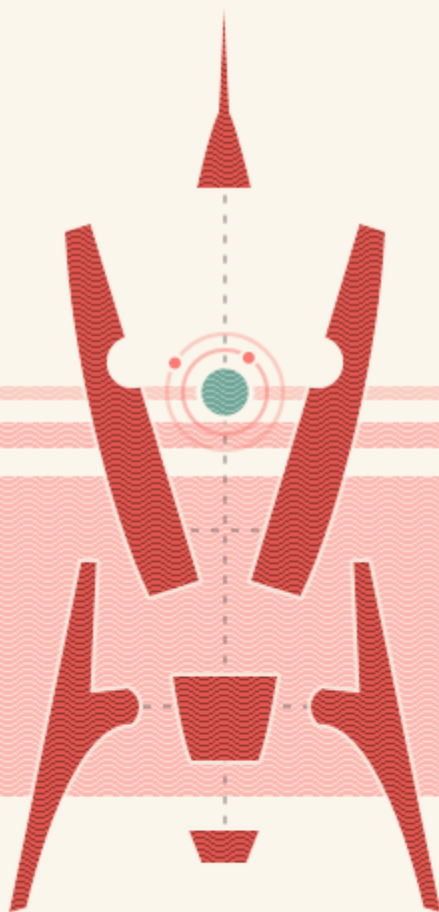




ATOM

flight manual



1.0

<> with ♥ by **GitHub**

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit [**http://creativecommons.org/licenses/by-sa/3.0/**](http://creativecommons.org/licenses/by-sa/3.0/) or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Introduction

Welcome to the Flight Manual for Atom, the Hackable Text Editor of the 21st Century.

In the very simplest terms, Atom is a hackable text editor. It is like any other text editor in that you generally use it to open plain text files like computer programs, modify them and save them again. It is also much more in that it's built on well known web technologies such as HTML and CoffeeScript and is quite easy to customize, extend and modify.

In this book, we'll cover everything from the basics that you need to know to edit files like an elementary school student all the way to building brand new Atom-like applications with the shell that underlies Atom. We'll also cover everything in between.

Strap into your seat and get your oxygen tank ready, we're about to take off.

Table of Contents

Introduction	iii
CHAPTER 1: Getting Started	11
Why Atom	11
The Nucleus of Atom	11
An Open Source Text Editor	12
Installing Atom	13
Atom on Mac	14
Atom on Windows	14
Atom on Linux	15
Atom from Source	16
Setting up a Proxy	16
Atom Basics	16
Basic Terminology	17
Command Palette	18
Settings and Preferences	19
Opening, Modifying and Saving Files	22
Opening Directories	24
Summary	27
CHAPTER 2: Using Atom	29

Atom Packages	29
Package Settings	30
Atom Themes	31
Command Line	33
Moving in Atom	34
Navigating by Symbols	35
Atom Bookmarks	36
Atom Selections	37
Editing and Deleting Text	38
Basic Manipulation	39
Deleting and Cutting	39
Multiple Cursors and Selections	40
Whitespace	41
Brackets	42
Encoding	43
Find and Replace	44
Snippets	46
Creating Your Own Snippets	47
Autocomplete	49
Folding	50
Panes	51
Grammar	52
Version Control in Atom	53
Checkout HEAD revision	53
Git status list	54
Commit editor	55
Status bar icons	56
Line diffs	56
Open on GitHub	57
Writing in Atom	58
Spell Checking	58

Previews	60
Snippets	61
Basic Customization	62
Configuring with CSON	62
Style Tweaks	63
Customizing Key Bindings	65
Global Configuration Settings	66
Language Specific Configuration Settings	68
Summary	70
CHAPTER 3: Hacking Atom	71
Tools of the Trade	71
The Init File	72
Package: Word Count	73
Package Generator	73
Developing our Package	81
Basic Debugging	85
Testing	86
Publishing	87
Summary	89
Package: Modifying Text	89
Basic Text Insertion	90
Add the ASCII Art	92
Summary	93
Creating a Theme	93
Getting Started	93
Creating a Syntax Theme	93
Creating an Interface Theme	94
Development workflow	95
Theme Variables	97
Iconography	100

Debugging	101
Update to the latest version	101
Check for linked packages	101
Check Atom and package settings	101
TimeCop	103
Check the keybindings	103
Check if the problem shows up in safe mode	105
Check your config files	105
Check for errors in the developer tools	105
Diagnose performance problems with the dev tools CPU profiler	106
Check that you have a build toolchain installed	107
Writing specs	107
Create a new spec	108
Asynchronous specs	109
Running specs	110
Converting from TextMate	111
Converting a TextMate Bundle	111
Converting a TextMate Theme	111
Summary	112
CHAPTER 4: Behind Atom	113
Configuration API	113
Reading Config Settings	113
Writing Config Settings	114
Keymaps In-Depth	114
Structure of a Keymap File	114
Removing Bindings	116
Forcing Chromium’s Native Keystroke Handling	117
Overloading Key Bindings	117
Step-by-Step: How Keydown Events are Mapped to Commands	118
Scoped Settings, Scopes and Scope Descriptors	118

Scope names in syntax tokens	118
Scope Selectors	119
Scope Descriptors	119
Serialization in Atom	120
Package Serialization Hook	120
Serialization Methods	121
Versioning	122
Developing Node Modules	122
Linking a Node Module Into Your Atom Dev Environment	122
Interacting With Other Packages Via Services	123
Maintaining Your Packages	124
Unpublish a Version	124
Adding a Collaborator	124
Transferring Ownership	125
Unpublish Your Package	125
Rename Your Package	125
Summary	126
Upgrading to 1.0 APIs	127
Index	147

Getting Started 1

This chapter will be about getting started with Atom.

Why Atom

There are a lot of text editors out there, why should you spend your time learning about and using Atom?

Editors like Sublime and TextMate offer convenience but only limited extensibility. On the other end of the spectrum, Emacs and Vim offer extreme flexibility, but they aren't very approachable and can only be customized with special-purpose scripting languages.

We think we can do better. Our goal is a zero-compromise combination of hackability and usability: an editor that will be welcoming to an elementary school student on their first day learning to code, but also a tool they won't outgrow as they develop into seasoned hackers.

As we've used Atom to build Atom, what began as an experiment has gradually matured into a tool we can't live without. On the surface, Atom is the modern desktop text editor you've come to expect. Pop the hood, however, and you'll discover a system begging to be hacked on.

The Nucleus of Atom

The web is not without its faults, but two decades of development has forged it into an incredibly malleable and powerful platform. So when we set out to write a text editor that we ourselves would want to extend, web technology was the obvious choice. But first, we had to free it from its chains.

THE NATIVE WEB

Web browsers are great for browsing web pages, but writing code is a specialized activity that warrants dedicated tools. More importantly, the browser se-

verely restricts access to the local system for security reasons, and for us, a text editor that couldn't write files or run local subprocesses was a non-starter.

For this reason, we didn't build Atom as a traditional web application. Instead, Atom is a specialized variant of Chromium designed to be a text editor rather than a web browser. Every Atom window is essentially a locally-rendered web page.

All the APIs available to a typical Node.js application are also available to the code running in each window's JavaScript context. This hybrid provides a really unique client-side development experience.

Since everything is local, you don't have to worry about asset pipelines, script concatenation, and asynchronous module definitions. If you want to load some code, just require it at the top of your file. Node's module system makes it easy to break the system down into lots of small, focused packages.

JAVASCRIPT, MEET C++

Interacting with native code is also really simple. For example, we wrote a wrapper around the Oniguruma regular expression engine for our TextMate grammar support. In a browser, that would have required adventures with NaCl or Esprima. Node integration made it easy.

In addition to the Node APIs, we also expose APIs for native dialogs, adding application and context menu items, manipulating the window dimensions, etc.

WEB TECH: THE FUN PARTS

Another great thing about writing code for Atom is the guarantee that it's running on the newest version of Chromium. That means we can ignore issues like browser compatibility and polyfills. We can use all the web's shiny features of tomorrow, today.

For example, the layout of our workspace and panes is based on flexbox. It's an emerging standard and has gone through a lot of change since we started using it, but none of that mattered as long as it worked.

With the entire industry pushing web technology forward, we're confident that we're building Atom on fertile ground. Native UI technologies come and go, but the web is a standard that becomes more capable and ubiquitous with every passing year. We're excited to dig deeper into its toolbox.

An Open Source Text Editor

We see Atom as a perfect complement to GitHub's primary mission of building better software by working together. Atom is a long-term investment, and Git-

Hub will continue to support its development with a dedicated team going forward. But we also know that we can't achieve our vision for Atom alone. As Emacs and Vim have demonstrated over the past three decades, if you want to build a thriving, long-lasting community around a text editor, it has to be open source.

The entire Atom editor is free and open source and is available under the <https://github.com/atom> organization.

Installing Atom

To get started with Atom, we'll need to get it on our system. This section will go over installing Atom on Mac, Windows and Linux, as well as the basics of how to build it from source.

Installing Atom should be fairly simple on any of these systems. Generally you can simply go to <https://atom.io> and at the top of the page you should see a download button as in **Figure 1-1**.

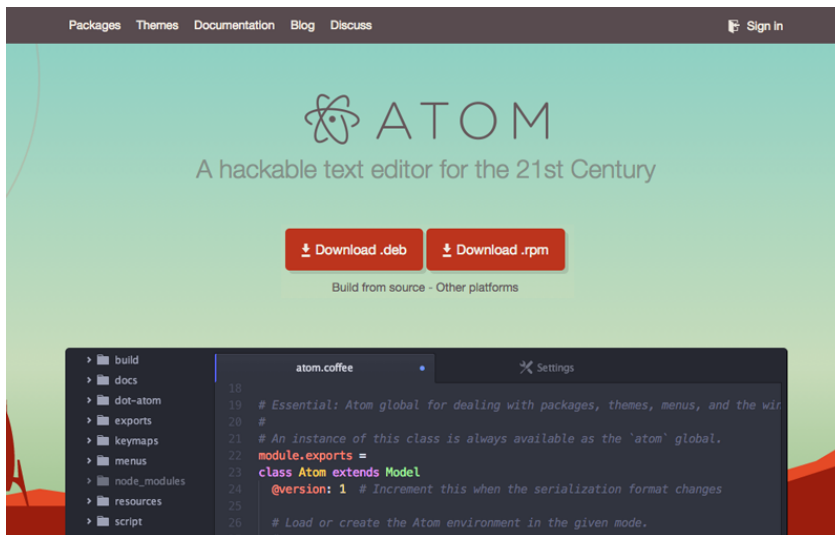


FIGURE 1-1

Download buttons on atom.io

The buttons should be specific to your platform and easily installable. However, let's go over them here in a bit of detail.

Atom on Mac

Atom was originally built for Mac and should be a simple setup process. You can either hit the download button from the atom.io site or you can go to the Atom releases page at:

<https://github.com/atom/atom/releases/latest>

Here you can download the `atom-mac.zip` file explicitly.

Once you have that file, you can click on it to extract the binary and then drag the new Atom application into your “Applications” folder.

When you first open Atom, it will try to install the `atom` and `apm` commands for use in the terminal. In some cases, Atom might not be able to install these commands because it needs an administrator password. To check if Atom was able to install the `atom` command, for example, open a terminal window and type `which atom`. If the `atom` command has been installed, you’ll see something like this:

```
$ which atom
/usr/local/bin/atom
$
```

If the `atom` command wasn’t installed, the `which` command won’t return anything:

```
$ which atom
$
```

To install the `atom` and `apm` commands, run “Window: Install Shell Commands” from the **Command Palette**, which will prompt you for an administrator password.

Atom on Windows

Atom comes with a windows installer. You can download the installer from <https://atom.io> or from:

<https://github.com/atom/atom/releases/latest>

This will install Atom, add the `atom` and `apm` commands to your PATH, create shortcuts on the desktop and in the start menu, and also add an Open with Atom context menu in the Explorer.

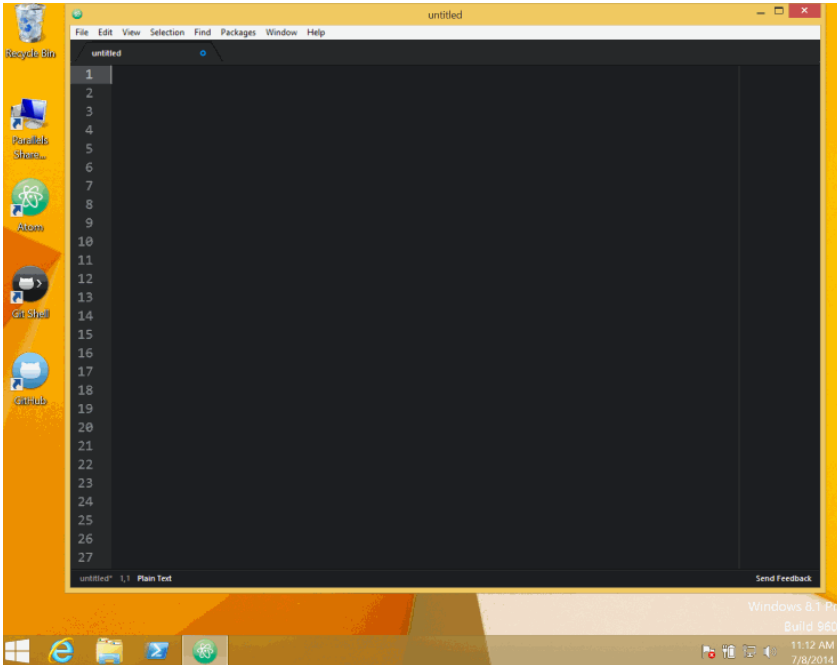


FIGURE 1-2
Atom on Windows

If you just want to download a .zip of the latest Atom release for Windows, you can also get it from the **Atom releases page** at <https://github.com/atom/atom/releases>.

Atom on Linux

To install Atom on Linux, you can download a **Debian package** or **RPM package** either from the **main Atom website** at atom.io or from the Atom project releases page at <https://github.com/atom/atom/releases>.

On Debian, you would install the Debian package with `dpkg -i`:

```
$ sudo dpkg -i atom-amd64.deb
```

On RedHat or another RPM based system, you would use the `rpm -i` command:

```
$ rpm -i atom.x86_64.rpm
```

Atom from Source

If none of those options works for you or you just want to build Atom from source, you can also do that.

There are detailed and up to date build instructions for Mac, Windows, Linux and FreeBSD at: <https://github.com/atom/atom/tree/master/docs/build-instructions>

In general, you need Git, a C++ toolchain, and Node to build it. See the repository documentation for detailed instructions.

Setting up a Proxy

If you're using a proxy, you can configure **apm** (Atom Package Manager) to use it by setting the `https-proxy` config in your `~/.atom/.apmrc` file:

```
https-proxy = https://9.0.2.1:0
```

If you are behind a firewall and seeing SSL errors when installing packages, you can disable strict SSL by putting the following in your `~/.atom/.apmrc` file:

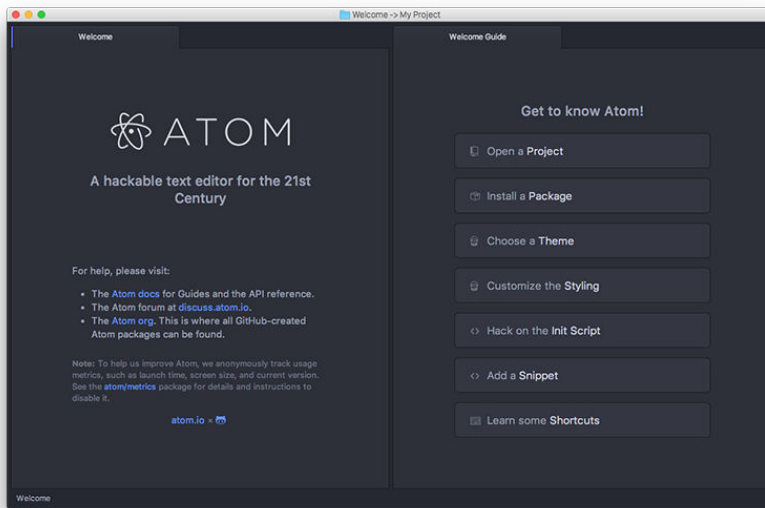
```
strict-ssl = false
```

You can run `apm config get https-proxy` to verify it has been set correctly, and running `apm config list` lists all custom config settings.

Atom Basics

Now that Atom is installed on your system, let's fire it up, configure it and get acquainted with the editor.

When you launch Atom for the first time, you should get a screen that looks like this:

**FIGURE 1-3**

Atom's welcome screen

This is the Atom welcome screen and gives you a pretty good starting point for how to get started with the editor.

Basic Terminology

First of all, let's get acquainted with some of the terminology we'll be using in this manual.

Buffer

A buffer is the text content of a file in Atom. It's basically the same as a file for most descriptions, but it's the version Atom has in memory. For instance, you can change the text of a buffer and it isn't written to its associated file until you save it.

Pane

A pane is a visual section of Atom. If you look at the welcome screen we just launched, you can see four Panes - the tab bar, the gutter (which has line numbers in it), the status bar at the bottom and finally the text editor.

Command Palette

In that welcome screen, we are introduced to probably the most important command in Atom, the “Command Palette”. If you hit `cmd-shift-P` while focused in an editor pane, the command palette will pop up.

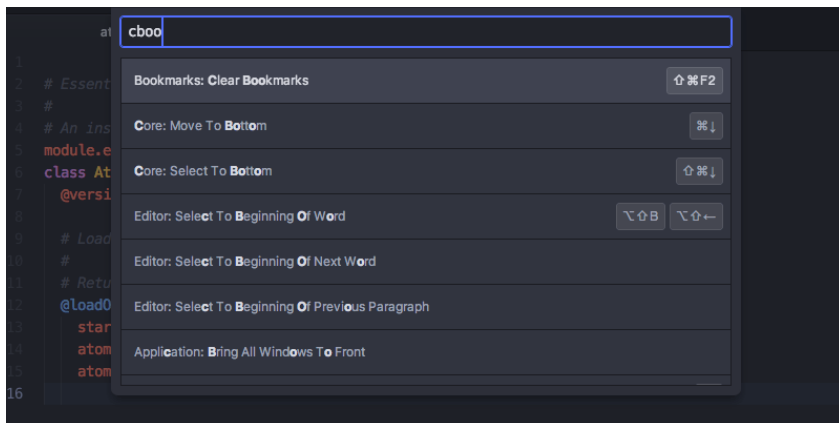
Throughout the book we will use shortcut keybindings like `cmd-shift-P` to demonstrate how to run a command. These are the default keybindings for Atom on Mac. They may occasionally be slightly different depending on your platform.

You can use the Command Palette to look up the correct keybindings if it doesn’t work for some reason.

This search-driven menu can do just about any major task that is possible in Atom. Instead of clicking around all the application menus to look for something, you can just hit `cmd-shift-P` and search for the command.

FIGURE 1-4

The Command Palette



Not only can you see and quickly search through thousands of possible commands, but you can also see if there is a keybinding associated with it. This is great because it means you can guess your way to doing interesting things while also learning the shortcut key strokes for doing it.

For the rest of the book, we will try to be clear as to the text you can search for in the Command Palette in addition to the keybinding for different commands.

Settings and Preferences

Atom has a number of settings and preferences you can modify on its Settings screen.

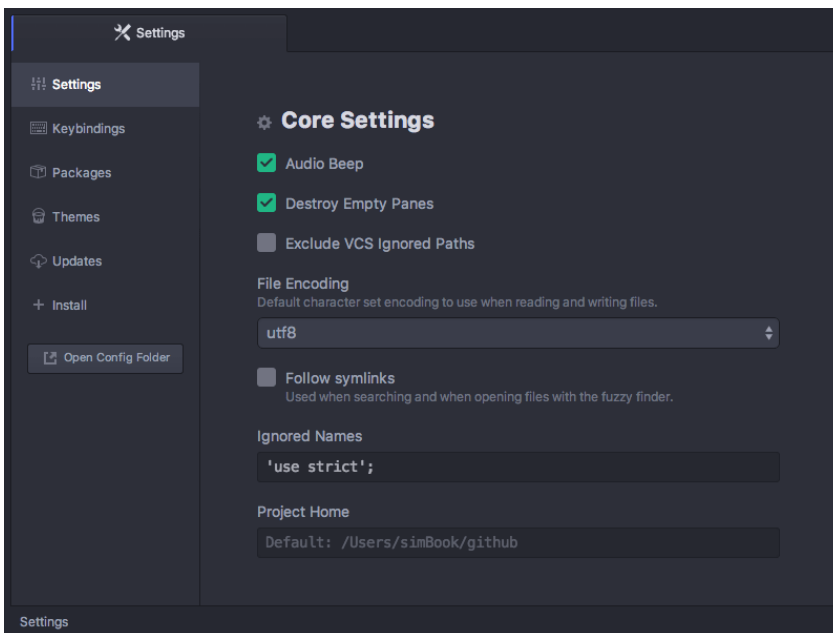


FIGURE 1-5

Atom's settings screen

This includes things like changing the color scheme or theme, specifying how to handle wrapping, font settings, tab size, scroll speed and much more. You can also use this screen to install new packages and themes, which we'll cover in **"Atom Packages"**.

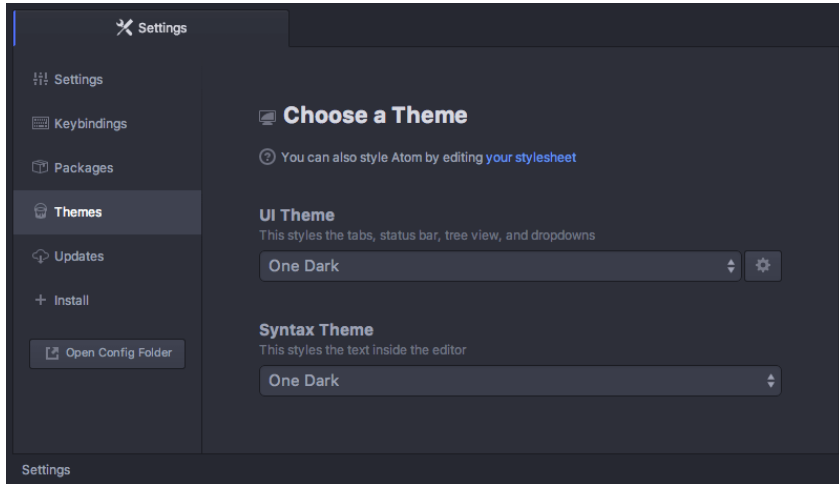
To open the Settings screen, you can go to the *Preferences* menu item under the main "Atom" menu in the menu bar. You can also search for `settings-view:open` in the command palette or use the `cmd-,` keybinding.

CHANGING THE COLOR THEME

The Settings view also lets you change the color themes for Atom. Atom ships with 4 different UI color themes, dark and light variants of the Atom and One theme, as well as 8 different syntax color themes. You can modify the active theme or install new themes by clicking on the "Themes" menu item in the sidebar of the Settings view.

FIGURE 1-6

Changing the theme from Settings

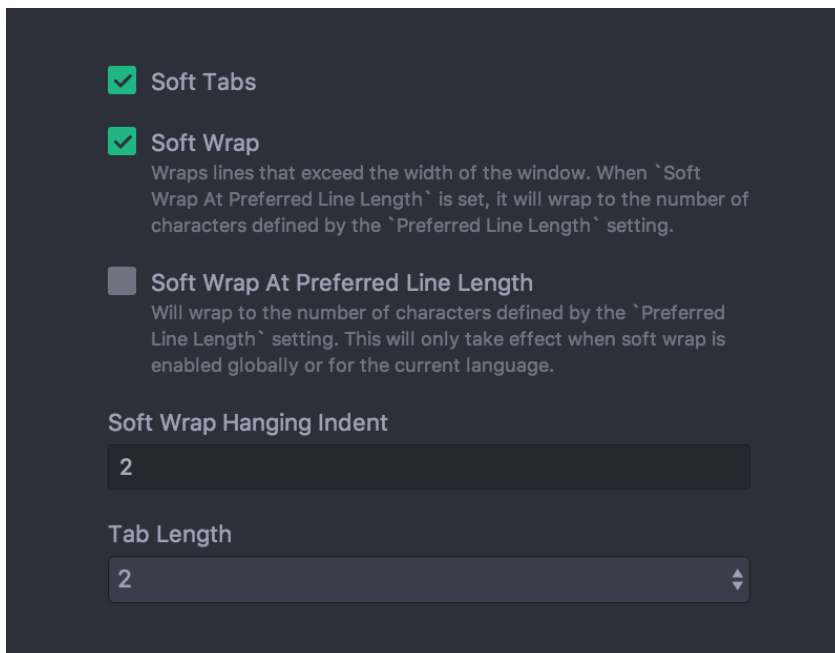


The UI themes modify the color of UI elements like the tabs and the tree view, while the syntax themes modify the syntax highlighting of text you load into the editor. To change the theme, simply pick something different in the dropdowns.

There are also dozens of themes on Atom.io that you can choose from if you want something different. We will also cover customizing a theme in “**Style Tweaks**” and creating your own theme in “**Creating a Theme**”.

SOFT WRAP

You can also use the Settings view to specify your whitespace and wrapping preferences.

**FIGURE 1-7**

Changing the theme from Settings

Enabling “Soft Tabs” will insert spaces instead of actual tab characters when you hit the tab key and the “Tab Length” specifies how many spaces to insert when you do so, or how many spaces to represent a tab as if “Soft Tabs” is disabled.

The “Soft Wrap” option will wrap lines that are too long to fit in your current window. If soft wrapping is disabled, the lines will simply run off the side of the screen and you will have to scroll the window to see the rest of the content. If “Soft Wrap At Preferred Line Length” is toggled, the lines will wrap at 80 characters instead of the end of the screen. You can also change the default line length to a value other than 80 on this screen.

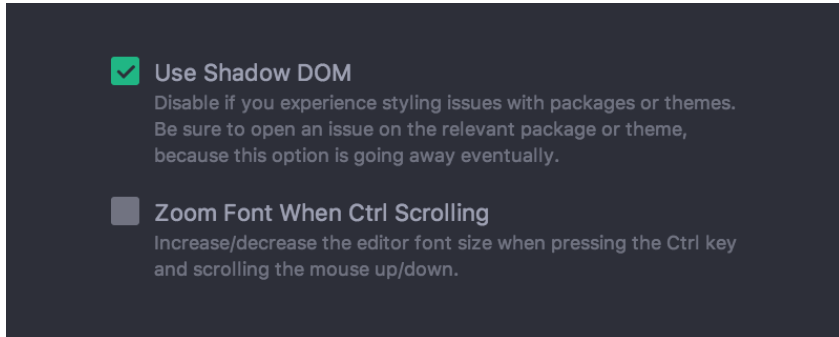
In “**Basic Customization**” we will see how to set different wrap preferences for different types of files (for example, if you want to wrap Markdown files but not code files).

BETA FEATURES

As Atom is developed, there are occasionally new features that are tested before they are mainlined for everyone. In some cases, those changes are shipped turned off by default but can be enabled in the Settings view should you wish to try them out.

FIGURE 1-8

Beta features in the Settings view



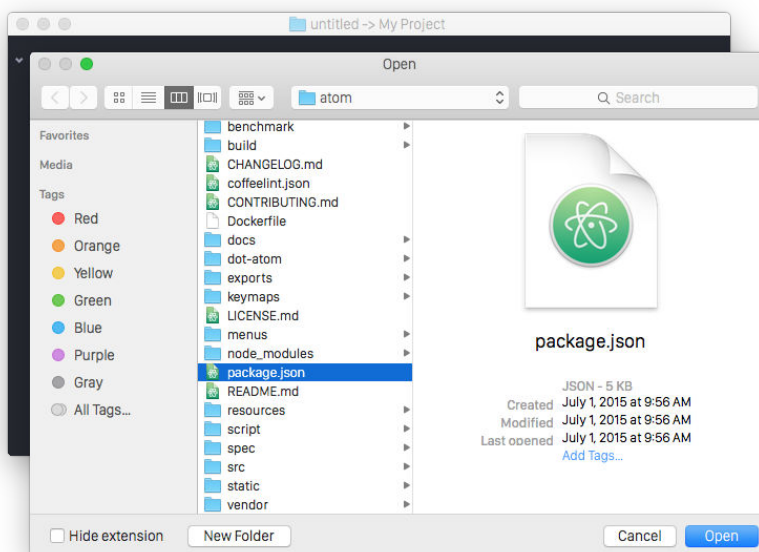
This is mainly useful for Package developers to get access to a feature or change before it ships to the general population to ensure their package still works with the new functionality. However, it may be interesting to try some of these features out occasionally if you're interested in what's coming soon.

Opening, Modifying and Saving Files

Now that your editor is looking and acting how you want, let's start opening up and editing files. This is a text editor after all, right?

OPENING A FILE

There are several ways to open a file in Atom. You can do it by choosing "File >> Open" from the menu bar or by hitting `cmd-O` to choose a file from the system dialog.

**FIGURE 1-9***Open file by dialog*

This is useful for opening a file that is not contained in the project you're currently in (more on that next), or if you're starting from a new window for some reason.

Another way to open a file in Atom is from the command line using the `atom` command. If you're on a Mac, the Atom menu bar has a command named "Install Shell Commands" which installs the `atom` and `apm` commands **if Atom wasn't able to install them itself**. On Windows and Linux, the `atom` and `apm` commands are installed automatically as a part of Atom's **installation process**.

You can run the `atom` command with one or more file paths to open up those files in Atom.

```
$ atom -h
Atom Editor v0.152.0
```

```
Usage: atom [options] [path ...]
```

One or more paths to files or folders may be specified. If there is an existing Atom window that contains all of the given folders, the paths will be opened in that window. Otherwise, they will be opened in a new window.

```
...
```

This is a great tool if you're used to the terminal or you work from the terminal a lot. Just fire off `atom [files]` and you're ready to start editing.

EDITING AND SAVING A FILE

Editing a file is pretty straightforward. You can click around and scroll with your mouse and type to change the content. There is no special editing mode or key commands.

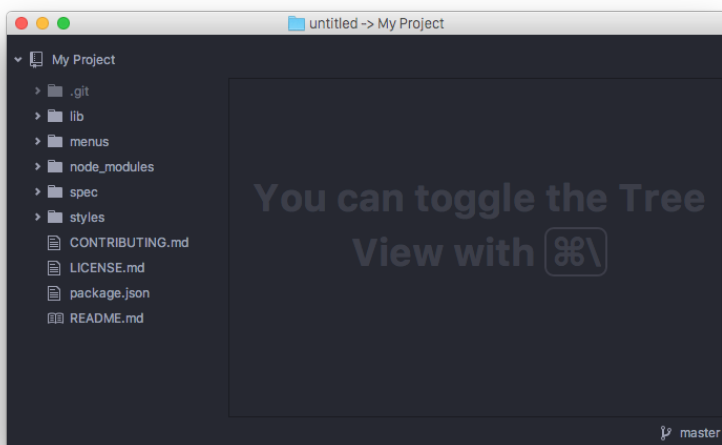
To save a file you can choose “File >> Save” from the menu bar or `cmd-s` to save the file. If you choose “Save As” or hit `cmd-shift-s` then you can save the current content in your editor under a different file name. Finally, you can choose `cmd-alt-s` to save all the open files in Atom.

Opening Directories

Atom doesn't just work with single files though; you will most likely spend most of your time working on projects with multiple files. To open a directory, choose “File >> Open” from the menu bar and select a directory from the dialog. You can also add more than one directory to your current Atom window, by choosing “File >> Add Project Folder...” from the menu bar or hitting `cmd-shift-O`.

You can open any number of directories from the command line by passing their paths to the `atom` command line tool. For example, you could run the command `atom ./hopes ./dreams` to open both the `hopes` and the `dreams` directories at the same time.

When you open Atom with one or more directories, you will automatically get a Tree view on the side of your window.

FIGURE 1-10*Tree view in an open project*

The Tree view allows you to explore and modify the file and directory structure of your project. You can open, rename, delete and create new files from this view.

You can also hide and show it with `cmd-\` or the `tree-view:toggle` command from the Palette, and `ctrl-0` will focus it. When the Tree view has focus you can press `a`, `m`, or `delete` to add, move or delete files and folders. You can also simply right-click on a file or folder in the Tree view to see many of the various options, including all of these plus showing the file in your native filesystem or copying the file path to your system clipboard.

ATOM MODULES

Like many parts of Atom, the Tree view is not built directly into the editor, but is its own standalone package that is simply shipped with Atom by default.

You can find the source code to the Tree view here: <https://github.com/atom/tree-view>

This is one of the interesting things about Atom. Many of its core features are actually just packages implemented the same way you would implement any other functionality. This means that if you don't like the Tree view for example, it's fairly simple to write your own implementation of that functionality and replace it entirely.

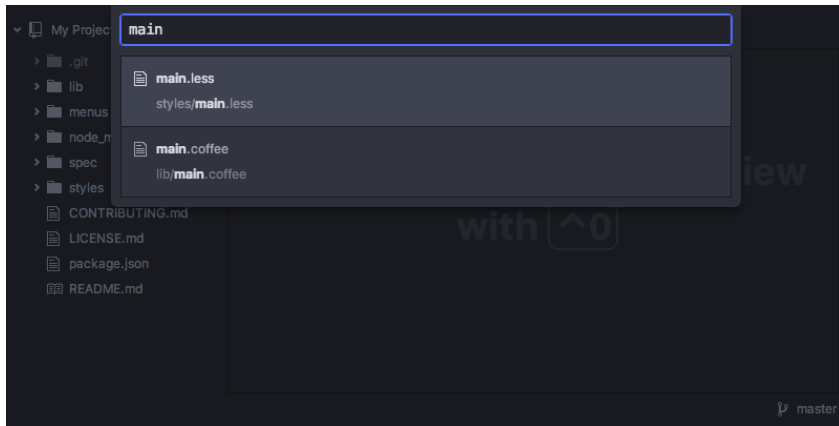
OPENING A FILE IN A PROJECT

Once you have a project open in Atom, you can easily find and open any file within that project.

If you hit either `cmd-T` or `cmd-P`, the Fuzzy Finder dialog will pop up. This will let you quickly search for any file in any directory your project by typing parts of the path.

FIGURE 1-11

Opening files with the Fuzzy Finder



You can also search through only the files currently opened (rather than every file in your project) with `cmd-B`. This searches through your “buffers” or open files. You can also limit this fuzzy search with `cmd-shift-B`, which searches only through the files which are new or have been modified since your last Git commit.

The fuzzy finder uses both the `core.ignoredNames` and `fuzzy-finder.ignoredNames` config settings to filter out files and folders that will not be shown. If you have a project with tons of files you don’t want it to search through, you can add patterns or paths to either of these config settings. We’ll learn more about config settings in “**Global Configuration Settings**”, but for now you can easily set these in the Settings view under Core Settings.

Both of those config settings are interpreted as glob patterns as implemented by the `minimatch` Node.js library.

You can read more about `minimatch` here: <https://github.com/isaacs/minimatch>

This package will also not show Git ignored files when the `core.excludeVcsIgnoredPaths` is enabled. You can easily toggle this in the Settings view, it’s one of the top options.

Summary

You should now have a basic understanding of what Atom is and what you want to do with it. You should also have it installed on your system and be able to use it for the most basic text editing operations.

Now you're ready to start digging into the fun stuff.

Using Atom 2

Now that we've covered the very basics of Atom, we are ready to see how to really start getting the most out of using it. In this chapter we'll look at how to find and install new packages in order to add new functionality, how to find and install new themes, how to work with and manipulate text in a more advanced way, how to customize the editor in any way you wish, how to work with Git for version control and more.

At the end of this chapter, you should have a customized environment that you're comfortable in and you should be able to change, create and move through your text like a master.

Atom Packages

First we'll start with the Atom package system. As we mentioned previously, Atom itself is a very basic core of functionality that ships with a number of useful packages that add new features like the **Tree View** and the **Settings View**.

In fact, there are more than 70 packages that comprise all of the functionality that is available in Atom by default. For example, the **Welcome dialog** that you see when you first start Atom, the **spell checker**, the **themes** and the **fuzzy finder** are all packages that are separately maintained and all use the same APIs that you have access to, as we'll see in great detail in **Chapter 3**.

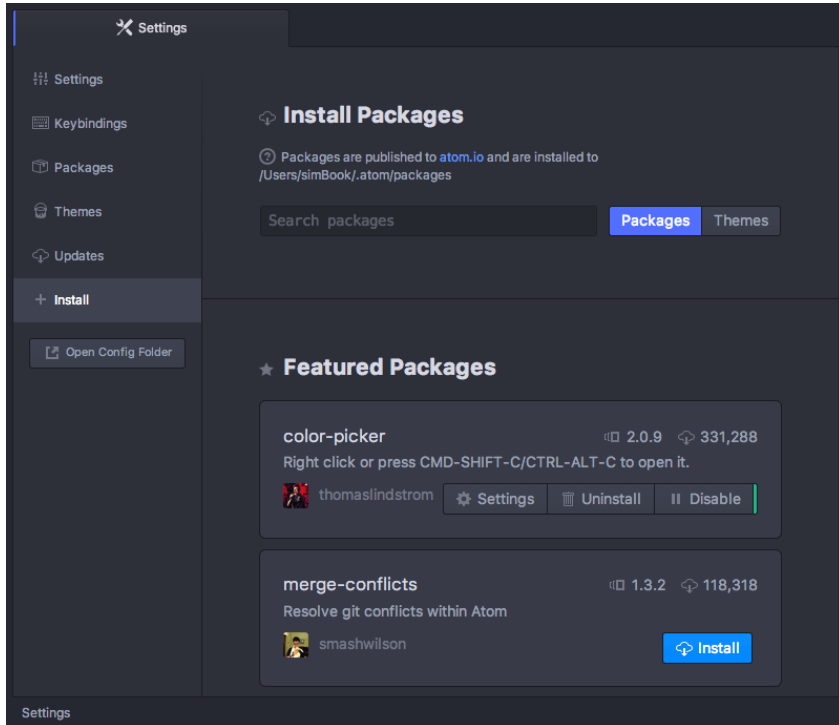
This means that packages can be incredibly powerful and can change everything from the very look and feel of the entire interface to the basic operation of even core functionality.

In order to install a new package, you can use the Install tab in the now familiar Settings view. Simply open up the Settings view (`cmd - ,`), click on the "Install" tab and type your search query into the box under Install Packages that hints "Search Packages".

The packages listed here have been published to **atom.io** which is the official registry for Atom packages. Searching on the settings pane will go to the **atom.io** package registry and pull in anything that matches your search terms.

FIGURE 2-1

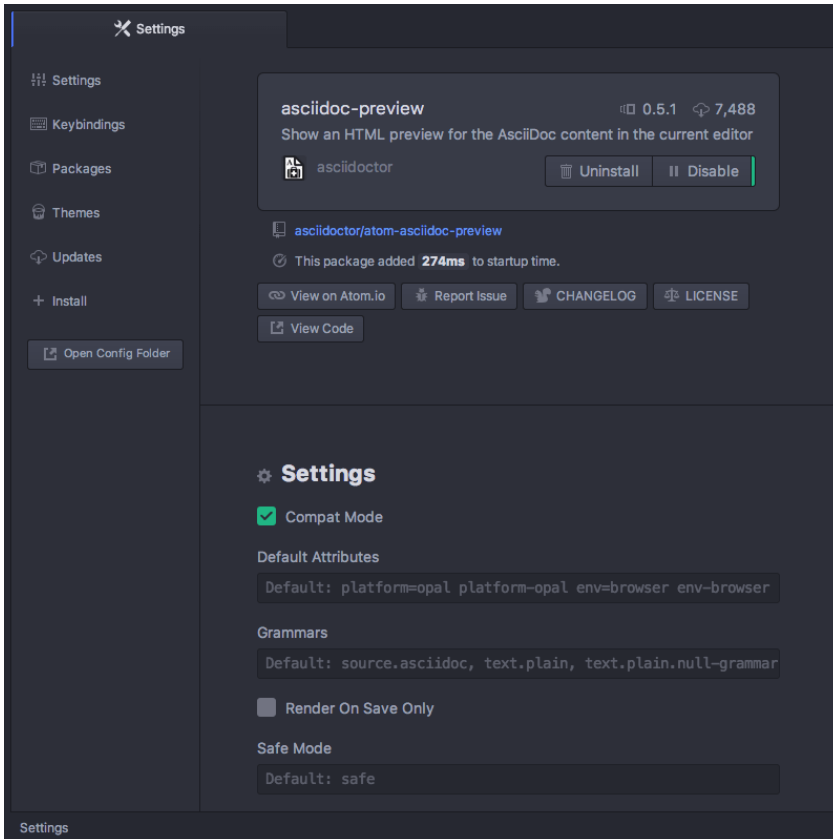
Package install screen



All of the packages will come up with an “Install” button. Clicking that will download the package and install it. Your editor will now have the functionality that the package provides.

Package Settings

Once a package is installed in Atom, it will show up in the side pane under the “Packages” tab, along with all the preinstalled packages that come with Atom. To filter the list in order to find one, you can type into the “Filter packages by name” textbox.

**FIGURE 2-2**

Package settings screen

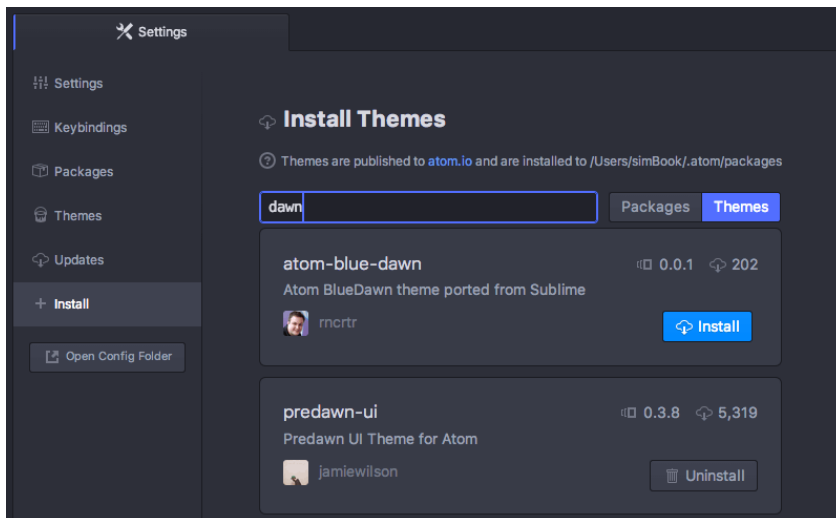
Clicking on the “Settings” button for a package will give you the settings screen for that package specifically. Here you have the option of changing some of the default variables for the package, seeing what all the command keybindings are, disabling the package temporarily, looking at the source code, seeing the current version of the package, reporting issues and uninstalling the package.

If a new version of any of your packages is released, Atom will automatically detect it and you can upgrade the package from either this screen or from the “Updates” tab. This helps you easily keep all your installed packages up to date.

Atom Themes

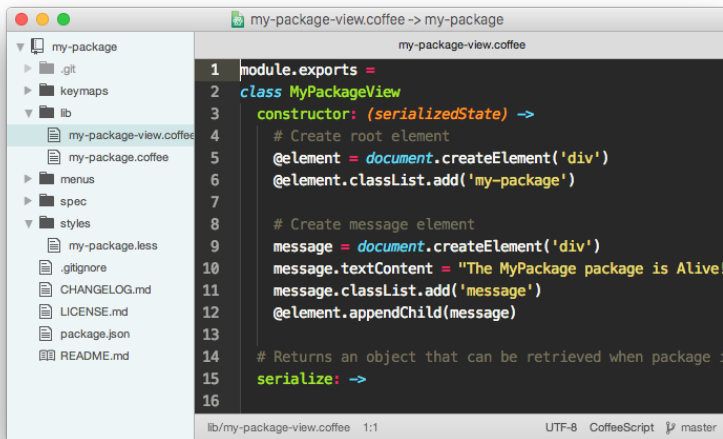
You can also find and install new themes for Atom from the Settings view. These can be either UI themes or syntax highlighting themes and you can search for

them from the “Install” tab, just like searching for new packages. Make sure to press the “Themes” toggle next to the search box.

FIGURE 2-3*Theme search screen*

Clicking on the theme title will take you to a profile page for the theme on atom.io, which usually has a screenshot of the theme. This way you can see what it looks like before installing it.

Clicking on “Install” will install the theme and make it available in the Theme dropdowns as we saw in “**Changing the Color Theme**”.

**FIGURE 2-4**

*Example of the Unity
UI theme with
Monokai syntax
theme*

Command Line

You can also install packages or themes from the command line using `apm`.

Check that you have `apm` installed by running the following command in your terminal:

```
$ apm help install
```

You should see a message print out with details about the `apm install` command.

If you do not, launch Atom and run the *Atom > Install Shell Commands* menu to install the `apm` and `atom` commands.

You can also install packages by using the `apm install` command:

- `apm install <package_name>` to install the latest version.
- `apm install <package_name>@<package_version>` to install a specific version.

For example `apm install emmet@0.1.5` installs the 0.1.5 release of the **Emmet** package.

You can also use `apm` to find new packages to install. If you run `apm search`, you can search the package registry for a search term.

```
$ apm search coffee
Search Results For 'coffee' (5)
└─ coffee-trace Add smart trace statements to coffee files with one keypress each.
└─ coffee-navigator Code navigation panel for Coffee Script (557 downloads, 8 stars)
└─ atom-compile-coffee This Atom.io Package compiles .coffee Files on save to .js
└─ coffee-lint CoffeeScript linter (3336 downloads, 18 stars)
└─ git-grep `git grep` in atom editor (1224 downloads, 9 stars)
```

You can use `apm view` to see more information about a specific package.

```
$ apm view git-grep
git-grep
└─ 0.7.0
└─ git://github.com/mizchi/atom-git-grep
└─ `git grep` in atom editor
└─ 1224 downloads
└─ 9 stars
```

Run ``apm install git-grep`` to install this package.

Moving in Atom

While it's pretty easy to move around Atom by clicking with the mouse or using the arrow keys, there are some keybindings that may help you keep your hands on the keyboard and navigate around a little faster.

First of all, Atom ships with many of the basic Emacs keybindings for navigating a document. To go up and down a single character, you can use `ctrl-P` and `ctrl-N`. To go left and right a single character, you can use `ctrl-B` and `ctrl-F`. These are the equivalent of using the arrow keys, though some people prefer not having to move their hands to where the arrow keys are located on their keyboard.

In addition to single character movement, there are a number of other movement keybindings.

`alt-B, alt-left`

Move to beginning of word

`alt-F, alt-right`

Move to end of word

`cmd-right, ctrl-E`

Move to end of line

`cmd-left, ctrl-A`

Move to first character of line

`cmd-up`

Move to top of file

`cmd-down`

Move to bottom of file

You can also move directly to a specific line (and column) number with `ctrl-G`. This will bring up a dialog that asks which line you would like to jump to. You can also use the `row:column` syntax to jump to a character in that line as well.

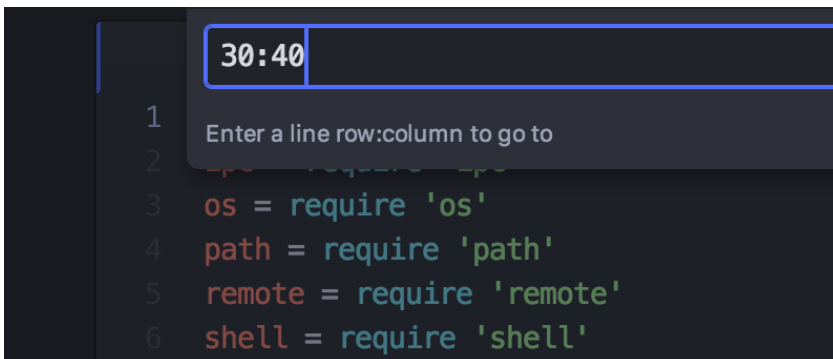


FIGURE 2-5

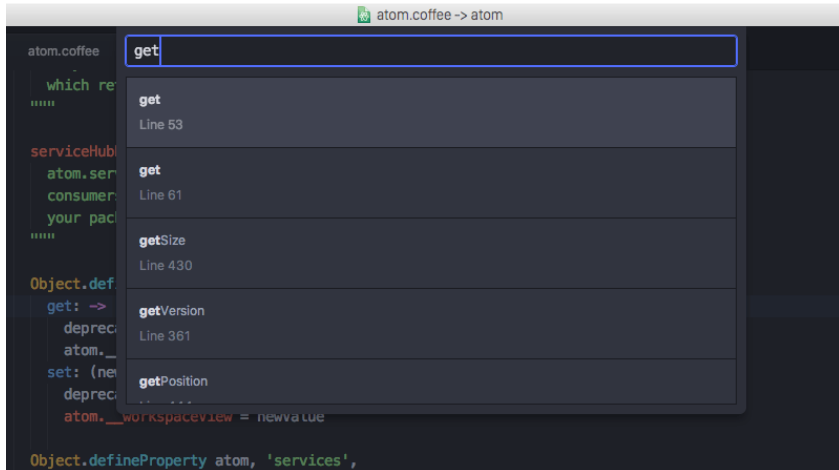
Go directly to a line

Navigating by Symbols

You can also jump around a little more informatively. To jump to a symbol such as a method definition, press `cmd-r`. This opens a list of all symbols in the current file, which you can fuzzy filter similarly to `cmd-t`. To search for symbols across your project, use `cmd-shift-r`.

FIGURE 2-6

*Search by symbol
across your project*



You can also use `ctrl-alt-down` to jump directly to the declaration of the method or function under the cursor.

First you'll need to make sure you have a **tags** (or **TAGS**) file generated for your project. This can be done by installing **ctags** and running a command such as `ctags -R src/` from the command line in your project's root directory.

If you're on a Mac using **Homebrew**, you can just run `brew install ctags`.

You can customize how tags are generated by creating your own `.ctags` file in your home directory (`~/ .ctags`). An example can be found [here](#).

The symbols navigation functionality is implemented in the **atom/symbols-view** package.

Atom Bookmarks

Atom also has a great way to bookmark specific lines in your project so you can jump back to them quickly.

If you press `cmd-F2`, Atom will toggle a “bookmark” on the current line. You can set these throughout your project and use them to quickly find and jump to important lines of your project. A small bookmark symbol is added to the line gutter, like on line 22 of **Figure 2-7**.

If you hit `F2`, Atom will jump to the next bookmark in the file you currently have focused. If you use `shift-F2` it will cycle backwards through them instead.

You can also see a list of all your project's current bookmarks and quickly filter them and jump to any of them by hitting `ctrl-F2`.

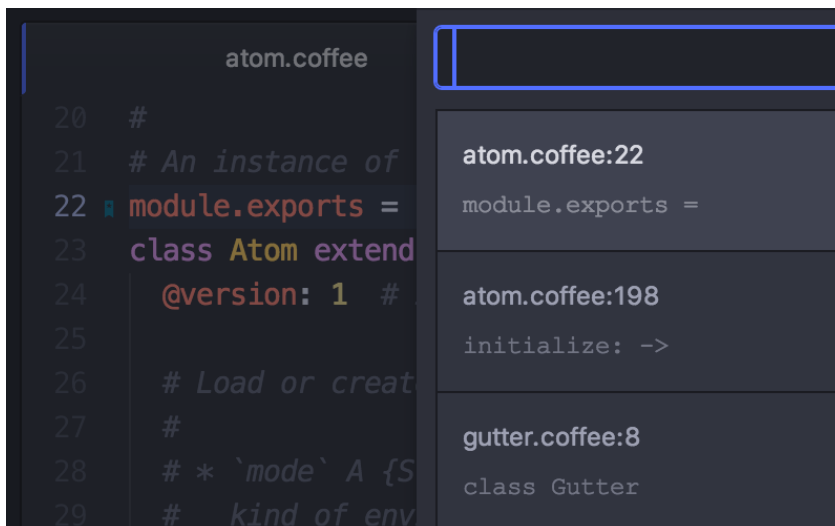


FIGURE 2-7

View and filter bookmarks.

The bookmarks functionality is implemented in the **atom/bookmarks** package.

Atom Selections

Text selections in Atom support a number of actions, such as scoping deletion, indentation and search actions, and marking text for actions such as quoting and bracketing.

Selections mirror many of the movement commands. They're actually exactly the same keybindings as the movement commands, but with a `shift` key added in.

`ctrl-shift-P`

Select up

`ctrl-shift-N`

Select down

`ctrl-shift-B`

Select previous character

`ctrl-shift-F`

Select next character

`alt-shift-B, alt-shift-left`

Select to beginning of word

`alt-shift-F, alt-shift-right`

Select to end of word

`ctrl-shift-E, cmd-shift-right`

Select to end of line

`ctrl-shift-A, cmd-shift-left`

Select to first character of line

`cmd-shift-up`

Select to top of file

`cmd-shift-down`

Select to bottom of file

In addition to the cursor movement selection commands, there are also a few commands that help with selecting specific areas of content.

`cmd-A`

Select the entire buffer

`cmd-L`

Select entire line

`ctrl-shift-W`

Select current word

Editing and Deleting Text

So far we've looked at a number of ways to move around and select regions of a file, so now let's actually change some of that text. Obviously you can type in order to insert characters, but there are also a number of ways to delete and manipulate text that could come in handy.

Basic Manipulation

There are a handful of cool keybindings for basic text manipulation that might come in handy. These range from moving around lines of text and duplicating lines to changing the case.

`ctrl-T`

Transpose characters. This swaps the two characters on either side of the cursor.

`cmd-J`

Join the next line to the end of the current line

`ctrl-cmd-up`, `ctrl-cmd-down`

Move the current line up or down

`cmd-shift-D`

Duplicate the current line

`cmd-K`, `cmd-U`

Upper case the current word

`cmd-K`, `cmd-L`

Lower case the current word

Atom also has built in functionality to re-flow a paragraph to hard-wrap at a given maximum line length. You can format the current selection to have lines no longer than 80 (or whatever number `editor.preferredLineLength` is set to) characters using `cmd-alt-Q`. If nothing is selected, the current paragraph will be reflowed.

Deleting and Cutting

You can also delete or cut text out of your buffer with some shortcuts. Be ruthless.

`ctrl-shift-K`

Delete current line

`cmd-delete`

Delete to end of line (`cmd-fn-backspace` on mac)

`ctrl-K`

Cut to end of line

`cmd-backspace`

Delete to beginning of line

`alt-backspace`, `alt-H`

Delete to beginning of word

`alt-delete`, `alt-D`

Delete to end of word

Multiple Cursors and Selections

One of the cool things that Atom can do out of the box is support multiple cursors. This can be incredibly helpful in manipulating long lists of text.

`cmd-click`

Add new cursor

`cmd-shift-L`

Convert a multi-line selection into multiple cursors

`ctrl-shift-up`, `ctrl-shift-down`

Add another cursor above/below the current cursor

`cmd-D`

Select the next word in the document that is the same as the currently selected word

`ctrl-cmd-G`

Select all words in a document that are the same as the one under the current cursor(s)

Using these commands you can place cursors in multiple places in your document and effectively execute the same commands in multiple places at once.


```
525 # * `width` The window's width {Number}.
526 # * `height` The window's height {Number}.
527 getWindowDimensions: ->
528   browserWindow = @getCurrentWindow()
529   [x, y] = browserWindow.getPosition()
530   [width, height] = browserWindow.getSize()
531   maximized = browserWindow.isMaximized()
532   {x, y, width, height, maximized}
533
```

FIGURE 2-8

*Using multiple
cursors*

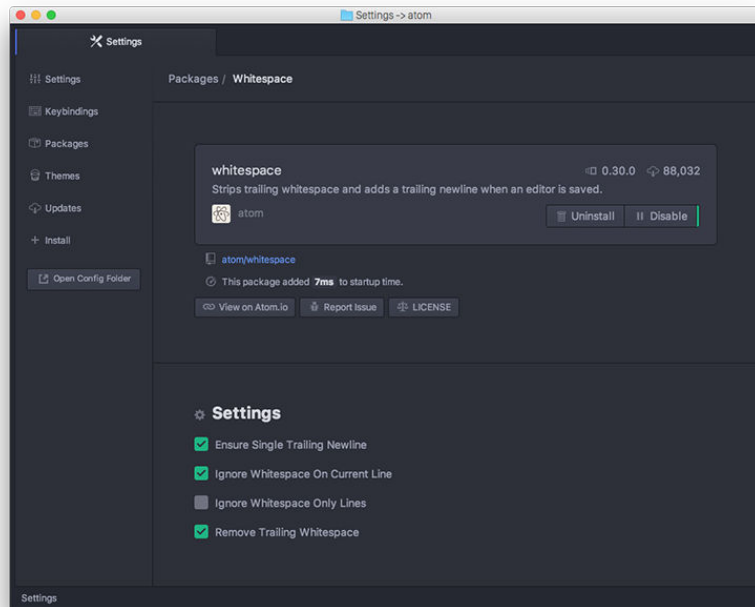
This can be incredibly helpful in doing many type of repetitive tasks such as renaming variables or changing the format of some text. You can use this with almost any plugin or command - for example, changing case and moving or duplicating lines.

You can also use the mouse to select text with the command key pressed down to select multiple regions of your text simultaneously.

Whitespace

Atom comes with several commands to help you manage the whitespace in your document. One very useful pair of commands converts leading spaces into tabs and converts leading tabs into spaces. If you're working with a document that has mixed whitespace, these commands are great for helping to normalize the file. There are no keybindings for the whitespace commands, so you will have to search your command palette for "Convert Spaces to Tabs" (or vice versa) to run one of these commands.

The whitespace commands are implemented in the **atom/whitespace** package. The settings for the whitespace commands are managed on the page for the **whitespace** package.

FIGURE 2-9*Managing your
whitespace settings*

Note that the “Remove Trailing Whitespace” option is on by default. This means that every time you save any file opened in Atom, it will strip all trailing whitespace from the file. If you want to disable this, go to the `whitespace` package in your settings panel and uncheck that option.

Atom will also by default ensure that your file has a trailing newline. You can also disable this option on that screen.

Brackets

Atom ships with intelligent and easy to use bracket handling.

It will by default highlight `[]`, `()`, and `{ }` style brackets when your cursor is over them. It will also highlight matching XML and HTML tags.

Atom will also automatically autocomplete `[]`, `()`, and `{ }`, `“”`, `‘’`, `“”`, `‘’`, `«»`, `◊`, and backticks when you type the leading one. If you have a selection and you type any of these opening brackets or quotes, Atom will enclose the selection with the opening and closing brackets or quotes.

There are a few other interesting bracket related commands that you can use.

ctrl-m

Jump to the bracket matching the one adjacent to the cursor. It jumps to the nearest enclosing bracket when there's no adjacent bracket.

ctrl-cmd-m

Select all the text inside the current brackets

alt-cmd-.

Close the current XML/HTML tag

The brackets functionality is implemented in the **atom/bracket-matcher** package. Like all of these packages, to change defaults related to bracket handling, or to disable it entirely, you can navigate to this package in the Settings view.

Encoding

Atom also ships with some basic file encoding support should you find yourself working with non-UTF-8 encoded files, or should you wish to create one.

ctrl-shift-U

Toggle menu to change file encoding

If you pull up the file encoding dialog, you can choose an alternate file encoding to save your file in.

When you open a file, Atom will try to auto-detect the encoding. If Atom can't identify the encoding, the encoding will default to UTF-8, which is also the default encoding for new files.

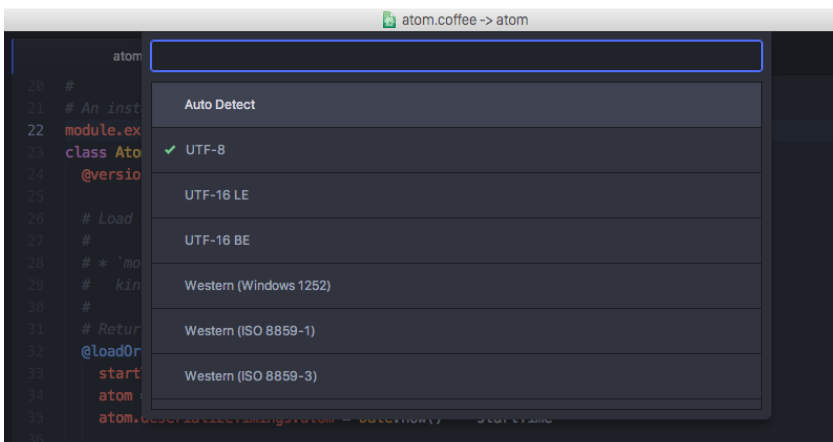


FIGURE 2-10

Changing your file encoding

If you pull up the encoding menu and change the active encoding to something else, the file will be written out in that encoding the next time you save the file.

The encoding selector is implemented in the **atom/encoding-selector** package.

Find and Replace

Finding and replacing text in your file or project is quick and easy in Atom.

cmd - F

Search within a buffer

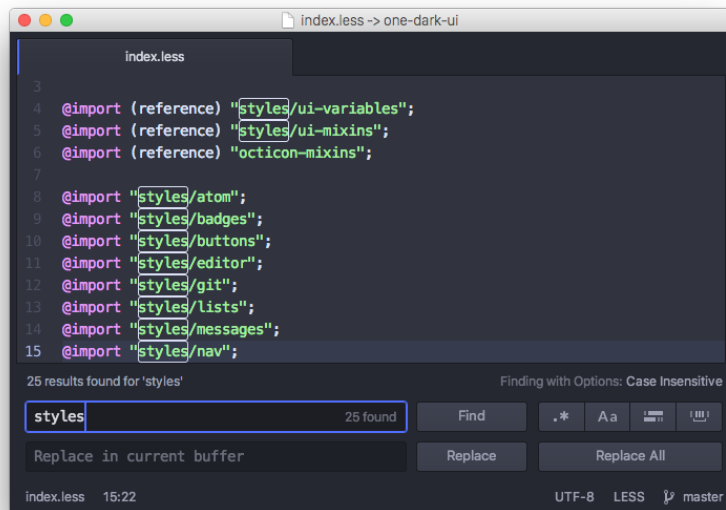
cmd - shift - f

Search the entire project

If you launch either of those commands, you'll be greeted with the "Find and Replace" pane at the bottom of your screen.

FIGURE 2-11

Find and replace text in the current file



To search within your current file you can hit **cmd - F**, type in a search string and hit enter (or **cmd - G** or the "Find Next" button) multiple times to cycle

through all the matches in that file. The “Find and Replace” pane also contains buttons for toggling case sensitivity, performing regular expression matching and scoping selections.

If you type a string in the “Replace in current buffer” text box, you can replace matches with a different string. For example, if you wanted to replace every instance of the string “Scott” with the string “Dragon”, you would enter those values in the two text boxes and hit the “Replace All” button to perform the replacements.

You can also find and replace throughout your entire project if you invoke the panel with `cmd-shift-F`.

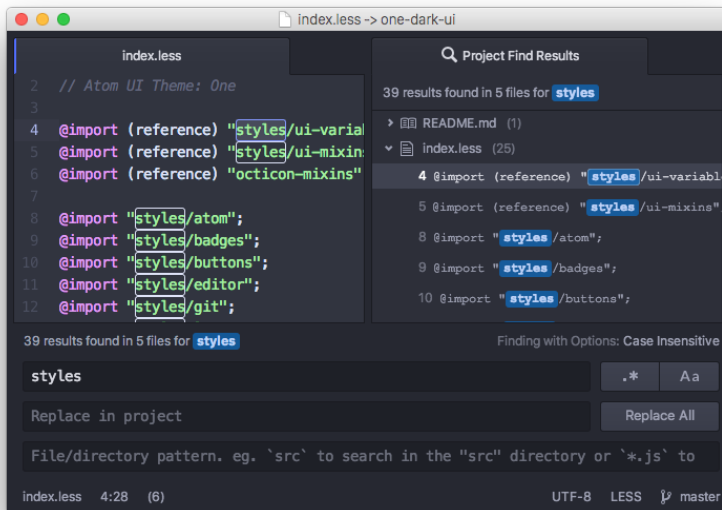


FIGURE 2-12

Find and replace text in your project

This is a great way to find out where in your project a function is called, an anchor is linked to or a specific misspelling is located. Click on the matching line to jump to that location in that file.

You can limit a search to a subset of the files in your project by entering a **glob pattern** into the “File/Directory pattern” text box. When you have multiple project folders open, this feature can also be used to search in only one of those folders. For example, if you had the folders `/path1/folder1` and `/path2/folder2` open, you could enter a pattern starting with `folder1` to search only in the first folder.

Hit escape while focused on the Find and Replace pane to clear the pane from your workspace.

The Find and Replace functionality is implemented in the **atom/find-and-replace** package and uses the **atom/scandal** package to do the actual searching.

Snippets

Snippets are an incredibly powerful way to quickly generate commonly needed code syntax from a shortcut.

The idea is that you can type something like `habtm` and then hit the `tab` key and that will expand into `has_and_belongs_to_many`.

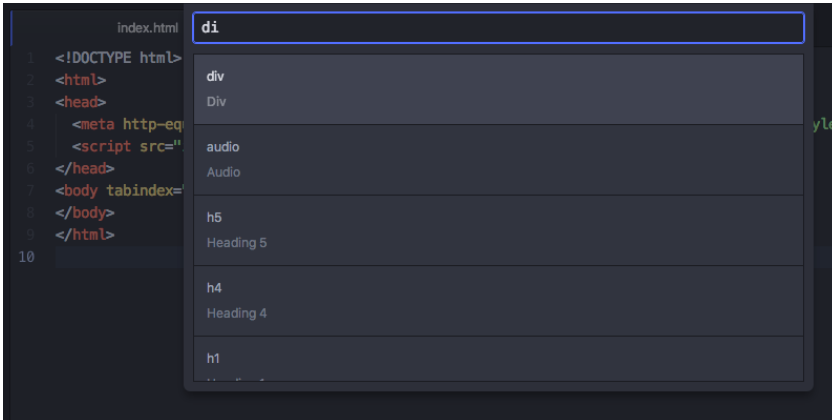
Many of the packages come bundled with their own snippets that are specific to that mode. For example, the `language-html` package that provides support for HTML syntax highlighting and grammar comes with dozens of snippets to create many of the various HTML tags you might want to use. If you create a new HTML file in Atom, you can type `html` and then hit `tab` and it will expand to:

```
<html>
  <head>
    <title></title>
  </head>
  <body>

  </body>
</html>
```

It will also position the cursor in the middle of the `title` tag so you can immediately start filling out the tag. Many snippets have multiple focus points that you can move through with the `tab` key as well - for instance, in the case of this HTML snippet, once you've filled out the title tag you can hit `tab` and the cursor will move to the middle of the body tag.

To see all the available snippets for the file type that you currently have open, you can type `alt-shift-S`.

**FIGURE 2-13**

View all available snippets

You can also use fuzzy search to filter this list down by typing in the selection box. Selecting one of them will execute the snippet where your cursor is (or multiple cursors are).

Creating Your Own Snippets

So that's pretty cool, but what if there is something the language package didn't include or something that is custom to the code you write? Luckily it's incredibly easy to add your own snippets.

There is a text file in your `~/ .atom` directory called `snippets.cson` that contains all your custom snippets that are loaded when you launch Atom. However, you can also easily open up that file by selecting the *Atom > Open Your Snippets* menu.

SNIPPET FORMAT

So let's look at how to write a snippet. The basic snippet format looks like this:

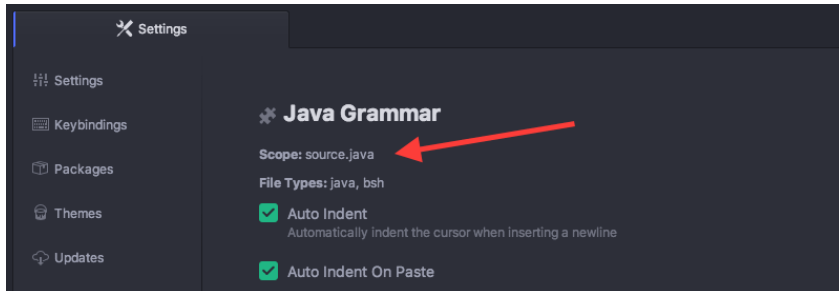
```
'source.js':
  'console.log':
    'prefix': 'log'
    'body': 'console.log(${1:"crash"});$2'
```

The outermost keys are the selectors where these snippets should be active. The easiest way to determine what this should be is to go to the language package of the language you want to add a snippet for and look for the “Scope” string.

For example, if we wanted to add a snippet that would work for Java files, we would look up the `language - java` package in our Settings view and we can see the Scope is `source.java`. Then the top level snippet key would be that prepended by a period (like a CSS class selector would do).

FIGURE 2-14

Finding the selector scope for a snippet



The next level of keys are the snippet names. These are used for describing the snippet in a more readable way in the snippet menu. It's generally best to use some sort of short human readable string here.

Under each snippet name is a prefix that should trigger the snippet and a body to insert when the snippet is triggered.

Each `$` followed by a number is a tab stop. Tab stops are cycled through by pressing `tab` once a snippet has been triggered.

The above example adds a `log` snippet to JavaScript files that would expand to:

```
console.log("crash");
```

The string `"crash"` would be initially selected and pressing `tab` again would place the cursor after the `;`

Snippet keys, unlike CSS selectors, can only be repeated once per level. If there are duplicate keys at the same level, then only the last one will be read. See [Configuring with CSON](#) for more information.

MULTI-LINE SNIPPET BODY

You can also use multi-line syntax using `"""` for larger templates:

```
'.source.js':
  'if, else if, else':
```



```
'prefix': 'ieie'
'body': """
    if (${1:true}) {
        $2
    } else if (${3:false}) {
        $4
    } else {
        $5
    }
"""
```

As you might expect, there is a snippet to create snippets. If you open up a snippets file and type `snip` and then hit `tab`, you will get the following text inserted:

```
'.source.js':
  'Snippet Name':
    'prefix': 'hello'
    'body': 'Hello World!'
```

Bam, just fill that bad boy out and you have yourself a snippet. As soon as you save the file, Atom should reload the snippets and you will immediately be able to try it out.

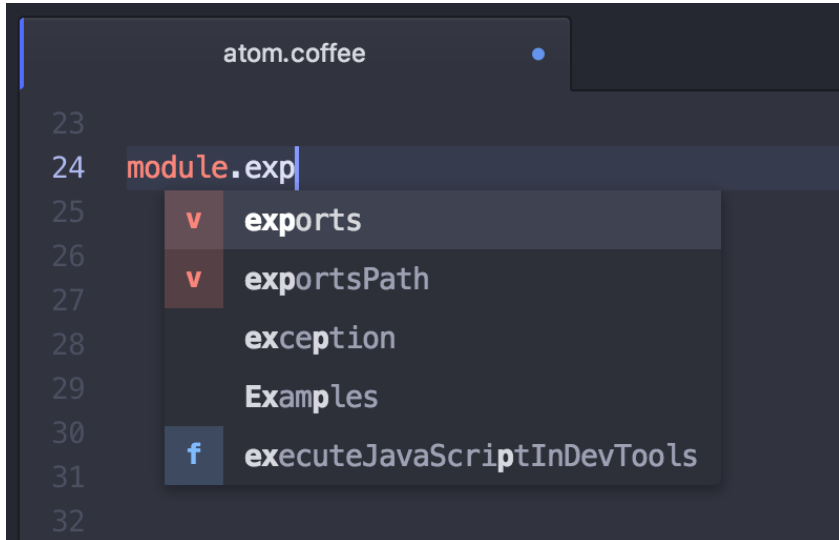
The snippets functionality is implemented in the **atom/snippets** package.

For more examples, see the snippets in the **language-html** and **language-javascript** packages.

Autocomplete

If you're still looking to save some typing time, Atom also ships with simple autocompletion functionality.

The autocompleter lets you view and insert possible completions in the editor using `ctrl-space`.

FIGURE 2-15*Autocomplete menu*

By default, the autocompleter will look through the current open file for strings that match what you're starting to type.

If you want more options, in the Settings panel for the `autocomplete-plus` package you can toggle a setting to make the autocompleter look for strings in all your open buffers rather than just the current file.

The Autocomplete functionality is implemented in the **atom/autocomplete-plus** package.

Folding

If you want to see an overview of the structure of the code file you're working on, folding can be a helpful tool. Folding hides blocks of code such as functions or looping blocks in order to simplify what is on your screen.

You can fold blocks of code by clicking the arrows that appear when you hover your mouse cursor over the gutter. You can also fold and unfold from the keyboard with the `alt-cmd-[` and `alt-cmd-]` keybindings.

```

599 # Call this method when establishing a real application with
600 > startEditorWindow: ->#
627
628 > unloadEditorWindow: ->#
639
640 > removeEditorWindow: ->#
649
650 openInitialEmptyEditorIfNecessary: ->
651     if @getLoadSettings().initialPaths?.length is 0 and @world
652         @workspace.open(null)

```

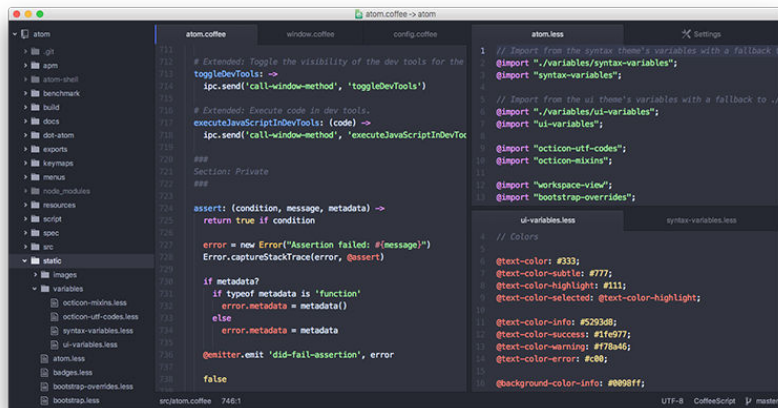
FIGURE 2-16*Code folding example*

To fold everything, use `alt-cmd-shift-{` and to unfold everything use `alt-cmd-shift-}`. You can also fold at a specific indentation level with `cmd-k cmd-N` where N is the indentation depth.

Finally, you can fold arbitrary sections of your code or text by making a selection and then hitting `ctrl-alt-cmd-F` or choosing “Fold Selection” in the Command Palette.

Panes

You can split any editor pane horizontally or vertically by using `cmd-k arrow` where the arrow is the direction to split the pane. Once you have a split pane, you can move focus between them with `cmd-k cmd-arrow` where the arrow is the direction the focus should move to.

FIGURE 2-17*Multiple panes*

Each pane has its own “items” or files, which are represented by tabs. You can move the files from pane to pane by dragging them with the mouse and dropping them in the pane you want that file to be in.

To close a pane, close all its editors with `cmd-w`, then press `cmd-w` one more time to close the pane. You can configure panes to auto-close when empty in the Settings view.

Grammar

The “grammar” of a buffer is what language Atom thinks that file content is. Types of grammars would be Java or Markdown. We looked at this a bit when we created some snippets in “**Snippets**”.

If you load up a file, Atom does a little work to try to figure out what type of file it is. Largely this is accomplished by looking at its file extension (`.md` is generally a Markdown file, etc), though sometimes it has to inspect the content a bit to figure it out if it’s ambiguous.

If you load up a file and Atom can’t determine a grammar for the file, it will default to *Plain Text*, which is the simplest one. If it does default to *Plain Text* or miscategorize a file, or if for any reason you wish to change the active grammar of a file, you can pull up the Grammar selector with `ctrl-shift-L`.

**FIGURE 2-18***Grammar selector*

Once the grammar of a file is changed manually, Atom will remember that until you set it back to auto-detect or choose a different grammar manually.

The Grammar selector functionality is implemented in the **atom/grammar-selector** package.

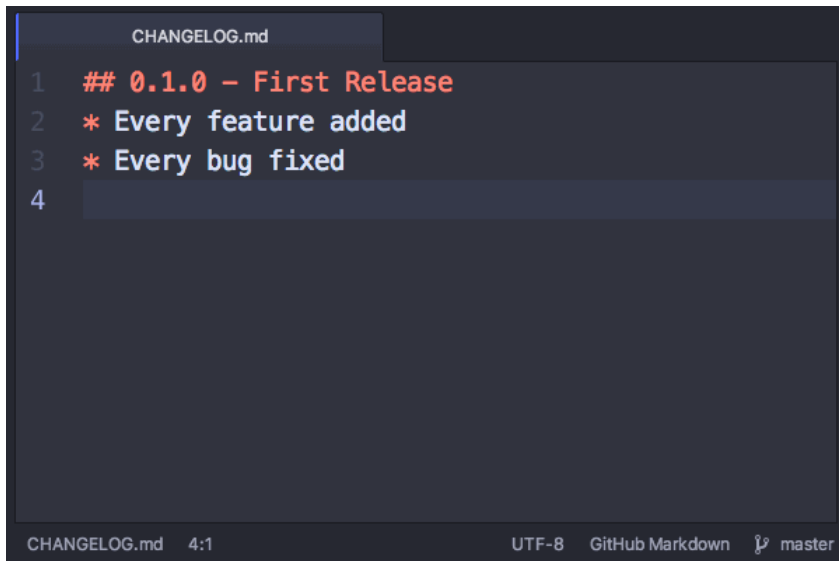
Version Control in Atom

Version control is an important aspect of any project and Atom comes with basic **Git** and **GitHub** integration baked in.

Checkout HEAD revision

The `cmd-alt-Z` keybinding checks out the HEAD revision of the file in the editor.

This is a quick way to discard any saved and staged changes you've made and restore the file to the version in the HEAD commit. This is essentially the same as running `git checkout HEAD -- <path>` and `git reset HEAD -- <path>` from the command line for that path.

FIGURE 2-19*Git checkout HEAD*

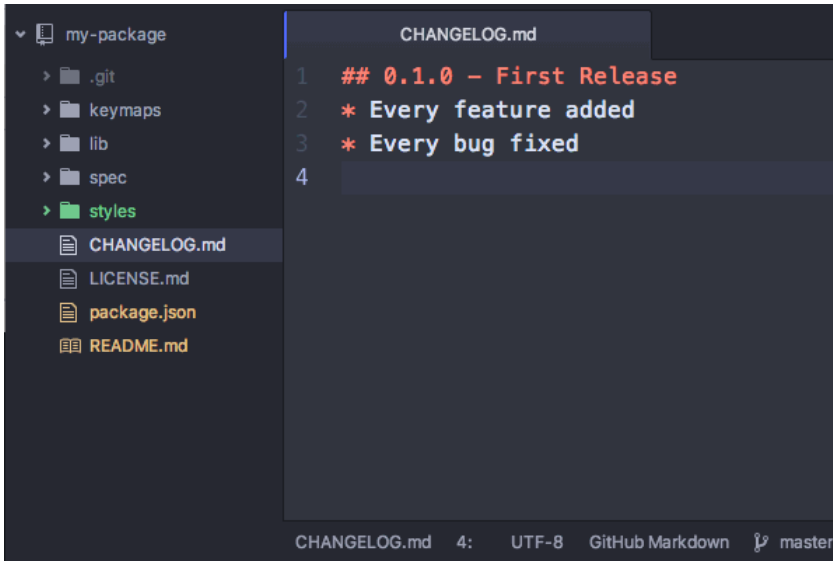
```
CHANGELOG.md
1  ## 0.1.0 - First Release
2  * Every feature added
3  * Every bug fixed
4
CHANGELOG.md 4:1 UTF-8 GitHub Markdown master
```

This command goes onto the undo stack so you can use `cmd-Z` afterwards to restore the previous contents.

Git status list

Atom ships with the fuzzy-finder package which provides `cmd-T` to quickly open files in the project and `cmd-B` to jump to any open editor.

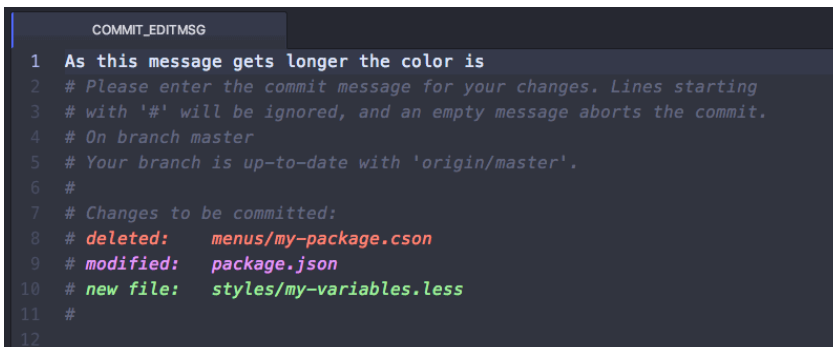
The package also comes with `cmd-shift-B` which pops up a list of all the untracked and modified files in the project. These will be the same files that you would see on the command line if you ran `git status`.

**FIGURE 2-20***Git status list*

An octicon will appear to the right of each file letting you know whether it is untracked or modified.

Commit editor

Atom can be used as your Git commit editor and ships with the `language-git` package which adds syntax highlighting to edited commit, merge, and rebase messages.

**FIGURE 2-21***Git commit message highlighting*

You can configure Atom to be your Git commit editor with the following command:

```
$ git config --global core.editor "atom --wait"
```

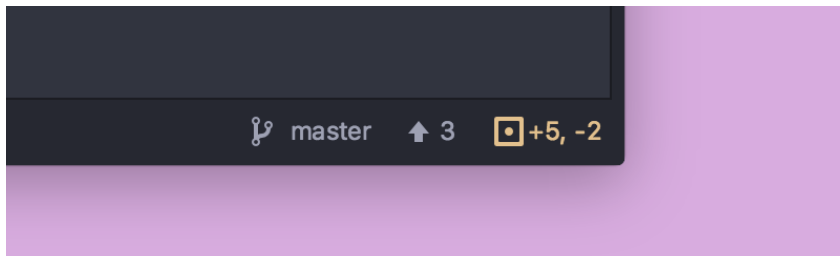
The **language-git** package will help you with your brevity by colorizing the first lines of commit messages when they're longer than 50 and 65 characters.

Status bar icons

The **status-bar** package that ships with Atom includes several Git decorations that display on the right side of the status bar.

FIGURE 2-22

Git Status Bar



The currently checked out branch name is shown with the number of commits the branch is ahead of or behind its upstream branch.

An icon is added if the file is untracked, modified, or ignored. The number of lines added and removed since the file was last committed will be displayed as well.

Line diffs

The included **git-diff** package colorizes the gutter next to lines that have been added, edited, or removed.

**FIGURE 2-23***Git line diffs*

This package also adds `alt-g down` and `alt-g up` keybindings that allow you to move the cursor to the next/previous diff hunk in the current editor.

Open on GitHub

If the project you're working on is on GitHub, there are also some very useful integrations you can use. Most of the commands will take the current file you're viewing and open a view of that file on GitHub - for instance, the blame or commit history of that file.

`alt-G O`

Open file on GitHub

`alt-G B`

Open blame of file on GitHub

`alt-G H`

Open history of file on GitHub

`alt-G C`

Copy the URL of the current file on GitHub

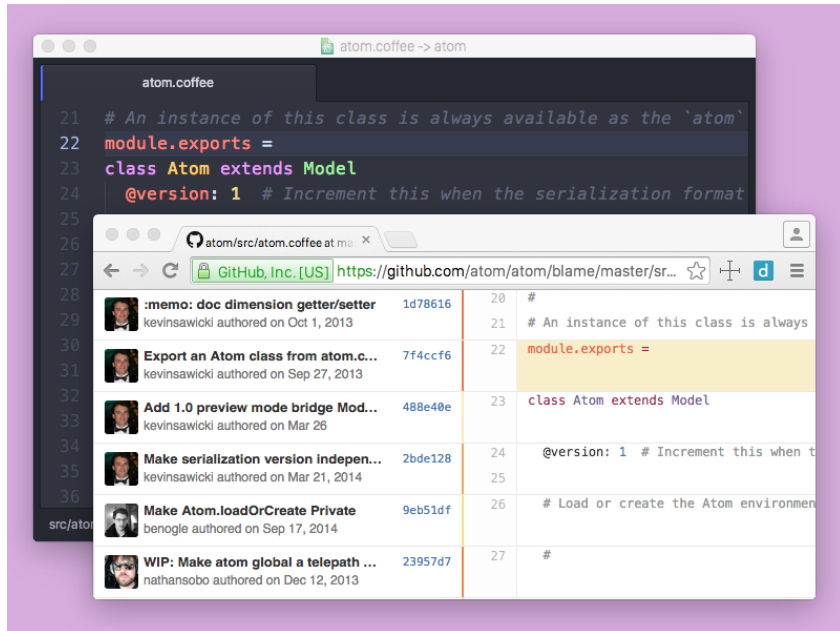
`alt-G R`

Branch compare on GitHub

The branch comparison simply shows you the commits that are on the branch you're currently working on locally that are not on the mainline branch.

FIGURE 2-24

*Open Blame of file
on GitHub*



Writing in Atom

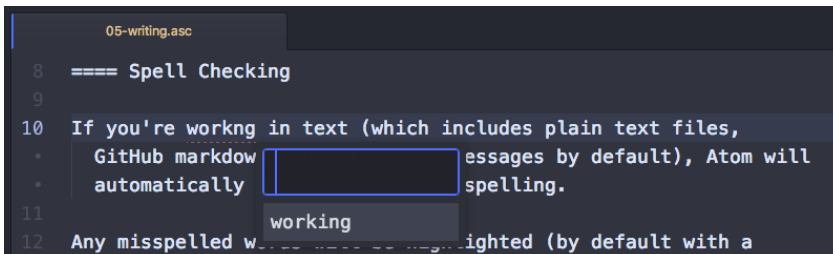
Though it is probably most common to use Atom to write software code, Atom can also be used to write prose quite effectively. Most often this is done in some sort of markup language such as Markdown or AsciiDoc (which this manual is written in). Here we'll quickly cover a few of the tools Atom provides for helping you write prose.

In these docs, we'll concentrate on writing in Markdown; however, other prose markup languages like AsciiDoc have packages that provide similar functionality.

Spell Checking

If you're working in text (which includes plain text files, GitHub markdown and Git commit messages by default), Atom will automatically try to check your spelling.

Any misspelled words will be highlighted (by default with a dashed red line beneath the word) and you can pull up a menu of possible corrections by hitting cmd-: (or by choosing "Correct Spelling" from the right-click context menu or from the Command Palette).

**FIGURE 2-25**

Checking your spelling

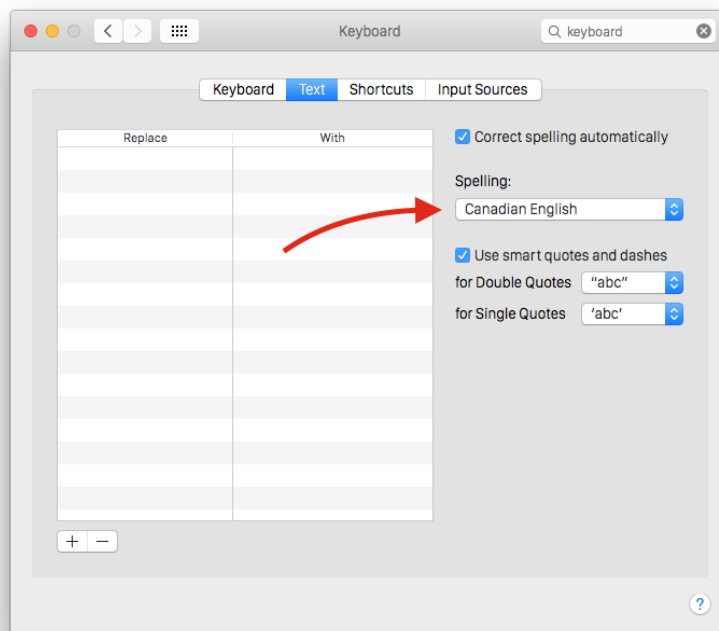
To add more types of files to the list of what Atom will try to spell check, go to the Spell Check package settings in your Settings view and add any grammars you want to spell check.

The default grammars to spell check are “text.plain, source.gfm, text.git-commit” but you can add something like “source.asciidoc” if you wish to check those types of files too.

The Atom spell checker uses the system dictionary, so if you want it to check your spelling in another language or locale, you can change it easily.

FIGURE 2-26

Changing your spell checking dictionary



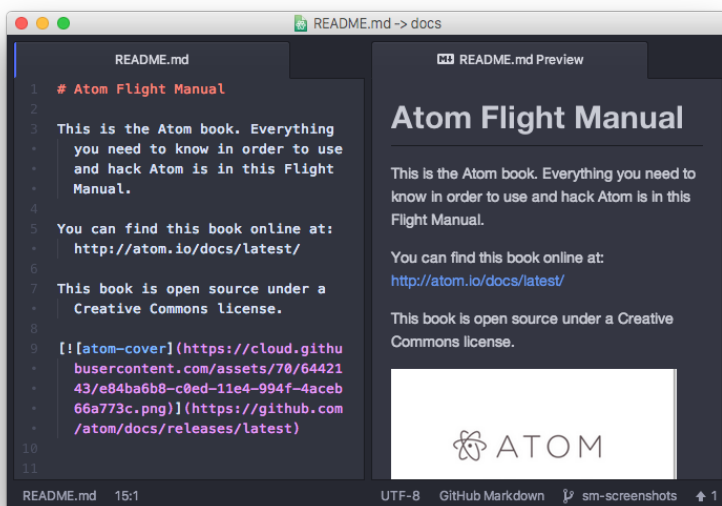
The spell checking is implemented in the **atom/spell-check** package.

Previews

When writing prose in a markup language, it's often very useful to get an idea of what the content will look like when it's rendered. Atom ships with a markdown preview plugin by default.

`ctrl-shift-M`

Will toggle Preview mode for Markdown.

FIGURE 2-27*Preview your prose*

As you edit the text, the preview will also update by default. This makes it fairly easy to check your syntax as you type.

You can also copy the rendered HTML from the preview pane into your system clipboard. There is no keybinding for it, but you can find it in the Command Palette by searching for “Markdown Preview Copy HTML”.

Markdown preview is implemented in the **atom/markdown-preview** package.

Snippets

There are also a number of great snippets available for writing Markdown quickly.

If you type `img` and hit `tab` you get a Markdown-formatted image embed code like ``. If you type `table` and hit `tab` you get a nice example table to fill out.

Header One	Header Two	
:-----	:-----	
Item One	Item Two	

Although there are only a handful of Markdown snippets (**b** for bold, *i* for italic, `code` for a code block, etc), they save you from having to look up the more obscure syntaxes. Again, you can easily see a list of all available snippets for the type of file you’re currently in by hitting *alt-shift-S*.

Basic Customization

Now that we are feeling comfortable with just about everything built into Atom, let’s look at how to tweak it. Perhaps there is a keybinding that you use a lot but feels wrong or a color that isn’t quite right for you. Atom is amazingly flexible, so let’s go over some of the simpler flexes it can do.

Configuring with CSON

All of Atom’s config files (with the exception of your **style sheet** and your **Init Script**) are written in CSON, short for CoffeeScript Object Notation.¹ Just like its namesake JSON (JavaScript Object Notation)², CSON is a text format for storing structured data in the form of simple objects made up of key-value pairs.

```
key:
  key: value
  key: value
  key: [value, value]
```

Objects are the backbone of any CSON file, and are delineated with either indentation (as in the above example), or with curly brackets (`{}`). A key’s value can either be a String, a Number, an Object, a Boolean, `null`, or an Array of any of these data types.

Unlike CSS’s selectors, CSON’s keys can only be repeated once per object. If there are duplicate keys, then the last usage of that key overwrites all others, as if they weren’t there. The same holds true for Atom’s config files.

¹ <https://github.com/bevry/cson#what-is-cson>

² <http://json.org/>

Avoid this:

```
# DON'T DO THIS
'.source.js':
  'console.log':
    'prefix': 'log'
    'body': 'console.log(${1:"crash"});$2'

# Only this snippet will be loaded
'.source.js':
  'console.error':
    'prefix': 'error'
    'body': 'console.error(${1:"crash"});$2'
```

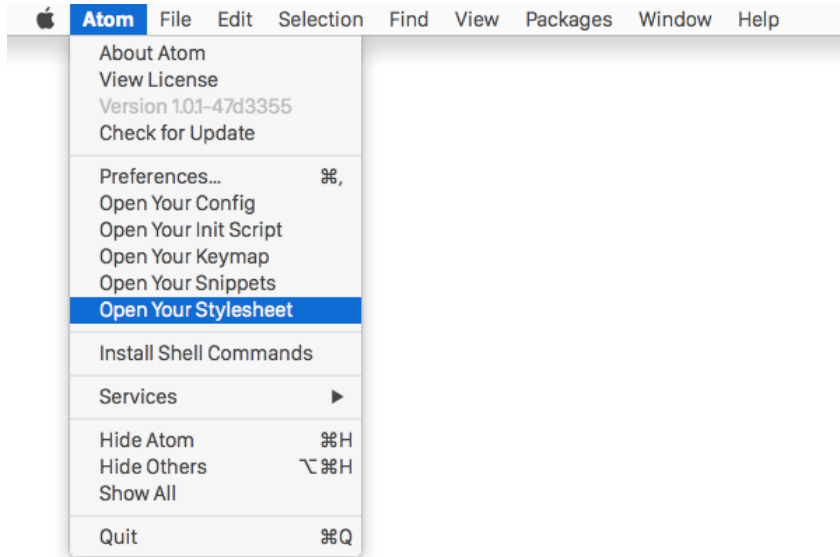
Use this instead:

```
# DO THIS: Both of these will be loaded
'.source.js':
  'console.log':
    'prefix': 'log'
    'body': 'console.log(${1:"crash"});$2'
  'console.error':
    'prefix': 'error'
    'body': 'console.error(${1:"crash"});$2'
```

Style Tweaks

If you want to apply quick-and-dirty personal styling changes without creating an entire theme that you intend to publish, you can add styles to the `styles.less` file in your `~/atom` directory.

You can open this file in an editor from the *Atom > Open Your Stylesheet* menu.

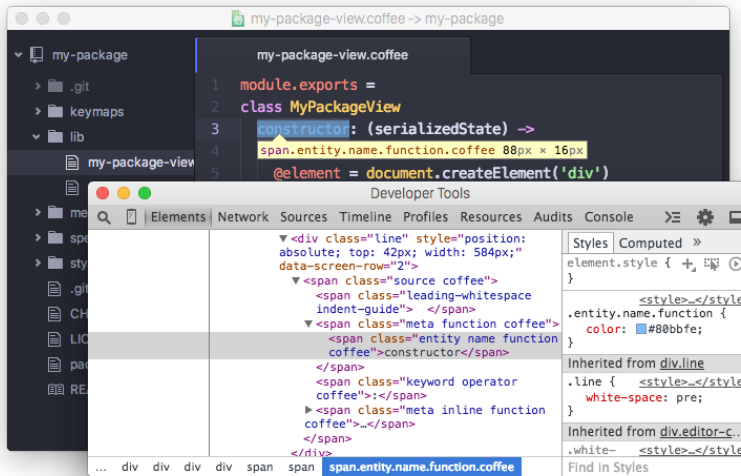
FIGURE 2-28*Open your stylesheet*

For example, to change the color of the cursor, you could add the following rule to your `~/atom/styles.less` file:

```
atom-text-editor::shadow .cursor {  
  border-color: pink;  
}
```

The easiest way to see what classes are available to style is to inspect the DOM manually via the developer tools. We'll go over the developer tools in great detail in the next chapter, but for now let's take a simple look.

You can open the Developer Tools by hitting `alt-cmd-I`, which will bring up the Chrome developer tools panel.

**FIGURE 2-29***Developer Tools*

You can now easily inspect all the elements in your current editor. If you want to update the style of something, you simply need to figure out what classes it has and add a Less rule to your styles file to modify it.

If you are unfamiliar with Less, it is a basic CSS preprocessor that makes some things in CSS a bit easier. You can learn more about it at lesscss.org. If you prefer to use CSS instead, this file can also be named `styles.css` and contain CSS.

Customizing Key Bindings

Atom keymaps work similarly to stylesheets. Just as stylesheets use selectors to apply styles to elements, Atom keymaps use selectors to associate keystrokes with events in specific contexts. Here's a small example, excerpted from Atom's built-in keymaps:

```

'atom-text-editor':
  'enter': 'editor:newline'

'atom-text-editor[mini] input':
  'enter': 'core:confirm'

```

This keymap defines the meaning of `enter` in two different contexts. In a normal editor, pressing `enter` emits the `editor:newline` event, which causes

the editor to insert a newline. But if the same keystroke occurs inside a select list’s mini-editor, it instead emits the `core:confirm` event based on the binding in the more-specific selector.

By default, `~/ .atom/keymap.cson` is loaded when Atom is started. It will always be loaded last, giving you the chance to override bindings that are defined by Atom’s core keymaps or third-party packages.

You can open this file in an editor from the *Atom > Open Your Keymap* menu.

You’ll want to know all the commands available to you. Open the Settings panel (`cmd-,`) and select the *Keybindings* tab. It will show you all the keybindings currently in use.

Global Configuration Settings

Atom loads configuration settings from the `config.cson` file in your `~/ .atom` directory, which contains CoffeeScript-style JSON: **CSON**.

```
'core':
  'excludeVcsIgnoredPaths': true
'editor':
  'fontSize': 18
```

The configuration itself is grouped by the package name or one of the two core namespaces: `core` and `editor`.

You can open this file in an editor from the *Atom > Open Your Config* menu.

CONFIGURATION KEY REFERENCE

- **core**
 - `disabledPackages`: An array of package names to disable
 - `excludeVcsIgnoredPaths`: Don’t search within files specified by *.gitignore*
 - `ignoredNames`: File names to ignore across all of Atom
 - `projectHome`: The directory where projects are assumed to be located
 - `themes`: An array of theme names to load, in cascading order
- **editor**
 - `autoIndent`: Enable/disable basic auto-indent (defaults to `true`)
 - `nonWordCharacters`: A string of non-word characters to define word boundaries

- `fontSize`: The editor font size
- `fontFamily`: The editor font family
- `invisibles`: A hash of characters Atom will use to render white-space characters. Keys are whitespace character types, values are rendered characters (use value `false` to turn off individual white-space character types)
 - `tab`: Hard tab characters
 - `cr`: Carriage return (for Microsoft-style line endings)
 - `eol`: `\n` characters
 - `space`: Leading and trailing space characters
- `preferredLineLength`: Identifies the length of a line (defaults to 80)
- `showInvisibles`: Whether to render placeholders for invisible characters (defaults to `false`)
- `showIndentGuide`: Show/hide indent indicators within the editor
- `showLineNumbers`: Show/hide line numbers within the gutter
- `softWrap`: Enable/disable soft wrapping of text within the editor
- `softWrapAtPreferredLineLength`: Enable/disable soft line wrapping at `preferredLineLength`
- `tabLength`: Number of spaces within a tab (defaults to 2)
- `fuzzyFinder`
 - `ignoredNames`: Files to ignore **only** in the fuzzy-finder
- `whitespace`
 - `ensureSingleTrailingNewline`: Whether to reduce multiple newlines to one at the end of files
 - `removeTrailingWhitespace`: Enable/disable stripping of white-space at the end of lines (defaults to `true`)
- `wrap-guide`
 - `columns`: Array of hashes with a `pattern` and `column` key to match the path of the current editor to a column position.

Language Specific Configuration Settings

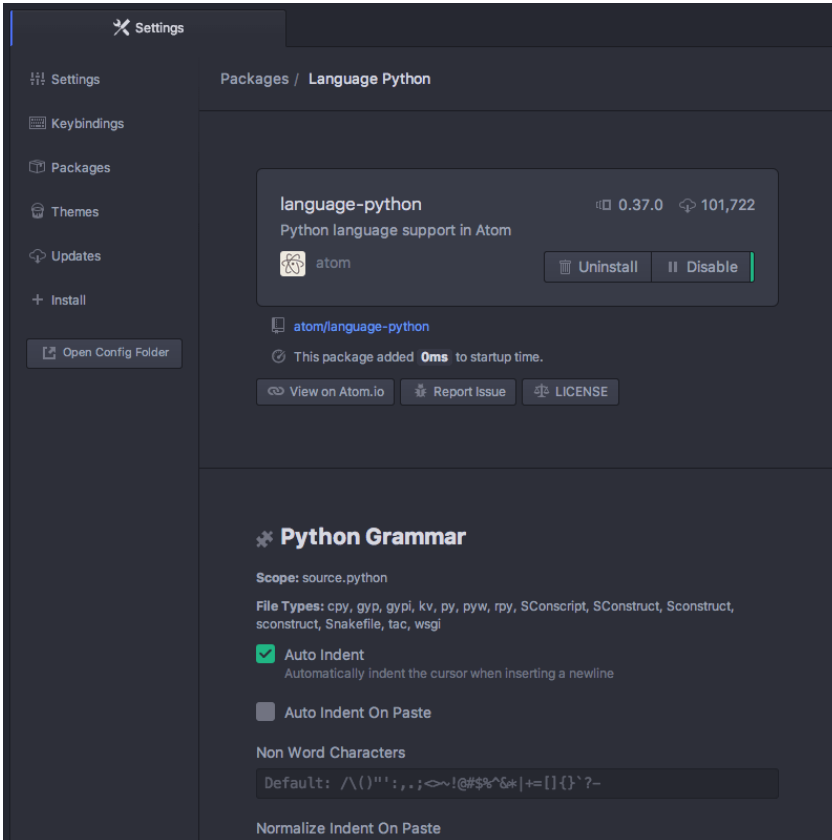
You can also set several configuration settings differently for different file types. For example, you may want Atom to soft wrap markdown files, have two-space tabs for ruby files, and four-space tabs for python files.

There are several settings now scoped to an editor's language. Here is the current list:

```
editor.tabLength  
editor.softWrap  
editor.softWrapAtPreferredLineLength  
editor.preferredLineLength  
editor.scrollPastEnd  
editor.showInvisibles  
editor.showIndentGuide  
editor.nonWordCharacters  
editor.invisibles  
editor.autoIndent  
editor.normalizeIndentOnPaste
```

LANGUAGE-SPECIFIC SETTINGS IN THE SETTINGS VIEW

You can edit these config settings in the settings view on a per-language basis. Just search for the language of your choice in the left panel, select it, and edit away!

**FIGURE 2-30**

Python specific settings

LANGUAGE-SPECIFIC SETTINGS IN YOUR CONFIG FILE

You can also edit the actual configuration file directly. Open your config file via the Command Palette, type “open config”, and hit enter.

Global settings are under a global key, and each language can have its own top-level key. This key is the language’s scope. Language-specific settings override anything set in the global section.

```
'global': # all languages unless overridden
  'editor':
    'softWrap': false
    'tabLength': 8

'.source.gfm': # markdown overrides
  'editor':
    'softWrap': true
```

```

'.source.ruby': # ruby overrides
  'editor':
    'tabLength': 2

'.source.python': # python overrides
  'editor':
    'tabLength': 4

```

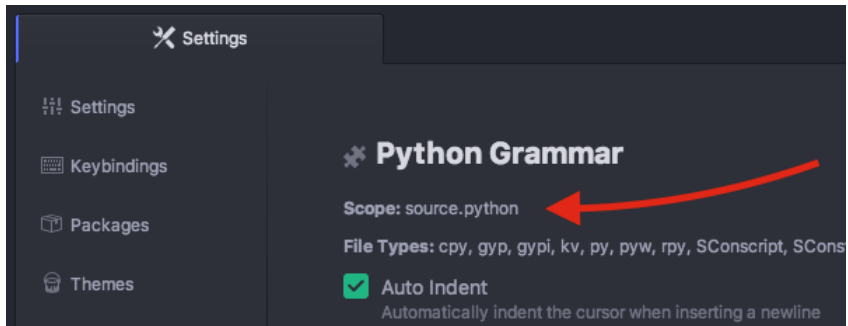
FINDING A LANGUAGE'S SCOPE NAME

In order to write these overrides effectively, you'll need to know the scope name for the language. We've already done this for finding a scope for writing a snippet in **"Snippet Format"**, but we can quickly cover it again.

The scope name is shown in the settings view for each language. Search for the language of your choice in the left panel, select it, and you should see the scope name under the language name heading:

FIGURE 2-31

Finding a language grammar



Summary

At this point you should be something of an Atom master user. You should be able to navigate and manipulate your text and files like a wizard. You should also be able to customize Atom backwards and forwards to make it look and act just how you want it to.

In the next chapter, we're going to kick it up a notch: we'll take a look at changing and adding new functionality to the core of Atom itself. We're going to start creating packages for Atom. If you can dream it, you can build it.

Hacking Atom 3

Now it's time to come to the “Hackable” part of the Hackable Editor. As we've seen throughout the second section, a huge part of Atom is made up of bundled packages. If you wish to add some functionality to Atom, you have access to the same APIs and tools that the core features of Atom has. From the **tree view** to the **command palette** to **find and replace** functionality, even the most core features of Atom are implemented as packages.

In this chapter, we're going to learn how to extend the functionality of Atom through writing packages. This will be everything from new user interfaces to new language grammars to new themes. We'll learn this by writing a series of increasingly complex packages together, introducing you to new APIs and tools and techniques as we need them.

If you're looking for an example using a specific API or feature, you can skip to the end of the chapter where we've indexed all the examples that way.

Tools of the Trade

To begin, there are a few things we'll assume you know, at least to some degree. Since all of Atom is implemented using web technologies, we have to assume you know web technologies such as JavaScript and CSS. Specifically, we'll be implementing everything in CoffeeScript and Less, which are preprocessors for Javascript and CSS respectively.

If you don't know CoffeeScript, but you are familiar with JavaScript, you shouldn't have too much trouble. Here is an example of some simple CoffeeScript code:

```
MyPackageView = require './my-package-view'

module.exports =
  myPackageView: null

activate: (state) ->
```

```

@myPackageView = new MyPackageView(state.myPackageViewState)

deactivate: ->
  @myPackageView.destroy()

serialize: ->
  myPackageViewState: @myPackageView.serialize()

```

We'll go over examples like this in a bit, but this is what the language looks like.

Just about everything you can do with CoffeeScript in Atom is also doable in JavaScript, but as most of the community uses CoffeeScript, you will probably want to write your packages in it. This will help you get contributions from the community and in many instances write simpler code.

You can brush up on CoffeeScript at coffeescript.org.

Less is an even simpler transition from CSS. It adds a number of useful things like variables and functions to CSS. You can brush up on your Less skills at lesscss.org. Our usage of Less won't get too complex in this book however, so as long as you know basic CSS you should be fine.

The Init File

When Atom finishes loading, it will evaluate `init.coffee` in your `~/ .atom` directory, giving you a chance to run CoffeeScript code to make customizations. Code in this file has full access to **Atom's API**. If customizations become extensive, consider creating a package, which we will cover in “**Package: Word Count**”.

You can open the `init.coffee` file in an editor from the *Atom > Open Your Init Script* menu. This file can also be named `init.js` and contain JavaScript code.

For example, if you have the Audio Beep configuration setting enabled, you could add the following code to your `init.coffee` file to have Atom greet you with an audio beep every time it loads:

```
atom.beep()
```

Because `init.coffee` provides access to Atom's API, you can use it to implement useful commands without creating a new package or extending an existing one. Here's a command which uses the **Selection API** and **Clipboard API** to construct a Markdown link from the selected text and the clipboard contents as the URL:


```
atom.commands.add 'atom-text-editor', 'markdown:paste-as-link', ->
  return unless editor = atom.workspace.getActiveTextEditor()

  selection = editor.getLastSelection()
  clipboardText = atom.clipboard.read()

  selection.insertText("#{selection.getText()}(#{clipboardText})")
```

Now, reload Atom and use the “**Command Palette**” to execute the new command by name (i.e., “Markdown: Paste As Link”). And if you’d like to trigger the command via a keyboard shortcut, you can define a **keymap for the command**.

Package: Word Count

Let’s get started by writing a very simple package and looking at some of the tools needed to develop one effectively. We’ll start by writing a package that tells you how many words are in the current buffer and display it in a small modal window.

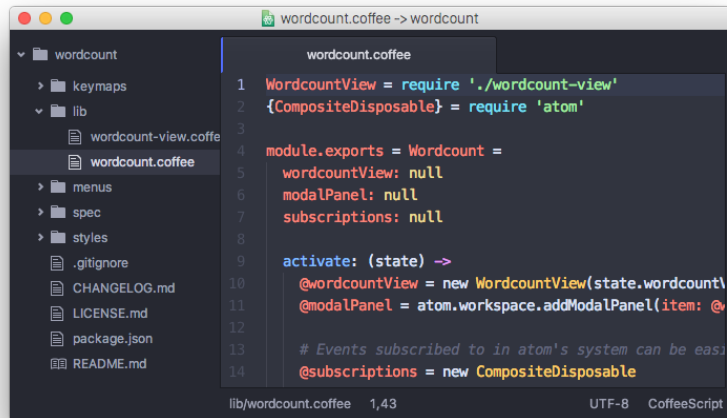
Package Generator

The simplest way to start a package is to use the built-in package generator that ships with Atom. As you might expect by now, this generator is itself a separate package implemented in **atom/package-generator**.

You can run the generator by invoking the command palette and searching for “Generate Package”. A dialog will appear asking you to name your new project. Atom will then create that directory and fill it out with a skeleton project and link it into your `.atom` directory so it’s loaded when you launch your editor next time.

FIGURE 3-1

Basic generated
Atom package



You can see that Atom has created about a dozen files that make up the package. Let's take a look at each of them to get an idea of how a package is structured, then we can modify them to get our word count functionality.

The basic package layout is as follows:

```

my-package/
  grammars/
  keymaps/
  lib/
  menus/
  spec/
  snippets/
  styles/
  index.coffee
  package.json
  
```

Not every package will have (or need) all of these directories and the package generator doesn't create snippets or grammars. Let's see what some of these are so we can start messing with them.

PACKAGE.JSON

Similar to **npm packages**, Atom packages contain a *package.json* file in their top-level directory. This file contains metadata about the package, such as the

path to its “main” module, library dependencies, and manifests specifying the order in which its resources should be loaded.

In addition to the regular **npm package.json keys** available, Atom package.json files have their own additions.

- **main**: the path to the CoffeeScript file that’s the entry point to your package. If this is missing, Atom will default to looking for an `index.coffee` or `index.js`.
- **styles**: an Array of Strings identifying the order of the style sheets your package needs to load. If not specified, style sheets in the *styles* directory are added alphabetically.
- **keymaps**: an Array of Strings identifying the order of the key mappings your package needs to load. If not specified, mappings in the *keymaps* directory are added alphabetically.
- **menus**: an Array of Strings identifying the order of the menu mappings your package needs to load. If not specified, mappings in the *menus* directory are added alphabetically.
- **snippets**: an Array of Strings identifying the order of the snippets your package needs to load. If not specified, snippets in the *snippets* directory are added alphabetically.
- **activationCommands**: an Object identifying commands that trigger your package’s activation. The keys are CSS selectors, the values are Arrays of Strings identifying the command. The loading of your package is delayed until one of these events is triggered within the associated scope defined by the CSS selector.
- **activationHooks**: an Array of Strings identifying hooks that trigger your package’s activation. The loading of your package is delayed until one of these hooks are triggered. Known hooks are:
- `language-package-name:grammar-used` (e.g) `language-javascript:grammar-used`

The `package.json` in the package we’ve just generated looks like this currently:

```
{
  "name": "wordcount",
  "main": "./lib/wordcount",
  "version": "0.0.0",
  "description": "A short description of your package",
  "activationCommands": {
    "atom-workspace": "wordcount:toggle"
  },
}
```

```

    "repository": "https://github.com/atom/wordcount",
    "license": "MIT",
    "engines": {
      "atom": ">0.50.0"
    },
    "dependencies": {
    }
  }

```

If you wanted to use `activationHooks`, you might have:

```

{
  "name": "wordcount",
  "main": "./lib/wordcount",
  "version": "0.0.0",
  "description": "A short description of your package",
  "activationHooks": ["language-javascript:grammar-used", "language-coffee-script:grammar-used"],
  "repository": "https://github.com/atom/wordcount",
  "license": "MIT",
  "engines": {
    "atom": ">0.50.0"
  },
  "dependencies": {
  }
}

```

One of the first things you should do is ensure that this information is filled out. The name, description, repository URL the project will be at, and the license can all be filled out immediately. The other information we'll get into more detail on as we go.

SOURCE CODE

If you want to extend Atom's behavior, your package should contain a single top-level module, which you export from whichever file is indicated by the `main` key in your `package.json` file. In the package we just generated, the main package file is `lib/wordcount.coffee`. The remainder of your code should be placed in the `lib` directory, and required from your top-level file. If the `main` key is not in your `package.json` file, it will look for `index.coffee` or `index.js` as the main entry point.

Your package's top-level module is a singleton object that manages the life-cycle of your extensions to Atom. Even if your package creates ten different views and appends them to different parts of the DOM, it's all managed from your top-level object.

Your package's top-level module should implement the following methods:

- `activate(state)`: This **required** method is called when your package is activated. It is passed the state data from the last time the window was serialized if your module implements the `serialize()` method. Use this to do initialization work when your package is started (like setting up DOM elements or binding events).
- `serialize()`: This **optional** method is called when the window is shutting down, allowing you to return JSON to represent the state of your component. When the window is later restored, the data you returned is passed to your module's `activate` method so you can restore your view to where the user left off.
- `deactivate()`: This **optional** method is called when the window is shutting down. If your package is watching any files or holding external resources in any other way, release them here. If you're just subscribing to things on window, you don't need to worry because that's getting torn down anyway.

STYLE SHEETS

Style sheets for your package should be placed in the *styles* directory. Any style sheets in this directory will be loaded and attached to the DOM when your package is activated. Style sheets can be written as CSS or **Less**, but Less is recommended.

Ideally, you won't need much in the way of styling. Atom provides a standard set of components which define both the colors and UI elements for any package that fits into Atom seamlessly. You can view all of Atom's UI components by opening the styleguide: open the command palette (`cmd-shift-P`) and search for *styleguide*, or just type `cmd-ctrl-shift-G`.

If you *do* need special styling, try to keep only structural styles in the package style sheets. If you *must* specify colors and sizing, these should be taken from the active theme's **ui-variables.less**.

An optional `styleSheets` array in your *package.json* can list the style sheets by name to specify a loading order; otherwise, style sheets are loaded alphabetically.

KEYMAPS

It's recommended that you provide key bindings for commonly used actions for your extension, especially if you're also adding a new command. In our new package, we have a keymap filled in for us already in the `keymaps/word-count.cson` file:

```
'atom-workspace':
  'ctrl-alt-o': 'wordcount:toggle'
```

This means that if you hit `ctrl-alt-o`, our package will run the `toggle` command. We'll look at that code next, but if you want to change the default key mapping, you can do that in this file.

Keymaps are placed in the *keymaps* subdirectory. By default, all keymaps are loaded in alphabetical order. An optional `keymaps` array in your *package.json* can specify which keymaps to load and in what order.

Keybindings are executed by determining which element the keypress occurred on. In the example above, the `wordcount:toggle` command is executed when pressing `ctrl-alt-o` only on the `atom-workspace` element. This means that if you're focused on something else like the Tree View or Settings pane for example, this key mapping won't work.

We'll cover more advanced keymapping stuff a bit later in “**Keymaps In-Depth**”.

MENUS

Menus are placed in the *menus* subdirectory. This defines menu elements like what pops up when you right click (a context-menu) or would go in the menu bar (application menu) to trigger functionality in your plugin.

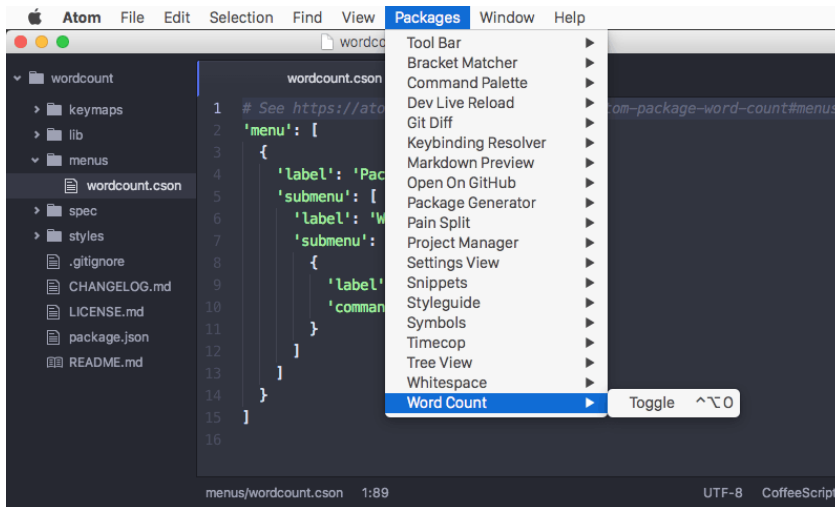
By default, all menus are loaded in alphabetical order. An optional `menus` array in your *package.json* can specify which menus to load and in what order.

Application Menu

It's recommended that you create an application menu item for common actions with your package that aren't tied to a specific element. If we look in the `menus/wordcount.cson` file that was generated for us, we'll see a section that looks like this:

```
'menu': [
  {
    'label': 'Packages'
    'submenu': [
      'label': 'Word Count'
      'submenu': [
        {
          'label': 'Toggle'
          'command': 'wordcount:toggle'
        }
      ]
    ]
  }
]
```

This section puts a “Toggle” menu item under a menu group named “Word Count” in the “Packages” menu.

**FIGURE 3-2**

Application menu item

When you select that menu item, it will run the `wordcount:toggle` command, which we’ll look at in a bit.

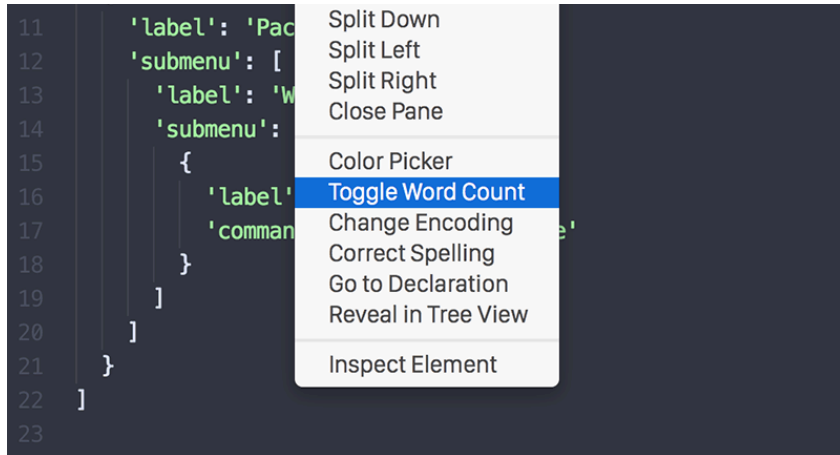
The menu templates you specify are merged with all other templates provided by other packages in the order which they were loaded.

Context Menu

It’s recommended to specify a context menu item for commands that are linked to specific parts of the interface. In our `menus/wordcount.cson` file, we can see an auto-generated section that looks like this:

```
'context-menu':
  'atom-text-editor': [
    {
      'label': 'Toggle Word Count'
      'command': 'wordcount:toggle'
    }
  ]
```

This adds a “Toggle Word Count” menu option to the menu that pops up when you right-click in an Atom text editor pane.

FIGURE 3-3*Context menu entry*

When you click that it will again run the `wordcount:toggle` method in your code.

Context menus are created by determining which element was selected and then adding all of the menu items whose selectors match that element (in the order which they were loaded). The process is then repeated for the elements until reaching the top of the DOM tree.

You can also add separators and submenus to your context menus. To add a submenu, provide a `submenu` key instead of a `command`. To add a separator, add an item with a single `type: 'separator'` key/value pair. For instance, you could do something like this:

```
'context-menu':
  'atom-workspace': [
    {
      label: 'Text'
      submenu: [
        {label: 'Inspect Element', command: 'core:inspect'}
        {type: 'separator'}
        {label: 'Selector All', command: 'core:select-all'}
        {type: 'separator'}
        {label: 'Deleted Selected Text', command: 'core:delete'}
      ]
    }
  ]
```


Developing our Package

Currently with the generated package we have, if we run that `toggle` command through the menu or the command palette, we'll get a little pop up that says "The Wordcount package is Alive! It's ALIVE!".

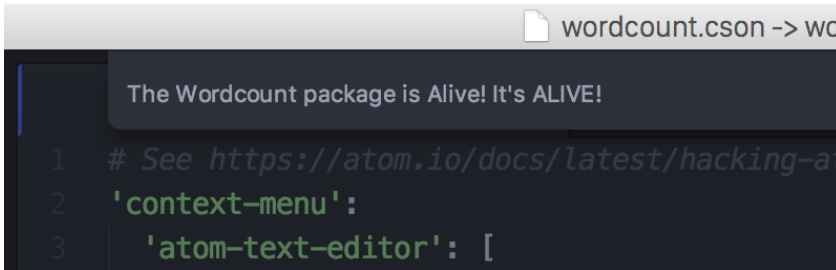


FIGURE 3-4

Wordcount Package is Alive dialog

UNDERSTANDING THE GENERATED CODE

Let's take a look at the code in our `lib` directory and see what is happening.

There are two files in our `lib` directory. One is the main file (`lib/wordcount.coffee`), which is pointed to in the package. `json` file as the main file to execute for this package. This file handles the logic of the whole plugin.

The second file is a View class (`lib/wordcount-view.coffee`), which handles the UI elements of the package. Let's look at this file first, since it's pretty simple.

```
module.exports =
class WordcountView
  constructor: (serializedState) ->
    # Create root element
    @element = document.createElement('div')
    @element.classList.add('wordcount')

    # Create message element
    message = document.createElement('div')
    message.textContent = "The Wordcount package is Alive! It's ALIVE!"
    message.classList.add('message')
    @element.appendChild(message)

    # Returns an object that can be retrieved when package is activated
    serialize: ->

    # Tear down any state and detach
```

```

destroy: ->
  @element.remove()

getElement: ->
  @element

```

Basically the only thing happening here is that when the View class is created, it creates a simple `div` element and adds the `wordcount` class to it (so we can find or style it later) and then adds the ``Wordcount package is Alive!`` text to it. There is also a `getElement` method which returns that `div`. The `serialize` and `destroy` methods don't do anything and we won't have to worry about that until another example.

Notice that we're simply using the basic browser DOM methods (ie, `createElement()`, `appendChild()`).

The second file we have is the main entry point to the package (again, because it's referenced in the `package.json` file). Let's take a look at that file.

```

WordcountView = require './wordcount-view'
{CompositeDisposable} = require 'atom'

module.exports = Wordcount =
  wordcountView: null
  modalPanel: null
  subscriptions: null

  activate: (state) ->
    @wordcountView = new WordcountView(state.wordcountViewState)
    @modalPanel = atom.workspace.addModalPanel(item: @wordcountView.getElement(), v

    # Events subscribed to in atom's system can be easily cleaned up with a Composite
    @subscriptions = new CompositeDisposable

    # Register command that toggles this view
    @subscriptions.add atom.commands.add 'atom-workspace',
      'wordcount:toggle': => @toggle()

  deactivate: ->
    @modalPanel.destroy()
    @subscriptions.dispose()
    @wordcountView.destroy()

  serialize: ->
    wordcountViewState: @wordcountView.serialize()

  toggle: ->
    console.log 'Wordcount was toggled!'

```

```

    if @modalPanel.isVisible()
        @modalPanel.hide()
    else
        @modalPanel.show()

```

There is a bit more going on here. First of all we can see that we are defining four methods. The only required one is `activate`. The `deactivate` and `serialize` methods are expected by Atom but optional. The `toggle` method is one Atom is not looking for, so we'll have to invoke it somewhere for it to be called, which you may recall we do both in the `activationCommands` section of the `package.json` file and in the action we have in the menu file.

The `deactivate` method simply destroys the various class instances we've created and the `serialize` method simply passes on the serialization to the `View` class. Nothing too exciting here.

The `activate` command does a number of things. For one, it is not called automatically when Atom starts up, it is first called when one of the `activationCommands` as defined in the `package.json` file are called. In this case, `activate` is only called the first time the `toggle` command is called. If nobody ever invokes the menu item or hotkey, this code is never called.

This method does two things. The first is that it creates an instance of the `View` class we have and adds the element that it creates to a hidden modal panel in the Atom workspace.

```

@wordcountView = new WordcountView(state.wordcountViewState)
@modalPanel = atom.workspace.addModalPanel(
    item: @wordcountView.getElement(),
    visible: false
)

```

We'll ignore the state stuff for now, since it's not important for this simple plugin. The rest should be fairly straightforward.

The next thing this method does is create an instance of the `CompositeDisposable` class so it can register all the commands that can be called from the plugin so other plugins could subscribe to these events.

```

# Events subscribed to in atom's system can be easily cleaned up with a CompositeDisposable
@subscriptions = new CompositeDisposable

# Register command that toggles this view
@subscriptions.add atom.commands.add 'atom-workspace', 'wordcount:toggle': => @toggle()

```

Next we have the `toggle` method. This method simply toggles the visibility of the modal panel that we created in the `activate` method.

```
toggle: ->
  console.log 'Wordcount was toggled!'

  if @modalPanel.isVisible()
    @modalPanel.hide()
  else
    @modalPanel.show()
```

This should be fairly simple to understand. We're looking to see if the modal element is visible and hiding or showing it depending on it's current state.

THE FLOW

So, let's review the actual flow in this package.

- Atom starts up
- Atom reads the package.json
- Atom reads keymaps, menus, main file
- User runs a package command
- Atom executes the `activate` method
 - Creates a `WordCount` view, which creates a `div`
 - Grabs that `div` and sticks it in a hidden modal
- Atom executes the package command
 - Sees that the modal is hidden, makes it visible
- User runs a package command again
- Atom executes the package command
 - Sees that the modal is visible, makes it hidden
- You shut down Atom
 - Serializations?

TODO: Verify this and perhaps make it a graphic?

COUNTING THE WORDS

So now that we understand what is happening, let's modify the code so that our little modal box shows us the current word count instead of static text.

We'll do this in a very simple way. When the dialog is toggled, we'll count the words right before displaying the modal. So let's do this in the `toggle` command. If we add some code to count the words and ask the view to update itself, we'll have something like this:

```
toggle: ->
  if @modalPanel.isVisible()
    @modalPanel.hide()
  else
    editor = atom.workspace.getActiveTextEditor()
    words = editor.getText().split(/\s+/).length
    @wordcountView.setCount(words)
    @modalPanel.show()
```

Let's look at the 3 lines we've added. First we get an instance of the current editor object (where our text to count is) by calling `atom.workspace.getActiveTextEditor()`.

Next we get the number of words by calling `getText()` on our new editor object, then splitting that text on whitespace with a regular expression and then getting the length of that array.

Finally, we tell our view to update the word count it displays by calling the `setCount()` method on our view and then showing the modal again. Since that method doesn't yet exist, let's create it now.

We can add this code to the end of our `wordcount-view.coffee` file:

```
setCount: (count) ->
  displayText = "There are #{count} words."
  @element.children[0].textContent = displayText
```

Pretty simple - we take the count number that was passed in and place it into a string that we then stick into the element that our view is controlling.

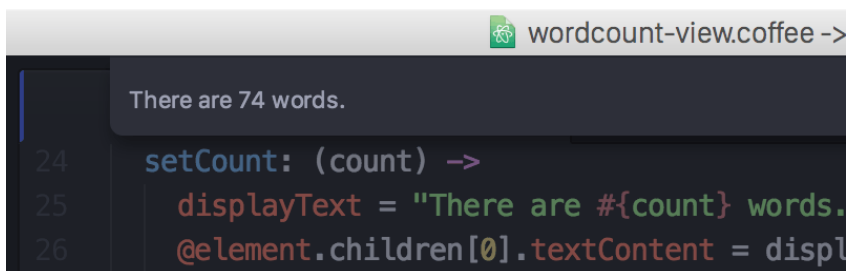


FIGURE 3-5

Word Count Working

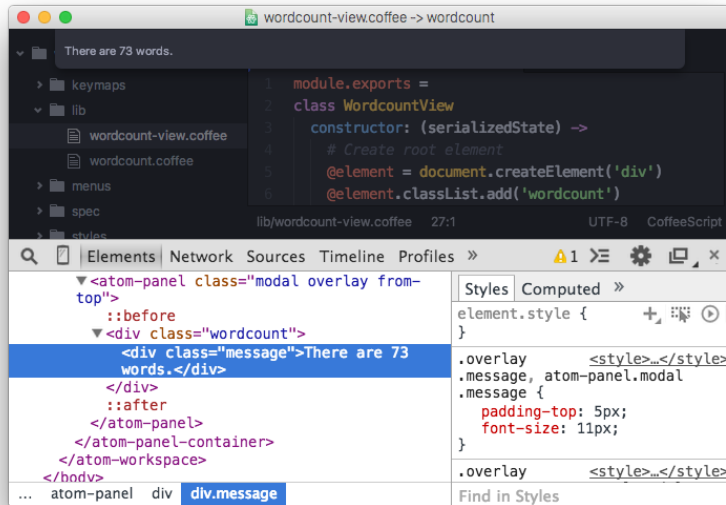
Basic Debugging

You'll notice a few `console.log` statements in the code. One of the cool things about Atom being built on Chromium is that you can use some of the same debugging tools available to you that you have when doing web development.

To open up the Developer Console, hit `alt-cmd-I`, or choose the menu option `View > Developer > Toggle Developer Tools`.

FIGURE 3-6

*Developer Tools
Debugging*



From here you can inspect objects, run code and view console output just as though you were debugging a web site.

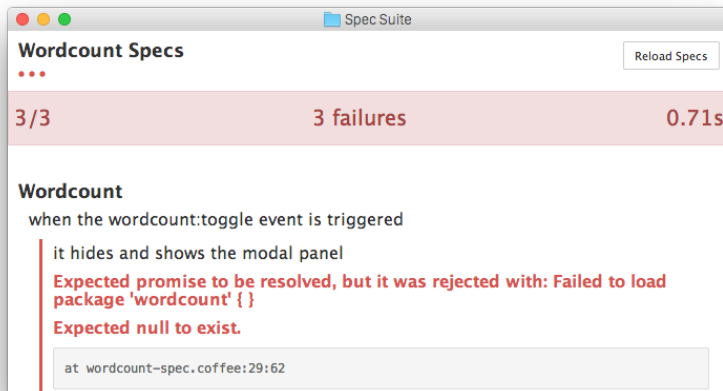
Testing

Your package should have tests, and if they're placed in the `spec` directory, they can be run by Atom.

Under the hood, **Jasmine** executes your tests, so you can assume that any DSL available there is also available to your package.

RUNNING TESTS

Once you've got your test suite written, you can run it by pressing `cmd-alt-ctrl-p` or via the `Developer > Run Package Specs` menu. Our generated package comes with an example test suite, so you can run this right now to see what happens.

**FIGURE 3-7***Spec Suite Results*

You can also use the `apm test` command to run them from the command line. It prints the test output and results to the console and returns the proper status code depending on whether the tests passed or failed.

MAKING OUR TESTS WORK

As you can see, our generated tests are all failing because of the changes we've made.

TODO: Fix the tests

Publishing

Now that our simple plugin is working and tested, let's go ahead and publish it so it's available to the world.

Atom bundles a command line utility called `apm` which we first used back in “**Command Line**” to search for and install packages via the command line. The `apm` command can also be used to publish Atom packages to the public registry and update them.

PREPARE YOUR PACKAGE

There are a few things you should double check before publishing:

- Your **package.json** file has `name`, `description`, and `repository` fields.

- Your **package.json** file has a `version` field with a value of `"0.0.0"`.
- Your **package.json** file has an `engines` field that contains an entry for Atom such as: `"engines": {"atom": ">=0.50.0"}`.
- Your package has a `README.md` file at the root.
- Your package is in a Git repository that has been pushed to **GitHub**. Follow **this guide** if your package isn't already on GitHub.

PUBLISH YOUR PACKAGE

Before you publish a package it is a good idea to check ahead of time if a package with the same name has already been published to **atom.io**. You can do that by visiting <https://atom.io/packages/my-package> to see if the package already exists. If it does, update your package's name to something that is available before proceeding.

Now let's review what the `apm publish` command does:

1. Registers the package name on atom.io if it is being published for the first time.
2. Updates the `version` field in the **package.json** file and commits it.
3. Creates a new **Git tag** for the version being published.
4. Pushes the tag and current branch up to GitHub.
5. Updates atom.io with the new version being published.

Now run the following commands to publish your package:

```
cd ~/github/my-package
apm publish minor
```

If this is the first package you are publishing, the `apm publish` command may prompt you for your GitHub username and password. This is required to publish and you only need to enter this information the first time you publish. The credentials are stored securely in your **keychain** once you login.

Your package is now published and available on atom.io. Head on over to <https://atom.io/packages/my-package> to see your package's page.

With `apm publish`, you can bump the version and publish by using

```
apm publish <version-type>
```

where `<version-type>` can be `major`, `minor` and `patch`.

The `major` option to the `publish` command tells `apm` to increment the first digit of the version before publishing so the published version will be `1.0.0` and the Git tag created will be `v1.0.0`.

The `minor` option to the `publish` command tells `apm` to increment the second digit of the version before publishing so the published version will be `0.1.0` and the Git tag created will be `v0.1.0`.

The `patch` option to the `publish` command tells `apm` to increment the third digit of the version before publishing so the published version will be `0.0.1` and the Git tag created will be `v0.0.1`.

Use `major` when you make a change that breaks backwards compatibility, like changing defaults or removing features. Use `minor` when adding new functionality or making improvements on existing code. Use `patch` when you fix a bug that was causing incorrect behaviour.

Check out **semantic versioning** to learn more about versioning your package releases.

You can also run `apm help publish` to see all the available options and `apm help` to see all the other available commands.

Summary

We’ve now generated, customized and published our first plugin for Atom. Congratulations! Now anyone can install our masterpiece from directly within Atom as we did in “**Atom Packages**”.

Package: Modifying Text

Now that we have our first package written, let’s go through examples of other types of packages we can make. This section will guide you through creating a simple command that replaces the selected text with **ascii art**. When you run our new command with the word “cool” selected, it will be replaced with:

```

                                o888
                                888
      ooooooo      ooooooo      ooooooo
888      888 888      888 888      888 888
888      888      888 888      888 888
      88ooo888      88ooo88      88ooo88  o888o

```

This should demonstrate how to do basic text manipulation in the current text buffer and how to deal with selections.

The final package can be viewed at <https://github.com/atom/ascii-art>.

Basic Text Insertion

To begin, press cmd-shift-P to bring up the **Command Palette**. Type “generate package” and select the “Package Generator: Generate Package” command, just as we did in “**Package Generator**”. Enter `ascii-art` as the name of the package.

Now let’s edit the package files to make our ASCII Art package do something interesting. Since this package doesn’t need any UI, we can remove all view-related code so go ahead and delete `lib/ascii-art-view.coffee`, `spec/ascii-art-view-spec.coffee`, and `styles/`.

Next, open up `lib/ascii-art.coffee` and remove all view code, so it looks like this:

```
{CompositeDisposable} = require 'atom'

module.exports =
  subscriptions: null

  activate: ->
    @subscriptions = new CompositeDisposable
    @subscriptions.add atom.commands.add 'atom-workspace',
      'ascii-art:convert': => @convert()

  deactivate: ->
    @subscriptions.dispose()

  convert: ->
    console.log 'Convert text!'
```

CREATE A COMMAND

Now let’s add a command. It is recommended that you namespace your commands with the package name followed by a `:`, so as you can see in the code, we called our command `ascii-art:convert` and it will call the `convert()` method when it’s called.

So far, that will simply log to the console. Let’s start by making it insert something into the text buffer.

```
convert: ->
  if editor = atom.workspace.getActiveTextEditor()
    editor.insertText('Hello, World!')
```

As in “**Counting the Words**”, we’re using `atom.workspace.getActiveTextEditor()` to get the object that represents the active text editor. If this

`convert()` method is called when not focused on a text editor, this will simply return a blank string, so we can skip the next line.

Next we insert a string into the current text editor with the `insertText()` method. This will insert the text wherever the cursor currently is in the current editor. If there is a selection, it will replace all selections with the “Hello, World!” text.

RELOAD THE PACKAGE

Before we can trigger `ascii-art:convert`, we need to load the latest code for our package by reloading the window. Run the command “Window: Reload” from the command palette or by pressing `ctrl-alt-cmd-l`.

TRIGGER THE COMMAND

Now open the command panel and search for the “Ascii Art: Convert” command. But it’s not there! To fix this, open `package.json` and find the property called `activationCommands`. Activation commands speed up load time by allowing Atom to delay a package’s activation until it’s needed. So remove the existing command and use `ascii-art:convert` in `activationCommands`:

```
"activationCommands": {
  "atom-workspace": "ascii-art:convert"
}
```

First, reload the window by running the command “Window: Reload” from the command palette. Now when you run the “Ascii Art: Convert” command it will output *Hello, World!*

ADD A KEY BINDING

Now let’s add a key binding to trigger the `ascii-art:convert` command. Open `keymaps/ascii-art.cson` and add a key binding linking `ctrl-alt-a` to the `ascii-art:convert` command. You can delete the pre-existing key binding since you don’t need it anymore.

When finished, the file should look like this:

```
'atom-text-editor':
  'ctrl-alt-a': 'ascii-art:convert'
```

Now reload the window and verify that the key binding works.

Add the ASCII Art

Now we need to convert the selected text to ASCII art. To do this we will use the **figlet node** module from **npm**. Open *package.json* and add the latest version of figlet to the dependencies:

```
"dependencies": {
  "figlet": "1.0.8"
}
```

After saving the file, run the command “Update Package Dependencies: Update” from the Command Palette. This will install the package’s node module dependencies, only figlet in this case. You will need to run “Update Package Dependencies: Update” whenever you update the dependencies field in your *package.json* file.

If for some reason this doesn’t work, you’ll see a message saying “Failed to update package dependencies” and you will find a new *npm-debug.log* file in your directory. That file should give you some idea as to what went wrong.

Now require the figlet node module in *lib/ascii-art.coffee* and instead of inserting *Hello, World!*, convert the selected text to ASCII art.

```
convert: ->
  if editor = atom.workspace.getActiveTextEditor()
    selection = editor.getSelectedText()

    figlet = require 'figlet'
    font = "o8"
    figlet selection, {font: font}, (error, art) ->
      if error
        console.error(error)
      else
        editor.insertText("\n#{art}\n")
```

Now reload the editor, select some text in an editor window and hit **ctrl-alt-a**. It should be replaced with a ridiculous ASCII art version instead.

There are a couple of new things in this example we should look at quickly. The first is the **editor.getSelectedText()** which, as you might guess, returns the text that is currently selected.

We then call the Figlet code to convert that into something else and replace the current selection with it with the **editor.insertText()** call.

Summary

In this section, we've made a UI-less package that takes selected text and replaces it with a processed version. This could be helpful in creating linters or checkers for your code.

Creating a Theme

Atom's interface is rendered using HTML, and it's styled via **Less** which is a superset of CSS. Don't worry if you haven't heard of Less before; it's just like CSS, but with a few handy extensions.

Atom supports two types of themes: *UI* and *syntax*. UI themes style elements such as the tree view, the tabs, drop-down lists, and the status bar. Syntax themes style the code inside the editor.

Themes can be installed and changed from the settings view which you can open by selecting the *Atom > Preferences...* menu and navigating to the *Install* section and the *Themes* section on the left hand side.

Getting Started

Themes are pretty straightforward but it's still helpful to be familiar with a few things before starting:

- Less is a superset of CSS, but it has some really handy features like variables. If you aren't familiar with its syntax, take a few minutes to **familiarize yourself**.
- You may also want to review the concept of a *package.json* (as covered in "**package.json**"). This file is used to help distribute your theme to Atom users.
- Your theme's *package.json* must contain a "theme" key with a value of "ui" or "syntax" for Atom to recognize and load it as a theme.
- You can find existing themes to install or fork on **atom.io**.

Creating a Syntax Theme

Let's create your first theme.

To get started, hit `cmd-shift-P`, and start typing "Generate Syntax Theme" to generate a new theme package. Select "Generate Syntax Theme," and you'll be asked for the path where your theme will be created. Let's call ours *motif-syntax*. *Tip:* syntax themes should end with *-syntax*.

Atom will pop open a new window, showing the *motif-syntax* theme, with a default set of folders and files created for us. If you open the settings view (`cmd-,`) and navigate to the *Themes* section on the left, you'll see the *Motif* theme listed in the *Syntax Theme* drop-down. Select it from the menu to activate it, now when you open an editor you should see that your new *motif-syntax* theme in action.

Open up *styles/colors.less* to change the various colors variables which have been already been defined. For example, turn `@red` into `#f4c2c1`.

Then open *styles/base.less* and modify the various selectors that have been already been defined. These selectors style different parts of code in the editor such as comments, strings and the line numbers in the gutter.

As an example, let's make the `.gutter background-color` into `@red`.

Reload Atom by pressing `cmd-alt-ctrl-l` to see the changes you made reflected in your Atom window. Pretty neat!

Tip: You can avoid reloading to see changes you make by opening an atom window in dev mode. To open a Dev Mode Atom window run `atom --dev` in the terminal, use `cmd-shift-o` or use the *View > Developer > Open in Dev Mode* menu. When you edit your theme, changes will instantly be reflected!

It's advised to *not* specify a `font-family` in your syntax theme because it will override the Font Family field in Atom's settings. If you still like to recommend a font that goes well with your theme, we recommend you do so in your README.

Creating an Interface Theme

Interface themes **must** provide a `ui-variables.less` file which contains all of the variables provided by the core themes, which we'll cover in "**Theme Variables**".

To create an interface UI theme, do the following:

1. Fork one of the following repositories:
 - **atom-dark-ui**
 - **atom-light-ui**
2. Clone the forked repository to the local filesystem
3. Open a terminal in the forked theme's directory
4. Open your new theme in a Dev Mode Atom window run `atom --dev` in the terminal or use the *View > Developer > Open in Dev Mode* menu
5. Change the name of the theme in the theme's `package.json` file
6. Name your theme end with a `-ui`. i.e. `super-white-ui`

7. Run `apm link` to symlink your repository to `~/ .atom/packages`
8. Reload Atom using `cmd-alt-ctrl-L`
9. Enable the theme via *UI Theme* drop-down in the *Themes* section of the settings view
10. Make changes! Since you opened the theme in a Dev Mode window, changes will be instantly reflected in the editor without having to reload.

Development workflow

There are a few of tools to help make theme development faster and easier.

LIVE RELOAD

Reloading by hitting `cmd-alt-ctrl-L` after you make changes to your theme is less than ideal. Atom supports **live updating** of styles on Dev Mode Atom windows.

To enable a Dev Mode window:

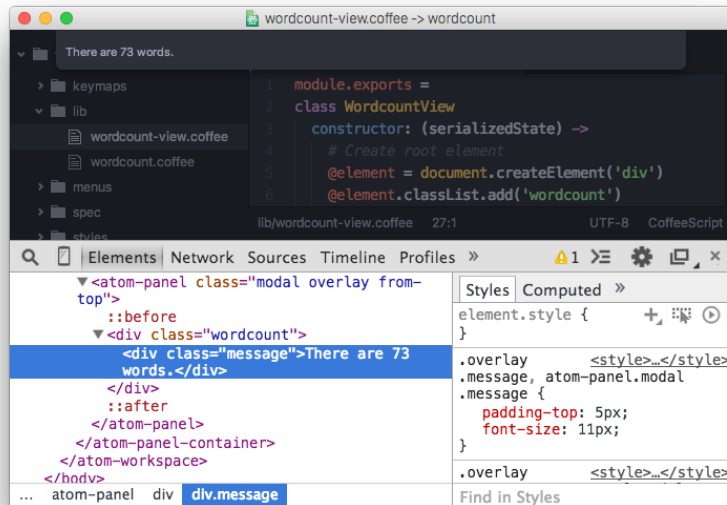
1. Open your theme directory in a dev window by either going to the *View > Developer > Open in Dev Mode* menu or by hitting the `cmd-shift-o` shortcut
2. Make a change to your theme file and save it. Your change should be immediately applied!

If you'd like to reload all the styles at any time, you can use the shortcut `cmd-ctrl-shift-r`.

DEVELOPER TOOLS

Atom is based on the Chrome browser, and supports Chrome's Developer Tools. You can open them by selecting the *View > Toggle Developer Tools* menu, or by using the `cmd-alt-i` shortcut.

The dev tools allow you to inspect elements and take a look at their CSS properties.

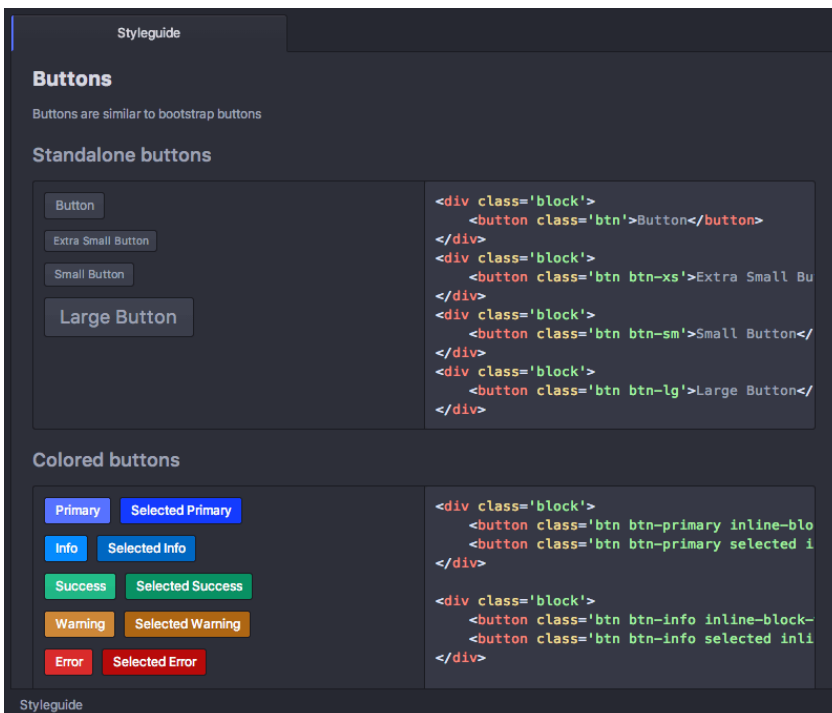
FIGURE 3-8*Dev Tools*

Check out Google’s **extensive tutorial** for a short introduction.

ATOM STYLEGUIDE

If you are creating an interface theme, you’ll want a way to see how your theme changes affect all the components in the system. The **styleguide** is a page that renders every component Atom supports.

To open the styleguide, open the command palette (`cmd-shift-P`) and search for *styleguide*, or use the shortcut `cmd-ctrl-shift-g`.

**FIGURE 3-9***Style Guide*

Theme Variables

Atom's UI provides a set of variables you can use in your own themes and packages.

USE IN THEMES

Each custom theme must specify a `ui-variables.less` file with all of the following variables defined. The top-most theme specified in the theme settings will be loaded and available for import.

USE IN PACKAGES

In any of your package's `.less` files, you can access the theme variables by importing the `ui-variables` file from Atom.

Your package should generally only specify structural styling, and these should come from **the style guide**. Your package shouldn't specify colors, padding sizes, or anything in absolute pixels. You should instead use the theme

variables. If you follow this guideline, your package will look good out of the box with any theme!

Here's an example `.less` file that a package can define using theme variables:

```
@import "ui-variables";

.my-selector {
  background-color: @base-background-color;
  padding: @component-padding;
}
```

VARIABLES

Text colors

- `@text-color`
- `@text-color-subtle`
- `@text-color-highlight`
- `@text-color-selected`
- `@text-color-info` - A blue
- `@text-color-success` - A green
- `@text-color-warning` - An orange or yellow
- `@text-color-error` - A red

Background colors

- `@background-color-info` - A blue
- `@background-color-success` - A green
- `@background-color-warning` - An orange or yellow
- `@background-color-error` - A red
- `@background-color-highlight`
- `@background-color-selected`
- `@app-background-color` - The app's background under all the editor components

Component colors

- `@base-background-color` -
- `@base-border-color` -
- `@pane-item-background-color` -

- @pane-item-border-color -
- @input-background-color -
- @input-border-color -
- @tool-panel-background-color -
- @tool-panel-border-color -
- @inset-panel-background-color -
- @inset-panel-border-color -
- @panel-heading-background-color -
- @panel-heading-border-color -
- @overlay-background-color -
- @overlay-border-color -
- @button-background-color -
- @button-background-color-hover -
- @button-background-color-selected -
- @button-border-color -
- @tab-bar-background-color -
- @tab-bar-border-color -
- @tab-background-color -
- @tab-background-color-active -
- @tab-border-color -
- @tree-view-background-color -
- @tree-view-border-color -
- @ui-site-color-1 -
- @ui-site-color-2 -
- @ui-site-color-3 -
- @ui-site-color-4 -
- @ui-site-color-5 -

Component sizes

- @disclosure-arrow-size -
- @component-padding -
- @component-icon-padding -
- @component-icon-size -

- `@component-line-height -`
- `@component-border-radius -`
- `@tab-height -`

Fonts

- `@font-size -`
- `@font-family -`

Iconography

Atom comes bundled with the **Octicons** icon set. Use them to add icons to your packages.

USAGE

Atom's usage of the Octicons differs just a bit from the **standard way**. The biggest difference is in the naming of the icon classes. Instead of the `octicon-` prefix, you would use a more generic `icon icon-` prefix.

As an example, to add a **monitor icon**, use the `icon icon-device-desktop` classes in your markup:

```
<span class="icon icon-device-desktop"></span>
```

Or in case you're using **SpacePen** it would be:

```
@span class: 'icon icon-device-desktop'
```

SIZE

Octicons look best with a `font-size` of 16px. It's already used as the default, so you don't need to worry about it. In case you prefer a different icon size, try to use multiples of 16 (32px, 48px etc.) for the sharpest result. Sizes in between are ok too, but might look a bit blurry for icons with straight lines.

USABILITY

Although icons can make your UI visually appealing, when used without a text label, it can be hard to guess its meaning. In cases where space for a text label is insufficient, consider adding a **tooltip** that appears on hover. Or a more subtle `title="label"` attribute would already help as well.

Debugging

Atom provides several tools to help you understand unexpected behavior and debug problems. This guide describes some of those tools and a few approaches to help you debug and provide more helpful information when **submitting issues**:

Update to the latest version

You might be running into an issue which was already fixed in a more recent version of Atom than the one you're using.

If you're building Atom from source, pull down the latest version of master and **re-build**.

If you're using released version, check which version of Atom you're using:

```
$ atom --version
0.178.0-37a85bc
```

Head on over to the **list of releases** and see if there's a more recent release. You can update to the most recent release by downloading Atom from the releases page, or with the in-app auto-updater. The in-app auto-updater checks for and downloads a new version after you restart Atom, or if you use the Atom > Check for Update menu option.

Check for linked packages

If you develop or contribute to Atom packages, there may be left-over packages linked to your `~/ .atom/packages` or `~/ .atom/dev/packages` directories. You can use:

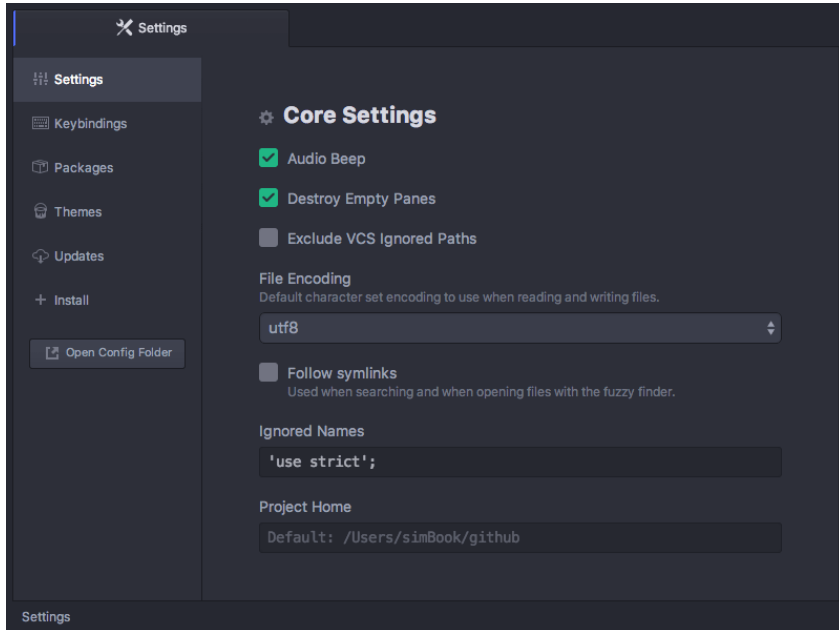
```
$ apm links
```

to list all linked development packages. You can remove the links using the `apm unlink` command. See `apm unlink --help` for details.

Check Atom and package settings

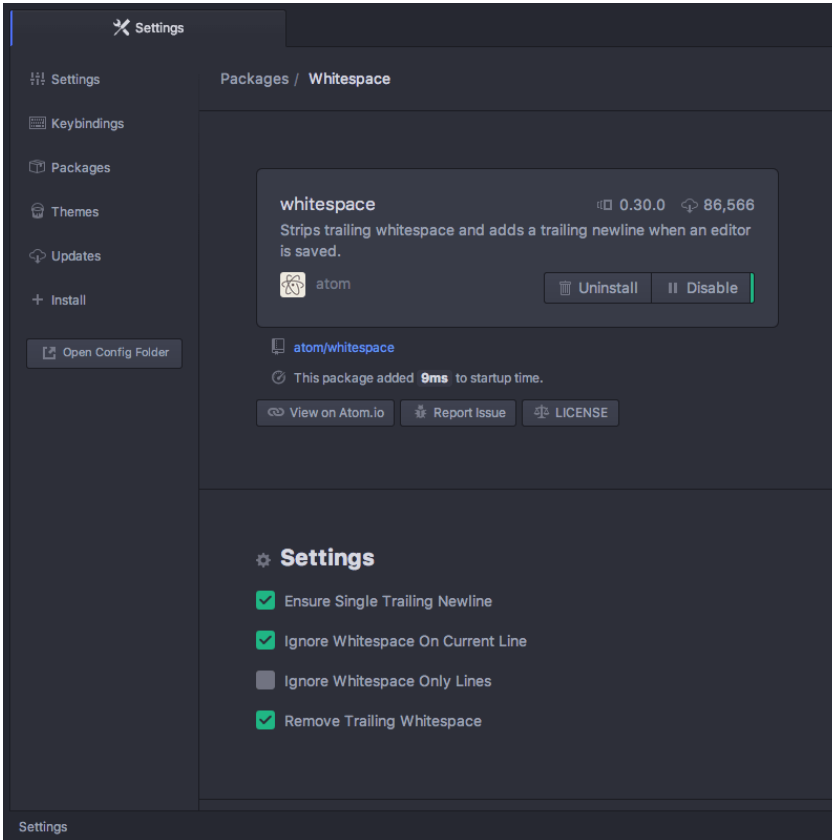
In some cases, unexpected behavior might be caused by misconfigured or unconfigured settings in Atom or in one of the packages.

Open Atom's Settings View with `cmd-`, or the Atom > Preferences menu option.

FIGURE 3-10*Settings View*

Check Atom’s settings in the Settings pane, there’s a description of each configuration option [here](#). For example, if you want Atom to use hard tabs (real tabs) and not soft tabs (spaces), disable the “Soft Tabs” option.

Since Atom ships with a set of packages and you can install additional packages yourself, check the list of packages and their settings. For example, if you’d like to get rid of the vertical line in the middle of the editor, disable the **Wrap Guide package**. And if you don’t like it when Atom strips trailing whitespace or ensures that there’s a single trailing newline in the file, you can configure that in the **Whitespace packages’** settings.

**FIGURE 3-11***Package Settings*

TimeCop

TODO: Document TimeCop

Check the keybindings

If a command is not executing when you hit a keystroke or the wrong command is executing, there might be an issue with the keybindings for that keystroke. Atom ships with the **Keybinding resolver**, a neat package which helps you understand which keybindings are executed.

Show the keybinding resolver with `cmd-. .` or with “Key Binding Resolver: Show” from the Command palette. With the keybinding resolver shown, hit a keystroke:

FIGURE 3-12*Keybinding Resolver*

Key Binding Resolver: cmd-c

✓ core:copy	body	/Users/simBook/github/atom/keymaps/darwin.cson
✗ native!	body .native-key-bindings	/Users/simBook/github/atom/keymaps/darwin.cson
✗ tree-view:copy	.platform-darwin .tree-view	/Users/simBook/github/atom/node_modules/tree-view

The keybinding resolver shows you a list of keybindings that exist for the keystroke, where each item in the list has the following:

- the command for the keybinding,
- the CSS selector used to define the context in which the keybinding is valid, and
- the file in which the keybinding is defined.

Of all the keybinding that are listed (grey color), at most one keybinding is matched and executed (green color). If the command you wanted to trigger isn't listed, then a keybinding for that command hasn't been defined. More keybindings are provided by **packages** and you can define your own keybindings as we saw in “**Customizing Key Bindings**”.

If multiple keybindings are matched, Atom determines which keybinding will be executed based on the **specificity of the selectors and the order in which they were loaded**. If the command you wanted to trigger is listed in the Keybinding resolver, but wasn't the one that was executed, this is normally explained by one of two causes:

- the keystroke was not used in the context defined by the keybinding's selector. For example, you can't trigger the “Tree View: Add File” command if the Tree View is not focused, or
- there is another keybinding that took precedence. This often happens when you install a package which defines keybinding that conflict with existing keybindings. If the package's keybindings have selectors with higher specificity or were loaded later, they'll have priority over existing ones.

Atom loads core Atom keybindings and package keybindings first, and user-defined keybindings after last. Since user-defined keybindings are loaded last, you can use your `keymap.cson` file to tweak the keybindings and sort out problems like these. For example, you can remove keybindings with **the `unset!` directive**.

If you notice that a package's keybindings are taking precedence over core Atom keybindings, it might be a good idea to report the issue on the package's GitHub repository.

Check if the problem shows up in safe mode

A large part of Atom's functionality comes from packages you can install. In some cases, these packages might be causing unexpected behavior, problems, or performance issues.

To determine if a package you installed is causing problems, start Atom from the terminal in safe mode:

```
$ atom --safe
```

This starts Atom, but does not load packages from `~/ .atom/packages` or `~/ .atom/dev/packages`. If you can no longer reproduce the problem in safe mode, it's likely it was caused by one of the packages.

To figure out which package is causing trouble, start Atom normally again and open Settings (`cmd-,`). Since Settings allow you to disable each installed package, you can disable packages one by one until you can no longer reproduce the issue. Restart (`cmd-q`) or reload (`cmd-ctrl-alt-l`) Atom after you disable each package to make sure it's completely gone.

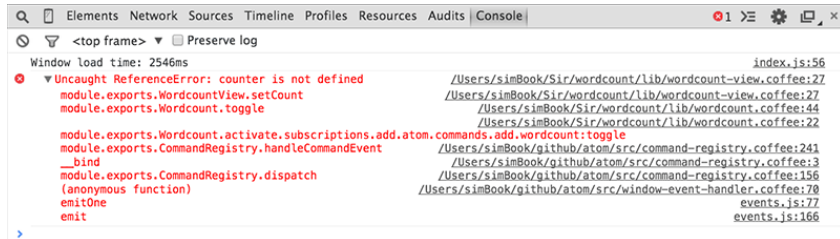
When you find the problematic package, you can disable or uninstall the package, and consider creating an issue on the package's GitHub repository.

Check your config files

You might have defined some custom functionality or styles in Atom's Init script or Stylesheet. In some situations, these personal hacks might be causing problems so try clearing those files and restarting Atom.

Check for errors in the developer tools

When an error is thrown in Atom, the developer tools are automatically shown with the error logged in the Console tab. However, if the dev tools are open before the error is triggered, a full stack trace for the error will be logged:

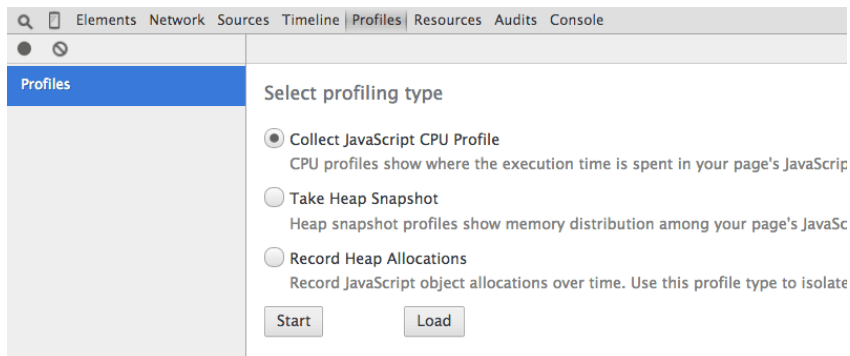
FIGURE 3-13*DevTools Error*

If you can reproduce the error, use this approach to get the full stack trace. The stack trace might point to a problem in your Init script or a specific package you installed, which you can then disable and report an issue on its GitHub repository.

Diagnose performance problems with the dev tools CPU profiler

If you're experiencing performance problems in a particular situation, your **reports** will be more valuable if you include a screenshot from Chrome's CPU profiler that gives some insight into what is slow.

To run a profile, open the dev tools ("Window: Toggle Dev Tools" in the **Command Palette**), navigate to the Profiles tab, select Collect JavaScript CPU Profile, and click Start.

FIGURE 3-14

Then refocus Atom and perform the slow action to capture a recording. When finished, click Stop. Switch to the Chart view, and a graph of the recorded actions will appear. Try to zoom in on the slow area, then take a screenshot

to include with your report. You can also save and post the profile data by clicking **Save** next to the profile's name (e.g. **Profile 1**) in the left panel.

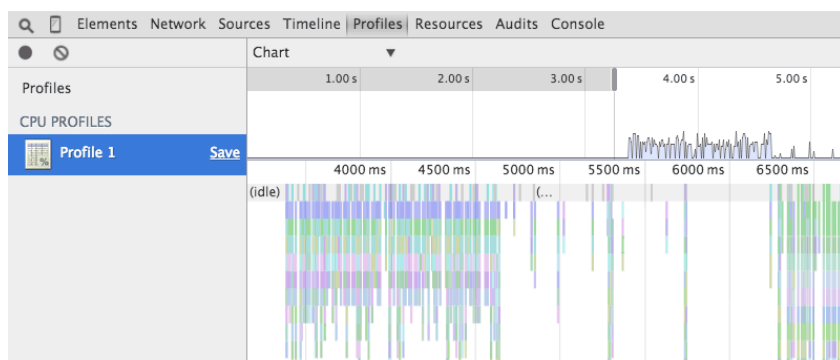


FIGURE 3-15

To learn more, check out the [Chrome documentation on CPU profiling](#).

Check that you have a build toolchain installed

If you are having issues installing a package using `apm install`, this could be because the package has dependencies on libraries that contain native code and so you will need to have a C++ compiler and Python installed to be able to install it.

You can run `apm install --check` to see if `apm` can build native code on your machine.

Check out the pre-requisites in the [build instructions](#) for your platform for more details.

Writing specs

We've looked at and written a few specs through the examples already. Now it's time to take a closer look at the spec framework itself. How exactly do you write tests in Atom?

Atom uses **Jasmine** as its spec framework. Any new functionality should have specs to guard against regressions.

Create a new spec

Atom specs and **package specs** are added to their respective spec directory. The example below creates a spec for Atom core.

CREATE A SPEC FILE

Spec files **must** end with `-spec` so add `sample-spec.coffee` to `atom/spec`.

ADD ONE OR MORE DESCRIBE METHODS

The `describe` method takes two arguments, a description and a function. If the description explains a behavior it typically begins with `when`; if it is more like a unit test it begins with the method name.

```
describe "when a test is written", ->
  # contents
```

or

```
describe "Editor::moveUp", ->
  # contents
```

ADD ONE OR MORE IT METHOD

The `it` method also takes two arguments, a description and a function. Try and make the description flow with the `it` method. For example, a description of `this should work` doesn't read well as `it this should work`. But a description of `should work` sounds great as `it should work`.

```
describe "when a test is written", ->
  it "has some expectations that should pass", ->
    # Expectations
```

ADD ONE OR MORE EXPECTATIONS

The best way to learn about expectations is to read the **Jasmine documentation** about them. Below is a simple example.

```
describe "when a test is written", ->
  it "has some expectations that should pass", ->
    expect("apples").toEqual("apples")
    expect("oranges").not.toEqual("apples")
```

Asynchronous specs

Writing Asynchronous specs can be tricky at first. Some examples.

PROMISES

Working with promises is rather easy in Atom. You can use our `waitForPromise` function.

```
describe "when we open a file", ->
  it "should be opened in an editor", ->
    waitForPromise ->
      atom.workspace.open('c.coffee').then (editor) ->
        expect(editor.getPath()).toContain 'c.coffee'
```

This method can be used in the `describe`, `it`, `beforeEach` and `afterEach` functions.

```
describe "when we open a file", ->
  beforeEach ->
    waitForPromise ->
      atom.workspace.open 'c.coffee'

  it "should be opened in an editor", ->
    expect(atom.workspace.getActiveTextEditor().getPath()).toContain 'c.coffee'
```

If you need to wait for multiple promises use a new `waitForPromise` function for each promise. (Caution: Without `beforeEach` this example will fail!)

```
describe "waiting for the packages to load", ->

  beforeEach ->
    waitForPromise ->
      atom.workspace.open('sample.js')
    waitForPromise ->
      atom.packages.activatePackage('tabs')
    waitForPromise ->
      atom.packages.activatePackage('tree-view')

  it 'should have waited long enough', ->
    expect(atom.packages.isPackageActive('tabs')).toBe true
    expect(atom.packages.isPackageActive('tree-view')).toBe true
```

ASYNCHRONOUS FUNCTIONS WITH CALLBACKS

Specs for asynchronous functions can be done using the `waitsFor` and `runs` functions. A simple example.

```
describe "fs.readdir(path, cb)", ->
  it "is async", ->
    spy = jasmine.createSpy('fs.readdirSpy')

    fs.readdir('/tmp/example', spy)
    waitsFor ->
      spy.callCount > 0
    runs ->
      exp = [null, ['example.coffee']]
      expect(spy.mostRecentCall.args).toEqual exp
      expect(spy).toHaveBeenCalledWith(null, ['example.coffee'])
```

For a more detailed documentation on asynchronous tests please visit the [Jasmine documentation](#).

Running specs

Most of the time you'll want to run specs by triggering the `window:run-package-specs` command. This command is not only to run package specs, it is also for Atom core specs. This will run all the specs in the current project's spec directory. If you want to run the Atom core specs and **all** the default package specs trigger the `window:run-all-specs` command.

To run a limited subset of specs use the `fdescribe` or `fit` methods. You can use those to focus a single spec or several specs. In the example above, focusing an individual spec looks like this:

```
describe "when a test is written", ->
  fit "has some expectations that should pass", ->
    expect("apples").toEqual("apples")
    expect("oranges").not.toEqual("apples")
```

RUNNING ON CI

It is now easy to run the specs in a CI environment like Travis and AppVeyor. See the [Travis CI For Your Packages](#) and [AppVeyor CI For Your Packages](#) posts for more details.

Converting from TextMate

It's possible that you have themes or grammars from **TextMate** that you like and use and would like to convert to Atom. If so, you're in luck because there are tools to help with the conversion.

Converting a TextMate Bundle

Converting a TextMate bundle will allow you to use its editor preferences, snippets, and colorization inside Atom.

Let's convert the TextMate bundle for the **R** programming language. You can find other existing TextMate bundles **on GitHub**.

You can convert the R bundle with the following command:

```
$ apm init --package ~/.atom/packages/language-r \
  --convert https://github.com/textmate/r.tmbundle
```

You can now browse to `~/.atom/packages/language-r` to see the converted bundle.

Your new package is now ready to use, launch Atom and open a `.r` file in the editor to see it in action!

Converting a TextMate Theme

This section will go over how to convert a **TextMate** theme to an Atom theme.

DIFFERENCES

TextMate themes use **plist** files while Atom themes use **CSS** or **Less** to style the UI and syntax in the editor.

The utility that converts the theme first parses the theme's plist file and then creates comparable CSS rules and properties that will style Atom similarly.

CONVERT THE THEME

Download the theme you wish to convert, you can browse existing TextMate themes on the **TextMate website**.

Now, let's say you've downloaded the theme to `~/Downloads/MyTheme.tmbundle`, you can convert the theme with the following command:

```
$ apm init --theme ~/.atom/packages/my-theme \  
--convert ~/Downloads/MyTheme.tmTheme
```

You can then browse to `~/.atom/packages/my-theme` to see the converted theme.

ACTIVATE THE THEME

Now that your theme is installed to `~/.atom/packages` you can enable it by launching Atom and selecting the *Atom > Preferences...* menu.

Select the *Themes* link on the left side and choose *My Theme* from the *Syntax Theme* dropdown menu to enable your new theme.

Your theme is now enabled, open an editor to see it in action!

Summary

(List of topics we covered / appendix?)

Summary.

Behind Atom 4

Now that we've written a number of packages and themes, let's take minute to take a closer look at some of the ways that Atom works in greater depth. Here we'll go into more of a deep dive on individual internal APIs and systems of Atom, even looking at some Atom source to see how things are really getting done.

Configuration API

Reading Config Settings

If you are writing a package that you want to make configurable, you'll need to read config settings via the `atom.config` global. You can read the current value of a namespaced config key with `atom.config.get`:

```
# read a value with `config.get`  
@showInvisibles() if atom.config.get "editor.showInvisibles"
```

Or you can subscribe via `atom.config.observe` to track changes from any view object.

```
{View} = require 'space-pen'  
  
class MyView extends View  
  attached: ->  
    @fontSizeObserveSubscription =  
      atom.config.observe 'editor.fontSize', (newValue, {previous}) =>  
        @adjustFontSize()  
  
  detached: ->  
    @fontSizeObserveSubscription.dispose()
```

The `atom.config.observe` method will call the given callback immediately with the current value for the specified key path, and it will also call it in the future whenever the value of that key path changes. If you only want to invoke the callback when the next time the value changes, use `atom.config.onDidChange` instead.

Subscription methods return **disposable** subscription objects. Note in the example above how we save the subscription to the `@fontSizeObserveSubscription` instance variable and dispose of it when the view is detached. To group multiple subscriptions together, you can add them all to a **CompositeDisposable** that you dispose when the view is detached.

Writing Config Settings

The `atom.config` database is populated on startup from `~/.atom/config.cson`, but you can programmatically write to it with `atom.config.set`:

```
# basic key update
atom.config.set("core.showInvisibles", true)
```

If you're exposing package configuration via specific key paths, you'll want to associate them with a schema in your package's main module. Read more about schemas in the [config API docs](#).

Keymaps In-Depth

Structure of a Keymap File

Keymap files are encoded as JSON or CSON files containing nested hashes. They work much like style sheets, but instead of applying style properties to elements matching the selector, they specify the meaning of keystrokes on elements matching the selector. Here is an example of some bindings that apply when keystrokes pass through `atom-text-editor` elements:

```
'atom-text-editor':
  'cmd-delete': 'editor:delete-to-beginning-of-line'
  'alt-backspace': 'editor:delete-to-beginning-of-word'
  'ctrl-A': 'editor:select-to-first-character-of-line'
  'ctrl-shift-e': 'editor:select-to-end-of-line'
  'cmd-left': 'editor:move-to-first-character-of-line'

'atom-text-editor:not([mini]):'
```

```
'cmd-alt-[': 'editor:fold-current-row'
'cmd-alt-]': 'editor:unfold-current-row'
```

Beneath the first selector are several bindings, mapping specific **keystroke patterns** to **commands**. When an element with the `atom-text-editor` class is focused and `cmd-delete` is pressed, a custom DOM event called `editor:delete-to-beginning-of-line` is emitted on the `atom-text-editor` element.

The second selector group also targets editors, but only if they don't have the `mini` attribute. In this example, the commands for code folding don't really make sense on mini-editors, so the selector restricts them to regular editors.

KEYSTROKE PATTERNS

Keystroke patterns express one or more keystrokes combined with optional modifier keys. For example: `ctrl-w v`, or `cmd-shift-up`. A keystroke is composed of the following symbols, separated by a `-`. A multi-keystroke pattern can be expressed as keystroke patterns separated by spaces.

Type	Examples
Character literals	a 4 \$
Modifier keys	cmd ctrl alt shift
Special keys	enter escape backspace delete tab home end pageup pagedown left right up down

COMMANDS

Commands are custom DOM events that are triggered when a keystroke matches a binding. This allows user interface code to listen for named commands without specifying the specific keybinding that triggers it. For example, the following code creates a command to insert the current date in an editor:

```
atom.commands.add 'atom-text-editor',
  'user:insert-date': (event) ->
    editor = @getModel()
    editor.insertText(new Date().toLocaleString())
```

`atom.commands` refers to the global `{CommandRegistry}` instance where all commands are set and consequently picked up by the command palette.

When you are looking to bind new keys, it is often useful to use the command palette (ctrl-shift-p) to discover what commands are being listened for in a given focus context. Commands are “humanized” following a simple algorithm, so a command like `editor:fold-current-row` would appear as “Editor: Fold Current Row”.

“COMPOSED” COMMANDS

A common question is, “How do I make a single keybinding execute two or more commands?” There isn’t any direct support for this in Atom, but it can be achieved by creating a custom command that performs the multiple actions you desire and then creating a keybinding for that command. For example, let’s say I want to create a “composed” command that performs a Select Line followed by Cut. You could add the following to your `init.coffee`:

```
atom.commands.add 'atom-text-editor', 'custom:cut-line', ->
  editor = atom.workspace.getActiveTextEditor()
  editor.selectLinesContainingCursors()
  editor.cutSelectedText()
```

Then let’s say we want to map this custom command to `alt-ctrl-z`, you could add the following to your keymap:

```
'atom-text-editor':
  'alt-ctrl-z': 'custom:cut-line'
```

SPECIFICITY AND CASCADE ORDER

As is the case with CSS applying styles, when multiple bindings match for a single element, the conflict is resolved by choosing the most **specific** selector. If two matching selectors have the same specificity, the binding for the selector appearing later in the cascade takes precedence.

Currently, there’s no way to specify selector ordering within a single keymap, because JSON objects do not preserve order. We eventually plan to introduce a custom CSS-like file format for keymaps that allows for ordering within a single file. For now, we’ve opted to handle cases where selector ordering is critical by breaking the keymap into two separate files, such as `snippets-1.cson` and `snippets-2.cson`.

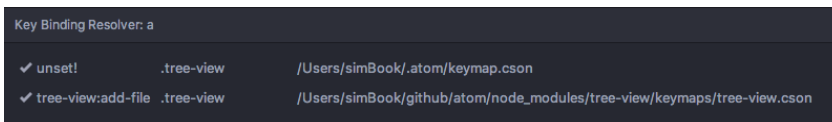
Removing Bindings

When the keymap system encounters a binding with the `unset!` directive as its command, it will treat the current element as if it had no key bindings matching

the current keystroke sequence and continue searching from its parent. If you want to remove a binding from a keymap you don't control, such as keymaps in Atom core or in packages, use the `unset!` directive.

For example, the following code removes the keybinding for `a` in the Tree View, which is normally used to trigger the `tree-view:add-file` command:

```
'.tree-view':
  'a': 'unset!'
```



Key Binding Resolver: a		
✓ unset!	.tree-view	/Users/simBook/.atom/keymap.cson
✓ tree-view:add-file	.tree-view	/Users/simBook/github/atom/node_modules/tree-view/keymaps/tree-view.cson

FIGURE 4-1

Keybinding resolver

Forcing Chromium's Native Keystroke Handling

If you want to force the native browser behavior for a given keystroke, use the `native!` directive as the command of a binding. This can be useful to enable the correct behavior in native input elements, for example. If you apply the `.native-key-bindings` class to an element, all the keystrokes typically handled by the browser will be assigned the `native!` directive.

Overloading Key Bindings

Occasionally, it makes sense to layer multiple actions on top of the same key binding. An example of this is the snippets package. Snippets are inserted by typing a snippet prefix such as `for` and then pressing `tab`. Every time `tab` is pressed, we want to execute code attempting to expand a snippet if one exists for the text preceding the cursor. If a snippet **doesn't** exist, we want `tab` to actually insert whitespace.

To achieve this, the snippets package makes use of the `.abortKeyBinding()` method on the event object representing the `snippets:expand` command.

```
# pseudo-code
editor.command 'snippets:expand', (e) =>
  if @cursorFollowsValidPrefix()
    @expandSnippet()
  else
    e.abortKeyBinding()
```

When the event handler observes that the cursor does not follow a valid prefix, it calls `e.abortKeyBinding()`, telling the keymap system to continue searching for another matching binding.

Step-by-Step: How Keydown Events are Mapped to Commands

- A keydown event occurs on a **focused** element.
- Starting at the focused element, the keymap walks upward towards the root of the document, searching for the most specific CSS selector that matches the current DOM element and also contains a keystroke pattern matching the keydown event.
- When a matching keystroke pattern is found, the search is terminated and the pattern's corresponding command is triggered on the current element.
- If `.abortKeyBinding()` is called on the triggered event object, the search is resumed, triggering a binding on the next-most-specific CSS selector for the same element or continuing upward to parent elements.
- If no bindings are found, the event is handled by Chromium normally.

Scoped Settings, Scopes and Scope Descriptors

Atom supports language-specific settings. You can soft wrap only Markdown files, or set the tab length to 4 in Python files.

Language-specific settings are a subset of something more general we call “scoped settings”. Scoped settings allow targeting down to a specific syntax token type. For example, you could conceivably set a setting to target only Ruby comments, only code inside Markdown files, or even only JavaScript function names.

Scope names in syntax tokens

Each token in the editor has a collection of scope names. For example, the aforementioned JavaScript function name might have the scope names `function` and `name`. An open paren might have the scope names `punctuation`, `parameters`, `begin`.

Scope names work just like CSS classes. In fact, in the editor, scope names are attached to a token's DOM node as CSS classes.

Take this piece of JavaScript:

```
function functionName() {
  console.log('Log it out');
}
```

In the dev tools, the first line's markup looks like this.

```
▼ <span class="source js">
  ▼ <span class="meta function js">
    <span class="storage type function js">function</span>
    <span class="entity name function js">functionName</span>
    <span class="punctuation definition parameters begin js">(</span>
    <span class="punctuation definition parameters end js">)</span>
  </span>
  <span class="meta brace curly js">{</span>
</span>
```

FIGURE 4-2

Markup

All the class names on the spans are scope names. Any scope name can be used to target a setting's value.

Scope Selectors

Scope selectors allow you to target specific tokens just like a CSS selector targets specific nodes in the DOM. Some examples:

```
'source.js' # selects all javascript tokens
'source.js .function.name' # selects all javascript function names
'.function.name' # selects all function names in any language
```

Config::set accepts a `scopeSelector`. If you'd like to set a setting for JavaScript function names, you can give it the js function name `scopeSelector`:

```
atom.config.set('source.js .function.name', 'my-package.my-setting', 'special value')
```

Scope Descriptors

A scope descriptor is an **Object** that wraps an Array of `String`s`. The Array describes a path from the root of the syntax tree to a token including *all* scope names for the entire path.

In our JavaScript example above, a scope descriptor for the function name token would be:

```
['source.js', 'meta.function.js', 'entity.name.function.js']
```

Config::get accepts a `scopeDescriptor`. You can get the value for your setting scoped to JavaScript function names via:

```
scopeDescriptor = ['source.js', 'meta.function.js', 'entity.name.function.js']
value = atom.config.get(scopeDescriptor, 'my-package.my-setting')
```

But, you do not need to generate scope descriptors by hand. There are a couple methods available to get the scope descriptor from the editor:

- **Editor::getRootScopeDescriptor** to get the language's descriptor. eg. `[".source.js"]`
- **Editor::scopeDescriptorForBufferPosition** to get the descriptor at a specific position in the buffer.
- **Cursor::getScopeDescriptor** to get a cursor's descriptor based on position. eg. if the cursor were in the name of the method in our example it would return `["source.js", "meta.function.js", "entity.name.function.js"]`

Let's revisit our example using these methods:

```
editor = atom.workspace.getActiveTextEditor()
cursor = editor.getLastCursor()
valueAtCursor = atom.config.get(cursor.getScopeDescriptor(), 'my-package.my-setting')
valueForLanguage = atom.config.get(editor.getRootScopeDescriptor(), 'my-package.my-setting')
```

Serialization in Atom

When a window is refreshed or restored from a previous session, the view and its associated objects are **deserialized** from a JSON representation that was stored during the window's previous shutdown. For your own views and objects to be compatible with refreshing, you'll need to make them play nicely with the serializing and deserializing.

Package Serialization Hook

Your package's main module can optionally include a `serialize` method, which will be called before your package is deactivated. You should return JSON, which will be handed back to you as an argument to `activate` next time it is called. In the following example, the package keeps an instance of `MyObject` in the same state across refreshes.


```

module.exports =
  activate: (state) ->
    @myObject =
      if state
        atom.deserializers.deserialize(state)
      else
        new MyObject("Hello")

  serialize: ->
    @myObject.serialize()

```

Serialization Methods

```

class MyObject
  atom.deserializers.add(this)

  @deserialize: ({data}) -> new MyObject(data)
  constructor: (@data) ->
  serialize: -> { deserializer: 'MyObject', data: @data }

```

.SERIALIZE()

Objects that you want to serialize should implement `.serialize()`. This method should return a serializable object, and it must contain a key named `deserializer` whose value is the name of a registered deserializer that can convert the rest of the data to an object. It's usually just the name of the class itself.

@DESERIALIZE(DATA)

The other side of the coin is the `deserialize` method, which is usually a class-level method on the same class that implements `serialize`. This method's job is to convert a state object returned from a previous call `serialize` back into a genuine object.

ATOM.DESERIALIZERS.ADD(KLASS)

You need to call the `atom.deserializers.add` method with your class in order to make it available to the deserialization system. Now you can call the global `deserialize` method with state returned from `serialize`, and your class's `deserialize` method will be selected automatically.

Versioning

```
class MyObject
  atom.deserializers.add(this)

  @version: 2
  @deserialize: (state) -> ...
  serialize: -> { version: @constructor.version, ... }
```

Your serializable class can optionally have a class-level `@version` property and include a `version` key in its serialized state. When deserializing, Atom will only attempt to call `deserialize` if the two versions match, and otherwise return undefined. We plan on implementing a migration system in the future, but this at least protects you from improperly deserializing old state.

Developing Node Modules

Atom contains a number of packages that are Node modules instead of Atom packages. If you want to make changes to the Node modules, for instance `atom-keymap`, you have to link them into the development environment differently than you would a normal Atom package.

Linking a Node Module Into Your Atom Dev Environment

Here are the steps to run a local version of a node module **not an apm** within Atom. We're using `atom-keymap` as an example:

```
$ git clone https://github.com/atom/atom-keymap.git
$ cd atom-keymap
$ npm install
$ npm link
$ apm rebuild # This is the special step, it makes the npm work with Atom's version
$ cd WHERE-YOU-CLONED-ATOM
$ npm link atom-keymap
$ atom # Should work!
```

After this, you'll have to `npm install` and `apm rebuild` when you make a change to the node module's code.

Interacting With Other Packages Via Services

Atom packages can interact with each other through versioned APIs called *services*. To provide a service, in your `package.json`, specify one or more version numbers, each paired with the name of a method on your package's main module:

```
{
  "providedServices": {
    "my-service": {
      "description": "Does a useful thing",
      "versions": {
        "1.2.3": "provideMyServiceV1",
        "2.3.4": "provideMyServiceV2",
      }
    }
  }
}
```

In your package's main module, implement the methods named above. These methods will be called any time a package is activated that consumes their corresponding service. They should return a value that implements the service's API.

```
module.exports =
  activate: -> # ...

  provideMyServiceV1: ->
    adaptToLegacyAPI(myService)

  provideMyServiceV2: ->
    myService
```

Similarly, to consume a service, specify one or more **version ranges**, each paired with the name of a method on the package's main module:

```
{
  "consumedServices": {
    "another-service": {
      "versions": {
        "^1.2.3": "consumeAnotherServiceV1",
        ">=2.3.4 <2.5": "consumeAnotherServiceV2",
      }
    }
  }
}
```

These methods will be called any time a package is activated that **provides** their corresponding service. They will receive the service object as an argument. You will usually need to perform some kind of cleanup in the event that the package providing the service is deactivated. To do this, return a `Disposable` from your service-consuming method:

```
{Disposable} = require 'atom'

module.exports =
  activate: -> # ...

  consumeAnotherServiceV1: (service) ->
    useService(adaptServiceFromLegacyAPI(service))
    new Disposable -> stopUsingService(service)

  consumeAnotherServiceV2: (service) ->
    useService(service)
    new Disposable -> stopUsingService(service)
```

Maintaining Your Packages

While publishing is, by far, the most common action you will perform when working with the packages you provide, there are other things you may need to do.

Unpublish a Version

If you mistakenly published a version of your package or perhaps you find a glaring bug or security hole, you may want to unpublish just that version of your package. For example, if your package is named `package-name` and the bad version of your package is `v1.2.3` then the command you would execute is:

```
apm unpublish package-name@1.2.3
```

This will remove just this particular version from the <https://atom.io> package registry. Anyone who has already downloaded this version will still have it, but it will no longer be available for installation by others.

Adding a Collaborator

Some packages get too big for one person. Sometimes priorities change and someone else wants to help out. You can let others help or create co-owners by **adding them as a collaborator** on the GitHub repository for your package.

Note: Anyone that has push access to your repository will have the ability to publish new versions of the package that belongs to that repository.

You can also have packages that are owned by a **GitHub organization**. Anyone who is a member of an organization's **team** which has push access to the package's repository will be able to publish new versions of the package.

Transferring Ownership

This is a permanent change. There is no going back!

If you want to hand off support of your package to someone else, you can do that by **transferring the package's repository** to the new owner.

Unpublish Your Package

It is important that you unpublish your package *before* deleting your repository. If you delete the repository first, you will lose access to the package and will not be able to recover it without assistance.

If you no longer want to support your package and cannot find anyone to take it over, you can unpublish your package from <https://atom.io>. For example, if your package is named `package-name` then the command you would execute is:

```
apm unpublish package-name
```

This will remove your package from the <https://atom.io> package registry. Anyone who has already downloaded a copy of your package will still have it and be able to use it, but it will no longer be available for installation by others.

Rename Your Package

If you need to rename your package for any reason, you can do so with one simple command – `apm publish --rename` changes the name field in your package's `package.json`, pushes a new commit and tag, and publishes your renamed package. Requests made to the previous name will be forwarded to the new name.

Once a package name has been used, it cannot be re-used by another package even if the original package is unpublished.

```
apm publish --rename new-package-name
```

Summary

You should now have a better understanding of some of the core Atom APIs and systems.

Upgrading to 1.0 APIs A

Atom is rapidly approaching 1.0. Much of the effort leading up to the 1.0 has been cleaning up APIs in an attempt to future proof, and make a more pleasant experience developing packages. If you have developed packages or syntaxes for Atom before the 1.0 API, you can find some tips on upgrading your work in this appendix.

Upgrading Your Package

This document will guide you through the large bits of upgrading your package to work with 1.0 APIs.

TL;DR

We've set deprecation messages and errors in strategic places to help make sure you don't miss anything. You should be able to get 95% of the way to an updated package just by fixing errors and deprecations. There are a couple of things you can do to get the full effect of all the errors and deprecations.

USE ATOM-SPACE-PEN-VIEWS

If you use any class from `require 'atom'` with a `$` or `View` in the name, add the `atom-space-pen-views` module to your package's `package.json` file's dependencies:

```
{
  "dependencies": {
    "atom-space-pen-views": "^2.0.3"
  }
}
```

Then run `apm install` in your package directory.

REQUIRE VIEWS FROM ATOM-SPACE-PEN-VIEWS

Anywhere you are requiring one of the following from `atom` you need to require them from `atom-space-pen-views` instead.

```
# require these from 'atom-space-pen-views' rather than 'atom'  
$  
$$  
$$$  
View  
TextView  
ScrollView  
SelectListView
```

So this:

```
# Old way  
{$, TextView, View, GitRepository} = require 'atom'
```

Would be replaced by this:

```
# New way  
{GitRepository} = require 'atom'  
{$, TextView, View} = require 'atom-space-pen-views'
```

RUN SPECS AND TEST YOUR PACKAGE

You wrote specs, right!? Here's where they shine. Run them with `cmd-shift-P`, and search for `run package specs`. It will show all the deprecation messages and errors.

UPDATE THE ENGINES FIELD IN PACKAGE.JSON

When you are deprecation free and all done converting, upgrade the `engines` field in your `package.json`:

```
{  
  "engines": {  
    "atom": ">=0.174.0 <2.0.0"  
  }  
}
```


EXAMPLES

We have upgraded all the core packages. Please see [this issue](#) for a link to all the upgrade PRs.

Deprecations

All of the methods in Atom core that have changes will emit deprecation messages when called. These messages are shown in two places: your **package specs**, and in **Deprecation Cop**.

SPECS

Just run your specs, and all the deprecations will be displayed in yellow.

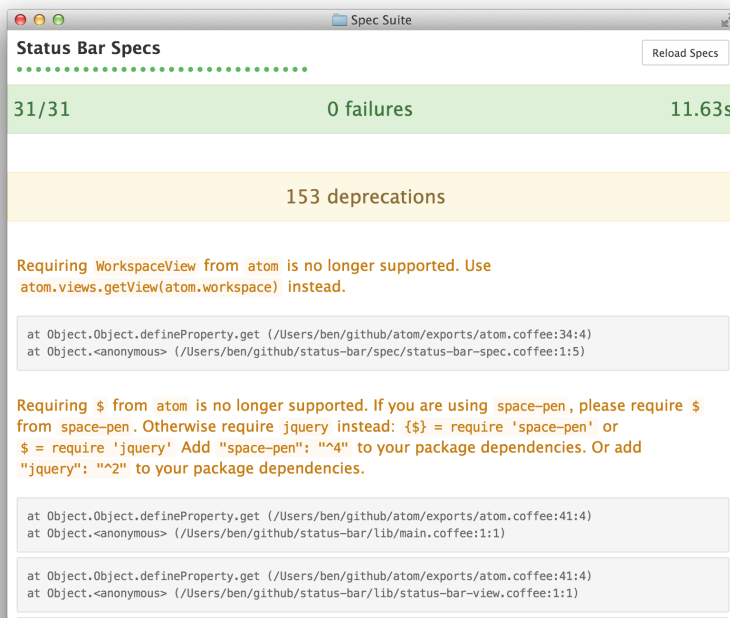


Figure 1-1.

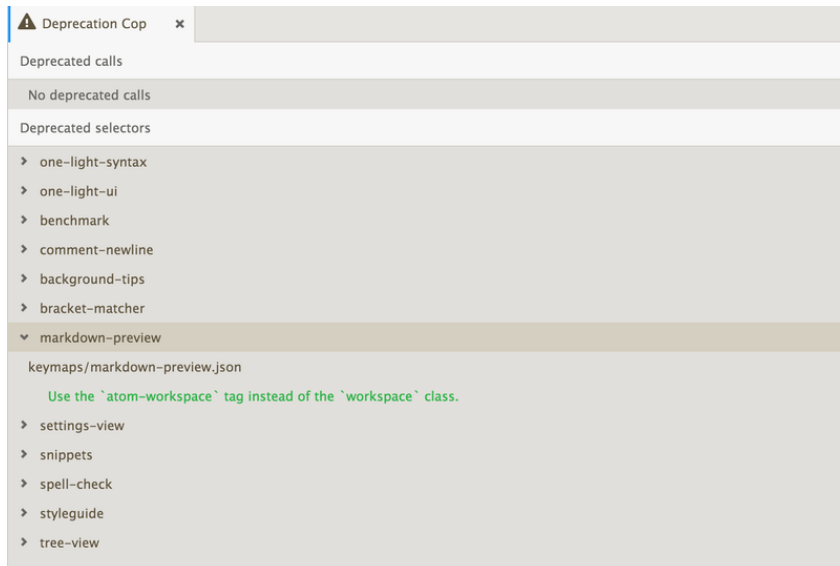
Deprecations in specs

DEPRECATION COP

Run an atom window in dev mode (`atom -d`) with your package loaded, and open Deprecation Cop (search for `deprecation` in the command palette). Deprecated methods will be appear in Deprecation Cop only after they have been called.

Figure 1-2.

Deprecation Cop



When deprecation cop is open, and deprecated methods are called, a Refresh button will appear in the top right of the Deprecation Cop interface. So exercise your package, then come back to Deprecation Cop and click the Refresh button.

Upgrading your Views

Previous to 1.0, views were baked into Atom core. These views were based on jQuery and space-pen. They looked something like this:

```
# The old way: getting views from atom
{$., TextEditorView, View} = require 'atom'

module.exports =
  class SomeView extends View
```

```

@content: ->
  @div class: 'find-and-replace', =>
    @div class: 'block', =>
      @subview 'myEditor', new TextEditorView(mini: true)
#...

```

THE NEW

require 'atom' no longer provides view helpers or jQuery. Atom core is now *view agnostic*. The preexisting view system is available from a new npm package: `atom-space-pen-views`.

`atom-space-pen-views` now provides jQuery, space-pen views, and Atom specific views:

```

# These are now provided by atom-space-pen-views
$
$$
$$$
View
TextEditorView
ScrollView
SelectListView

```

ADDING THE MODULE DEPENDENCIES

To use the new views, you need to specify the `atom-space-pen-views` module in your package's `package.json` file's dependencies:

```

{
  "dependencies": {
    "atom-space-pen-views": "^2.0.3"
  }
}

```

`space-pen` bundles jQuery. If you do not need `space-pen` or any of the views, you can require jQuery directly.

```

{
  "dependencies": {
    "jquery": "^2"
  }
}

```

CONVERTING YOUR VIEWS

Sometimes it is as simple as converting the requires at the top of each view page. I assume you read the *TL;DR* section and have updated all of your requires.

UPGRADING CLASSES EXTENDING ANY SPACE-PEN VIEW

afterAttach and beforeRemove updated

The afterAttach and beforeRemove hooks have been replaced with attached and detached and the semantics have changed.

afterAttach was called whenever the node was attached to another DOM node, even if that parent node wasn't present in the DOM. afterAttach also was called with a boolean indicating whether or not the element and its parents were on the DOM. Now the attached hook is *only* called when the node and all of its parents are actually on the DOM, and is not called with a boolean.

beforeRemove was only called when `$.fn.remove` was called, which was typically used when the node was completely removed from the DOM. The new detached hook is called whenever the DOM node is *detached*, which could happen if the node is being detached for reattachment later. In short, if beforeRemove is called the node is never coming back. With detached it might be attached again later.

```
# Old way
{View} = require 'atom'
class MyView extends View
  afterAttach: (onDom) ->
    #...

  beforeRemove: ->
    #...

# New way
{View} = require 'atom-space-pen-views'
class MyView extends View
  attached: ->
    # Always called with the equivalent of @afterAttach(true)!
    #...

  detached: ->
    #...
```

subscribe and subscribeToCommand methods removed

The subscribe and subscribeToCommand methods have been removed. See the [Eventing and Disposables](#) section for more info.

UPGRADING TO THE NEW TEXTEDITORVIEW

All of the atom-specific methods available on the `TextView` have been moved to the `TextEditor`, available via `TextView::getModel`. See the [TextView docs](#) and [TextEditor docs](#) for more info.

UPGRADING CLASSES EXTENDING SCROLLVIEW

The `ScrollView` has very minor changes.

You can no longer use `@off` to remove default behavior for `core:move-up`, `core:move-down`, etc.

```
# Old way to turn off default behavior
class ResultsView extends ScrollView
  initialize: (@model) ->
    super()
    # turn off default scrolling behavior from ScrollView
    @off 'core:move-up'
    @off 'core:move-down'
    @off 'core:move-left'
    @off 'core:move-right'
```

```
# New way to turn off default behavior
class ResultsView extends ScrollView
  initialize: (@model) ->
    disposable = super()
    # turn off default scrolling behavior from ScrollView
    disposable.dispose()
```

- Check out [an example](#) from find-and-replace.
- See the [docs](#) for all the options.

UPGRADING CLASSES EXTENDING SELECTLISTVIEW

Your `SelectListView` might look something like this:

```
# Old!
class CommandPaletteView extends SelectListView
  initialize: ->
    super()
    @addClass('command-palette overlay from-top')
```

```

    atom.workspaceView.command 'command-palette:toggle', => @toggle()

confirmed: ({name, jQuery}) ->
  @cancel()
  # do something with the result

toggle: ->
  if @hasParent()
    @cancel()
  else
    @attach()

attach: ->
  @storeFocusedElement()

  items = [] # TODO: build items
  @setItems(items)

  atom.workspaceView.append(this)
  @focusFilterEditor()

confirmed: ({name, jQuery}) ->
  @cancel()

```

This attaches and detaches itself from the dom when toggled, canceling magically detaches it from the DOM, and it uses the classes `overlay` and `from-top`.

The new `SelectListView` no longer automatically detaches itself from the DOM when cancelled. It's up to you to implement whatever cancel behavior you want. Using the new APIs to mimic the semantics of the old class, it should look like this:

```

# New!
class CommandPaletteView extends SelectListView
  initialize: ->
    super()
    # no more need for the `overlay` and `from-top` classes
    @addClass('command-palette')
    atom.commands.add 'atom-workspace', 'command-palette:toggle', => @toggle()

  # You need to implement the `cancelled` method and hide.
  cancelled: ->
    @hide()

  confirmed: ({name, jQuery}) ->
    @cancel()
    # do something with the result

```

```

toggle: ->
  # Toggling now checks panel visibility,
  # and hides / shows rather than attaching to / detaching from the DOM.
  if @panel?.isVisible()
    @cancel()
  else
    @show()

show: ->
  # Now you will add your select list as a modal panel to the workspace
  @panel ?= atom.workspace.addModalPanel(item: this)
  @panel.show()

  @storeFocusedElement()

  items = [] # TODO: build items
  @setItems(items)

  @focusFilterEditor()

hide: ->
  @panel?.hide()

```

- And check out the **conversion of CommandPaletteView** as a real-world example.
- See the **SelectListView docs** for all options.

Using the model layer rather than the view layer

The API no longer exposes any specialized view objects or view classes. `atom.workspaceView`, and all the view classes: `WorkspaceView`, `EditorView`, `PaneView`, etc. have been globally deprecated.

Nearly all of the atom-specific actions performed by the old view objects can now be managed via the model layer. For example, here's adding a panel to the interface using the `atom.workspace` model instead of the `workspaceView`:

```

# Old!
div = document.createElement('div')
atom.workspaceView.appendToTop(div)

# New!
div = document.createElement('div')
atom.workspace.addTopPanel(item: div)

```

For actions that still require the view, such as dispatching commands or munging css classes, you'll access the view via the `atom.views.getView()`

method. This will return a subclass of `HTMLElement` rather than a jQuery object or an instance of a deprecated view class (e.g. `WorkspaceView`).

```
# Old!
workspaceView = atom.workspaceView
editorView = workspaceView.getActiveEditorView()
paneView = editorView.getPaneView()

# New!
# Generally, just use the models
workspace = atom.workspace
editor = workspace.getActiveTextEditor()
pane = editor.getPane()

# If you need views, get them with `getView`
workspaceElement = atom.views.getView(atom.workspace)
editorElement = atom.views.getView(editor)
paneElement = atom.views.getView(pane)
```

Updating Specs

`atom.workspaceView`, the `WorkspaceView` class and the `EditorView` class have been deprecated. These two objects are used heavily throughout specs, mostly to dispatch events and commands. This section will explain how to remove them while still retaining the ability to dispatch events and commands.

REMOVING WORKSPACEVIEW REFERENCES

`WorkspaceView` has been deprecated. Everything you could do on the view, you can now do on the `Workspace` model.

Requiring `WorkspaceView` from `atom` and accessing any methods on it will throw a deprecation warning. Many specs lean heavily on `WorkspaceView` to trigger commands and fetch `EditorView` objects.

Your specs might contain something like this:

```
# Old!
{WorkspaceView} = require 'atom'
describe 'FindView', ->
  beforeEach ->
    atom.workspaceView = new WorkspaceView()
```

Instead, we will use the `atom.views.getView()` method. This will return a plain `HTMLElement`, not a `WorkspaceView` or jQuery object.


```
# New!
describe 'FindView', ->
  workspaceElement = null
  beforeEach ->
    workspaceElement = atom.views.getView(atom.workspace)
```

ATTACHING THE WORKSPACE TO THE DOM

The workspace needs to be attached to the DOM in some cases. For example, view hooks only work (`attached()` on `View`, `attachedCallback()` on custom elements) when there is a descendant attached to the DOM.

You might see this in your specs:

```
# Old!
atom.workspaceView.attachToDom()
```

Change it to:

```
# New!
jasmine.attachToDOM(workspaceElement)
```

REMOVING EDITORVIEW REFERENCES

Like `WorkspaceView`, `EditorView` has been deprecated. Everything you needed to do on the view you are now able to do on the `TextEditor` model.

In many cases, you will not even need to get the editor's view anymore. Any of those instances should be updated to use the `TextEditor` instance instead. You should really only need the editor's view when you plan on triggering a command on the view in a spec.

Your specs might contain something like this:

```
# Old!
describe 'Something', ->
  [editorView] = []
  beforeEach ->
    editorView = atom.workspaceView.getActiveView()
```

We're going to use `atom.views.getView()` again to get the editor element. As in the case of the `workspaceElement`, `getView` will return a subclass of `HTMLElement` rather than an `EditorView` or `jQuery` object.

```
# New!
describe 'Something', ->
  [editor, editorElement] = []
  beforeEach ->
```

```
editor = atom.workspace.getActiveTextEditor()
editorElement = atom.views.getView(editor)
```

DISPATCHING COMMANDS

Since the `editorElement` objects are no longer jQuery objects, they no longer support `trigger()`. Additionally, Atom has a new command dispatcher, `atom.commands`, that we use rather than commandeering jQuery's `trigger` method.

From this:

```
# Old!
workspaceView.trigger 'a-package:toggle'
editorView.trigger 'find-and-replace:show'
```

To this:

```
# New!
atom.commands.dispatch workspaceElement, 'a-package:toggle'
atom.commands.dispatch editorElement, 'find-and-replace:show'
```

Eventing and Disposables

A couple large things changed with respect to events:

1. All model events are now exposed as event subscription methods that return **Disposable** objects
2. The `subscribe()` method is no longer available on space-pen View objects
3. An `Emitter` is now provided from `require 'atom'`

CONSUMING EVENTS

All events from the Atom API are now methods that return a **Disposable** object, on which you can call `dispose()` to unsubscribe.

```
# Old!
editor.on 'changed', ->

# New!
disposable = editor.onDidChange ->

# You can unsubscribe at some point in the future via `dispose()`
disposable.dispose()
```

Deprecation warnings will guide you toward the correct methods.

Using a CompositeDisposable

You can group multiple disposables into a single disposable with a CompositeDisposable.

```
{CompositeDisposable} = require 'atom'

class Something
  constructor: ->
    editor = atom.workspace.getActiveTextEditor()
    @disposables = new CompositeDisposable
    @disposables.add editor.onDidChange ->
    @disposables.add editor.onDidChangePath ->

  destroy: ->
    @disposables.dispose()
```

REMOVING VIEW::SUBSCRIBE AND SUBSCRIBER::SUBSCRIBE CALLS

There were a couple permutations of subscribe(). In these examples, a CompositeDisposable is used as it will commonly be useful where conversion is necessary.

subscribe(unsubscribable)

This one is very straight forward.

```
# Old!
@subscribe editor.on 'changed', ->

# New!
disposables = new CompositeDisposable
disposables.add editor.onDidChange ->
```

subscribe(modelObject, event, method)

When the modelObject is an Atom model object, the change is very simple. Just use the correct event method, and add it to your CompositeDisposable.

```
# Old!
@subscribe editor, 'changed', ->

# New!
disposables = new CompositeDisposable
disposables.add editor.onDidChange ->
```

subscribe(jQueryObject, selector(optional), event, method)

Things are a little more complicated when subscribing to a DOM or jQuery element. Atom no longer provides helpers for subscribing to elements. You can use jQuery or the native DOM APIs, whichever you prefer.

```
# Old!
@subscribe $(window), 'focus', ->

# New!
{Disposable, CompositeDisposable} = require 'atom'
disposables = new CompositeDisposable

# New with jQuery
focusCallback = ->
$(window).on 'focus', focusCallback
disposables.add new Disposable ->
  $(window).off 'focus', focusCallback

# New with native APIs
focusCallback = ->
window.addEventListener 'focus', focusCallback
disposables.add new Disposable ->
  window.removeEventListener 'focus', focusCallback
```

PROVIDING EVENTS: USING THE EMITTER

You no longer need to require `emissary` to get an emitter. We now provide an `Emitter` class from `require 'atom'`. We have a specific pattern for use of the `Emitter`. Rather than mixing it in, we instantiate a member variable, and create explicit subscription methods. For more information see the **Emitter docs**.

```
# New!
{Emitter} = require 'atom'

class Something
  constructor: ->
    @emitter = new Emitter

  destroy: ->
    @emitter.dispose()

  onDidChange: (callback) ->
    @emitter.on 'did-change', callback

  methodThatFiresAChange: ->
    @emitter.emit 'did-change', {data: 2}
```

```
# Using the evented class
something = new Something
something.onDidChange (eventObject) ->
  console.log eventObject.data # => 2
something.methodThatFiresAChange()
```

Subscribing To Commands

`$.fn.command` and `View::subscribeToCommand` are no longer available. Now we use `atom.commands.add`, and collect the results in a `CompositeDisposable`. See **the docs** for more info.

```
# Old!
atom.workspaceView.command 'core:close core:cancel', ->

# When inside a View class, you might see this
@subscribeToCommand 'core:close core:cancel', ->

# New!
@disposables.add atom.commands.add 'atom-workspace',
  'core:close': ->
  'core:cancel': ->

# You can register commands directly on individual DOM elements in addition to
# using selectors. When in a View class, you should have a `@element` object
# available. `@element` is a plain HTMLElement object
@disposables.add atom.commands.add @element,
  'core:close': ->
  'core:cancel': ->
```

Upgrading your stylesheet's selectors

Many selectors have changed, and we have introduced the **Shadow DOM** to the editor. See the **Upgrading Your UI Theme And Package Selectors guide** for more information in upgrading your package stylesheets.

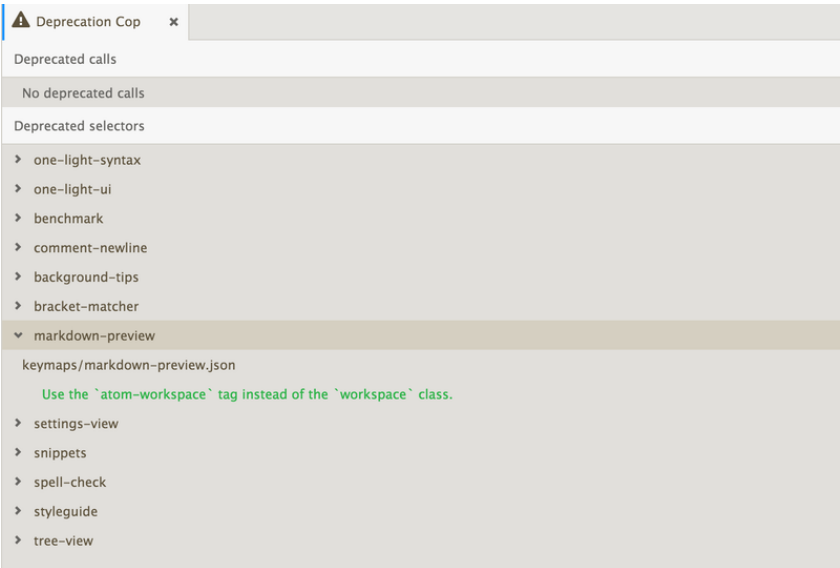
Upgrading Your UI Theme Or Package Selectors

In addition to changes in Atom's scripting API, we'll also be making some breaking changes to Atom's DOM structure, requiring style sheets and keymaps in both packages and themes to be updated.

Deprecation Cop

Deprecation cop will list usages of deprecated selector patterns to guide you. You can access it via the command palette (cmd-shift-p, then search for Deprecation). It breaks the deprecations down by package:

Figure 1-3.
Deprecation Cop



Custom Tags

Rather than adding classes to standard HTML elements to indicate their role, Atom now uses custom element names. For example, `<div class="workspace">` has now been replaced with `<atom-workspace>`. Selectors should be updated accordingly. Note that tag names have lower specificity than classes in CSS, so you'll need to take care in converting things.

Old Selector	New Selector
.editor	atom-text-editor
.editor.mini	atom-text-editor[mini]
.workspace	atom-workspace

Old Selector	New Selector
<code>.horizontal</code>	<code>atom-workspace-axis.horizontal</code>
<code>.vertical</code>	<code>atom-workspace-axis.vertical</code>
<code>.pane-container</code>	<code>atom-pane-container</code>
<code>.pane</code>	<code>atom-pane</code>
<code>.tool-panel</code>	<code>atom-panel</code>
<code>.panel-top</code>	<code>atom-panel.top</code>
<code>.panel-bottom</code>	<code>atom-panel.bottom</code>
<code>.panel-left</code>	<code>atom-panel.left</code>
<code>.panel-right</code>	<code>atom-panel.right</code>
<code>.overlay</code>	<code>atom-panel.modal</code>

Supporting the Shadow DOM

Text editor content is now rendered in the shadow DOM, which shields it from being styled by global style sheets to protect against accidental style pollution. For more background on the shadow DOM, check out the **Shadow DOM 101** on HTML 5 Rocks. If you need to style text editor content in a UI theme, you'll need to circumvent this protection for any rules that target the text editor's content. Some examples of the kinds of UI theme styles needing to be updated:

- Highlight decorations
- Gutter decorations
- Line decorations
- Scrollbar styling
- Anything targeting a child selector of `.editor`

During a transition phase, it will be possible to enable or disable the text editor's shadow DOM in the settings, so themes will need to be compatible with both approaches.

SHADOW DOM SELECTORS

Chromium provides two tools for bypassing shadow boundaries, the `::shadow` pseudo-element and the `/deep/` combinator. For an in-depth explanation of styling the shadow DOM, see the **Shadow DOM 201** article on HTML 5 Rocks.

`::shadow`

The `::shadow` pseudo-element allows you to bypass a single shadow root. For example, say you want to update a highlight decoration for a linter package. Initially, the style looks as follows:

```
// Without shadow DOM support
atom-text-editor .highlight.my-linter {
  background: hotpink;
}
```

In order for this style to apply with the shadow DOM enabled, you will need to add a second selector with the `::shadow` pseudo-element. You should leave the original selector in place so your theme continues to work with the shadow DOM disabled during the transition period.

```
// With shadow DOM support
atom-text-editor .highlight.my-linter,
atom-text-editor::shadow .highlight.my-linter {
  background: hotpink;
}
```

Check out the **find-and-replace** package for another example of using `::shadow` to pierce the shadow DOM.

`/deep/`

The `/deep/` combinator overrides **all** shadow boundaries, making it useful for rules you want to apply globally such as scrollbar styling. Here's a snippet containing scrollbar styling for the Atom Dark UI theme before shadow DOM support:

```
// Without shadow DOM support
.scrollbars-visible-always {
  ::-webkit-scrollbar {
    width: 8px;
    height: 8px;
  }

  ::-webkit-scrollbar-track,
  ::-webkit-scrollbar-corner {
    background: @scrollbar-background-color;
  }
}
```



```

::-webkit-scrollbar-thumb {
  background: @scrollbar-color;
  border-radius: 5px;
  box-shadow: 0 0 1px black inset;
}
}

```

To style scrollbars even inside of the shadow DOM, each rule needs to be prefixed with `/deep/`. We use `/deep/` instead of `::shadow` because we don't care about the selector of the host element in this case. We just want our styling to apply everywhere.

```

// With shadow DOM support using /deep/
.scrollbars-visible-always {
  /deep/ ::-webkit-scrollbar {
    width: 8px;
    height: 8px;
  }

  /deep/ ::-webkit-scrollbar-track,
  /deep/ ::-webkit-scrollbar-corner {
    background: @scrollbar-background-color;
  }

  /deep/ ::-webkit-scrollbar-thumb {
    background: @scrollbar-color;
    border-radius: 5px;
    box-shadow: 0 0 1px black inset;
  }
}
}

```

CONTEXT-TARGETED STYLE SHEETS

The selector features discussed above allow you to target shadow DOM content with specific selectors, but Atom also allows you to target a specific shadow DOM context with an entire style sheet. The context into which a style sheet is loaded is based on the file name. If you want to load a style sheet into the editor, name it with the `.atom-text-editor.less` or `.atom-text-editor.css` extensions.

```

my-ui-theme/
  styles/
    index.less           # loaded globally
    index.atom-text-editor.less # loaded in the text editor shadow DOM

```

Check out this **style sheet** from the `decoration-example` package for an example of context-targeting.

Inside a context-targeted style sheet, there's no need to use the `::shadow` or `/deep/` expressions. If you want to refer to the element containing the shadow root, you can use the `::host` pseudo-element.

During the transition phase, style sheets targeting the `atom-text-editor` context will **also** be loaded globally. Make sure you update your selectors in a way that maintains compatibility with the shadow DOM being disabled. That means if you use a `::host` pseudo element, you should also include the same style rule matches against `atom-text-editor`.

Upgrading Your Syntax Theme

Text editor content is now rendered in the shadow DOM, which shields it from being styled by global style sheets to protect against accidental style pollution. For more background on the shadow DOM, check out the **Shadow DOM 101** on HTML 5 Rocks.

Syntax themes are specifically intended to style only text editor content, so they are automatically loaded directly into the text editor's shadow DOM when it is enabled. This happens automatically when the theme's `package.json` contains a `theme: "syntax"` declaration, so you don't need to change anything to target the appropriate context.

When theme style sheets are loaded into the text editor's shadow DOM, selectors intended to target the editor from the **outside** no longer make sense. Styles targeting the `.editor` and `.editor-colors` classes instead need to target the `::host` pseudo-element, which matches against the containing `atom-text-editor` node. Check out the **Shadow DOM 201** article for more information about the `::host` pseudo-element.

Here's an example from Atom's light syntax theme. Note that the `atom-text-editor` selector intended to target the editor from the outside has been retained to allow the theme to keep working during the transition phase when it is possible to disable the shadow DOM.

```
atom-text-editor, ::host { /* ::host added */
  background-color: @syntax-background-color;
  color: @syntax-text-color;

  .invisible-character {
    color: @syntax-invisible-character-color;
  }
  /* more nested selectors... */
}
```

Index