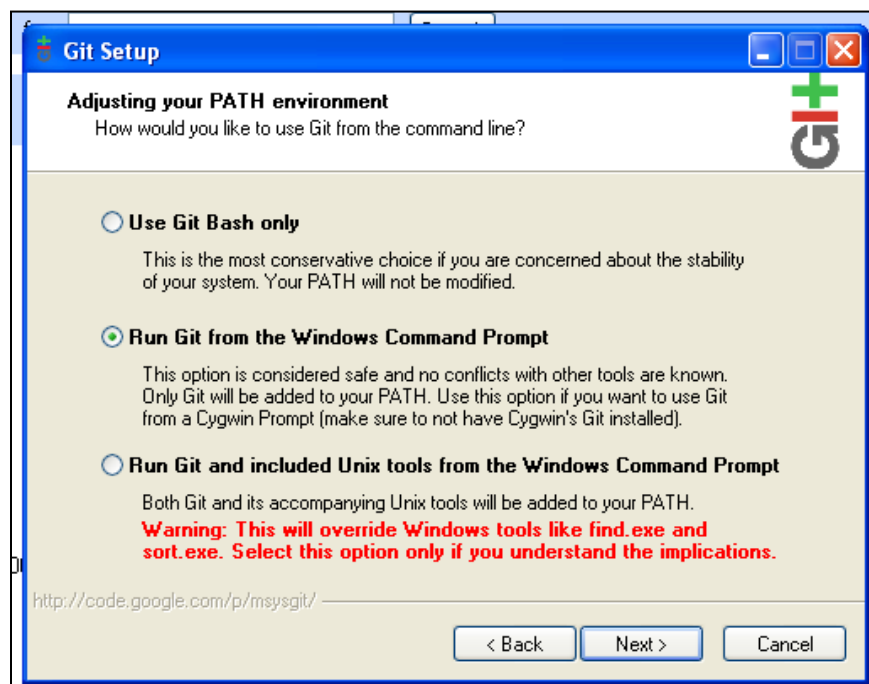


Figure 280: Select to option to run Git from the Windows command prompt

Next select the CR/LF behavior option:

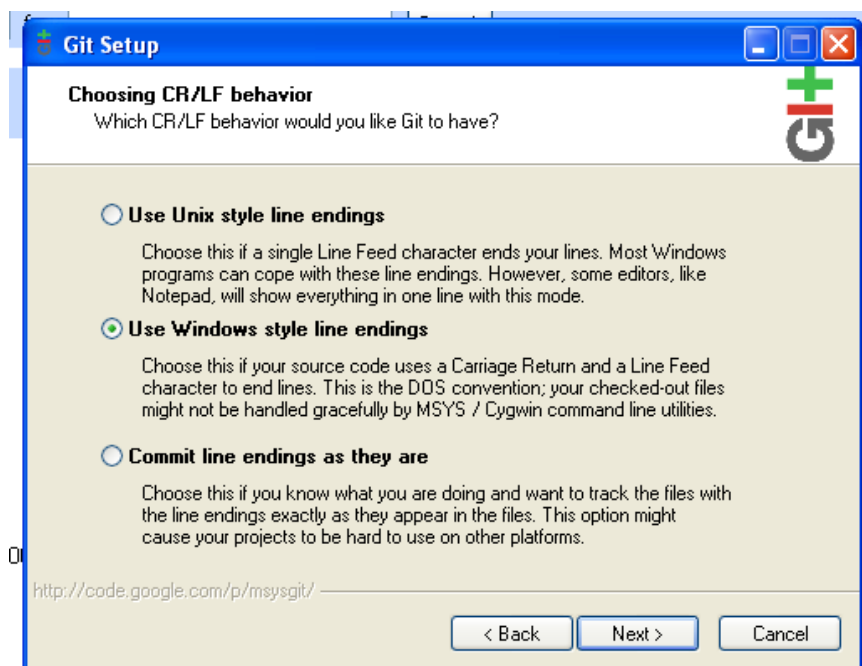


Figure 281: Select Windows style line endings

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:

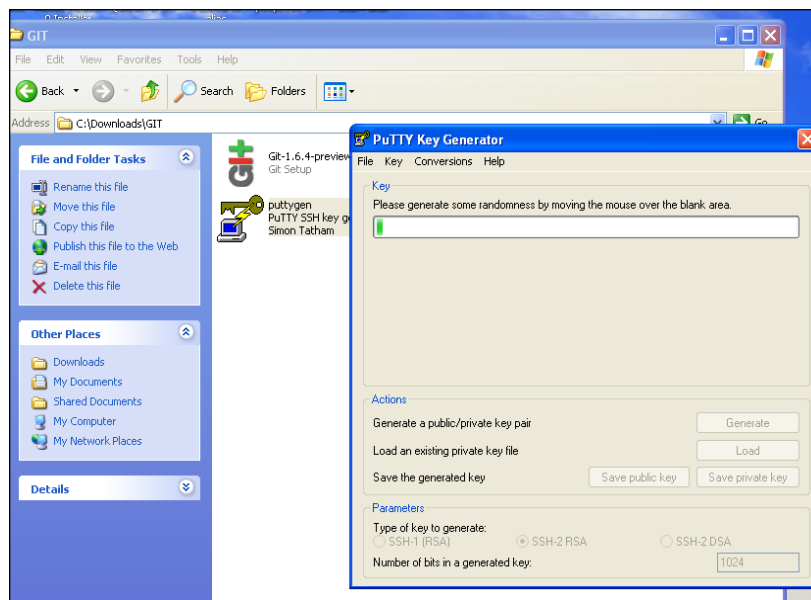


Figure 282: Running the PuTTY Key Generator install

Open up the application and start the process of generating key pairs:

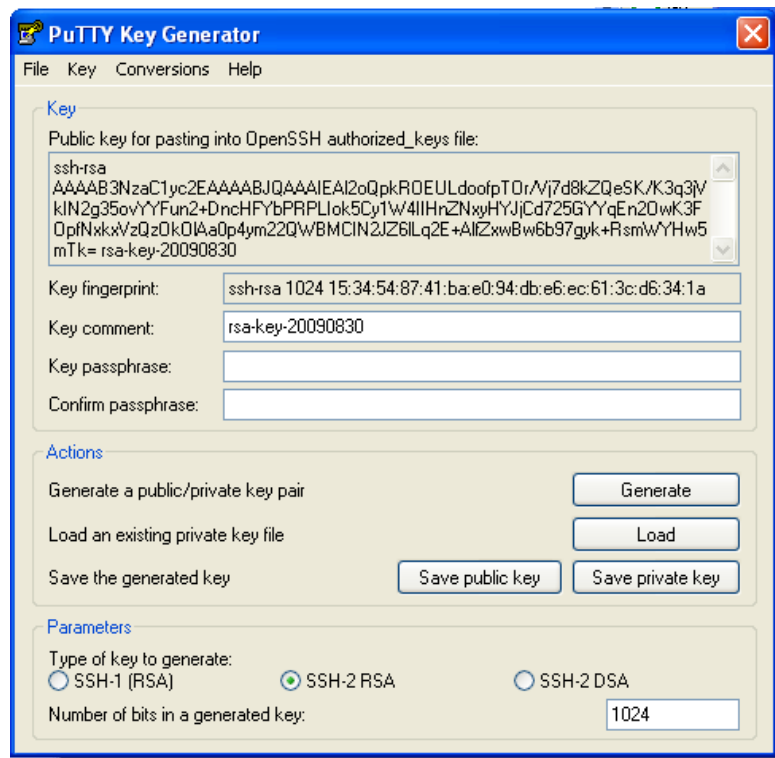


Figure 283: Generate SSH key pairs for use with Git

Saving the files with default names:

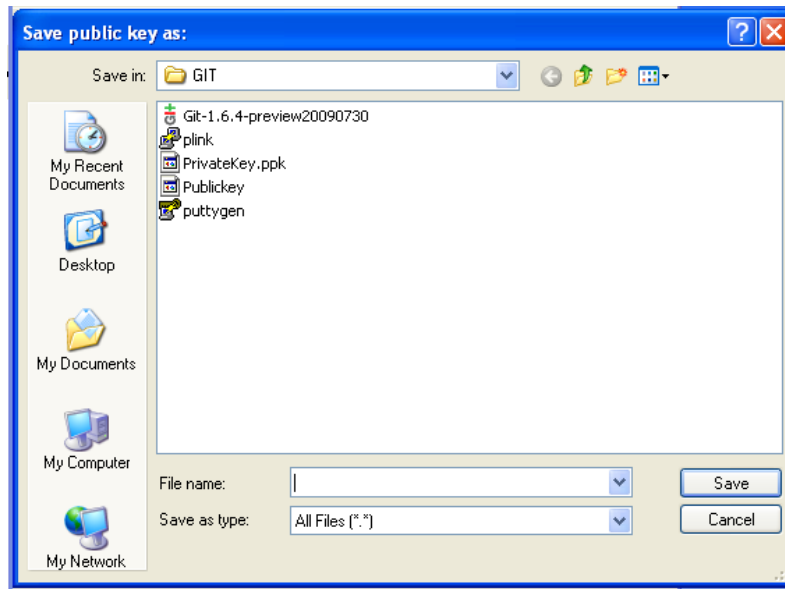


Figure 284: The default file names generated by PuTTYGen

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.

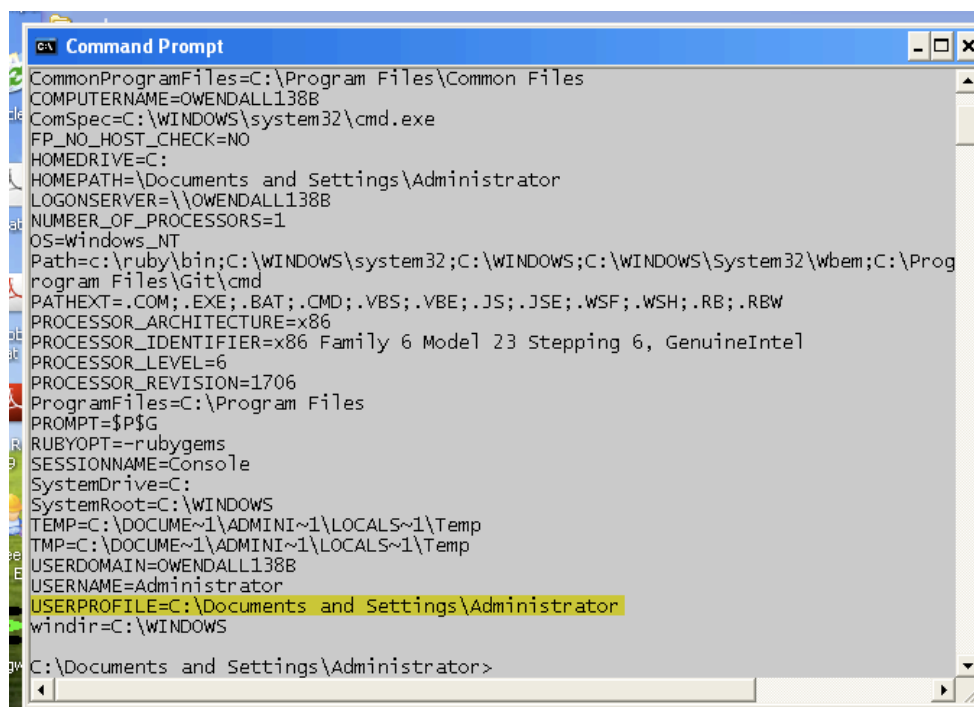


Figure 285: Locating your USERPROFILE setting

I was logged in as the user “Administrator” in windows when I tried to use the Heroku gem (see next chapter):

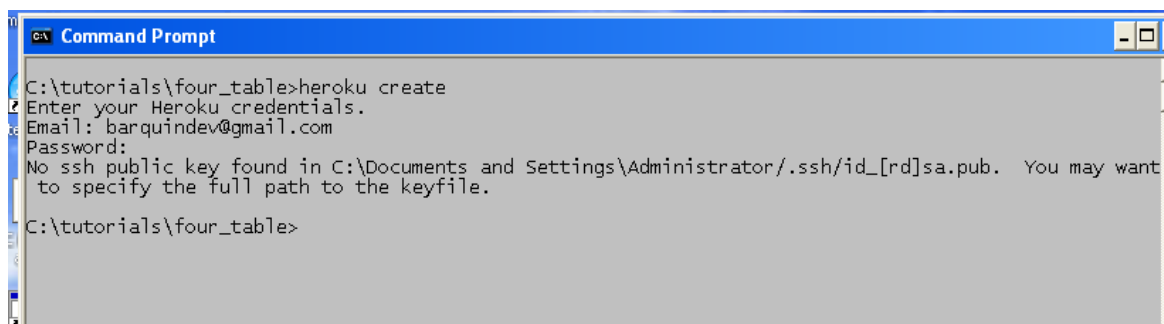


Figure 286: View of "no ssh public key found" error

So 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
```

So we can move the keys as follows:

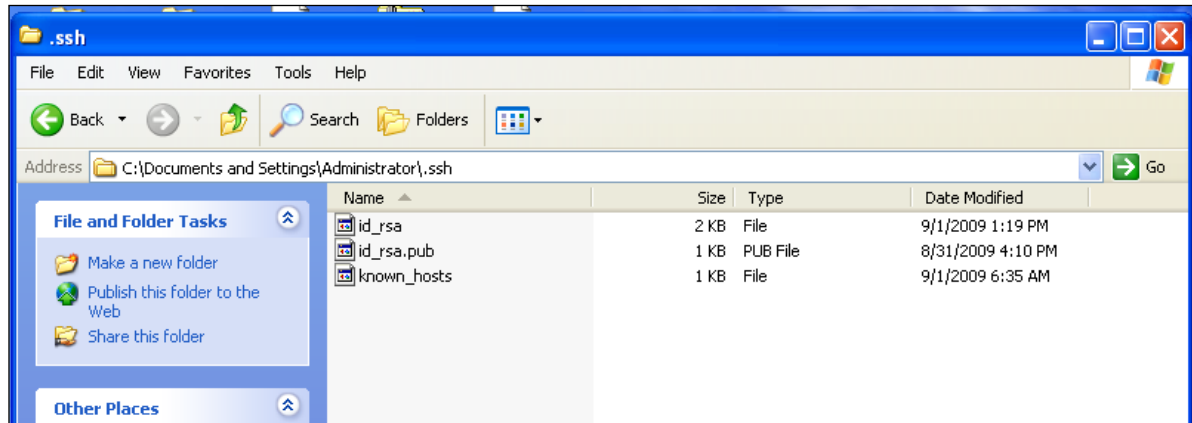


Figure 287: Naming your SSH key pairs

(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 chapter to learn how Git is used to deploy your application to Heroku.com.

Tutorial 27 – 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:

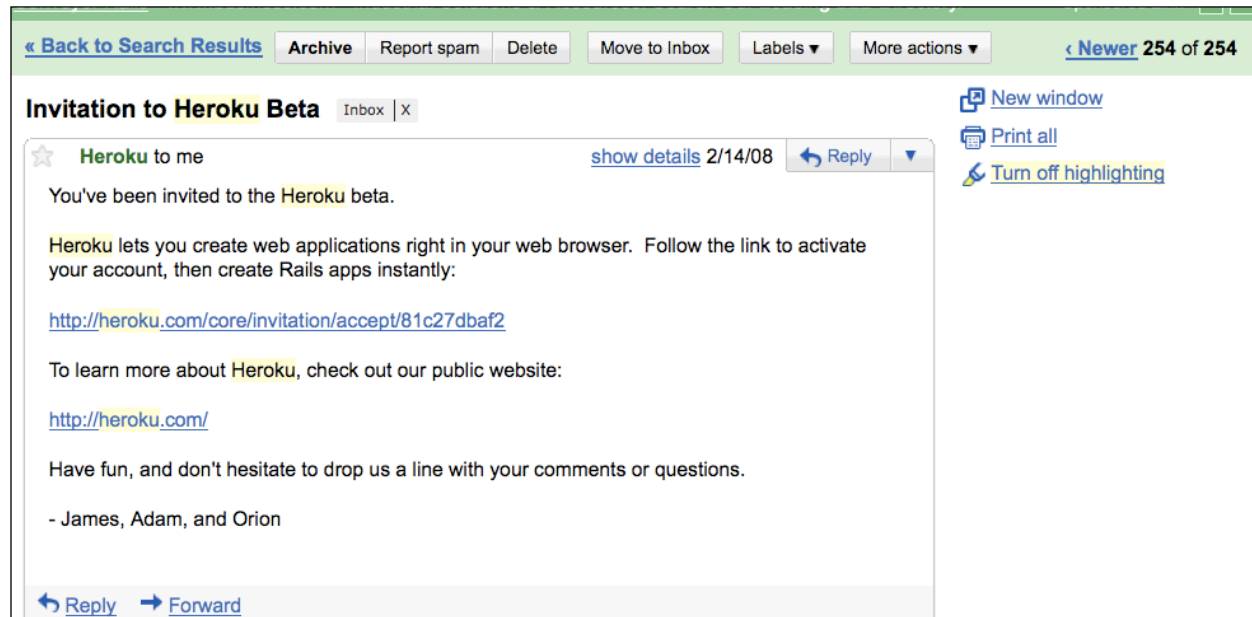


Figure 288: The original Heroku beta invitation

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>

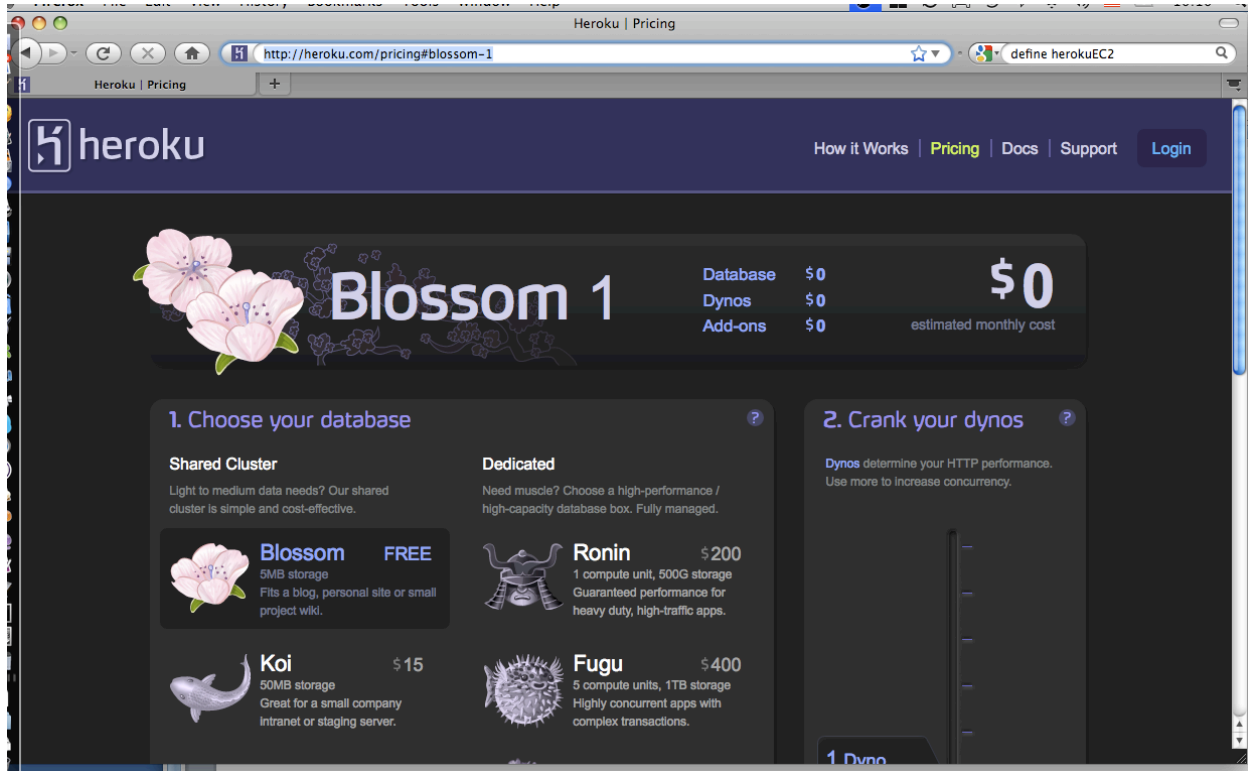


Figure 289: Using the free "Blossom" database hosting option on Heroku.com

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 transparently.

For this tutorial we will use the “`four_table`” application will built in 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 Chapter 23 – Installing and Using git.

Step 2: Create an Account at Heroku.com

Go to <http://heroku.com/signup>:

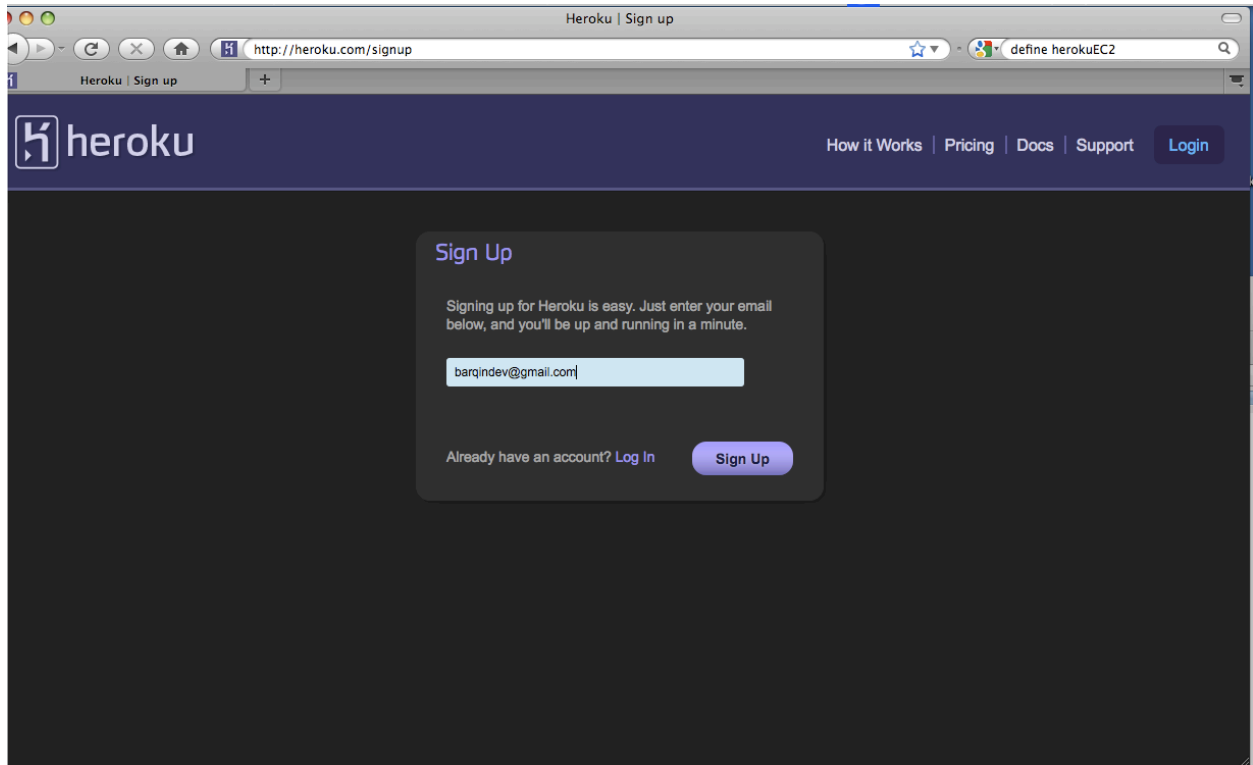


Figure 290: Sign Up for a Heroku account

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.

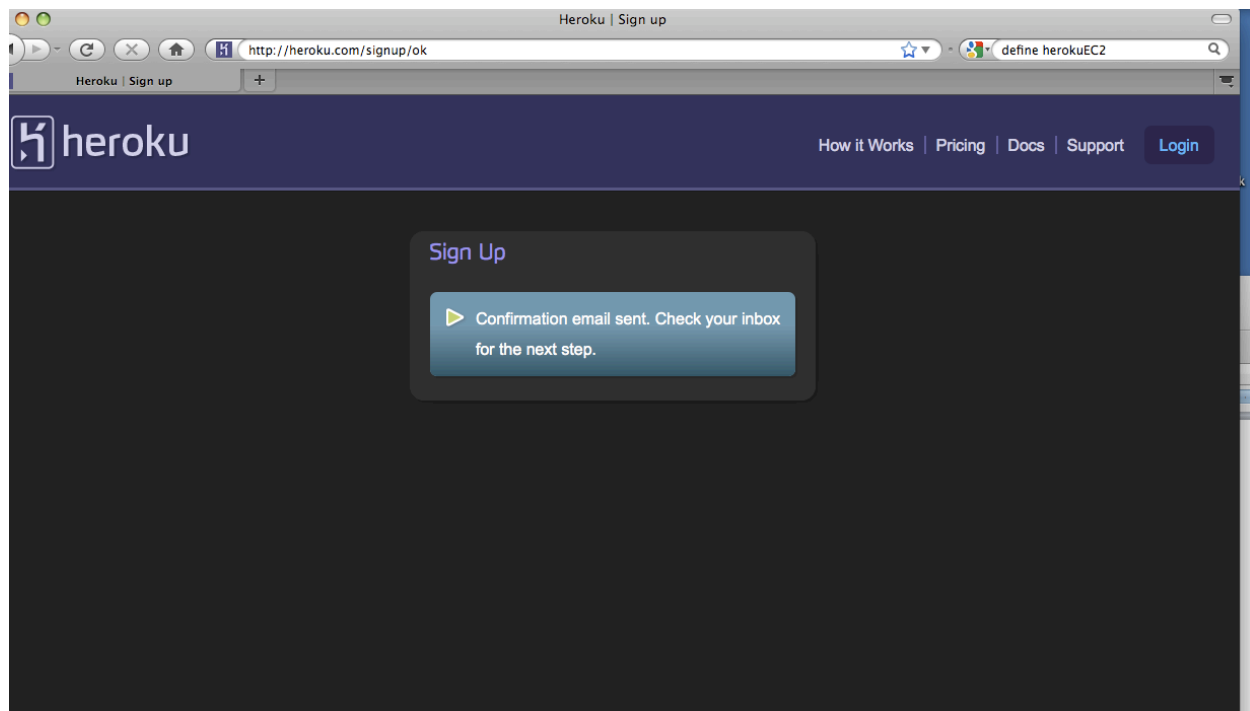


Figure 291: Heroku notification that "Confirmation email sent"

Going to you email to access the confirmation link you will need:

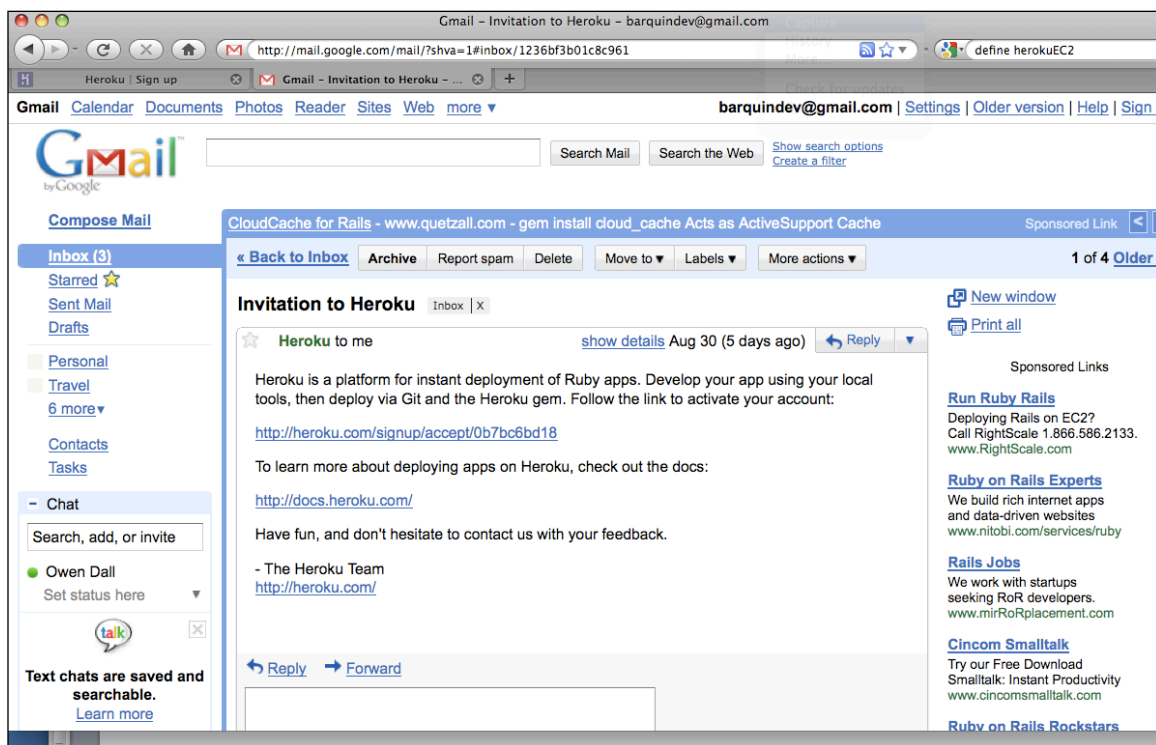


Figure 292: Locating your "Invitation to Heroku" email

When you click the confirmation link, you should see a screen similar to the following:

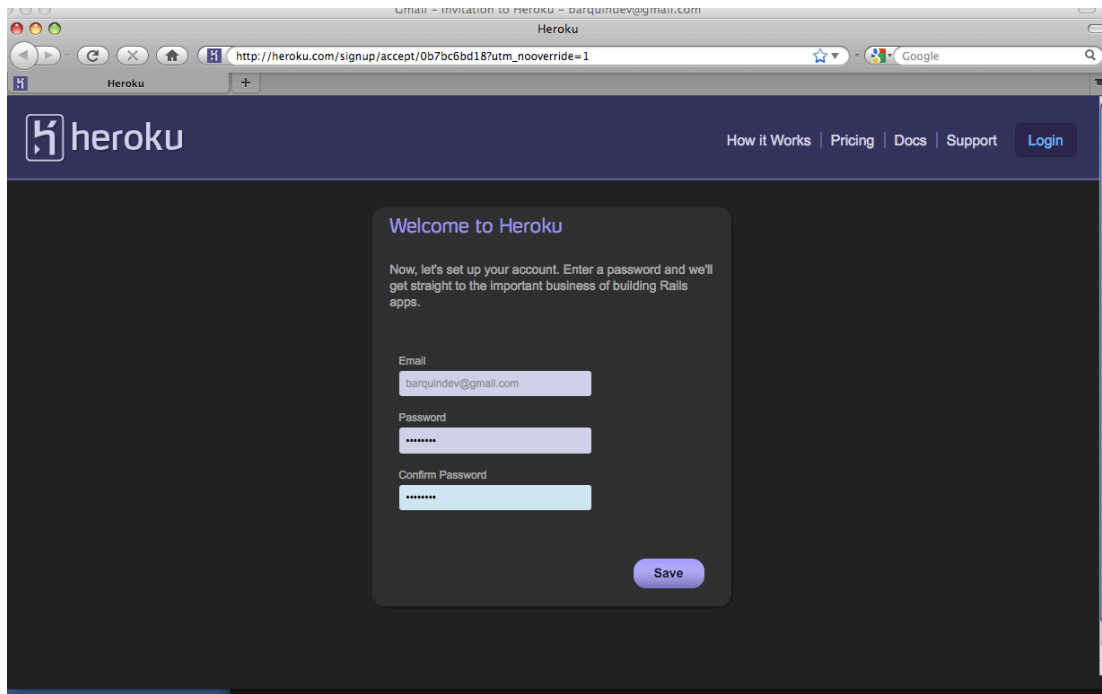


Figure 293: The "Welcome to Heroku" signup page

And then this when you finish:

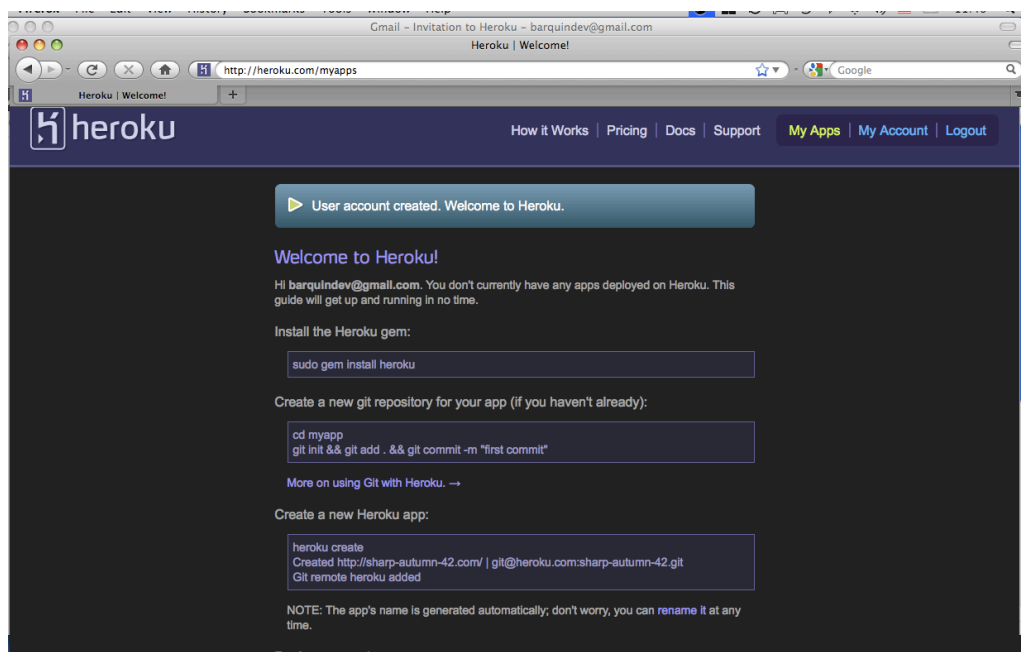


Figure 294: The "Account Created" message at Heroku.com

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 you command prompt and type the following command:

```
C:\ruby>gem install heroku
```

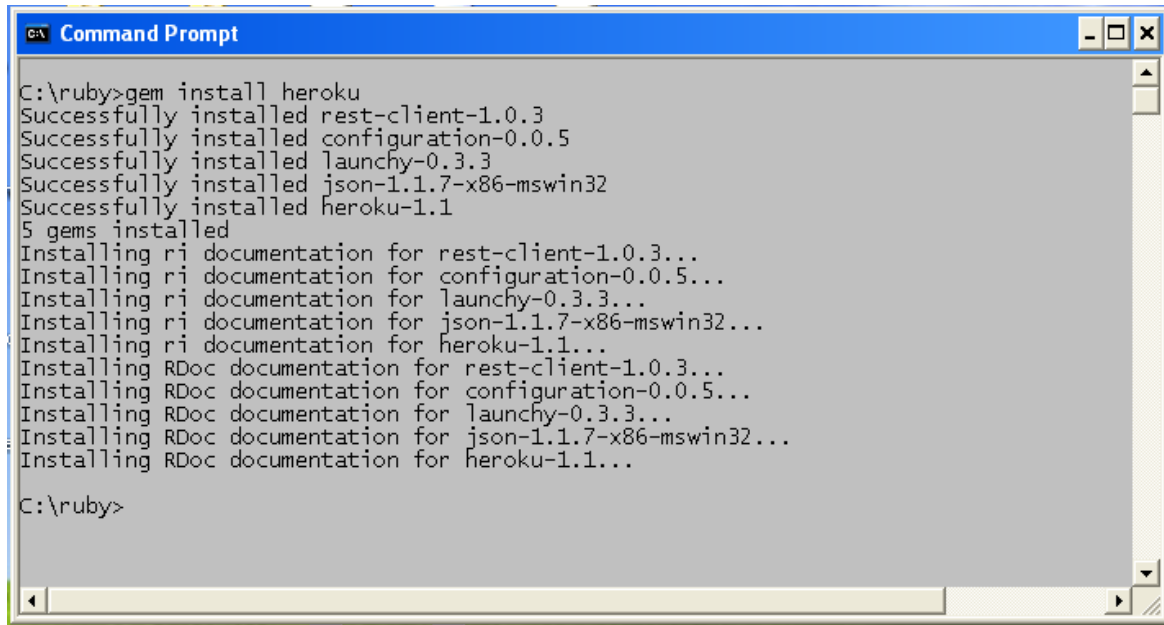


Figure 295: Installing the Heroku Ruby gem

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:

```
C:\tutorials\four_table> git init
```

Tell git to add all the files in all folders to the project:

```
C:\tutorials\four_table> git add .
```

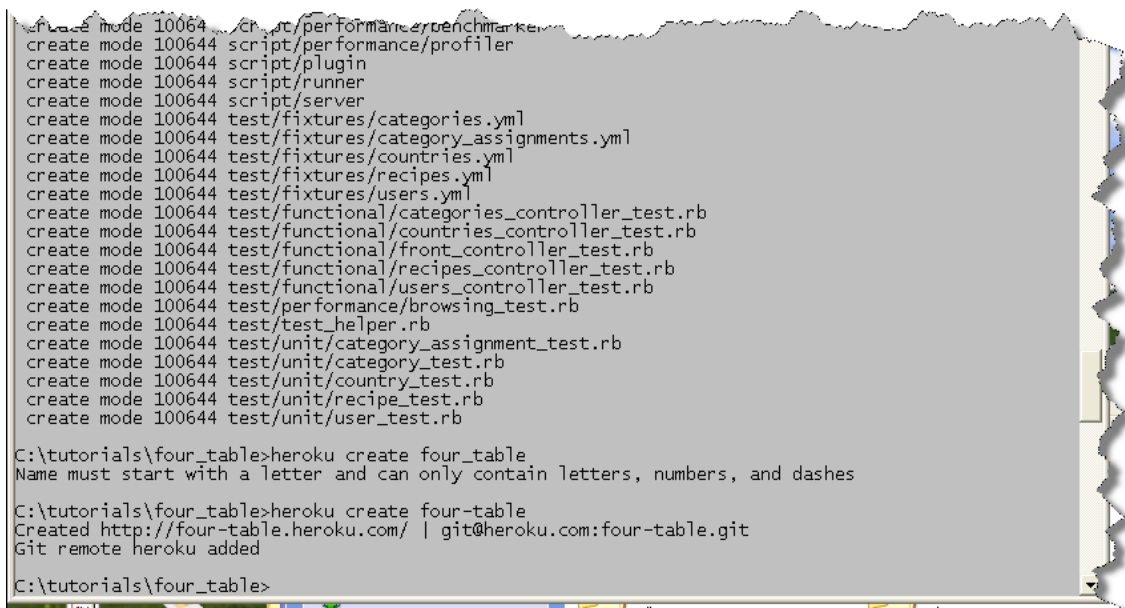
Tell git to commit these additions and enter an optional message that helps for version control:

```
C:\tutorials\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.

```
C:\tutorials\four_table> heroku create four_table
```



```
create mode 100644 script/performance/benchmarks
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/recipes.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 296: Console output from the "heroku create" command

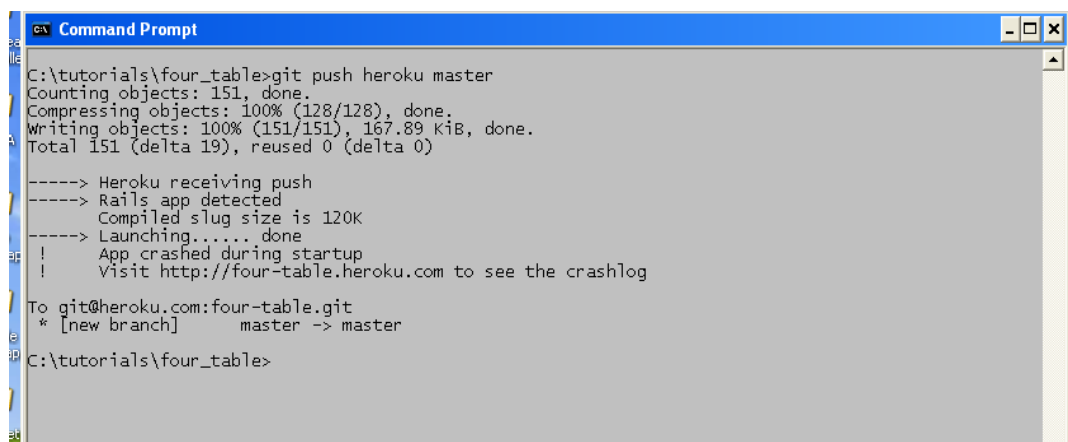
Note: The first time you try to create using the heroku gem you will be prompted to enter your user name and password that you provided heroku while creating an account:

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:

```
C:\tutorials\four_table> heroku create four-table
```

And then:

```
C:\tutorials\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..... done
! App crashed during startup
! Visit http://four-table.heroku.com to see the crashlog

To git@heroku.com:four-table.git
* [new branch] master -> master
C:\tutorials\four_table>
```

Figure 297: 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:

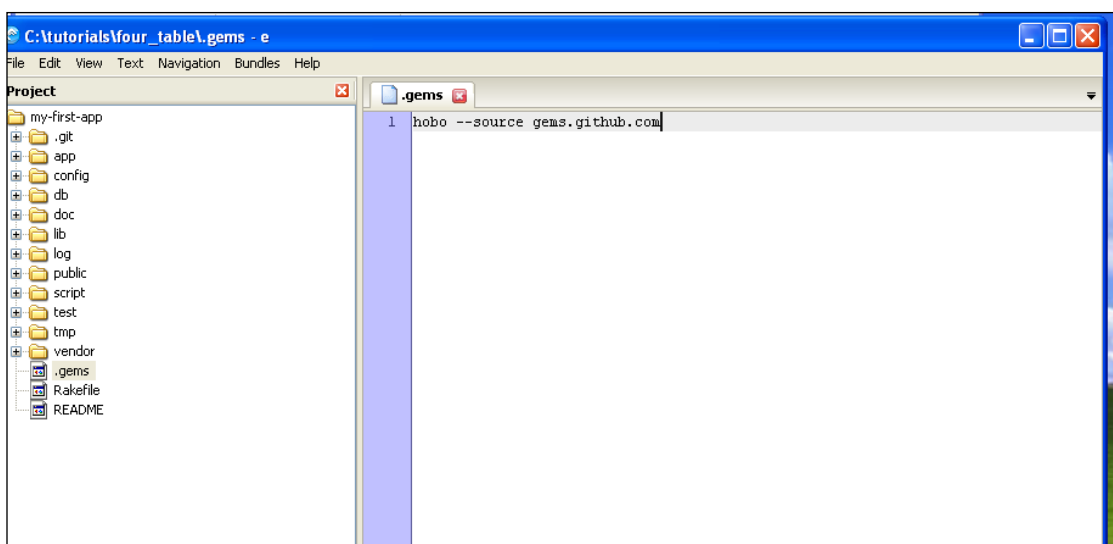


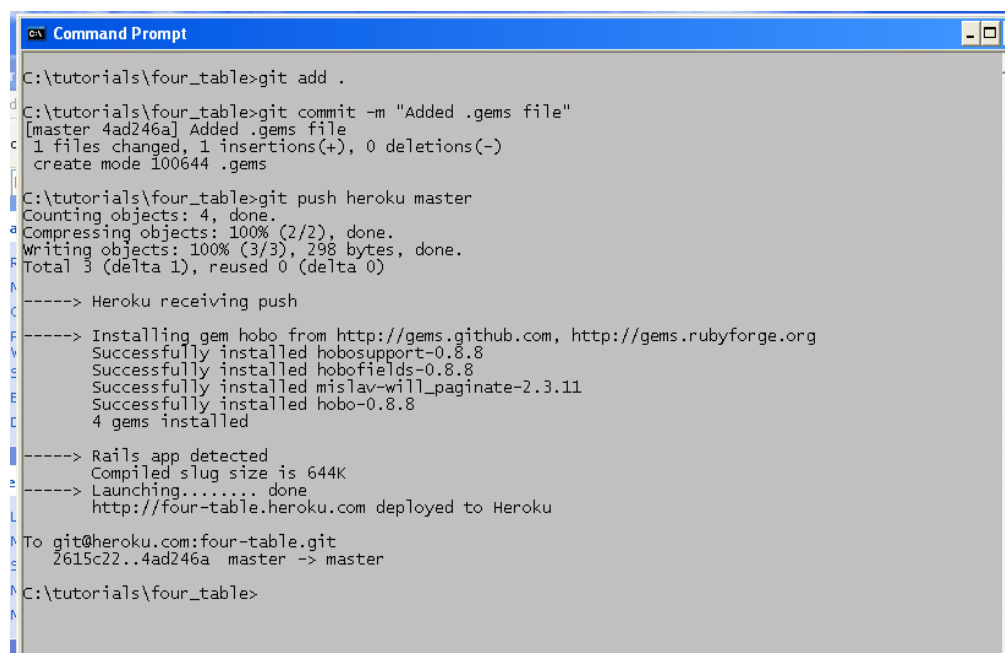
Figure 298: Telling Heroku where to find your application's gems

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:

```
C:\tutorials\four_table> git add .
C:\tutorials\four_table> git commit -m "Added .gems definition file"
C:\tutorials\four_table> git push heroku master
```



```
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>
```

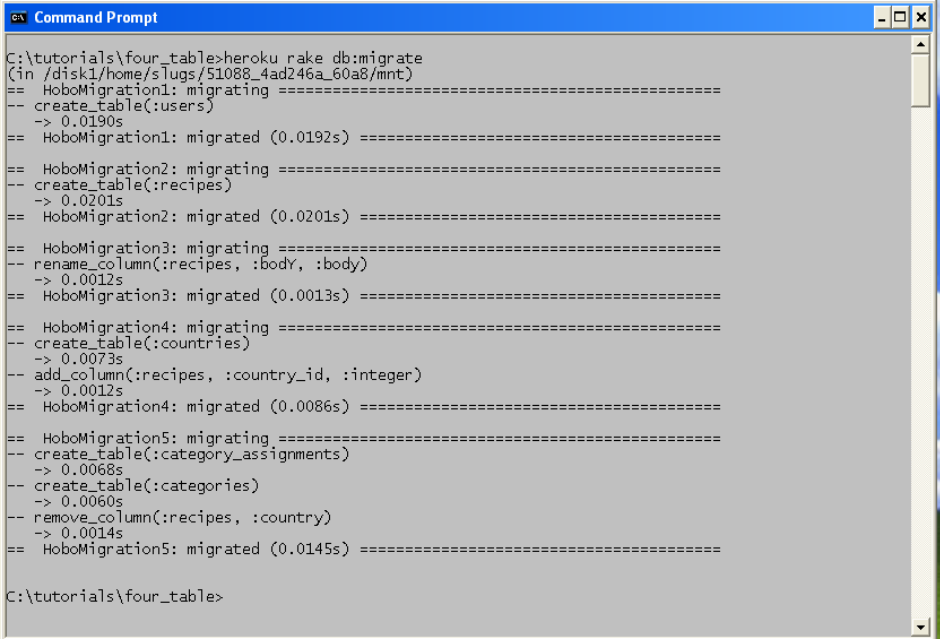
Figure 299: Adding your “.gems” config file to your git repository

Note that 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:

```
C:\tutorials\four_table> heroku rake db:migrate
```



```
C:\tutorials\four_table>heroku rake db:migrate
(in /disk1/home/slugs/51088_4ad246a_60a8/mnt)
== HoboMigration1: migrating =====
-- create_table(:users)
--> 0.0190s
== HoboMigration1: migrated (0.0192s) =====
== HoboMigration2: migrating =====
-- create_table(:recipes)
--> 0.0201s
== HoboMigration2: migrated (0.0201s) =====
== HoboMigration3: migrating =====
-- rename_column(:recipes, :body, :body)
--> 0.0012s
== HoboMigration3: migrated (0.0013s) =====
== HoboMigration4: migrating =====
-- create_table(:countries)
--> 0.0073s
-- add_column(:recipes, :country_id, :integer)
--> 0.0012s
== HoboMigration4: migrated (0.0086s) =====
== HoboMigration5: migrating =====
-- create_table(:category_assignments)
--> 0.0068s
-- create_table(:categories)
--> 0.0060s
-- remove_column(:recipes, :country)
--> 0.0014s
== HoboMigration5: migrated (0.0145s) =====

C:\tutorials\four_table>
```

Figure 300: Migrating your database schema to Heroku.com

Step 7: Test your application

Log into Heroku.com to see the application URL:

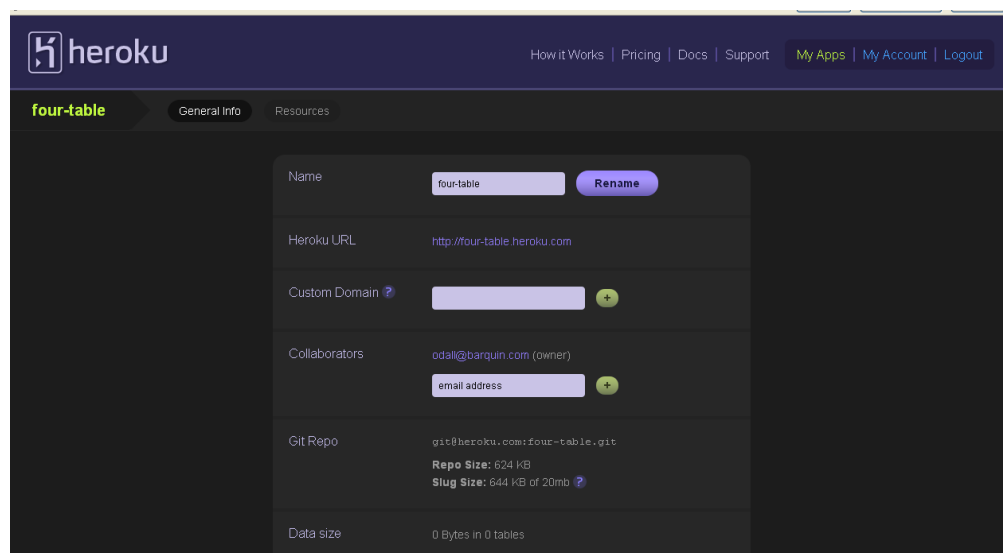


Figure 301: Testing your Heroku app

<http://four-table.herokuapp.com>

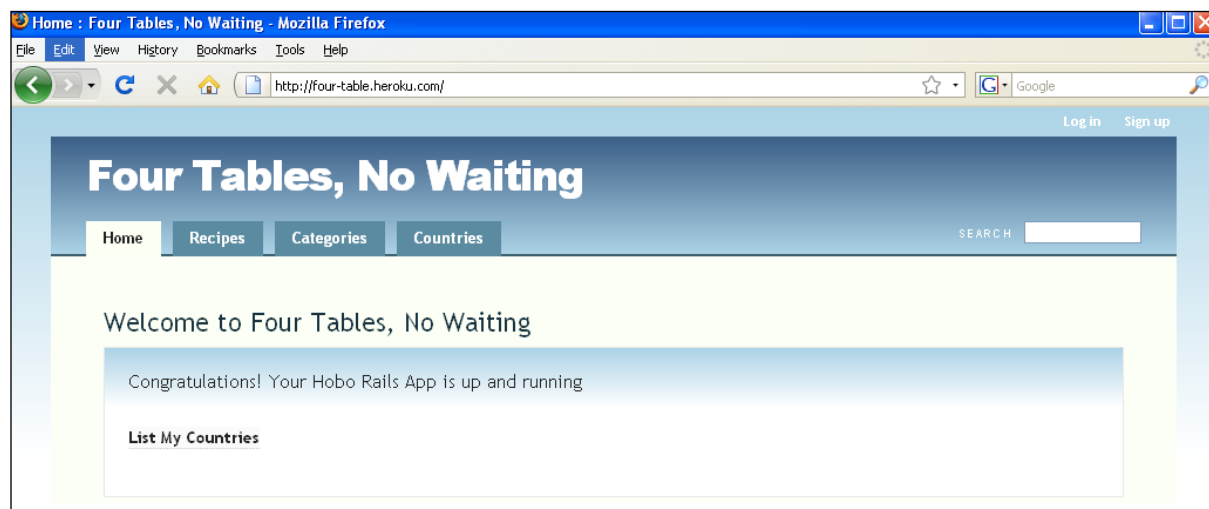


Figure 302: Running the "Four Table" app on Heroku.com

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:

```
C:\tutorials\four_table> gem install taps
```

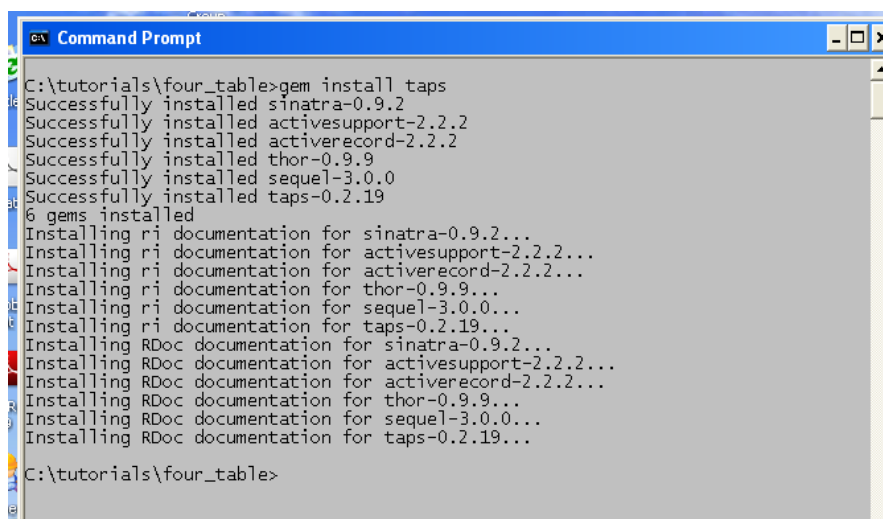


Figure 303: Installing the Taps gem to upload data to Heroku.com

Note that several other dependencies are also installed along with Taps.

Now you can use the following single command to upload your existing (local) data to your version on Heroku:

```
c:\tutorials\four_table> heroku db:push
```

```
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 304: 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 to see:

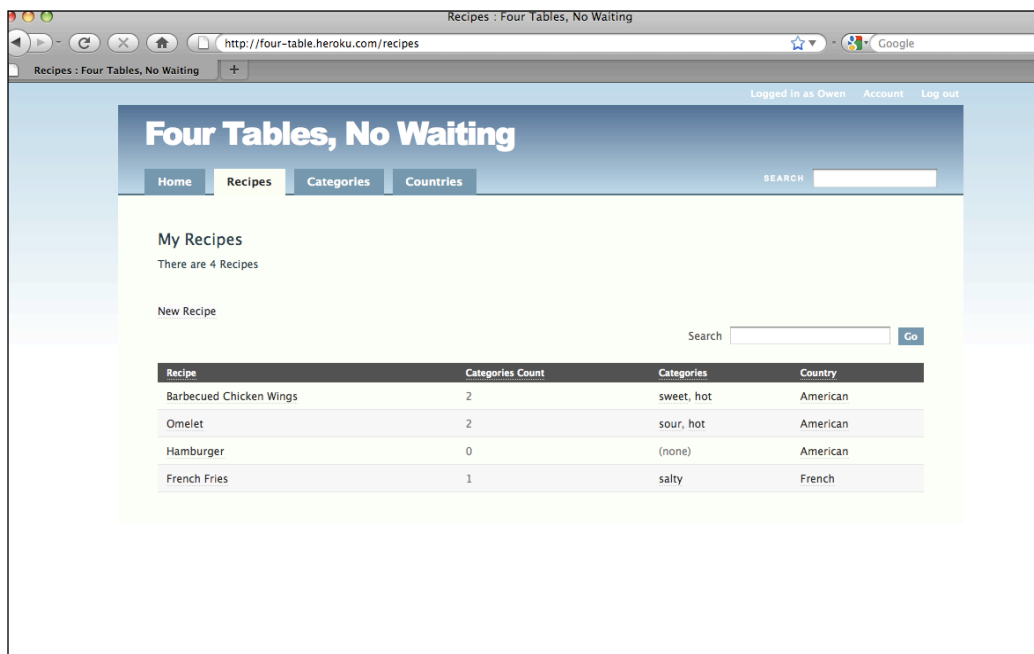


Figure 305: The "Four Table" app on Heroku.com with data

Now let's add a recipe for "Crab Cakes":

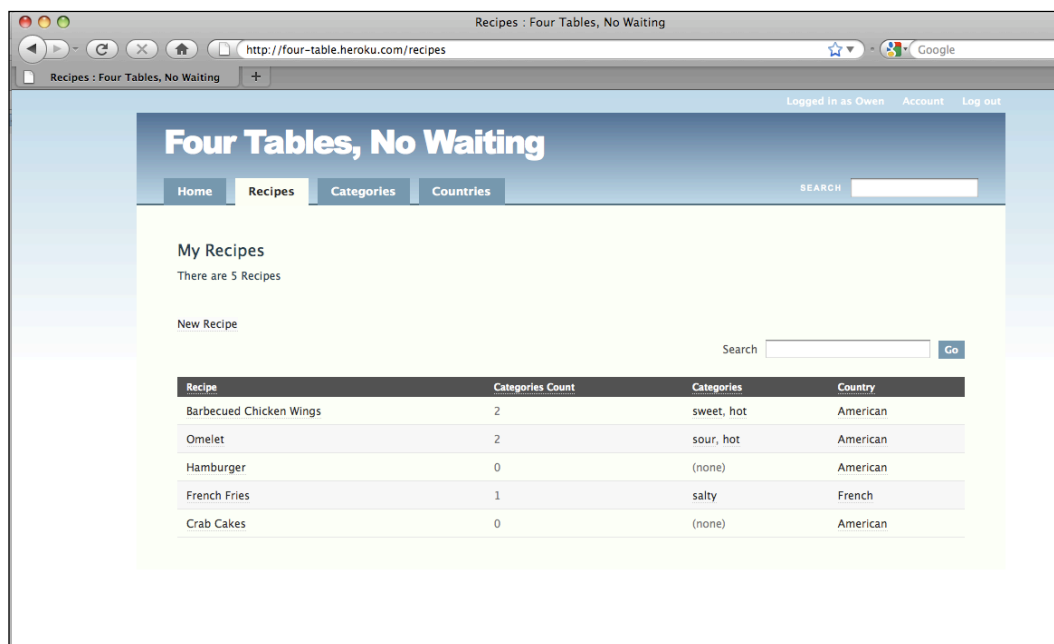


Figure 306: Add a recipe on Heroku.com

Step 9: Pull changed data from Heroku

I can use the “pull” option to backup my change on Heroku to my local database:

```
c:\tutorials\four_table> heroku db:pull
```

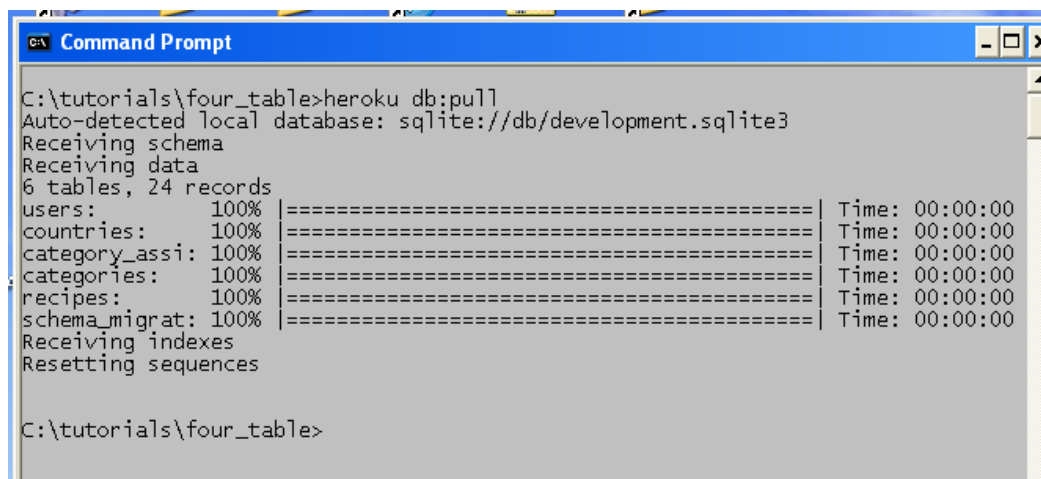


Figure 307: Pull changed data from Heroku.com to your local app

Pretty slick! I now have 24 records on the local version--including my precious recipe for crab cakes.

INDEX

A

account, 40, 61, 63, 88, 240, 299, 303
acting_user, 63, 64, 65, 66, 88, 197, 198, 272, 274
acting_user.signed_up?, 65, 66, 197, 198, 272, 274
action, 63, 67, 68, 69, 70, 71, 72, 73, 75, 76, 79, 95, 108,
109, 111, 112, 114, 133, 135, 138, 152, 153, 160, 169,
170, 171, 187, 189, 215, 218, 260, 270, 273, 274, 277
actions, 65, 67, 68, 69, 70, 72, 73, 74, 75, 76, 91, 103, 104,
108, 150, 162, 165, 166, 167, 170, 187, 188, 189, 190,
191, 193, 194, 218, 231, 239, 243, 273, 274, 277, 278,
280
--add-routes, 279
Administration Sub-Site, 279
agile, 175
application.css, 251, 253, 254, 256, 257, 261, 263, 268
application.dryml, 41, 76, 78, 79, 90, 106, 110, 111, 112,
113, 115, 116, 117, 127, 133, 135, 136, 140, 143, 152,
153, 154, 158, 163, 164, 165, 202, 203, 205, 220, 225,
229, 237, 252, 255, 258, 259, 263, 265, 268, 269, 271
associated record collection, 140
associations, 80, 81, 96, 177, 193, 282
attribute, 59, 71, 80, 86, 118, 122, 124, 125, 136, 141, 142,
146, 147, 149, 154, 158, 159, 160, 162, 163, 164, 170,
206, 212, 213, 217, 221, 224, 238, 243
attributes, 126, 207, 237
attrs, 259, 269
auto_actions, 67, 68, 69, 70, 72, 73, 74, 187, 188, 189, 190,
191, 194, 218, 231, 239, 243, 274, 280
auto-generated, 76, 97, 103, 104, 106, 107, 108, 133, 136,
152, 153, 156, 160, 161, 162, 165, 169, 171, 208
Auto-Generated Tag, 101, 104

B

background:, 251, 253, 254, 256, 257, 258, 261, 264, 266
bar chart, 233, 235
before_filter, 231
belongs_to, 83, 90, 91, 92, 99, 178, 179, 218, 231, 272, 282
body, 61, 80, 81, 83, 91, 109, 111, 114, 124, 135, 136, 137,
138, 139, 141, 143, 144, 150, 152, 154, 155, 157, 158,
161, 162, 164, 167, 176, 204, 206, 210, 238, 251, 253,
257, 259, 263, 264, 270, 274, 275

C

can_edit?, 152, 169
card, 104, 107, 108, 113, 114, 115, 136, 137, 138, 139, 140,
143, 144, 205, 206, 210
cards.dryml, 40, 41, 76, 103, 106, 113, 143
changed, 42, 69, 111, 112, 114, 115, 130, 131, 132, 163,
196, 198, 217, 251, 309
changed?, 198
Changing Field Names, 32, 48
Changing Field Names and Displaying Hints, 32, 48

Charts, 227, 228, 230
children, 97, 99
CKEditor, 173, 215, 223, 224, 225, 226
collection, 111, 114, 115, 133, 135, 136, 137, 138, 139,
140, 141, 142, 144, 146, 147, 148, 149, 152, 155, 157,
162, 176, 206, 209, 210, 212, 213, 230, 232, 234
Constraints, 30
content, 77, 111, 116, 119, 122, 123, 124, 125, 131, 135,
136, 138, 146, 152, 157, 159, 161, 162, 165, 204, 206,
207, 209, 213, 246, 247, 258, 259, 260, 262, 263, 264,
265, 266, 268, 269
content-body, 111, 124, 135, 152, 158, 159, 161, 171, 209
content-header, 111, 124, 135, 152, 161, 204
context, 139, 141, 145, 163, 164, 165, 171, 202, 206, 210
controls, 45, 65, 95, 162, 273, 274
Create, 20, 34, 39, 40, 42, 44, 52, 53, 60, 61, 62, 65, 90,
122, 134, 153, 161, 163, 171, 176, 189, 199, 204, 218,
240, 299, 304
create_permitted?, 63, 64, 65, 88, 197
Creating Tags from Tags, 101, 125
CSS, 51, 103, 139, 140, 159, 246, 248, 250, 253, 254, 256,
260, 261, 263, 266, 267
CSS stylesheet, 246, 248

D

database schema, 37, 60, 84, 305, 306
database.yml, 21, 22, 23
def index, 150
default message text, 122
--default-name, 237, 240, 241, 243
Delete, 146
delete-button, 161
destroy_permitted?, 63, 64, 65, 66, 196, 197
Directories and Generators, 32, 34
display a list of records, 103
display a single record, 75
display collections of record, 103, 140
Display model data in table form, 146
Displaying Hints, 32, 48
div.page-header, 253, 254, 256, 257, 258
do_transition_action, 277
drop-down, 40, 79, 80, 81, 95, 162, 164, 175
DRY, 95
DRYML, 31, 50, 101, 102, 103, 105, 113, 118, 120, 124,
125, 129, 131, 133, 134, 202, 205, 206, 211, 217, 221,
245, 258, 265, 268
DRYML Guide, 211

E

Edit Page, 101, 161
edit.dryml, 221
Editing Auto-Generated Tags, 101, 104
Editing the Navigation Tabs, 32, 76

edit-link, 152, 169
empty-message, 213
enum_string, 80, 81, 193, 215
environment.rb, 221
ERB, 202
error-messages, 162, 165, 166, 167
extending a tag, 127

F

Field Validation, 32, 52
field_names, 49, 201
field-list, 152, 154, 155, 156, 157, 161, 162, 163, 164, 165, 166, 216, 217, 220
flash components, 227
flexibility, 103
force, 43, 47, 73, 83, 231, 234
form, 41, 45, 51, 52, 53, 65, 71, 73, 75, 95, 103, 104, 107, 108, 109, 115, 119, 146, 161, 162, 163, 164, 165, 166, 167, 168, 189, 198, 199, 200, 216, 220, 221, 226, 227, 243, 260, 261, 270, 277
Form Tag, 101, 161
format, 122, 123, 136, 200, 227
FusionCharts, 227, 228, 229, 230, 231, 232, 233, 235, 236

G

gem env, 12
gem list, 12
gem update --system, 11
gems, 12, 22, 31, 287, 288, 302, 304, 305
Git, 184, 289, 290, 291, 293
github.com, 287, 288, 304
Guest user, 274, 277
GUI, 81, 85, 90

H

has_many, 90, 91, 92, 99, 178, 179, 197, 199, 220
has_many
 through relationships, 99
header, 111, 114, 124, 135, 138, 143, 144, 152, 204, 252, 258
heading, 111, 114, 135, 138, 139, 143, 144, 152, 155, 161, 246, 249, 262, 263, 270
height:, 253, 254, 256, 257, 266
Heroku, 173, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309
heroku create, 303
heroku db
 pull, 309
 push, 308
heroku git push, 304
Heroku.com, 173, 296, 297, 298, 299, 301, 306, 307, 308, 309
Hobo, xi, xv, 8, 9, 20, 21, 23, 27, 31, 32, 34, 35, 36, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 54, 55, 58, 60, 61, 62, 63, 64, 65, 67, 68, 69, 70, 71, 73, 75, 76, 77, 79, 80, 81, 82, 83, 84, 86, 88, 90, 91, 92, 93, 95, 96, 97, 99, 102, 103, 104, 106, 107, 108, 109, 110, 111, 112,

113, 114, 115, 116, 118, 120, 124, 125, 127, 133, 134, 135, 138, 139, 140, 141, 142, 143, 145, 146, 147, 148, 149, 152, 153, 154, 157, 159, 160, 162, 163, 165, 167, 168, 169, 170, 171, 172, 176, 179, 187, 188, 189, 193, 195, 199, 200, 202, 204, 207, 212, 217, 218, 220, 221, 223, 224, 226, 237, 243, 244, 245, 246, 247, 248, 249, 271, 287, 288, 298, 304, 305

Hobo Controllers, 32, 67

Hobo Index Generation, 282

Hobo Lifecycles, 272, 278

hobo_migration, 34, 35, 36, 37, 41, 43, 44, 45, 46, 47, 48, 60, 61, 62, 84, 91, 106, 134, 179, 185, 193, 218, 220, 237, 240, 241, 243, 249, 273

hobo_model, 34, 41, 42, 43, 44, 46, 48, 61, 64, 67, 68, 69, 70, 72, 73, 74, 80, 81, 82, 83, 91, 92, 176, 177, 191, 197, 218, 231, 237, 238, 241, 280, 282

hobo_model_controller, 34, 67, 68, 69, 70, 72, 73, 74, 191, 231, 280

hobo_model_resource, 34, 41, 42, 43, 44, 61, 64, 82, 176, 177, 218, 237, 238, 241

HTML, 67, 77, 103, 104, 113, 118, 122, 124, 125, 129, 131, 136, 138, 140, 159, 168, 202, 205, 224, 248, 251

I

id_rsa.pub, 295
implicit, 139, 141, 202, 210
implicit context, 139, 141, 202, 210
include, 8, 34, 56, 69, 70, 103, 110, 199, 200, 229, 271
index.dryml, 112, 113, 118, 122, 124, 127, 129, 133, 134, 135, 140, 142, 146, 203, 204, 230, 232, 233, 259, 260, 263, 264, 273
Indexes, 29
instance variables, 214
Introduction to Permissions, 32, 60

J

join, 141, 142, 199, 205, 206, 212

L

label, 49, 53, 54, 150, 155, 157, 161, 162, 163, 165, 166, 167, 232, 234, 260, 270
lifecycle actions, 274, 277
Lifecycles, 272, 278
Lifecycles for Workflow, 272
limit, 8, 282
Listing Data in Table Form, 101, 146
Listing Data with the Index Tag, 101
log-in, 251

M

Making Your Own Tags, 101
Many-to-many relationships, 90
Matz, 31
merge-attrs, 76, 77, 78, 115

INDEX

method, 59, 63, 64, 71, 74, 80, 108, 112, 125, 127, 150,
193, 194, 198, 213, 214, 231, 234, 283, 284
--migrate, 237, 240, 241, 243
migration, 23, 34, 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46,
47, 48, 49, 60, 61, 62, 81, 82, 83, 84, 91, 92, 93, 106,
134, 179, 180, 181, 185, 193, 215, 218, 220, 237, 240,
241, 243, 249, 273, 282, 283, 284, 298
migrations, 36, 47, 91, 243, 282
Model Relationships, 32, 79, 90
model_name, 61, 65, 71, 107, 117, 200
Model-View-Controller, 31
MVC, 31, 177
MySQL, 16, 17, 18, 20, 21, 284
mysysgit, 289

N

name, xv, 23, 27, 40, 42, 43, 44, 46, 47, 48, 49, 52, 53, 54,
55, 57, 58, 61, 65, 67, 68, 71, 77, 82, 83, 84, 88, 89, 90,
91, 92, 93, 96, 107, 108, 110, 114, 115, 116, 117, 119,
127, 133, 138, 139, 142, 143, 144, 152, 154, 157, 163,
165, 175, 176, 178, 197, 200, 202, 203, 210, 212, 214,
217, 218, 223, 225, 232, 234, 237, 249, 260, 269, 273,
282, 283, 284, 294, 303, 304, 307
name:, 42, 82, 176, 218
named_scope, 231, 274
Navigation Tabs, 32, 76
New and Edit Pages With The Form Tag, 101
not_, 272, 273, 277

O

options, 22, 31, 59, 64, 65, 152, 215, 224, 230, 236, 260,
270, 273, 283, 284, 290, 297, 298
Oracle, 22, 23, 24, 25, 28, 29, 178, 181, 185, 282
Oracle Object Browser, 28
Oracle XE, 25
order, 53, 79, 90, 103, 115, 121, 122, 125, 150, 157, 213,
218, 220, 227, 228, 230, 258, 274, 275, 284
order_by, 150, 213, 274, 275

P

padding:, 253, 254, 256, 257, 258, 261, 264, 266
page-nav, 111, 135
pages.dryml, 40, 41, 76, 103, 106, 107, 108, 110, 111, 113,
115, 152, 153, 160, 161, 207, 208, 209, 212
param, 111, 114, 118, 120, 122, 123, 124, 125, 127, 130,
131, 135, 136, 138, 143, 144, 150, 152, 155, 159, 161,
162, 164, 165, 166, 167, 169, 207, 212, 213, 259, 260,
263, 265, 269, 270, 274, 275
parameter tag, 120, 122, 123, 125, 129, 130, 131, 132, 136,
139, 146, 155, 156, 157, 158, 163, 165, 171, 274
params, 118, 150, 213, 214, 260, 274, 275
parse_sort_param, 150, 213, 274, 275
Permissions, 32, 60, 63, 88, 193, 196, 197, 198, 272
Permissions for data integrity, 198
pie chart, 234, 235, 236
Plugins, 184
plural, 68, 71, 75, 77, 82, 107, 133

polymorphic, 161, 205, 282
post comments to more than one table, 237

R

Rails developers, 36
rake, 36, 47, 237, 240, 241, 243, 305
rake db
 migrate, 36, 47, 237, 240, 241, 243, 305
Rapid Parameter Tag, 156
Read, 224
Record Collections, 133
redirect_to, 277
relationship declarations, 90, 91, 95
Removing actions, 187
Reordering, 220
repeat tag, 141
Rich Text, 223
Roles, 193
ruby script/generate, 35, 42, 43, 45, 46, 48, 60, 61, 62, 82,
84, 176, 177, 179, 218, 220, 249, 279
ruby script/generate hobo_subsite, 279
ruby script/server, 39, 60, 80, 90, 104, 181

S

search, xi, xv, 34, 39, 44, 146, 149, 150, 212, 213, 214,
237, 259, 261, 269, 270, 274, 275
Show Page, 101, 152, 169
Show Page Tag, 101, 152, 169
show.dryml, 152, 153, 154, 155, 159, 160, 171, 207, 209,
212, 216
sign-up, 251
size, 59, 298
skip, 9, 59, 245
sqlite3.dll, 14
SQLite3-ruby gem, 14
state, 43, 44, 46, 47, 48, 72, 79, 90, 272, 273, 274, 275,
276, 283
stylesheets, 51, 246, 251, 268, 269
submit, 161, 162, 163, 165, 166, 167, 216, 227, 260, 270,
277
Sub-Site, 279, 281
subsite, 279

T

tag polymorphism, 163, 164
taglibs, 40, 76, 90, 102, 105, 106, 110, 127, 133, 135, 143,
202, 205, 207, 212, 225, 237, 249, 268
text-indent:, 253, 257
The <a> Tag Hyperlink, 101
theme, 245, 246, 247, 258
timestamps, 42, 45, 46, 48, 80, 81, 82, 83, 91, 92, 197
tnsnames.ora, 27
trailing colon (
) syntax, 145
transition, 272, 273, 274, 277, 278
transition actions, 273
transitions, 272, 278

U

UI, 44, 48, 50, 51, 58, 65, 69, 70, 90, 95, 105, 111, 115,
116, 198, 305
ul, 118, 124, 127, 136, 204, 210, 256, 263, 264, 265, 266,
270
update, 11, 23, 52, 59, 63, 64, 65, 66, 68, 70, 72, 73, 74, 75,
76, 88, 150, 175, 190, 197, 198, 213, 216, 217, 221,
231, 238, 239, 242, 243, 256, 257, 279, 280
update_permitted?, 63, 64, 65, 66, 88, 197, 198
upload data to Heroku.com, 307
User Comments, 173, 237
Users Controller, 194

V

validates_inclusion_of, 56, 58
validates_numericality_of, 54, 55, 58
validates_timeliness, 221
view hints, 49, 200
view_permitted?, 63, 64, 197
ViewHints, 49, 53, 97, 99, 200

W

Working with the Show Page Tag, 101, 152

X

XML, 104, 113, 227, 230, 231, 232, 234, 235