Extracted from:

TextMate

Power Editing for the Mac

Chapter 5

Built-in Automations

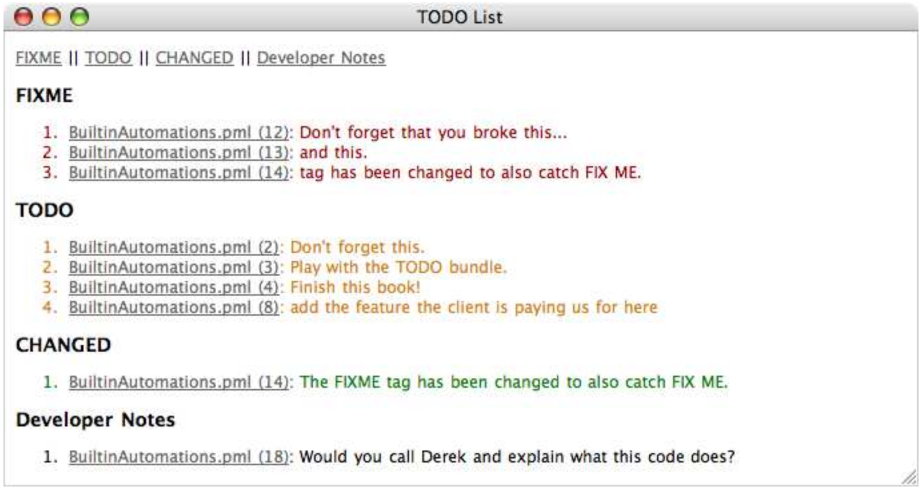
Soon you will learn to build any TextMate automation your heart desires, but first let's get a feel for what is possible by working with some of the built-in automations. TextMate currently ships with more than thirty *bundles*—a convenient grouping of related automations. In this chapter, you will look at bundles for different languages and activities. I don't have space to cover all of them in detail, but I will hit the highlights of many of the most popular bundles.

Bundles are directories of related files packaged on your hard drive. However, from day to day, you will interact with bundles from inside TextMate using either the Bundles menu or the shortcut automation menu with the small gear icon at the bottom of every editing window. Use ⌃⎋ to open the gear menu, navigate with the arrow keys, and select an item by pressing ↩.

It's important to know that bundle contents are available only when an editing window is open. This is a limitation of TextMate, even for commands that don't need a specific file on which to operate. If you go into the Bundles menu and find everything disabled, you probably just need to open a file. This issue will be addressed in a future version of TextMate.

5.1 The TODO Bundle

The TODO bundle consists of only two commands: Help and Show TODO List. The Help command describes Show TODO List. It's common for bundles to include a Help command like this, so be sure to look for them when you are exploring on your own. Take a moment to read that brief text now, and then I'll show you some additional tricks.



FIXME || TODO || CHANGED || Developer Notes

FIXME

1. BuiltinAutomations.pml (12): Don't forget that you broke this...
2. BuiltinAutomations.pml (13): and this.
3. BuiltinAutomations.pml (14): tag has been changed to also catch FIX ME.

TODO

1. BuiltinAutomations.pml (2): Don't forget this.
2. BuiltinAutomations.pml (3): Play with the TODO bundle.
3. BuiltinAutomations.pml (4): Finish this book!
4. BuiltinAutomations.pml (8): add the feature the client is paying us for here

CHANGED

1. BuiltinAutomations.pml (14): The FIXME tag has been changed to also catch FIX ME.

Developer Notes

1. BuiltinAutomations.pml (18): Would you call Derek and explain what this code does?

Figure 5.1: Show TODO List output

The Show TODO List command (^⇧T) is where the action is. When invoked, all files in the current project (or the current file if working outside a project) are saved, and TextMate starts scanning for tags. The tags the command looks for can have a couple of formats:

```
TODO Don't forget this.
TODO, Play with the TODO bundle.
TODO:  Finish this book!
```

It's also important to note that the tag doesn't have to be the first thing on the line. That allows you to hide TODO items in the comments of a programming language. For example:

```
# TODO:  add the feature the client is paying us for here
```

The command isn't limited to TODO tags either:

```
FIXME:  Don't forget that you broke this...
FIX ME:  and this.
CHANGED:  The FIXME tag has been changed to also catch FIX ME.
```

When invoked, these tags are located, sorted by type, color-coded, and hyperlinked back to the line of the file where they were found. The result appears in TextMate's HTML output window. You can see what I mean in Figure 5.1, which shows the output of the tag examples in this chapter.

Joe Asks...

What Happens When My Brain Is Full?

I'm sure you remember that spirited pep talk I gave early in the book about learning your keyboard shortcuts, and I really do mean that. Everyone has their limit of what they can effectively remember, though, and no matter what your limit is, this chapter will likely exceed it. I cover twelve bundles in these pages, and that's just a fraction of what comes with TextMate. That's a lot of automations, and I throw around keyboard shortcuts for all of them. Keep the shrink bills low, and be selective about what you actually commit to memory.

First, you may not need all these bundles. If you never build web applications with Rails, for example, feel free to skip that bundle. Don't skip TODO, Math, Text, Source, or TextMate, though, because I think they might surprise you.

Next, when you do find a bundle worth committing to memory, decide what level of attention it deserves. Is this a I-will-use-it-five-times-an-hour batch of commands? OK, spend the effort learning the keyboard shortcuts. However, you can still get mileage out of a bundle without knowing the shortcut. I use the TODO bundle often, but I had to look up the keyboard shortcut to add it to this chapter. That's just because the TODO hunting mode of my workflow is a mousy action, so I don't need to memorize the shortcut. Decide when you can and can't get away with the same.

Also, be sure to notice the patterns in TextMate's shortcuts. They are there on purpose. ⌃Q is always reformat in the current context; ⌘B and ⌘I are bold and italics anywhere that makes sense; ⌅ is continue list, comment, or whatever; and ⌃⌥⌘P is how you get a preview. The bundles have a lot more patterns than that. Learn them once, and use them everywhere.

Finally, you can hunt for a command just as I've trained you to do for files and symbols. ⌃⌘T will open a dialog box with the now-famous name-matching algorithm for bundle commands. Use as needed, but if you go after the same command three times in one day, I say it's time to learn that shortcut!

The TODO bundle is helpful with what you have seen so far, but the real power lies in how easily you can add your own tags to the mix. For example, the company I work for often embeds notes to the developers in the source code we write. A note is always tagged by the intended developer's first name. I can modify the Show TODO List command to pick up those notes with a few easy steps:

1. Open the Bundle Editor (^ ⌥ ⌘ B).
2. Click the folding arrow to the left of the TODO bundle's name to expose the bundle's contents.
3. Click Show TODO List so you can see the source in the Edit Command box.
4. Add a line like the following in the $tags = [...] definition right at the beginning of the script:
   ```
   { :label  => "Developer Notes",
     :color  => "#0090C8",
     :regexp => /JAMES[-\s,:]+(\w.*)$/i },
   ```

That will find the developer note tags used by my company, as long as they are addressed to me. I've added the ability to place hyphens after the tag as well, since we tend to do just that. These tags will turn a light blue, but you could modify the HTML color code to get your favorite hue. Here's a sample note from my company found by the addition we made to the command:

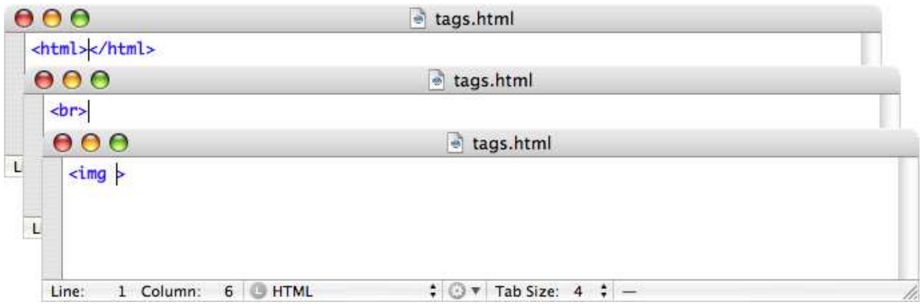
```
# JAMES--Would you call Derek and explain what this code does?
```

5.2 The HTML and CSS Bundles

The HTML bundle is TextMate's most popular bundle. So many people need to create a quick web page, and HTML is an easy markup language to learn. Even still, the markup can get repetitive, and we humans tend to make mistakes when dealing with such languages—well, only those humans who don't use TextMate as their editor, that is.

If you are going to learn only one command from the HTML bundle, definitely make it Insert Open/Close Tag (With Current Word). I'm not exaggerating when I say that command, available in any document via ^<, is 90% of what you need to write HTML, XHTML, or even XML quickly and effectively. I wrote this book primarily with that one command.

You can use Insert Open/Close Tag in two ways. First, you can type a tag name—html, div, or a, for example—and then trigger the command. The tag name will be transformed into an open and close tag pair. This



```
<html></html>
```

```
<br>
```

```
<img >
```

Figure 5.2: Insert Open/Close Tag

command is super smart, knowing to handle words like br and img with a single tag. It also always tries to drop your caret in the most logical place you need to be next. See Figure 5.2, and note the caret positions.

The other way to use the command is equally handy. Just invoke it with no word to the left of the caret, and it will generate and insert an open and close tag pair snippet. You can then type a name for the tag that will update both ends of the pair. Feel free to add tag attributes as needed, and know that they will not be copied to the closing tag. When you are ready, just press ⇥ to jump into the content portion of the tag.

Experiment a little with both forms before reading on so you can get a good feel for the command. The two uses are handy in different contexts, and you are well armed if you know both.

Two other commands are helpful for quickly building tags: Wrap Selection in Open/Close Tag (^ ⇧ W) and Wrap Each Selected Line in Open/Close Tag (^ ⇧ ⌘ W). They both do what their names suggest, adding open and close tag pairs in front of and behind the entire selection or at the start and end of each line in the selection. You can then type a tag name that will be mirrored to all inserted instances. These commands are nice for naturally typing several paragraphs or list items without stopping to worry about syntax and then marking them up after the fact.

Of course, the HTML bundle has many other commands. Among them are some snippets for inserting tags that commonly need a little extra baggage you don't want to have to type all the time. Good examples are Head, Style, and Script. Try typing head⇥ in an HTML document to see what I mean. It will insert a content-type tag and set you up to edit the document title. You trigger the other two snippets I mentioned the same way (style⇥ or script⇥), and all snippets of this kind are available in the Insert Tag submenu of the HTML bundle.

I just don't have the room to explain all the excellent helpers hiding in the HTML bundle, but I'll close this section with a few more quick tips:

- Wrap Selection as Link (⌃⇧L) works like the other wrap commands I discussed, but it will use the clipboard contents as the uniform resource locator (URL) for the link, assuming the clipboard contains a single line of text.
- For effortless linking, select the link text, and trigger Lookup Selection on Google and Link (⌃⇧⌘L). This command uses Google's "I'm Feeling Lucky" to find the first match in the search list and builds a hyperlink to that site.
- Type doctype⇥ at the beginning of your document to select one of those hard-to-remember statements.
- Type ⌘& for a menu of several useful commands involving entity and URL escapes.
- The common Mac shortcuts ⌘B and ⌘I work in HTML documents to create strong and em(phasis) tags.
- Use Window → Show Web Preview (⌃⌥⌘P) to check the result of your markup work with live updating, or use the Open Document in Running Browser(s) and Refresh Running Browser(s) (⌘R) commands in the HTML bundle to manage an external preview.
- Use Validate Syntax (W3C), available via ⌃⇧V, to ensure that your markup is free from errors.

You can see an example using several of the HTML automations to build a web page with minimal effort in Section 3.3, *Editing Multiple Lines at Once*, on page 41.

Adding Style

TextMate makes the CSS bundle available when you're editing a style sheet or even if you are just inside a style tag in an HTML document. CSS is not complex enough to require a powerful command suite like the HTML bundle has, but the bundle still has some useful snippets.

The hard part of CSS work, in my opinion, is remembering all the combinations of what goes after a given identifier. For example, after the margin identifier, you can put one argument to set all four margins; two to set vertical and horizontal margins; or four to set top, right, bottom, and left individually. Just to write that last sentence, though, I had to look up the order. I never remember it.

That's how TextMate can help you with CSS work. It remembers all the combinations. To get it to remind you, just trigger the menus. You can do that by typing a word like background, border, font, list, margin, padding, or text, followed by ⇥ in a style sheet or style tag. The menu will have all the common choices, and inserting one will include placeholders that prompt you for what to fill in at each point.

If you also have as much trouble remembering HTML color codes as I do, you will be happy to hear that the CSS bundle covers those too. You can choose Bundle → CSS → Insert Color (⇧ ⌘ C), and TextMate will display the standard Mac color chooser and allow you to make a choice. The color you select is converted to the expected string of six hexadecimal digits with a leading number sign and inserted at the caret position.

5.3 The Ruby Bundle

Because TextMate's own automations use Ruby heavily, TextMate has great support for the language via a full set of automations in the Ruby bundle.

The first thing you need in Ruby support is a way to run your scripts. TextMate ships with the RubyMate runtime environment invoked by Bundles → Ruby → Run (⌘ R in any Ruby document). This hands your code off to Ruby and displays program output in TextMate's HTML output window. Before the hand-off, though, TextMate modifies the standard Ruby environment to include some nice tie-ins to the graphical user interface (GUI). TextMate arranges to be notified of uncaught exceptions and hyperlinks the stack trace output back to the lines of your file. STDIN is also modified, so a call to gets() will trigger a GUI dialog box that sends your input down to the script. The environment even detects when you are running tests so it can color-code those results and hyperlink errors and failures.

RubyMate is great for running entire scripts, but Rubyists, spoiled by the ease of IRb, often want to evaluate some little snippet and see the results. You could switch to the Terminal and use IRb itself, but TextMate provides another option. When you run Bundles → Ruby → Execute and Update '# =>' Markers, TextMate filters the selected code, or the entire document in the event of no selection, through a script. That script updates # => markers you have placed at the ends of lines with the results of that expression. You can insert such a marker using a snippet bound to #⇥. The script will also annotate errors and show content printed to STDOUT. This is powerful tool for quick localized debugging or testing. For example, filtering the following code through the command:

```
RUBY_VERSION  # =>

data    = %w[one two three four five]
results = data.select { |n| n[0] == ?t }  # =>
```

yields the following:

```
RUBY_VERSION  # => "1.8.4"

data    = %w[one two three four five]
results = data.select { |n| n[0] == ?t }  # => ["two", "three"]
```

The majority of the Ruby bundle focuses on writing code, not running it. Many snippets exist for building Ruby code quickly. The good news is that the tab triggers follow mnemonic patterns to make them easier to remember.

When you are ready to create a new Ruby class or module, just press cla⇥ or mod⇥ for a menu of common skeletons. You can use a similar trigger for methods on def⇥, or you can use the variations defs⇥ for a class or module method and deft⇥ for a test method. The tab triggers r⇥, w⇥, and rw⇥ are shortcuts for Ruby's attr_reader(), attr_writer(), and attr_accessor() helpers. The snippets continue all the way down to simple language constructs available on triggers such as if⇥, case⇥, and while⇥. Get into the habit of using these, and you will never need to type **end** again.

Probably the most widely used Ruby snippets are in the iterator family of snippets. Again, the tab triggers follow patterns to make them easy to remember. Specifically, one-word iterators are available via the first three letters of the word, so inj⇥ will trigger inject(), and tim⇥ will trigger times(). The exception is each(), which uses the common abbreviation for the word ea⇥. If the iterator has more than one word, add the first letter of each additional word to the trigger, so sorb⇥ activates sort_by() and eawi⇥ activates each_with_index(). Though they may seem odd now, you will learn the patterns pretty fast with practice and will seldom need to look up snippet triggers again.

Note that all the snippets use the braces syntax and that there are no snippets for the "bang" variations. You can use two commands inside an iterator body to change it to the desired variation: Add ! to Method in Line (^ !) and Toggle 'do ... end' / '{ ... }' (^ {).

TextMate provides a similar suite of snippets for unit testing. You can build a skeleton test case file with tc⇥ or a test suite file with ts⇥. All the assertions work just like the iterators, building off the base of as⇥ for assert(), so you can trigger assert_in_delta() with asid⇥, for example.

Two facts to know about the automations in the Ruby bundle are that some of them insert requires for the needed standard library files when triggered, unless your document already has the require, of course, and some are based on fictional method names to make them easier to remember. You can suppress the autorequire behavior by adding - to the tab trigger. Both eas⇥ and eas-⇥ will insert an each_slice() snippet, but the first will make sure the current document requires the enumerator library. Examples of fictional methods include map_with_index(), class_from_name(), and word_wrap(), which all insert common idioms of Ruby code to handle these operations.

Finally, the bundle provides Documentation for Word (^ H) for easy access to Ruby's built-in documentation. Just place your caret in the word you want to look up, and then trigger the command for a hyperlinked HTML response.

5.4 The Rails Bundle

The Rails core team has done a good job of advertising TextMate in its screencasts, so it's not surprising that TextMate has become a favored choice for building Rails applications. Attracting all those Rails developers has also attracted some terrific automations for the Rails bundle.

To use the Rails bundle, you need to give TextMate the hint that you are a Rails developer. You need to do this because Rails files look like regular Ruby files to TextMate. When you have a Ruby file open, glance down at the language menu embedded in the bottom of the editing window. If that menu says *Ruby*, the Rails bundle isn't yet active. To kick it into gear, select Ruby on Rails from that same menu. In Rails mode, you have access to all the Ruby goodies plus the entire Rails bundle.

For an added boost to TextMate Rails development, I recommend installing the TextMate Footnotes plugin. This plugin will add links to the pages of your application under development mode that you can click to jump right to that file in TextMate for editing. There are also links to display the parameters and session inline in the page. All are helpful and won't affect your code when it is running in production mode.

To ask TextMate to add this functionality to your Rails project, open your project, select Bundles → Rails → Install Plugin (^ | and press 2), type footnotes into the plugin search field, and click the Go button (↩). Then click the download arrow button to the far right of the TextMate Footnotes match to install the plugin.

The Rails bundle includes timesaving automations for working with each layer of a Rails application. With models, the bundle has many snippets for validations and association methods such as has_one() and belongs_to(). You can find these snippets in the Models submenu of the Rails bundle. They have mnemonic tab triggers similar to those in the Ruby bundle.

My favorite feature of the Rails bundle's model layer is the support for migrations. Here again you have some snippets for quick entry, but the tab triggers are well thought out to maximize productivity. To see what I mean, assume you have a typical create table migration started with the following:

```
create_table :favorites do |t|

end
```

Now you're ready to add a handful of columns, so you put your caret in the table block and type mccc⇥ to trigger Create Several Columns:

```
create_table :favorites do |t|
  t.column :user_id, :integer
  mccc
end
```

Notice how that snippet sets up entry and drops in the trigger again for the next column. There is even a tab stop right after that mccc, so you will naturally end up there. When you are about to enter the last column, just tap ⌫ a couple of times, and add an ol before triggering the snippet. This mcol⇥ trigger will open a menu of column creation choices from which you can select Create Column in Table by pressing 9. Then you won't have the trailing snippet trigger to clean up.

A couple of the migrations are even smarter. Drop/Create Table (mtabt⇥, choice 6) and Remove/Add Column (mcol⇥, choice 8) will insert regular snippets for the change, with special triggers at the end to kick off macros. When the macros are triggered, your db/schema.rb file will be scanned by the bundle for the details of the table or column changed. Those details are reinserted as the migration's down() action so the table or column will be restored to its former state.

With views, the shortcut to master is ^>. When you trigger that snippet, you get <%= %>; however, pressing it again accesses a command that switches the tag to <% %>. This pair of automations is actually located in the Ruby bundle, but they are commonly used to edit the RHTML files of Rails projects.

Another nice view helper is Create Partial from Selection (^⇧H). You can use it, as the name implies, to separate a selection of code into a new partial file and insert a render() call to that file, but that's not all. If you invoke it without a selection, it will read all partial render() calls in the current document and inline the full partial just below each call. You can edit the contents of these documents and trigger the command again to return them to their files. See Figure 5.3, on the next page, for an example of how this plays out.

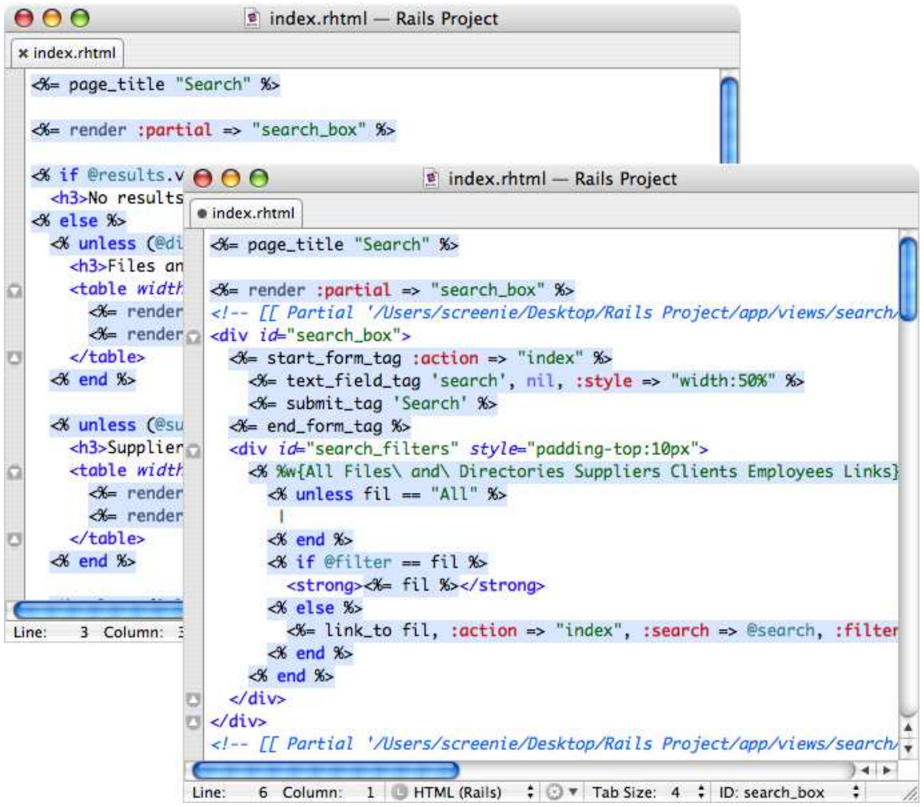
Controller work in the Rails bundle benefits from nice snippets for renders and redirects. Again, the triggers are mnemonic, and you can explore these snippets under Bundles → Rails → Controllers. Don't miss the params[...] (^P) and session[...] (^J) snippets in the top-level of the Rails bundle, because single-keystroke access is nice for those terms you type repeatedly.

Another challenge with Rails development is quickly navigating to the file you need to edit now. The Rails bundle adds some terrific shortcuts for this. I recommend committing one to memory. ⌥⇧⌘↓ will open a menu where you can select to jump to a Controller, Helper, Functional Test, Unit Test, View, or more. That makes getting around a breeze with just one shortcut to remember. Be sure to look in the Go To submenu of the Rails bundle, though, for other interesting navigation commands.

5.5 The Subversion Bundle

Most people have come to understand the value of version control, and if you're going to use version control these days, Subversion is a popular choice.[1] When you have a Subversion repository checked out, TextMate has a bundle of features that can help you get the most out of it.

1. If you are not yet familiar with Subversion, *Pragmatic Version Control Using Subversion* by Mike Mason (Pragmatic Bookshelf, 2005) is a great way to get started.



index.rhtml — Rails Project

```
<%= page_title "Search" %>

<%= render :partial => "search_box" %>

<% if @results.v
  <h3>No results
<% else %>
  <% unless (@di
    <h3>Files an
    <table width
      <%= render
      <%= render
    </table>
  <% end %>

  <% unless (@su
    <h3>Supplier
    <table width
      <%= render
      <%= render
    </table>
  <% end %>
```

Line: 3 Column: 3

index.rhtml — Rails Project

```
<%= page_title "Search" %>

<%= render :partial => "search_box" %>
<!-- [[ Partial '/Users/screenie/Desktop/Rails Project/app/views/search/
<div id="search_box">
  <%= start_form_tag :action => "index" %>
    <%= text_field_tag 'search', nil, :style => "width:50%" %>
    <%= submit_tag 'Search' %>
  <%= end_form_tag %>
  <div id="search_filters" style="padding-top:10px">
    <% %w{All Files\ and\ Directories Suppliers Clients Employees Links}
      <% unless fil == "All" %>
        |
      <% end %>
      <% if @filter == fil %>
        <strong><%= fil %></strong>
      <% else %>
        <%= link_to fil, :action => "index", :search => @search, :filter
      <% end %>
    <% end %>
  </div>
</div>
<!-- [[ Partial '/Users/screenie/Desktop/Rails Project/app/views/search/
```

Line: 6 Column: 1 HTML (Rails) Tab Size: 4 ID: search_box

Figure 5.3: Editing partials inline

As I said, you still need to handle the initial checkout before TextMate can help you with a repository. Luckily, that's just one command you need to feed the Terminal:

```
$ svn --username your.name.here checkout repository.url.here
```

After you have a checkout, just drag the top-level repository folder onto TextMate to create a project. This is a standard TextMate project just like the ones discussed in Chapter 2, *Projects*, on page 24. However, because this project is a Subversion checkout, you have access to the Subversion commands[2] for the files and directories contained within.

2. If TextMate has trouble locating your installed copy of Subversion, you can guide it to the binary by setting the TM_SVN variable to the path of the executable. See Section 9.2, *TextMate's Environment Variables*, on page 121, for details on how to set environment variables inside TextMate.

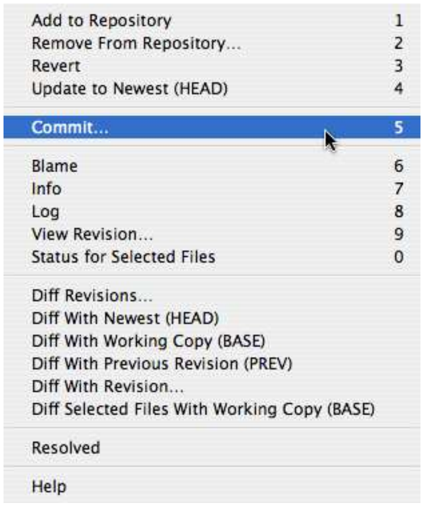

Add to Repository 1
Remove From Repository... 2
Revert 3
Update to Newest (HEAD) 4
Commit... 5
Blame 6
Info 7
Log 8
View Revision... 9
Status for Selected Files 0
Diff Revisions...
Diff With Newest (HEAD)
Diff With Working Copy (BASE)
Diff With Previous Revision (PREV)
Diff With Revision...
Diff Selected Files With Working Copy (BASE)
Resolved
Help

Figure 5.4: Subversion menu

Let's examine where you can find those commands and what they will do for you. All the Subversion commands are mapped to the same keystroke, ^ ⇧ A, so that triggers a nice menu of choices. You can see what this menu looks like in Figure 5.4.

These commands work just as they do when interacting with Subversion in the Terminal, but you get to do everything from the comfort of TextMate. Commands that need input from you to operate will display GUI windows when invoked. For example, Commit will open a window that accepts your commit message and allows you to change the files included in the commit, and Diff with Revision will show a window that allows you to select the version to which to compare the current file. Many commands show their output using TextMate's HTML output window.

Commands that require file selections to work on use the files and directories selected in the project drawer. Therefore, if you want to update the entire project to the newest revision, make sure the top-level directory is selected in the project drawer.

Probably the biggest advantage of using the TextMate bundle over the shell is that the Diff commands pipe their results into a new TextMate