A Using the Bookware

Frances Allen (1932–) received the 2006 Turing Award for pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.

All the things I do are of a piece. I'm exploring the edges, finding new ways of doing things. It keeps me very, very engaged.

—Fran Allen, from Computer History Museum Fellow Award Plaque, 2000

Concepts

Not only is this book all about creating SaaS: it also relies heavily on SaaS, as well as IaaS (***infrastructure as a service***) and PaaS (***platform as a service***), all of which are models of ***cloud computing***.

This appendix describes the cloud technologies that not only simplify your life as a student, but also are essential parts of the ecosystem you will use when deploying your real SaaS apps. As of this writing, all of these cloud services offer a zero-cost usage tier that is sufficient for doing the work in this book. In particular, we cover four "building blocks" of SaaS for developers:

- Cloud-based ***Integrated Development Environments***(IDEs) let you get started in minutes and access your development tools from anywhere via a browser. We recommend and provide setup instructions for the popular Cloud9[1] IDE.
- GitHub[2] is a SaaS site that lets you back up your ***version-controlled*** projects as well as collaborate on them with other developers.
- Heroku[3] is a ***platform as a service*** provider where you can deploy your Rails apps.
- ***Secure Shell*** is a widely-used protocol that allows secure access to remote services by using a cryptographic keypair rather than a password. We use it to access most SaaS services, including GitHub and Heroku.

Linux, a ubiquitous open-source implementation of the highly influential Unix operating system, forms the basis of of virtually all non-Microsoft cloud-based development tools, including the ones we describe here. Unix was created by programmers for programmers, and its design choices and the resulting "programmer experience" are prominent in developer tools even on non-Unix platforms. Many of the "survival skills" you must master therefore have their roots in the Unix design philosophy and culture.

A.1 General Guidance: Read, Ask, Search, Post!

Open source software changes rapidly. Despite our attempts to keep the book and examples up-to-date, it is possible that by the time you read this, some of the tools' behaviors will have changed slightly and some of the examples might require minor modifications to work correctly. Although we take steps in this book to minimize the pain, such as using Test-Driven Development (Chapter **??**) to catch problems quickly and providing scripts to preinstall all the gems used in the examples, errors *will* occur. You can react most productively when something goes unexpectedly wrong by remembering the acronym **RASP**: Read, Ask, Search, Post.

Read the error message. Error messages can look disconcertingly long, but a long error message is often your friend because it gives a strong hint of the problem. There will be places to look in the online information associated with the class given the error message.

Ask a coworker. If you have friends in the class, or have instant messaging enabled, put the message out there.

Search for the error message. You'd be amazed at how often experienced developers deal with an error by using a search engine such as Google or a programmers' forum such as StackOverflow[4] to look up key words or key phrases in the error message.

Post a question on a site like StackOverflow[5] (*after* searching to see if a similar question has been asked!), sites that specialize in helping out developers and allow you to vote for the most helpful answers to particular questions so that they eventually percolate to the top of the answer list.

A.2 Overview of the Bookware

The bookware consists of three parts.

The first is a uniform development environment including all the tools referenced in the book. This environment consists of a Cloud9 ***Integrated Development Environment*** (IDE) configured using a script we provide that installs all the necessary components.

The second comprises a set of excellent SaaS sites aimed at developers: GitHub[6], Heroku[7], and Pivotal Tracker[8].

The third is supplementary material connected to the book, which is free whether you've purchased the book or not:

- The book's web site (`http://saasbook.info`[9]) contains the latest errata for each book version, links to supplementary material online, a bug reporting mechanism if you find errors, and high-resolution renderings of the figures and tables in case you have trouble reading them on your ebook reader
- GitHub Gists contain syntax-highlighted, copy-and-pastable code excerpts for every example in the book.
- Screencasts available on YouTube and referenced throughout the book illustrate various tools and demos in action.

A.3 Development Using the Cloud9 IDE

When the Alpha Edition of this book became available in 2012, a major shift to cloud computing was already well underway. Even applications that had been "staples" of shrink-wrapped software, such as Microsoft Word and Quicken, were either being challenged by in-browser cloud-based rivals such as Google Docs or supplemented by cloud-based alternatives such as Quicken Online. In 2016, the migration to cloud-based apps has even overtaken Integrated Development Environments (IDEs). A typical cloud-based IDE provides access to a virtual machine running some variant of Unix, hosted on an Infrastructure as a Service (IaaS) such as Amazon Web Services (AWS), and accessed via a rich browser-based user interface.

Cloud-based IDEs have several advantages: you can use an inexpensive laptop but still develop on a fast and powerful server-class computer; you can work anywhere that you have an Internet-connected browser, even if you don't carry a laptop computer; and some cloud-based IDEs even allow multiple developers to use the same VM simultaneously from their own computers, facilitating pair programming and other peer activities such as helping a colleague debug their code. The chief disadvantage of a cloud-based IDE compared with a shrink-wrapped IDE is that you must be connected to the Internet to use it, but many modern cloud-based IDEs deliver a reasonable user experience even over a mobile phone data connection.

The bookware for this book is designed to be installed on the Cloud9[10] IDE. The book's website, `saasbook.info`, contains a "Bookware" tab with the instructions to configure your C9 environment, starting from a "bare bones" Linux image and running a script we provide. You can also use this script as a starting point if you prefer to install the software on your own computer, but you must be familiar with Unix command line utilities to install the bookware locally; there is no GUI.

■ ***Elaboration: Free and Open Source Software***

Linux was originally created by Finnish programmer Linus Torvalds, who wanted to create a free and full-featured version of the famous Unix operating system for his own use. The GNU project was started by Richard Stallman, creator of the Emacs editor and founder of the Free Software Foundation (which stewards GNU), an illustrious developer with very strong opinions about the role of open source software. Both Linux and GNU are constantly being improved by contributions from thousands of collaborators worldwide; in fact, Torvalds later created Git to manage this large-scale collaboration. Despite the apparent lack of centralized authority in their development, the robustness of GNU and Linux compare favorably to proprietary software developed under a traditional centralized model. This phenomenon is explored in Eric Raymond's *The Cathedral and the Bazaar*[11], which some consider the seminal manifesto of the Free and Open Source Software (FOSS) movement.

A.4 Working With Code: Editors

Although Cloud9 and other IDEs provide a GUI for many tasks, in this book we use command-line tools for all tasks, for three reasons. First, command line tools work in any Unix-based IDE. Second, we place heavy emphasis in the book on automation to avoid mistakes and improve productivity; GUI tasks often cannot be automated, whereas command line tools can be composed into scripts, an approach central to the Unix philosophy. Third, understanding what tools are involved in each aspect of development helps roll back the "magic

curtain" of IDE GUIs. We believe this is helpful when learning a new system because if something goes wrong while using the GUI, to find the problem you need some understanding of how the GUI actually does the tasks.

Following this philosophy, you may be considering installing and learning a standalone code editor, either as an alternative to Cloud9's built-in editor or because you want to develop on your own computer without installing an IDE. If so, you will save yourself a great deal of grief by working with an editor that supports syntax highlighting and automatic indentation for the language you use, as Cloud9's built-in editor does. Two of the most popular editors for code and text in the Unix world are `vim` and `emacs`; both can be started from the command line. `vim` stands for "*vi im*proved," since it began as a much-enhanced version of the early Unix editor ***vi***, written in 1976 by Unix legend, Sun co-founder, and Berkeley alumnus ***Bill Joy***. This collection[12] of tutorials and screencasts introduces some of `vim`'s features. ***Emacs*** is the granddaddy of customizable editors and one of the earliest creations of the illustrious ***Richard Stallman***. The canonical tutorial[13] is provided by the Free Software Foundation, though many others are available.

Whatever you do, don't try to use a word processor such as Microsoft Word as a code editor. Many word processors "helpfully" convert regular quotes (") to "smart quotes," sequences of hyphens (`--`) to em-dashes (—), and other conversions that will make your code incorrect, cause syntax errors, and bring you grief. Don't do it.

A.5 Getting Started With Secure Shell (ssh)

`sh` was written by Steve Bourne in 1977 to replace Unix co-creator Ken Thompson's original shell. `bash` is a portable and compatible GNU replacement for `sh` whose name stands for ***Bourne-Again Shell***.

The ***shell*** is the Unix program that lets you type commands and write scripts to automate simple tasks, and before the widespread adoption of GUIs, the shell was the only way to interact with a Unix system. When Unix was born, there was no Internet; users could only run a shell by logging in from a ***dumb terminal*** connected physically to the computer. By 1983 the Internet had reached many universities and companies, so a new tool called ***Remote Shell*** or `rsh` appeared that allowed you to login or run commands on an Internet-connected remote computer on which you had an account. Here is an example of using `rsh` from the command line:

https://gist.github.com/caa3b8754ed07b316a47

```
1 rsh -l fox eecs.berkeley.edu ps -ef
```

rsh first appeared in version 4.2 of the Berkeley Software Distribution (BSD), the open-source implementation of Unix created at UC Berkeley.

This command would attempt to login as user `fox` on the computer `eecs.berkeley.edu`, run the command `ps -ef` (which gives information about which applications are running on that computer), and print the output locally. Omitting `ps -ef` would establish an interactive shell session on the remote computer.

But `rsh` is insecure: access to the remote computer usually required transmitting your password unencrypted or "in the clear" over the Internet, leaving it vulnerable to "sniffer" programs eavesdropping on the network to harvest passwords. In 1995, Tatu Ylönen at the Helsinki University of Technology developed ***Secure Shell*** or `ssh` as a secure "drop-in replacement" for `rsh`. As with `rsh`, once the connection to the remote computer is established you can either run an interactive shell or run arbitrary commands whose output is delivered securely back to your computer over the encrypted connection; in the latter case, we sometimes say the data is ***tunnelled*** over `ssh`. But rather than relying on a password, `ssh` relies on a keypair and key exchange using the same techniques and algorithms as SSL/TLS (Section **??**), so your private key never leaves your computer.

OpenSSL is an open-source volunteer-maintained library used by `ssh` and many SSL/TLS implementations. Fortunately, `ssh` doesn't use the part of OpenSSL containing the catastrophic

Because `ssh` is secure, ubiquitous, and doesn't require exposing your password or any

other secrets, many services rely on it for remote access, either as the default method (GitHub) or the only method (Heroku). Private and public keys come in pairs, and both halves are important. If you lose the private key paired to a given public key, any resources that relied on your possession of that key will become *irrevocably inaccessible forever.* If you lose the public key paired to a given private key, you *may* be able to retrieve a copy of it from one of the other services to which you still have access—though some services don't even allow you to view public keys you've uploaded, for added security. So treat your keypairs like a passport: personal, valuable, and long-lived.

Thus a key step (get it?) in preparing your development environment is ensuring that you have a keypair whose private half is on your development computer and whose public half is added to all the services you want to access while developing, such as GitHub for managing code or Heroku for deploying. When you create an account on Cloud9, a keypair is created for you; you can see its public part by going to the homepage for your Cloud9 account (*not* for a specific workspace), clicking on the gear icon in the upper right corner to see your Account Settings, and then clicking "SSH Keys" in the left-hand navigation bar. Your public key will appear in a text box.

You can then follow GitHub's excellent instructions[14] for adding this public key to your GitHub account, so that you can access GitHub from within Cloud9. (We will cover GitHub basics in Section A.7.)

Some people use different `ssh` keypairs for different services, to avoid putting all their eggs in one basket in case one private key is compromised. `help.github.com` has tutorials on this topic as well as on creating a new keypair, if you're not using Cloud9.

A.6 Getting Started With Git for Version Control

Version control, also called source code control or software configuration management (SCM), is the process of keeping track of the history of changes to a set of files. It can tell who made each change and when, reconstruct one or more files as they existed at some point in the past, or selectively combine changes made by different people. A version control system (VCS) is a tool that helps manage this process. For individual developers, SCM provides a timestamped and annotated history of changes to the project and an easy way to undo changes that introduce bugs. Chapter **??** discusses the many additional benefits of SCM for small teams.

SCM or VCS? Confusingly, the abbreviations SCM and VCS are often used interchangeably.

We will be using Git for version control. Cloud-based Git hosting services like GitHub, while not required for Git, are highly desirable because they enable small teams to collaborate conveniently (as Chapter **??** describes) and give individual developers a place to back up their code. This section covers the basics of Git. The next section covers basic setup instructions for GitHub, though other cloud-based Git services are available as well.

Linus Torvalds invented Git to assist with version control on the Linux project. You should read this section even if you've used other VCSs like Subversion, as Git's conceptual model is quite different.

Like all version control systems, a key concept in Git is the project ***repository***, usually shortened to ***repo***, which holds the complete change history of some set of files that make up a project. To start using Git to track a project, you first `cd` to the project's top-level directory and use the command `git init`, which initializes an empty repo based in that directory. ***Tracked files*** are those that are a permanent part of the repo, so their revision information is maintained and they are backed up; `git add` is used to add a file to the set of tracked files. Not every project file needs to be tracked—for example, intermediate files created automatically as part of the development process, such as log files, are usually untracked.

Screencast A.6.1 illustrates the basic Git workflow. When you start a new project, `git init` sets up the project's root directory as a Git repo. As you create files in your project, for each new file you use `git add` *filename* to cause the new file to be tracked by Git. When you reach a point where you're happy with the current state of the project, you ***commit***

the changes: Git prepares a list of all of the changes that will be part of this commit, and opens that list in an editor so you can add a descriptive comment. Which editor to use is determined by a ***configuration setting***, as described below. Committing causes a snapshot of the tracked files to be recorded permanently along with the comments. This snapshot is assigned a ***commit ID***, a 40-digit hexadecimal number that, surprisingly, is unique in the universe (not just within this Git repo, but across all repos); an example might be `1623f899bda026eb9273cd93c359326b47201f62`. This commit ID is the canonical way to refer to the state of the project at that point in time, but as we'll see, Git provides more convenient ways to refer to a commit besides the cumbersome commit ID. One common way is to specify a prefix of the commit that is unique within this repo, such as `1623f8` for the example above.

The *SHA-1* algorithm is used to compute the 40-digit one-way hash of a representation of the entire tree representing the project at that point in time.

When you commit, you *must* provide a nonempty comment describing your changes, the so-called *commit message*. When you use the `git commit` command, Git opens whichever text editor is specified by the Git configuration so that you can edit this message. On Cloud9, the default is to use the somewhat quirky `nano` editor, but if you prefer to use Cloud9's built-in editor for your commit messages, make the following one-time change to Git's configuration to do so:

https://gist.github.com/56928ffcabb063cc86a30e2402c5a62b

```
1 git config --global core.editor 'c9 open --wait'
```

`--global` specifies that this option should apply to *all* your Git operations in *all* repos. (Most Git configuration variables can also be set on a per-repo basis.)

Screencast A.6.1: Basic Git flow for a single developer.
`http://youtu.be/rhvrUTP4s2c`
In this simple workflow, `git init` is used to start tracking a project with Git, `git add` and `git commit` are used to add and commit two files. One file is then modified, and when `git status` shows that a tracked file has some changes, `git diff` is used to preview the changes that would be committed. Finally `git commit` is used again to commit the new changes, and `git diff` is used to show the differences between the two committed versions of one of the files, showing that `git diff` can either compare two commits of a file or compare the current state of a file with some previous commit.

It's important to remember that while `git commit` permanently records a snapshot of the current repo state that can be reconstructed at any time in the future, it does *not* create a backup copy of the repo anywhere else, nor make your changes accessible to fellow developers. The next section describes how to use a cloud-based Git hosting service for those purposes.

■ ***Elaboration: Add, commit, and the Git index***

The simplified explanation of Git above omits discussion of the *index*, a staging area for changes to be committed. `git add` is used not only to add a new file to the project, but also to stage an existing file's state for committing. So if Alice modifies *existing* file `foo.rb`, she would need to `git add foo.rb` to cause her changes to be committed on the next `git commit`. The reason for separating the steps is that `git add` snapshots the file immediately, so even if the `commit` occurs later, the version that is committed corresponds to the file's state *at the time of* `git add`. (If you make subsequent changes to the file, you should use `git add` again to get those changes into the index.) We simplified the discussion by using `-a` option to `git commit`, which means "commit *all* current changes to tracked files, whether or not `git add` was used to add them." (`git add` is still necessary to add a new file.)

A.7 Getting Started With GitHub

A variety of cloud-based Git hosting services exist. We recommend and give instructions for GitHub. GitHub's free plan gives you as many projects (repos) as you want, but all are publicly readable. Paid plans allow you to have private repos. If you are a student or a teacher, you can get a limited number of private repos by requesting a free educational account[15].

To communicate with most cloud-based Git services, you add your public key to the service, usually through a browser-based interface. The corresponding private key on your development computer then allows you to create a remote copy of a repo there and push changes to it from your local repo. Other developers can, with your permission, both push their own changes and *pull* your changes and others' changes from that remote.

You'll need to do the following steps to setup GitHub. This section assumes you have already setup an `ssh` keypair as directed in Section A.5; you should perform these steps from any computer holding the private key from which you want to access GitHub.

1. Using the terminal or console, tell Git your name and email address, so that in a multi-person project each commit can be tied to the committer:

   https://gist.github.com/cdf75000ba29fd9ca859

   ```
   git config --global user.name 'Andy Yao'
   git config --global user.email 'yao@acm.org'
   ```

2. To create a GitHub repo that will be a remote of your existing project repo, fill out and submit the New Repository[16] form and note the repo name you chose. A good choice is a name that matches the top-level directory of your project, such as `myrottenpotatoes`.

3. Back on your development computer, in a terminal window `cd` to the top level of your project's directory (where you previously typed `git init`) and type the following, replacing `myusername` with your GitHub username and `myreponame` with the repository name you chose in the previous step:

   https://gist.github.com/73213878289bd7bddc74

   ```
   git remote add origin git@github.com:myusername/myreponame.git
   git push origin master
   ```

The first command tells Git that you're adding a new remote for your repo located at GitHub, and that the short name `origin` will be used from now on to refer to that remote. (This name is conventional among Git users for reasons explained in Chapter **??**.) The second command tells Git to *push* any changes from your local repo to the `origin` remote that aren't already there.

These account setup and key management steps only have to be done once. The process of creating a new repo and using `git remote` to add it must be done for each new project. Each time you use `git push` in a particular repo, you are propagating all changes to the repo since your last push to the remote, which has the nice side effect of keeping an up-to-date backup of your project.

Figure A.1 summarizes the basic Git commands introduced in this chapter, which should be enough to get you started as a solo developer. When you work in a team, you'll need to use additional Git features and commands introduced in Chapter **??**.

Command	What it does	When to use it
`git pull`	Fetch latest changes from other developers and merge into your repo	Each time you sit down to edit files in a team project
`git add` *file*	Stage *file* for commit	When you add a new file that is not yet tracked
`git status`	See what changes are pending commit and what files are untracked	Before committing, to make sure no important files are listed as "untracked" (if so, use `git add` to track them)
`git diff` *filename*	See the differences between the current version of a file and the last committed version	To see what you've changed, in case you break something. This command has many more options, some described in Chapter **??**.
`git commit -a`	Commit changes to *all* (`-a`) tracked files; an editor window will open where you can type a commit message describing the changes being committed	When you're at a stable point and want to snapshot the project state, in case you need to roll back to this point later
`git checkout` *filename*	Reverts a file to the way it looked after its last commit. **Warning:** any changes you've made since that commit will be lost. This command has many more options, some described in Chapter **??**.	When you need to "roll back" one or more files to a known-good version
`git push` *remote-name*	Push changes in your repo to the remote named *remote-name*, which if omitted will default to `origin` if you set up your repo according to instructions in Section A.7	When you want your latest changes to become available to other developers, or to back up your changes to the cloud

Figure A.1: Common Git commands. Some of these commands may seem like arbitrary incantations because they are very specific cases of much more general and powerful commands, and many will make more sense as you learn more of Git's features.

A.8 Deploying to the Cloud Using Heroku

The concepts in Chapter **??** are central to this discussion, so read that chapter first if you haven't already.

Cloud computing technologies like Heroku make SaaS deployment easier than it's ever been. Create a free Heroku[17] account if you haven't already; the free account provides enough functionality for the projects in this book. For deploying Rails apps, Heroku provides a gem called `heroku`, which is installed as part of the Cloud9 setup (see Section A.3). You first need to add your `ssh` public key to Heroku to enable deployment there. Once you've created a Heroku account, in your Cloud9 console window give the command `heroku keys:add`. You'll be prompted to enter your Heroku username (email address) and password. Once your public key has been added to Heroku in this way, you'll be able to deploy to Heroku directly from Cloud9.

Git branches are discussed in Chapter **??**.

Essentially, Heroku behaves like a Git remote (Section A.7) that only knows about a single branch called `master`, and pushing to that remote has the side-effect of deploying your app. When you do such a push, Heroku detects which framework your app is using to determine how to deploy the app. For Rails apps, Heroku runs `bundle` to install your app's gems, compiles your assets (described below), and starts the app.

Chapter **??** describes the three environments (development, production, testing) defined by Rails; when you deploy to Heroku or any other platform, your deployed app will run in the production environment. There are two changes you must make to accommodate a few important differences between your development environment and Heroku's production environment.

First, Heroku needs some additional gems to support these differences. Heroku requires some specific configuration settings for your app's production environment, which are captured in a gem called **rails_12factor**[18]. Furthermore Heroku uses the PostgreSQL database rather than SQLite. The following code excerpt shows how to change your app's `Gemfile` to accommodate these two differences. You must do this step for *each* new app you create that will be deployed on Heroku. As always, don't forget to run `bundle` after changing your Gemfile, and to commit and push your changes to both `Gemfile` and `Gemfile.lock`.

https://gist.github.com/f4e9bbf7a17c7e50780fc3c4fadbe4c0

```
# making your Gemfile safe for Heroku
ruby '2.2.0'   # just in case - tell Heroku which Ruby version we need
group :development, :test do
  # make sure sqlite3 gem ONLY occurs inside development & test groups
  gem 'sqlite3' # use SQLite only in development and testing
end
group :production do
  # make sure the following gems are in your production group:
  gem 'pg'              # use PostgreSQL in production (Heroku)
  gem 'rails_12factor'  # Heroku-specific production settings
end
```

After installing any needed gems, Heroku's next action on each deployment is to deal with your app's static assets, such as CSS files (Section **??**) and JavaScript files (Chapter **??**). Starting with Rails 3.1, Rails supports the higher-level language ***SCSS*** for creating CSS stylesheets and the ***CoffeeScript*** language for DRYing out JavaScript. Since browsers consume CSS and JavaScript but not SCSS or CoffeeScript, a sequence of steps collectively called the asset pipeline[19] performs the following code-generation tasks:

1. All CoffeeScript files in `app/assets`, if any, are converted to JavaScript.
2. All JavaScript files are concatenated into one large JavaScript file which is then ***minified*** to take up less space by removing whitespace and comments and perhaps re-

naming variables with shorter names. The resulting large JavaScript file is placed in `public/assets`.

3. All SCSS files in `app/assets`, if any, are translated to CSS.
4. All CSS files are concatenated into one large CSS file which is minified and placed in `public/assets`.
5. Rails arranges for the name of each of these single large files to include a "fingerprint" that uniquely identifies the file's content, allowing the static files to be cached by both browsers and servers (Section **??**) as long as the file's content doesn't change, which in production environments only happens when a new version of the app is deployed.
6. The behaviors of the Rails view helpers `javascript_include_tag` and `stylesheet_link_tag`, which usually appear in a layout such as `app/views/application.html.haml` (Section **??**), are modified to load these auto-generated files from the `public` directory, which in some production environments can be redirected to a separate static asset server or even a ***Content Distribution Network***.

The second change you must make to your app, therefore, is to specify which of three ways Heroku should manage the asset pipeline. The first way is for you to *precompile* the assets by running the asset pipeline locally on your computer and versioning the generated JavaScript and CSS files in Git. The second is to have Heroku prepare and compile the assets at runtime, the first time each type of asset is requested. This method can cause unpredictable performance when it happens, and neither we nor Heroku recommend it. The third method, which we recommend, is to let Heroku compile the static files just once at deploy time. This method is the most DRY: since you only keep your original files (JavaScript and/or CoffeeScript, CSS and/or SCSS) under version control, there is exactly one place where asset information can be changed. It also simplifies configuration if you're using Jasmine to test your JavaScript or CoffeeScript code.

To enable Heroku to precompile your assets at deploy time, add the following line in `config/environments/production.rb`:

https://gist.github.com/97a265eb158b18cf4965

```
# in config/environments/application.rb:
config.assets.initialize_on_precompile = false
```

This line prevents Heroku from trying to initialize the Rails environment before precompiling your assets: on Heroku, some ***environment variables*** on which Rails relies are not initialized until later, and their absence would cause an error during deployment. This article[20] contains some tips on troubleshooting asset pipeline problems at deploy time, including how to compile the asset pipeline locally to isolate problems. **Beware:** if you compile the asset pipeline locally, it will create the file `public/assets/manifest.yml`; make sure this file is *not* checked into Git, because its presence will tell Heroku that you're precompiling your own assets and don't want Heroku to do it for you!

Once you've made these two one-time changes in your app (and remembered to commit and push the results), deployment of each new app version follows a simple recipe, starting from your app's root directory:

1. **Make sure your app is running correctly and passing all your tests locally.** Remote debugging is always harder. Before you deploy, maximize your confidence in your local copy of the app!

Local (development)	Heroku (production)
`rails server`	`git push heroku master`
`rails console`	`heroku run console`
`rake db:migrate`	`heroku run rake db:migrate`
`more log/development.log`	`heroku logs`

Figure A.2: How to get the functionality of some useful development-mode commands for the deployed version of your app on Heroku.

2. If you have added or changed any gems, be sure you've successfully run `bundle` to make sure your app's dependencies are still satisfied, and that you've committed and pushed any changes to `Gemfile` and `Gemfile.lock`.

3. The *first* time you deploy an app, `heroku apps:create` *appname* creates a new Heroku application container called *appname*; if you omit the name, a whimsical name is preassigned, such as `luminous-coconut-237`. In any case, your app will be deployed at `http://`*appname*`.herokuapp.com`. You can change your app's name later by logging into your Heroku account and clicking My Apps.

4. Once you've committed your latest changes,
   `git push heroku master`
   deploys the head of your local repo's master branch to Heroku. (See this article[21] to deploy from a branch other than master, if you're following the branch-per-release methodology of Section **??**.)

5. `heroku ps`
   checks the process status (`ps`) of your deployed app. The **State** column should say something like "Up for 10s" meaning that your app has been available for 10 seconds. You can also use `heroku logs` to display the log file of your app, a useful technique if something goes wrong in production that worked fine in development.

6. `heroku run rake db:migrate`
   On any deployment where you have changed the database schema (Sections **??** and **??**), including the first-time deployment, this command will cause the app's database to be created or updated. If there are no pending migrations, the command safely does nothing. Heroku also has instructions on how to import the data from your development database[22] to your production database on your first deployment.

Figure A.2 summarizes how some of the useful commands you've been using in development mode can be applied to the deployed app on Heroku.

■ ***Elaboration: Production best practices***

In this streamlined introduction, we're omitting two best practices that Heroku recommends[23] for "hardening" your app in production. First, our Heroku deployment still uses WEBrick as the presentation tier; Heroku recommends using the streamlined `thin` webserver for better performance. Second, since subtle differences between SQLite3 and PostgreSQL functionality may cause migration-related problems as your database schemas get more complex, Heroku advises using PostgreSQL in both development and production, which would require installing and configuring PostgreSQL in your development environment. In general, it's a good idea to keep your development and production environments as similar as possible to avoid hard-to-debug problems in which something works in the development environment but fails in the production environment.

A.9 Checklist: Starting a New Rails App

Throughout the book we recommend several tools for developing, testing, deploying, and monitoring the code quality of your app. In this section, we pull together in one place a step-by-step list for creating a new app that takes advantage of all these tools. This section will only make sense after you have read all the referenced sections, so use it as a reference and don't worry if you don't understand all the steps now. Steps are annotated with the section number(s) in which the tool or concept is first introduced.

Set up your app: (§**??**)

1. `rails -v` to ensure you're running the desired version of Rails. If not run `gem install rails -v` *x.x.x* with *x.x.x* set to the version you want; 3.2.19 for example.
2. `rails new` *appname* `-T` to create the new app. `-T` skips creating the `test` subdirectory used by the **Test::Unit** testing framework, since we recommend using RSpec instead.
3. `cd` *appname* to navigate into your new app's root directory. From now on, all shell commands should be issued from this directory.
4. Edit the `Gemfile` to lock the versions of Ruby and Rails, for example:

   https://gist.github.com/8af52b226d5413ffb384134c03bd10e1

   ```
   # in Gemfile:
   ruby '2.2.2'    # Ruby version you're running
   rails '4.2.1'  # Rails version for this app
   ```

   If you ended up changing the version(s) already present in the Gemfile, run `bundle install --without production` to make sure you have compatible versions of Rails and other gems.
5. Make sure your app runs by executing `rails server -p $PORT -b $IP` and visiting the app's root URI. You should see the Rails welcome page. (If not using Cloud9, just use `rails server`.)
6. `git init` to set up your app's root directory as a GitHub repo. (§A.6, Screencast A.6.1)

Connect your app to GitHub, CodeClimate, Travis CI, and Heroku:

1. If working on an existing/legacy app, fork its GitHub repo and add your team to the fork as collaborators.
2. If starting a new app, create a new GitHub repo via GitHub's web interface, and do the initial commit and push of your new app's repo. Make sure you include a `.gitignore` file (which tells Git *not* to version certain files) that contains at least the files shown in this example[24]. (§A.7)
3. Ensure your repo contains a file at the top level called `LICENSE` containing the text of, or a link to, the license you want to use. We recommend choosing one of these popular open source licenses[25].
4. Point CodeClimate at your app's GitHub repo (§**??**). Add a CodeClimate badge to your repo's `README.md` ("splash page") so you can always see the latest CodeClimate results. Make sure your repo includes a top-level `.codeclimate.yml` file that tells CodeClimate to *exclude* from analysis any external libraries (`vendor/*`, `jquery.js`, and so on) and minified JavaScript files (usually named `*.min.js`).
5. Set up a free Travis CI account and point it at your app's GitHub repo (§**??**). Add a Travis CI badge to `README.md` to see the latest status of running tests.
6. Set up a Pivotal Tracker project to track user stories and velocity. (§**??**)
7. Make the changes necessary to deploy to production on Heroku. (§A.8)
8. Run `bundle install --without production` if you've changed your `Gemfile`. Commit the changes to `Gemfile` and `Gemfile.lock`. On future changes to the Gemfile, you can just say `bundle` with no arguments, since Bundler will remember the option to skip production gems. (§**??**)
9. `heroku apps:create` *appname* to create your new app on Heroku (§A.8)
10. `git push heroku master` to ensure the app deploys correctly. (§A.8)

Set up your testing environment:

1. Add support in your Gemfile for Cucumber (§**??**), RSpec (§**??**), interactive debugging (§**??**), Coveralls (which relies on SimpleCov; §**??**), Guard (§**??**), FactoryGirl (§**??**), and Jasmine if you plan to use JavaScript (§**??**).

https://gist.github.com/3c72d79d489727f27dbf28165feee1d0

```
group :development, :test do
  gem 'coveralls', :require => false # to use Coveralls.io for test coverage
  gem 'simplecov', :require => false
  gem 'jasmine-rails' # if you plan to use JavaScript/CoffeeScript
end
# setup Cucumber, RSpec, Guard support
group :test do
  gem 'rspec-rails'
  gem 'guard-rspec'
  gem 'cucumber-rails', :require => false
  gem 'cucumber-rails-training-wheels' # basic imperative step defs
  gem 'database_cleaner' # required by Cucumber
  gem 'factory_girl_rails' # if using FactoryGirl
end
```

(See Section **??** for additional gems to support fixtures and AJAX stubbing in your JavaScript tests.)

2. Run `bundle`, since you've changed your `Gemfile`. Commit the changes to `Gemfile` and `Gemfile.lock`.

3. If all is well, create the subdirectories and files used by RSpec, Cucumber, Jasmine, and if you're using them, the basic Cucumber imperative steps:

https://gist.github.com/dd1bde21bb174a51eb1fba7601128a8a

```
rails generate rspec:install
rails generate cucumber:install
rails generate cucumber_rails_training_wheels:install
rails generate jasmine_rails:install
bundle exec guard init rspec
```

4. If you're using Coveralls and SimpleCov, which we recommend, add the following lines to the appropriate testing-related configuration files:

https://gist.github.com/60183c413b2c6bb7154a8d61cb81eed0

```
# This information is from the article:
#   https://coveralls.zendesk.com/hc/en-us/articles/201769485-Ruby-Rails

# add at TOP of spec/rails_helper.rb
# AND at the TOP of features/support/env.rb:
require 'coveralls'
Coveralls.wear_merged!('rails')

# in your Rakefile, define a task that will push coverage info
# AFTER running both the specs and the features:

require 'coveralls/rake/task'
Coveralls::RakeTask.new
task :test_with_coveralls => [:spec, :features, 'coveralls:push']

# Finally, in the 'script:' section of .travis.yml, make this task
# the default/only testing task:
script:
  - bundle exec rake test_with_coveralls
```

5. If you're using FactoryGirl to manage factories (§**??**), add its setup code:

https://gist.github.com/80bc0375ba137524b59c

```
# For RSpec, create this file as spec/support/factory_girl.rb
RSpec.configure do |config|
  config.include FactoryGirl::Syntax::Methods
end
```

https://gist.github.com/850c9992f29d2f03d052

```
# For Cucumber, add at the end of features/support/env.rb:
World(FactoryGirl::Syntax::Methods)
```

6. `git add` and then commit any files created or modified by these steps.

7. Ensure Heroku deployment still works: `git push heroku master`

You're now ready to create and apply the first migration (§**??**), then re-deploy to Heroku and apply the migration in production (`heroku run rake db:migrate`).

Add other useful Gems:

Some that we recommend include:

- `railroady` draws diagrams of your class relationships such as has-many, belongs-to, and so on (§**??**)
- `omniauth` adds portable third-party authentication (§**??**)
- `devise` adds user self-signup pages, and optionally works with `omniauth`
- `figaro` provides a Heroku-friendly way to manage your app's "secrets" (sensitive information such as API keys and other credentials) for your application. This blog post[26] suggests a safe way to use it to manage your app secrets while benefiting from version control.

A.10 Fallacies and Pitfalls

Pitfall: **Versioning unencrypted sensitive files.**

Files containing sensitive information, such as API keys or other credentials, should *never* be checked into GitHub "in the clear" (without encryption). If the files must be checked in, they should be encrypted at rest. The discussion of the Figaro gem in the previous section explains further.

Pitfall: **Making check-ins (commits) too large.**

Git makes it quick and easy to do a commit, so you should do them frequently and make each one small, so that if some commit introduces a problem, you don't have to also undo all the other changes. For example, if you modified two files to work on feature A and three other files to work on feature B, do two separate commits in case one set of changes needs to be undone later. In fact, advanced Git users use `git add` to "cherry pick" a subset of changed files to include in a commit: add the specific files you want, and *omit* the `-a` flag to `git commit`.

Pitfall: **Forgetting to add files to the repo.**

If you create a new file but forget to add it to the repo, your copy of the code will still work but your file won't be tracked or backed up. Before you do a commit or a push, use `git status` to see the list of Untracked Files, and `git add` any files in that list that *should* be tracked. You can use the `.gitignore`[27] file to avoid being warned about files you never want to track, such as binary files or temporary files.

Pitfall: **Confusing commit with push.**

`git commit` captures a snapshot of the staged changes in *your* copy of a repo, but no one else will see those changes until you use `git push` to propagate them to other repo(s) such as the origin.

Pitfall: **Hidden assumptions that differ between development and production environments.**

Chapter **??** explains how Bundler and the Gemfile automate the management of your app's dependencies on external libraries and how migrations automate making changes to

your database. Heroku relies on these mechanisms for successful deployment of your app. If you manually install gems rather than listing them in your Gemfile, those gems will be missing or have the wrong version on Heroku. If you change your database manually rather than using migrations, Heroku won't be able to make the production database match your development database. Other dependencies of your app include the type of database (Heroku uses PostgreSQL), the versions of Ruby and Rails, the specific Web server used as the presentation tier, and more. While frameworks like Rails and deployment platforms like Heroku go to great lengths to shield your app from variation in these areas, using automation tools like migrations and Bundler, rather than making manual changes to your development environment, maximizes the likelihood that you've documented your dependencies so you can keep your development and production environments in sync. If it can be automated and recorded in a file, it should be!

A.11 To Learn More

- The Git Community Book[28] is a good online reference that can also be downloaded as a PDF file.

Notes

[1] http://c9.io
[2] http://github.com
[3] http://heroku.com
[4] http://stackoverflow.com
[5] http://stackoverflow.com
[6] http://github.com
[7] http://heroku.com
[8] http://pivotaltracker.com
[9] http://saasbook.info
[10] https://c9.io
[11] http://catb.org/~esr/writings/homesteading/cathedral-bazaar/
[12] http://code.tutsplus.com/articles/25-vim-tutorials-screencasts-and-resources--net-14631
[13] http://www.gnu.org/software/emacs/tour/
[14] https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/#platform-linux
[15] http://github.com/edu
[16] https://github.com/repositories/new
[17] http://heroku.com
[18] https://github.com/heroku/rails_12factor
[19] http://guides.rubyonrails.org/asset_pipeline.html
[20] https://devcenter.heroku.com/articles/rails-asset-pipeline
[21] https://devcenter.heroku.com/articles/git#deploying-code
[22] http://devcenter.heroku.com/articles/taps
[23] http://devcenter.heroku.com/articles/rails3
[24] https://github.com/github/gitignore/blob/master/Rails.gitignore
[25] https://opensource.org/licenses
[26] https://saasbook.blogspot.com/2016/08/keeping-secrets.html
[27] http://book.git-scm.com/4_ignoring_files.html
[28] http://book.git-scm.com/