Startup Engineering
Balaji S. Srinivasan
Stanford University

Class: class.coursera.org/startup-001
Signup: coursera.org/course/startup
Forums: bit.ly/startup-eng-forum

Deployment, DNS, and Custom Domains

Deployment: Dev, Staging, and Production

The easiest way to edit a website is to simply edit code on the live server for that website. That means SSHing directly into the server for `example.com`, loading up `emacs` or `vim`, editing the HTML/CSS/JS, and then just saving the file and reloading `example.com`. This style of web development is commonly practiced by graduate students on academic websites and does have the advantage of extreme simplicity. However, there are several problems with it. If there is only one copy of the codebase and it is located on the production website, it becomes difficult to:

- test features before end users see them
- roll back code that has introduced bugs
- restore code or data from catastrophic crashes of the server
- incorporate contributions from multiple engineers
- perform A/B testing of different incompatible features

We can solve these problems by introducing the concepts of (1) distributed version control of the codebase (e.g. via `git`) and (2) a dev/staging/production flow. The use of distributed version control allows us to save, restore, test, and merge multiple versions of a *codebase* by making use of git branch (1, 2, 3, 4). Combining this with separate servers for dev/staging/production allows us to thoroughly test changes to a *website* before rolling them out to end users. In a bit more detail, here's how our dev/staging/production flow (Figures 1 and 2) will work in the class:

1. *Development.*
    - You will develop on an SSH/HTTP/HTTPS-accessible EC2 machine with a text editor like `emacs`, and preview your work in the browser, using Chrome User Agent Switcher and Resolution Test to mimic mobile browsers.
    - The `screen` utility will be used to preserve your remote session so that you can disconnect from your dev instance and then reconnect from the same or another machine.
    - You will save your website's codebase along the way in a `git` repository (say `example-site.git`), hosted on Github. Importantly, and this is a new concept, you will use git branch to create a `develop` branch of your codebase rather than making edits directly to the `master` branch.

- You will also save the commands required to set up the EC2 machine (`setup.git`) and configure your development environment (`dotfiles.git`) in git repositories, to make the entire development process highly reproducible. This will permit other engineers to run/edit/debug your code and get exactly the same results.
- As you make changes in develop, you will create/commit/push edits, and run a local web server to permit the previewing of these edits in a browser.

2. *Staging.*

   - Once you have a "release candidate", that is, a version of the code that you think is worthy of evaluating for the production website, you will merge the code into a `staging` branch in `example-site.git` and optionally use git tag to tag a release candidate, with a systematic name like `staging-june-13-2013-r01`.
   - Using Heroku's syntax for setting up staging environments, you will then push to Heroku. This gives us a clone of exactly how the live site will look, as it might be subtly different from the EC2 dev preview.
   - In our class, staging is a separate Heroku app available at something like http://example-site-staging.herokuapp.com.
   - We may find some last minute bugs; if so, we make these edits/commits in the `develop` branch and then merge them into `staging`, rather than editing `staging` directly. This forces discipline; in particular it ensures that your bug fix is not incompatible with changes made to `develop` by others since your deploy to `staging`.

3. *Production.*

   - Once you and your automated and manual testers confirm that the site works as intended on the staging server (e.g. by clicking and trying to break things, or running headless tests against the staging URL), then you have a confirmed release candidate of `example-site.git` in the `staging` branch. This is now merged into `master`.
   - You may wish to further distinguish this revision by using another `git tag` with an internal naming scheme (e.g. `deploy-june-13-2013`).
   - You will then push this revision to the production server, which is viewed by end users. For the purposes of this class, that is a Heroku app like `http://example-site.herokuapp.com`.
   - Finally, if you have set up a custom domain, this is when end users see your changes propagate to `example.com` (see the last section in this lecture).

Figure 1: *Modern web development proceeds in three phases: dev, staging, and production.*

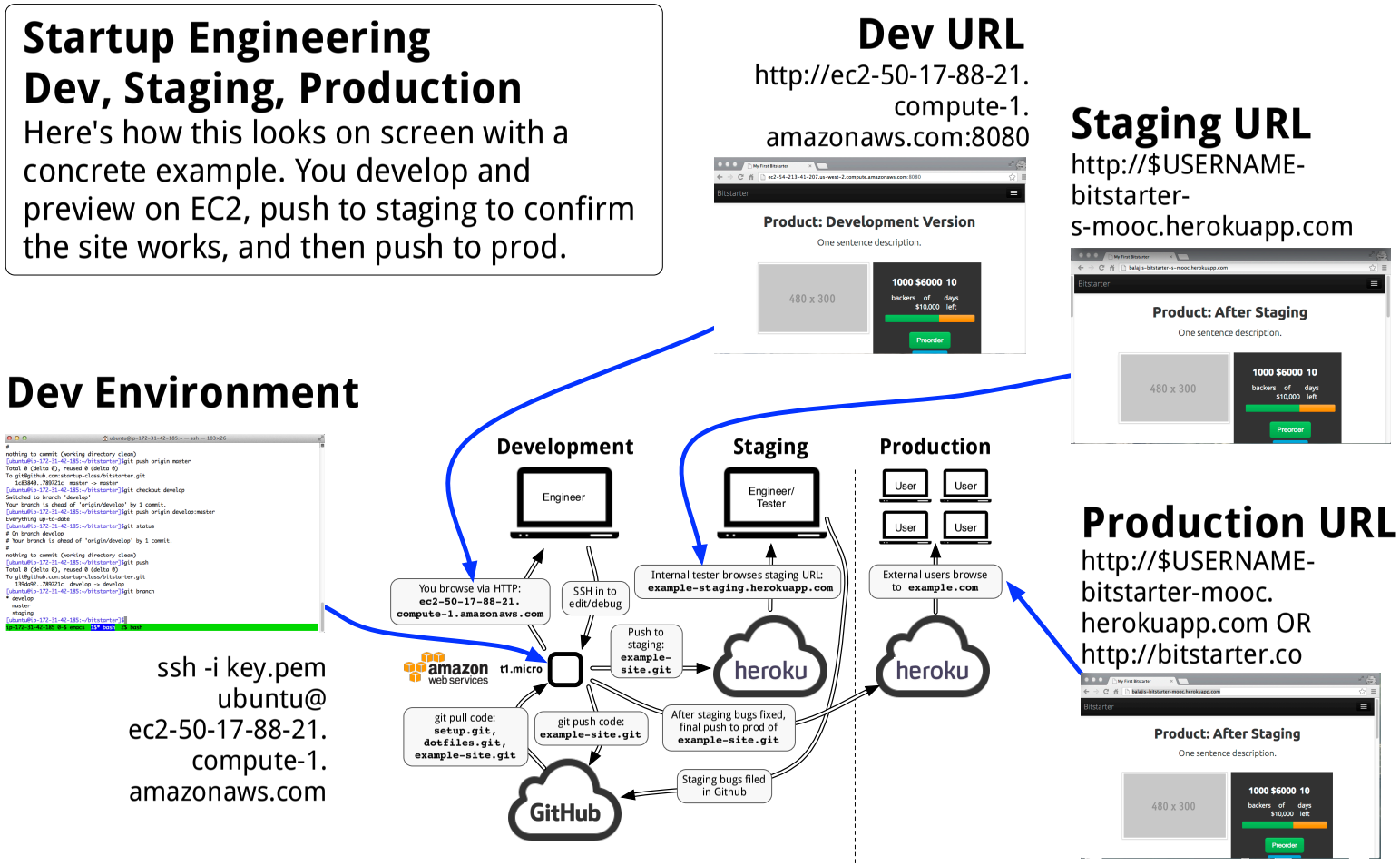

Figure 2: *Here's how dev/staging/prod looks on screen. See also Figure 1.*

Sidebar: Comparing EC2 vs. local laptops for development

Note that for the class we use EC2 as both editing environment and dev server, and Heroku as our staging and production servers. We do this for several reasons:

1. *Standardized environment.* As noted in Lecture 2, EC2 will work for any student with a credit card and evens out the background differences between student machines. In subsequent versions of the class we may also support local VMs, which are similar in concept.

2. *Shareable development URLs.* EC2 instances make it easy to set up a development server on the public internet (e.g. at `http://ec2-54-213-41-207.us-west-2.compute.amazonaws.com:8080` or the equivalent). This is very convenient for testing out certain kinds of apps (especially mobile apps that need a dev API) or sharing in-development URLs with people outside your organization. It's possible to do something like this with a laptop but harder.

3. *Scriptable infrastructure.* Terminating and restoring pristine EC2 instances also introduces the crucial concept of scriptable infrastructure; think of our simple setup.sh as the beginning of devops. Your dev environment needs to be scripted for it to be reproducible by other engineers, and you're not going to wipe/restore someone's local laptop every time just to ensure that (e.g.) you have exactly the same `PATH` settings, library versions, and the like. You can run a VM locally, but that is somewhat slow. Thus EC2 provides development discipline.

4. *Portable, general environment.* Learning how to work in a Linux terminal development environment means you can be productive in a new language or toolchain fairly rapidly. The differences between different emacs or vim modes are trivial relative to the differences between dedicated C# or Python editors. This is important as many projects require the use of multiple languages or coding across multiple environments.

5. *Large or private datasets.* When working with large datasets or databases with privacy constraints, sometimes you can download a small and/or anonymized subsample that is suitable for offline debugging. However, if that proves infeasible (as it is for search, social, genomics, advertising, or many other big data problems), you need to be able to work on a remote server (either one in your own colo or datacenter, or on EC2 or a similar IAAS provider). This experience will be very similar to the class experience, where we are just using a local laptop as a thin client.

That said, in practice (outside the class) engineers often choose to develop on a local laptop for convenience when possible (Figure 3), while previewing locally by connecting on localhost. At the risk of belaboring the obvious: if you can get things to work on your local machine and don't want to use EC2 for development, then go ahead and do that. We are simply presenting one solution that works; you should feel free to invent your own tech stack, use the tools you're already familiar with, or mix and match some of our choices with some of yours. However, given the size of the class we will not be able to give technical support for nonstandard configurations; we hope you understand!

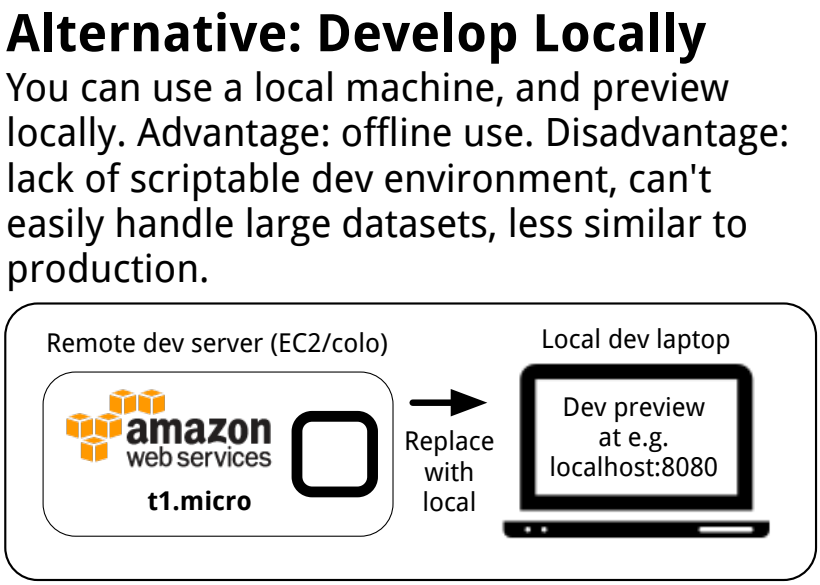

Figure 3: *You can replace the remote dev instance with a local laptop. This has advantages and disadvantages; see main text.*

Preliminaries: SSH/HTTP/HTTPS-accessible EC2 dev instance

We're now going to do a worked example of dev/staging/production with your code from HW3. To allow local previewing of your files under development, we'll first set up a dev instance which allows HTTP connections not just on port 22 (which handles SSH traffic), but also port 80 (which handles HTTP), port 443 (which handles HTTPS) and port 8080 (a typical debug/dev port for HTTP). This will allow you to edit code on EC2 and debug it by previewing in a browser, rather than copying it locally or always pushing to Heroku. To set this up you will want to go to the AWS Dashboard, set up a new security group, and then launch an instance[1] with that security group. Take a look at Figures 4-14:

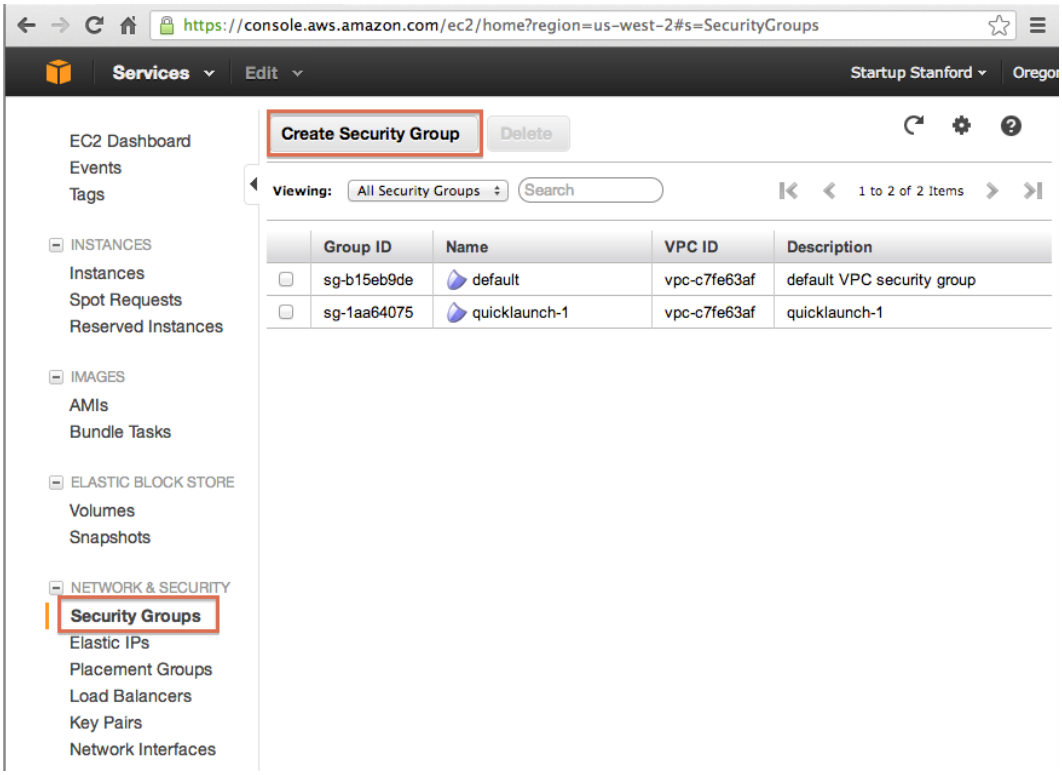

Figure 4: *Go to the EC2 dashboard and create a new security group.*

[1] You can also change the security group of an existing instance without starting up a new one, but it's pedagogically easier to just show how to do this from scratch rather than resuming (as everyone's VM will be in a different state).

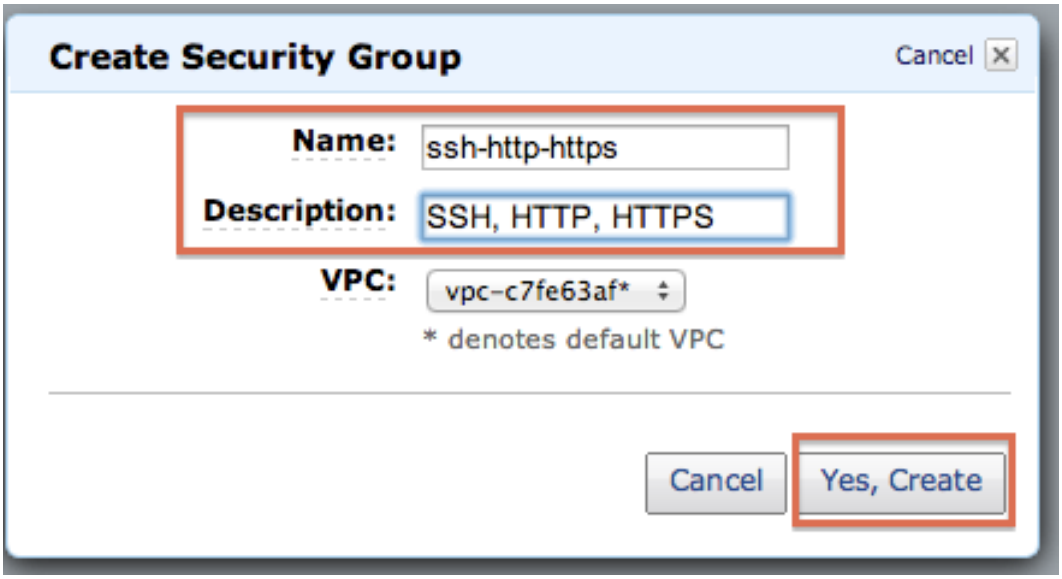

Figure 5: *Call it ssh-http-https.*

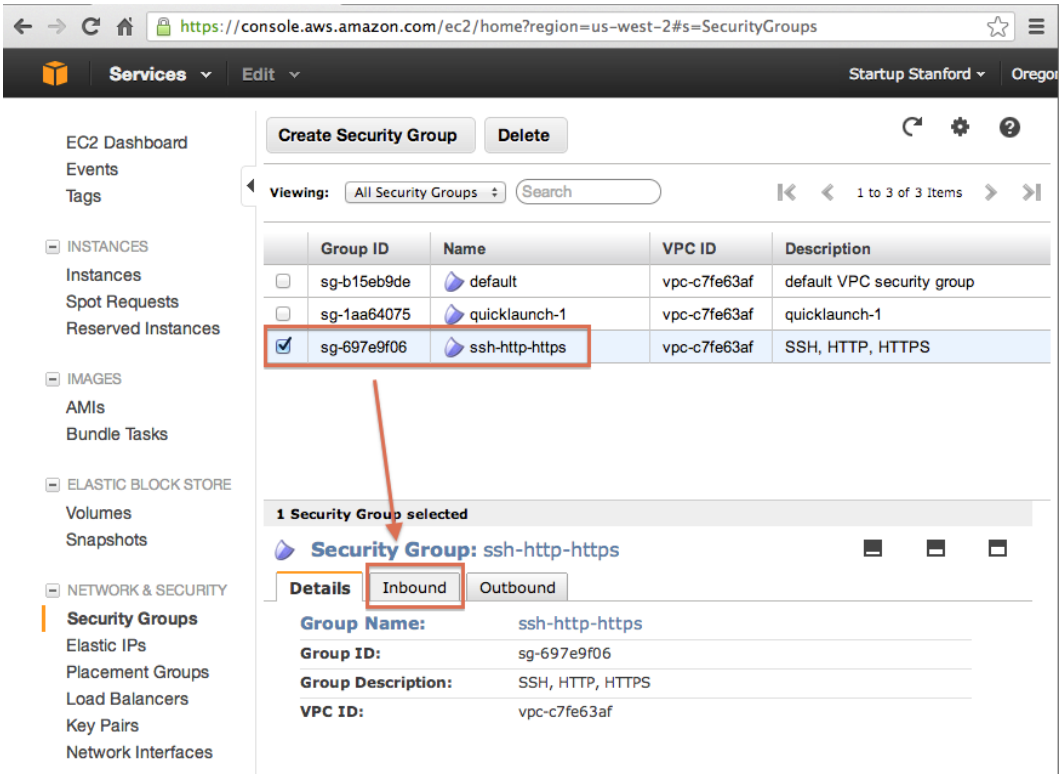

Figure 6: *Click it and click the 'Inbound' tab.*

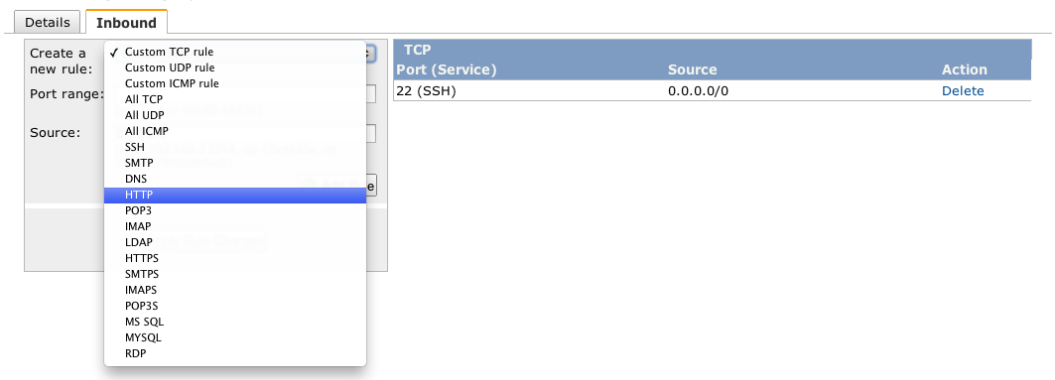

Figure 7: *Click the dropdowns in sequence and open up the HTTP and HTTPS ports.*

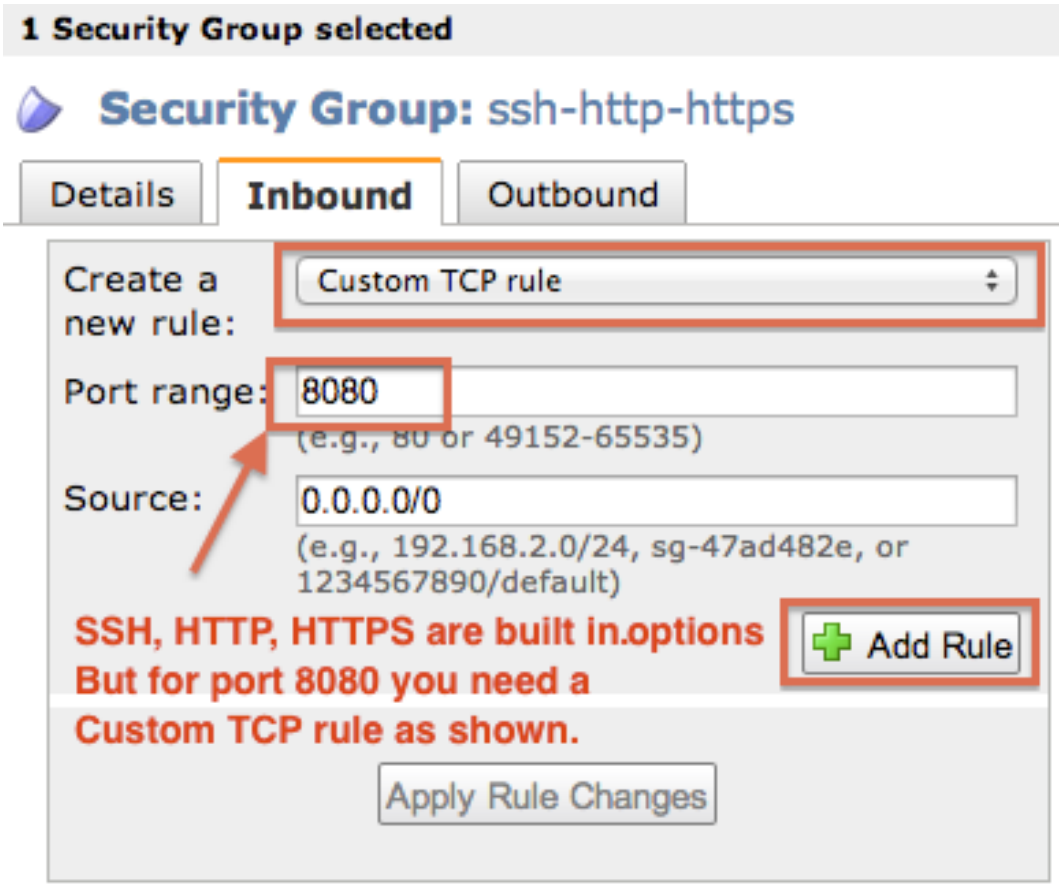

Figure 8: *For port 8080, you will need to set up a custom TCP rule, as shown.*

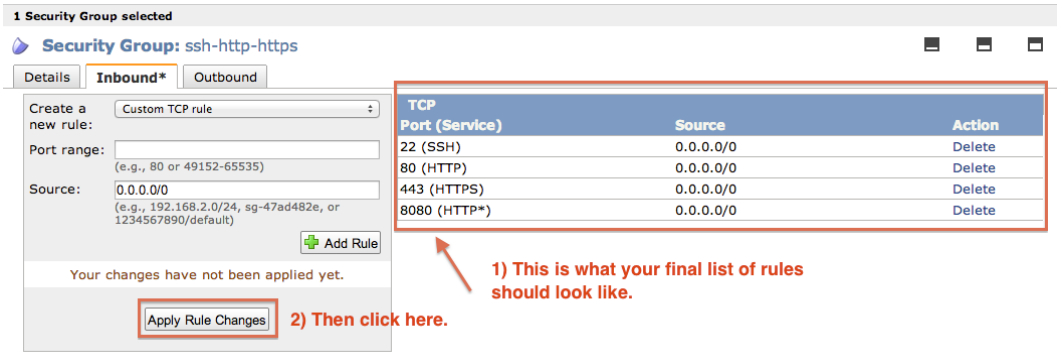

Figure 9: *You should have four ports open at the end: 22, 80, 443, and 8080.*

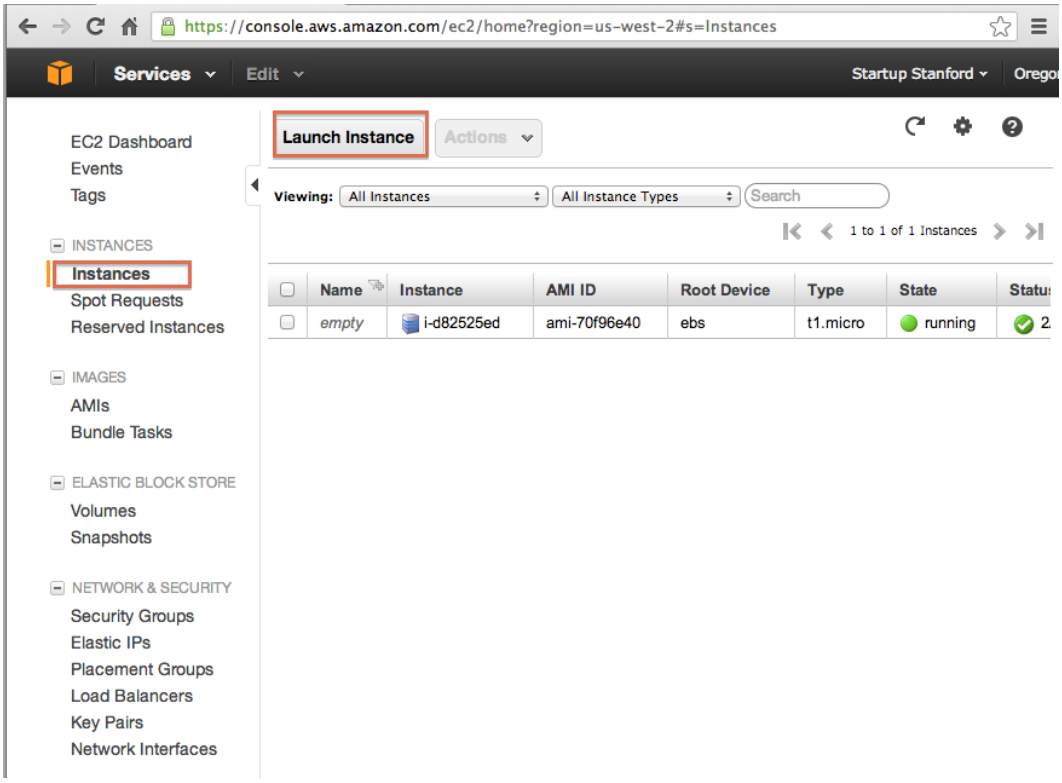

Figure 10: *Now terminate your old instances and launch a new instance (it is also possible to apply the security group to a running instance).*

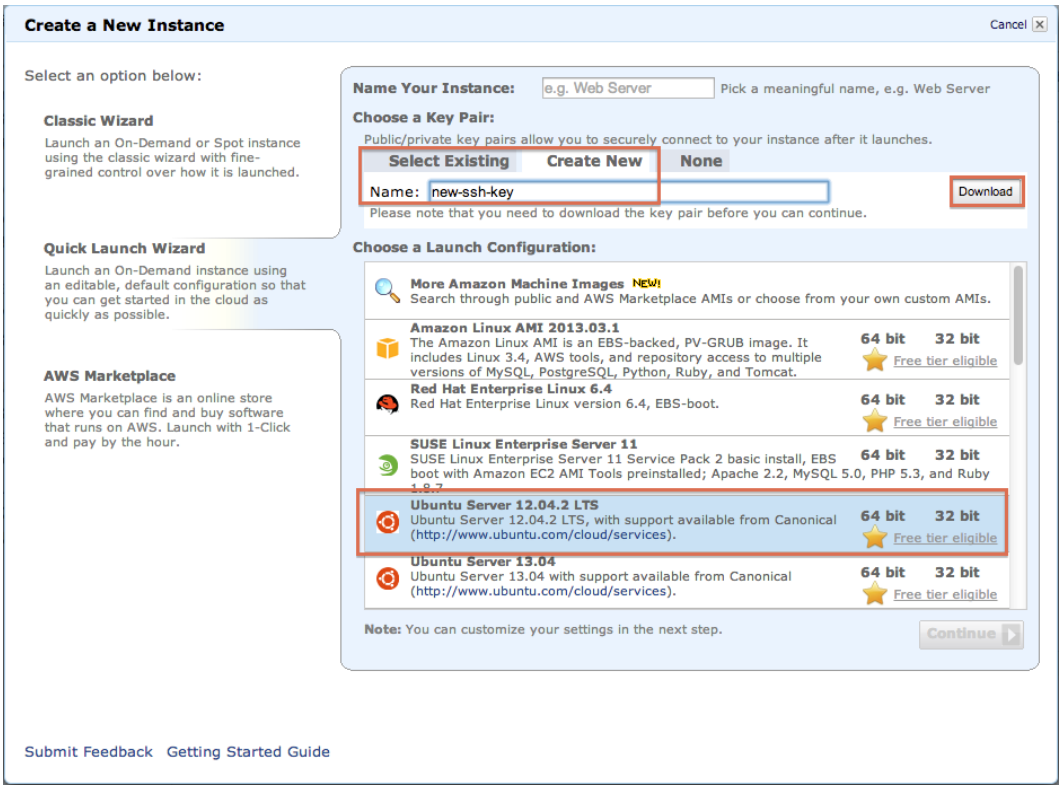

Figure 11: *Pick Ubuntu 12.04.2 LTS and download the pem key as usual.*

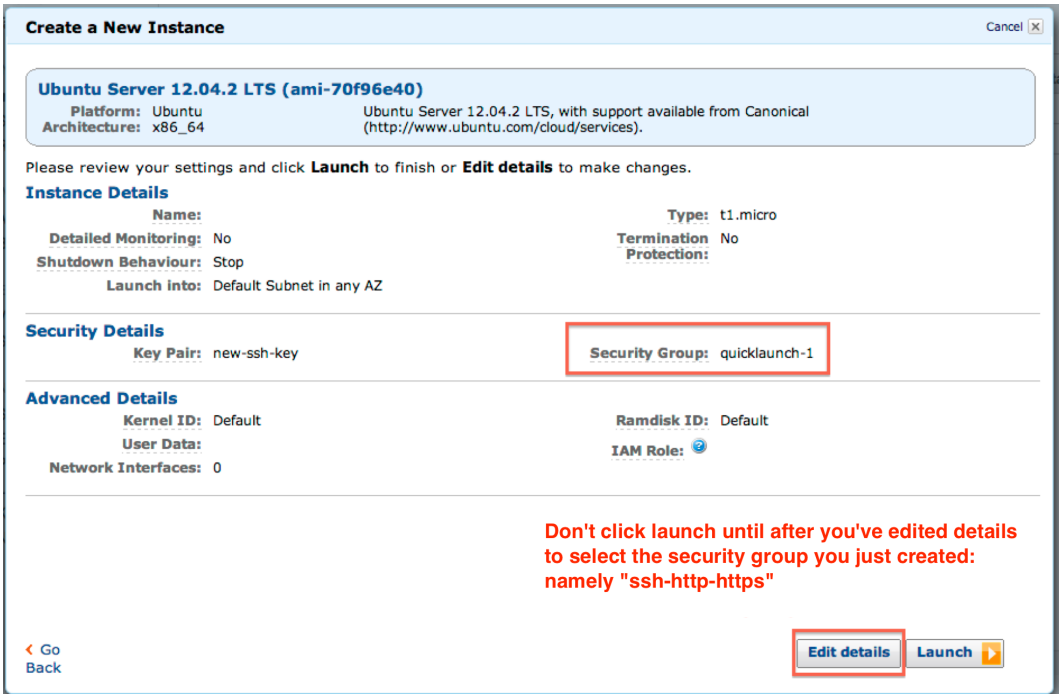

Figure 12: *Make sure to click 'Edit details' as the security group you just configured probably will not be selected by default.*

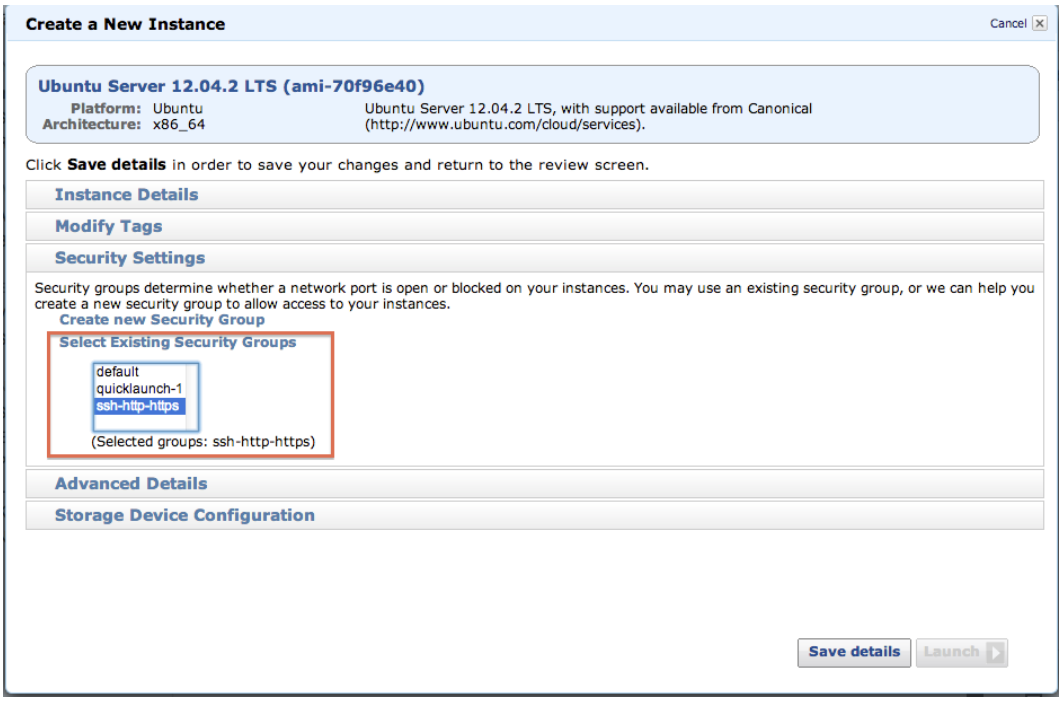

Figure 13: *Pick ssh-http-https in the Security Settings accordion.*

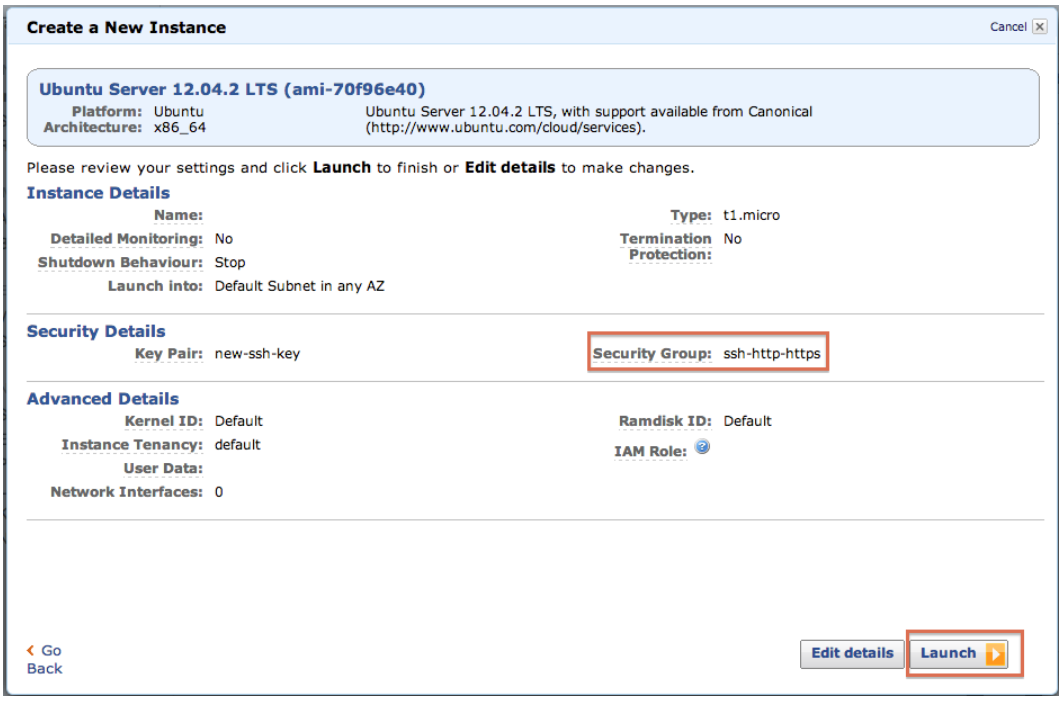

Figure 14: *Now launch your instance.*

You'll now want to run setup.sh to reinstall everything (including `dotfiles.git`) and then use `git clone` to pull down your bitstarter repository that you created in from github. Now edit your `bitstarter/web.js` file to configure it to port 8080 and then run `node web.js`. You should now be able to see your `index.html` in the browser, as shown in the following screenshots:

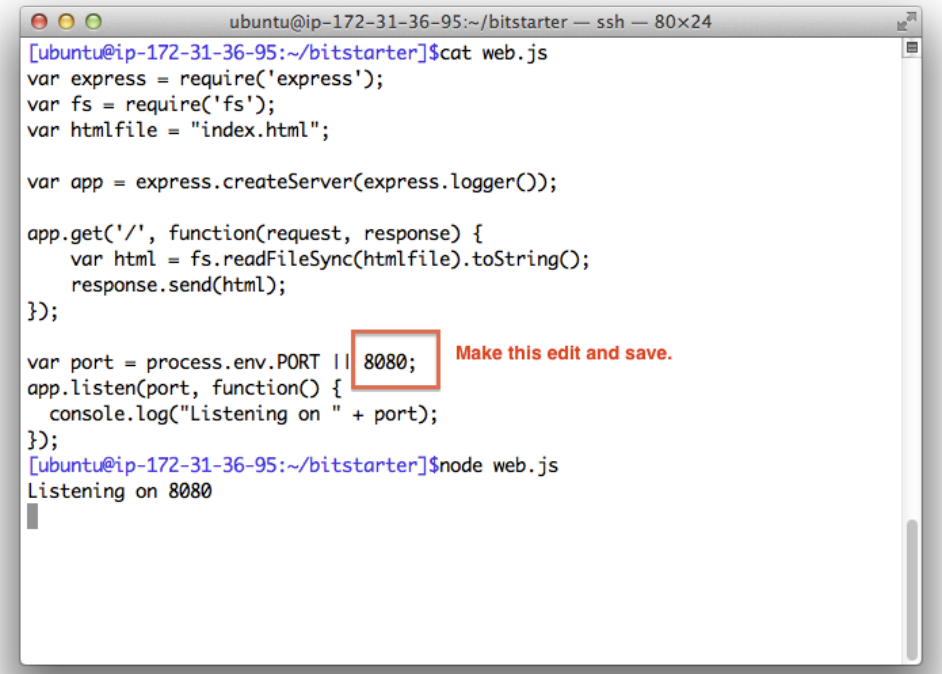

Figure 15: *Change your web.js to serve on port 8080, and run node web.js*

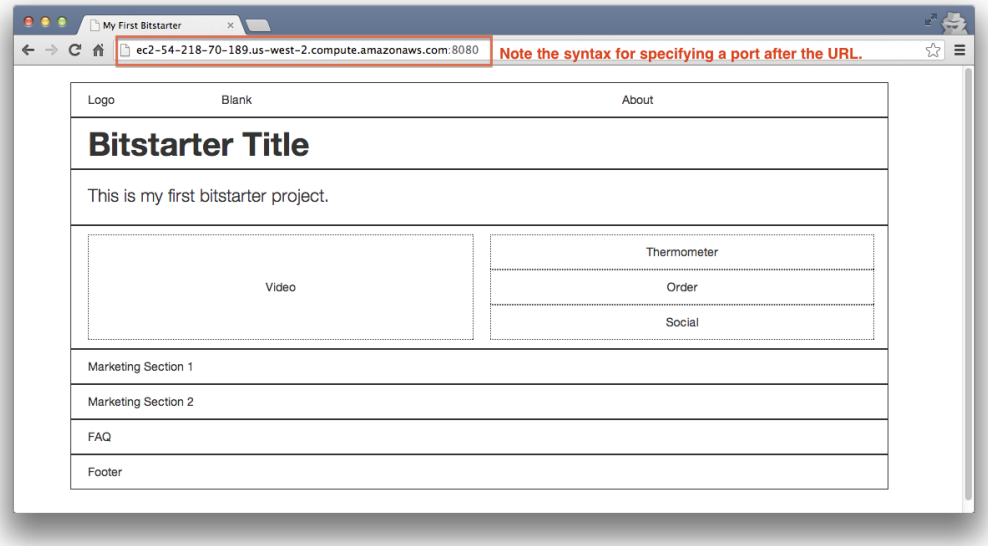

Figure 16: *Confirm that you can view your site in a browser.*

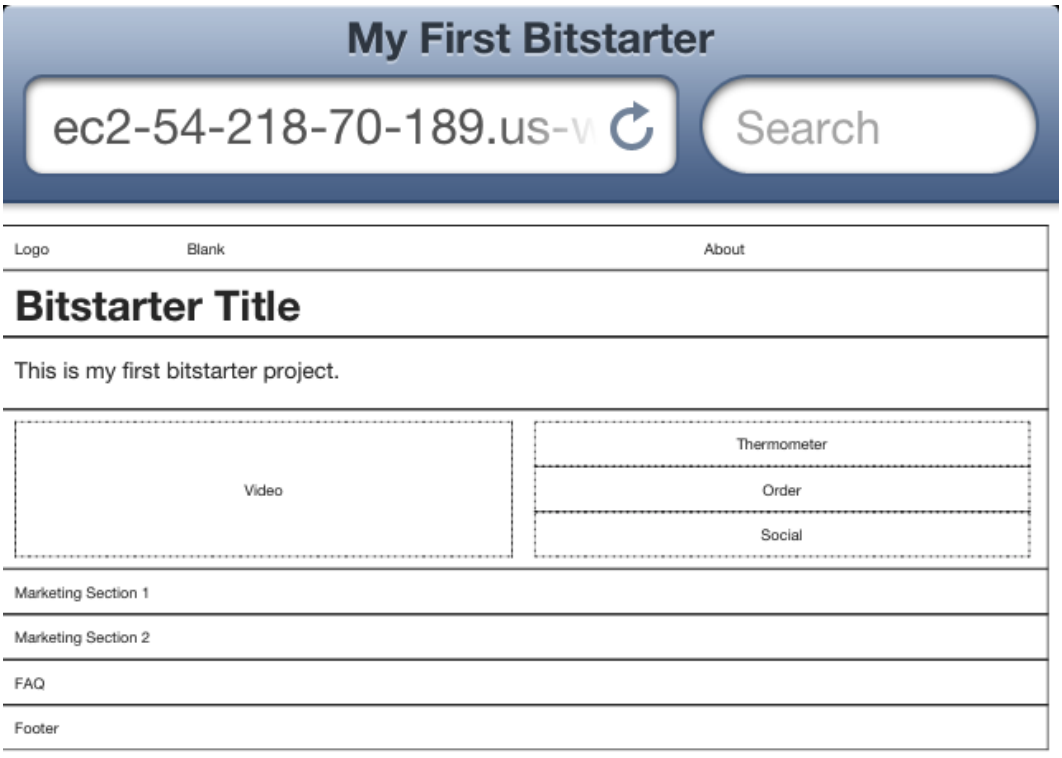

Figure 17: *Confirm that you can view your site in a mobile browser.*

Note that the last screenshot shows a mobile phone browsing to the URL. This works because by opening up the security group, you made your EC2 instance accessible over port 8080 over the public internet. However, it's a little inconvenient to keep refreshing your mobile phone. Instead, one of the best ways to preview how your website will look with a mobile client is by installing Chrome's User Agent Switcher and its Resolution Test tools. Then navigate to the same URL and set things up as shown in Figures 18-19. One way to confirm that you've configured this properly is to go to airbnb.com in Chrome with the User Agent Switcher set to iPhone 4 and see if you get automatically redirected to m.airbnb.com.

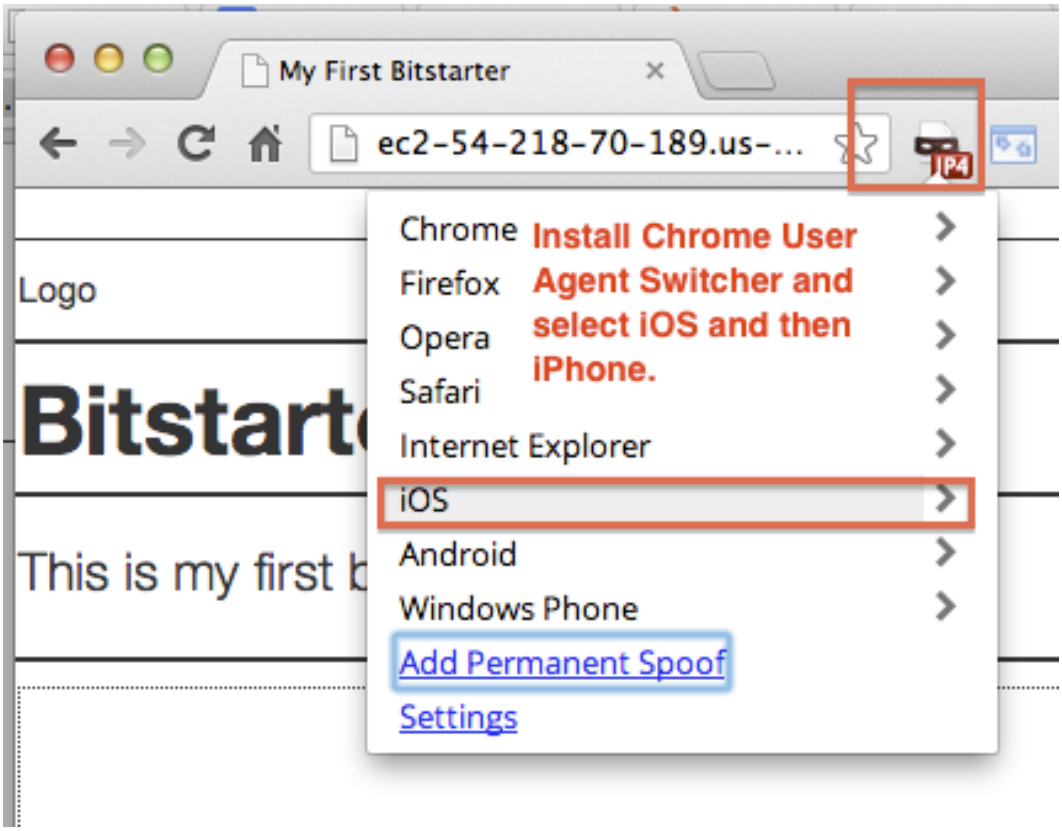

Figure 18: *To do mobile site previews on your desktop, set up Chrome's User Agent Switcher and select iOS -> iPhone 4.*

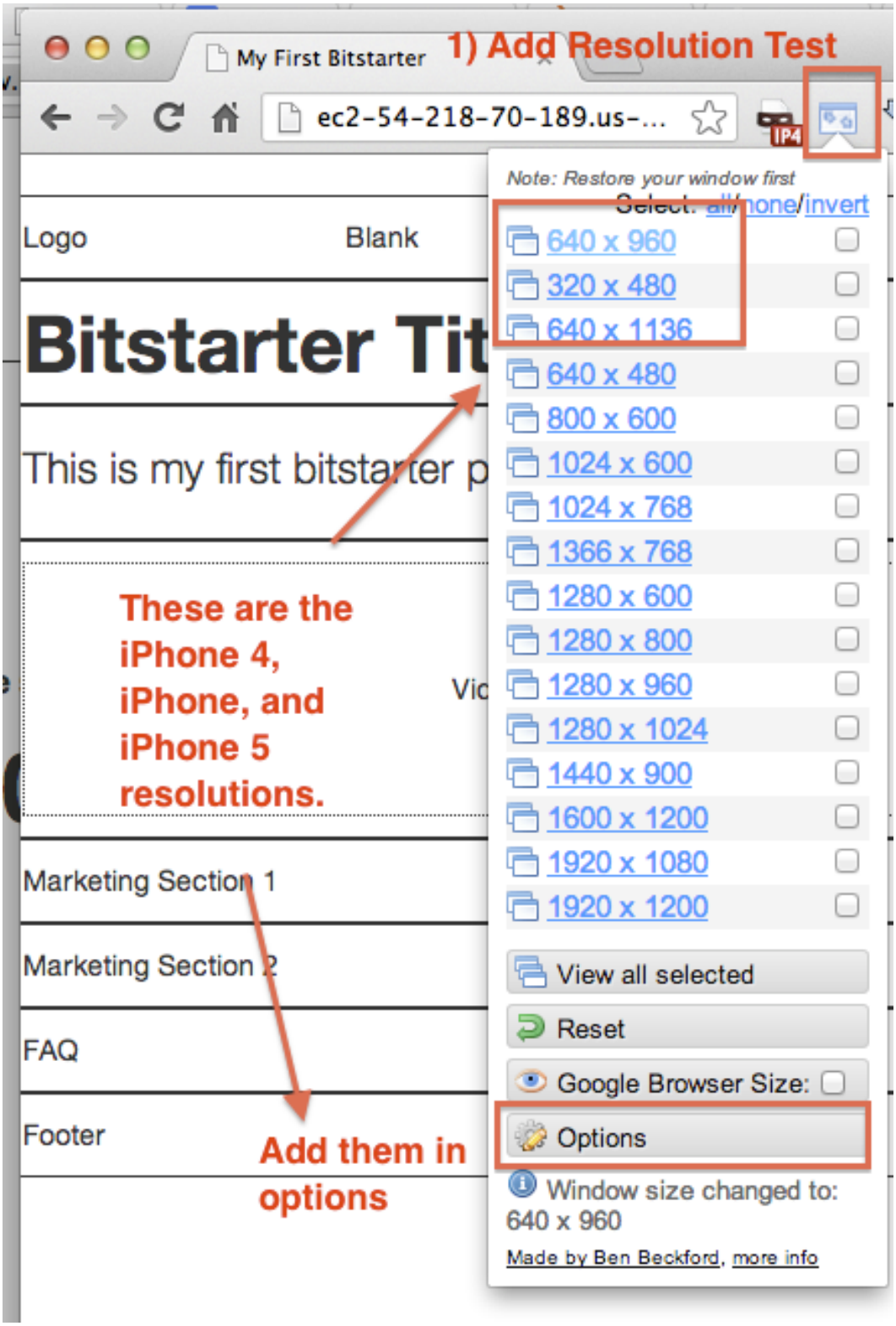

Figure 19: *To get the screen to the right resolution, use the Resolution Test Chrome extension as shown.*

At this point you've just set up a dev instance capable of editing websites and previewing both desktop and mobile versions over port 8080.

Creating and managing git branches

One of the most useful features of git is that it allows you to manage many simultaneous edits to multiple versions of a codebase. Suppose you have a production website and you want to add features. With git, a single engineer can use a three branch model (Figure 20) to make sure that all new features are thoroughly tested in dev and staging before being pushed to production. Moreover, this model scales up to support many engineers independently making non-overlapping changes to that website, previewing them on separate staging servers, and then interleaving them one-by-one, all while the production website remains unchanged. The fundamental tool for this process is the `git branch` command. Please read the following reference and 1, 2, 3, 4 as supplements; this will help you follow along with the screenshots in the next section.

Git Branching Model
Dev, Staging, Production

For the purposes of the class, we'll use a three branch model to implement the dev/staging/production flow. You will do primary edits of code in the **`develop`** branch. When you have a release candidate, git merge these edits into the **`staging`** branch and push to the staging server to preview what the live site should look like. Finally, when that looks good, we git merge the changes into the **`master`** branch and push to the production server.

The overall concept is that edits should flow one way: from the **`develop`** to the **`staging`** and then to the **`master`** branch. Bugs get caught before they hit the live site, and you can now roll back to an earlier production release.

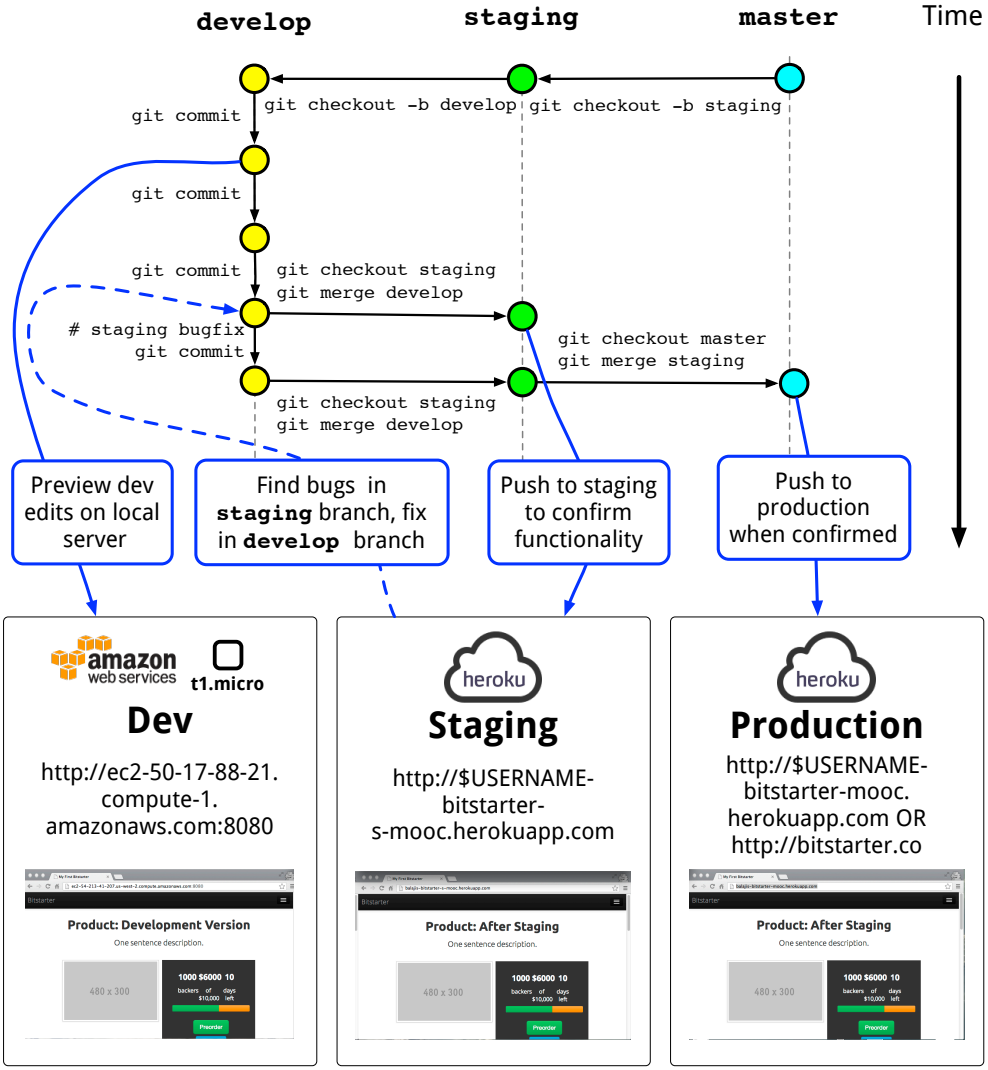

Figure 20: *We'll be using the following relatively simple model to manage changes to the live site. See nvie.com for a more complex one.*

Worked Example: Dev, Staging, and Production

Now that we've got our dev instance and understand a bit about git branching, we can illustrate the dev/staging/production flow. See the annotated interactive session and then the following screenshots (Figures 21-30). A few notes:

1. *Use your bitstarter repo.* Note that we have intentionally kept `github.com/startup-class/bitstarter` private for now to give you an incentive to do the interactive session yourself.
2. *Consult previous lectures.* Note also that the first portion of the interactive session below is straightforward given the previous lectures (especially Interactive Start and the Linux Command Line). The parts we are screenshotting begin at the `git checkout -b develop` line.
3. *Naming conventions.* When following along with the interactive session, please use the following naming convention for your Heroku apps. This will make it easy for us to manage on the Heroku end.

```
# Heroku apps have a thirty character limit
http://$YOUR_GITHUB_USERNAME-bitstarter-s-mooc.herokuapp.com  # staging
http://$YOUR_GITHUB_USERNAME-bitstarter-mooc.herokuapp.com    # production
```

Now let's go through the session. You should follow along, type in commands yourself, and review the screenshots (Figures 21-30) along the way.

```
# Assuming you launched a new EC2 instance with the ssh-http-https security
# group, run these commands to set up the machine.
sudo apt-get install -y git-core
git clone https://github.com/startup-class/setup.git
./setup/setup.sh

# Next, create an SSH key and (by copy/pasting with the mouse)
# add it to Github at https://github.com/settings/ssh
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub

# Now you can clone via SSH from github.
# Cloning over SSH allows you to push/pull changes.
# Note that you should substitute your own username and email.
git clone git@github.com:$USERNAME/bitstarter.git
git config --global user.name $USERNAME
git config --global user.email $EMAIL
exit # log out and log back in to enable node

# Next, change into the bitstarter directory and
# get all npm dependencies. Replace port 5000
# with 8080 if this is not already done, to allow
# serving over that port.
```

```
cd bitstarter
npm install
sed -i 's/5000/8080/g' web.js

# Create a development branch and push it to github
# The -b flag creates a new branch (http://git-scm.com/book/ch3-2.html)
# The -u sets that branch to track remote changes (http://goo.gl/sQ6OI)
git checkout -b develop
git branch
git push -u origin develop

# Create a staging branch and push it to github
git checkout -b staging
git branch
git push -u origin staging

# Login and add the SSH key you created previously to Heroku
# Then create heroku apps.
#
# IMPORTANT: Heroku has a 30 character limit, so use the naming convention:
# GITHUB_USERNAME-bitstarter-s-mooc for your staging app
# GITHUB_USERNAME-bitstarter-mooc for your production app
#
# Also see:
# https://devcenter.heroku.com/articles/keys
# http://devcenter.heroku.com/articles/multiple-environments
heroku login
heroku keys:add
heroku apps:create $USERNAME-bitstarter-s-mooc --remote staging-heroku
heroku apps:create $USERNAME-bitstarter-mooc --remote production-heroku

# Now return to dev branch, make some edits, push to github.
#
# NOTE: we use 'git checkout develop' to simply change into the develop
# branch without creating it anew with the '-b' flag.
#
# NOTE ALSO: We use 'git push origin develop' rather than 'git push -u
# origin develop' because we already set up the branch to track remote
# changes once. This means that if others edit the 'develop' branch we can
# merge their changes locally with git pull --rebase.
git checkout develop
git branch
emacs -nw index.html
git commit -a -m "Edited product desc in index.html"
git push origin develop

# Start node server in another screen tab to preview dev edits in browser.
```

```
# We use /usr/bin/env to avoid any issues from our rlwrap alias. Note
# that you can also run this as a background process with
# /usr/bin/env node web.js &, though you will want to redirect
# the log output.
#
# Then, in your browser go to the equivalent of:
# http://ec2-54-213-41-207.us-west-2.compute.amazonaws.com:8080
# Also see the branch you just pushed at:
# https://github.com/USERNAME/bitstarter/tree/develop
/usr/bin/env node web.js

# Once you have made enough edits that you like, it's time to merge commits
# into staging and then push to the staging server.
# See here to understand the 'git push staging-heroku staging:master' syntax:
# http://devcenter.heroku.com/articles/multiple-environments
# http://git-scm.com/book/ch9-5.html
git branch
git checkout staging # not checkout -b, because branch already exists
git merge develop  # merge changes from develop into staging.
git push staging-heroku staging:master

# Now inspect staging at USERNAME-bitstarter-s-mooc.herokuapp.com
# If you need to make edits, do them in develop, preview, and then merge into staging.
# Try to keep the flow of edits one way.
#
# Here is how we fix bugs detected in staging. First, we 'git checkout
# develop' again.  We make fixes (with emacs), save (with git commit -a -m),
# push them to github (git push origin develop), and then return to staging
# (git checkout staging).
#
# Now that we are back in the 'staging' branch, a status we can check with
# 'git branch', we again issue the all-important 'git merge develop'. This
# merges changes from the develop branch into staging.
git checkout develop
emacs -nw index.html # make edits and save
git commit -a -m "Fixed staging bugs in develop branch."
git push origin develop # push develop commits to github
git checkout staging
git merge develop  # merge changes from develop into staging.
git push origin staging # push staging commits to github
git push staging-heroku staging:master # push again to Heroku

# Once we confirm the website version deployed on staging works as intended,
# we merge into master and then push to production.
git checkout master
git merge staging
git push production-heroku master:master
```

```
118
119 # Assuming all goes well, go back to develop and edit some more.
120 git checkout develop
121 emacs -nw index.html
```

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout -b develop
Switched to a new branch 'develop'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
* develop
  master
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push -u origin develop
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
 * [new branch]      develop -> develop
Branch develop set up to track remote branch develop from origin.
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout -b staging
Switched to a new branch 'staging'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
  develop
  master
* staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push -u origin staging
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
 * [new branch]      staging -> staging
Branch staging set up to track remote branch staging from origin.
```

Figure 21: *Begin by moving into your bitstarter directory and creating a* `develop` *and* `staging` *branch. Push these to the* `origin` *as shown, which is Github in this case (see the* `bitstarter/.git/config` *file for details).*

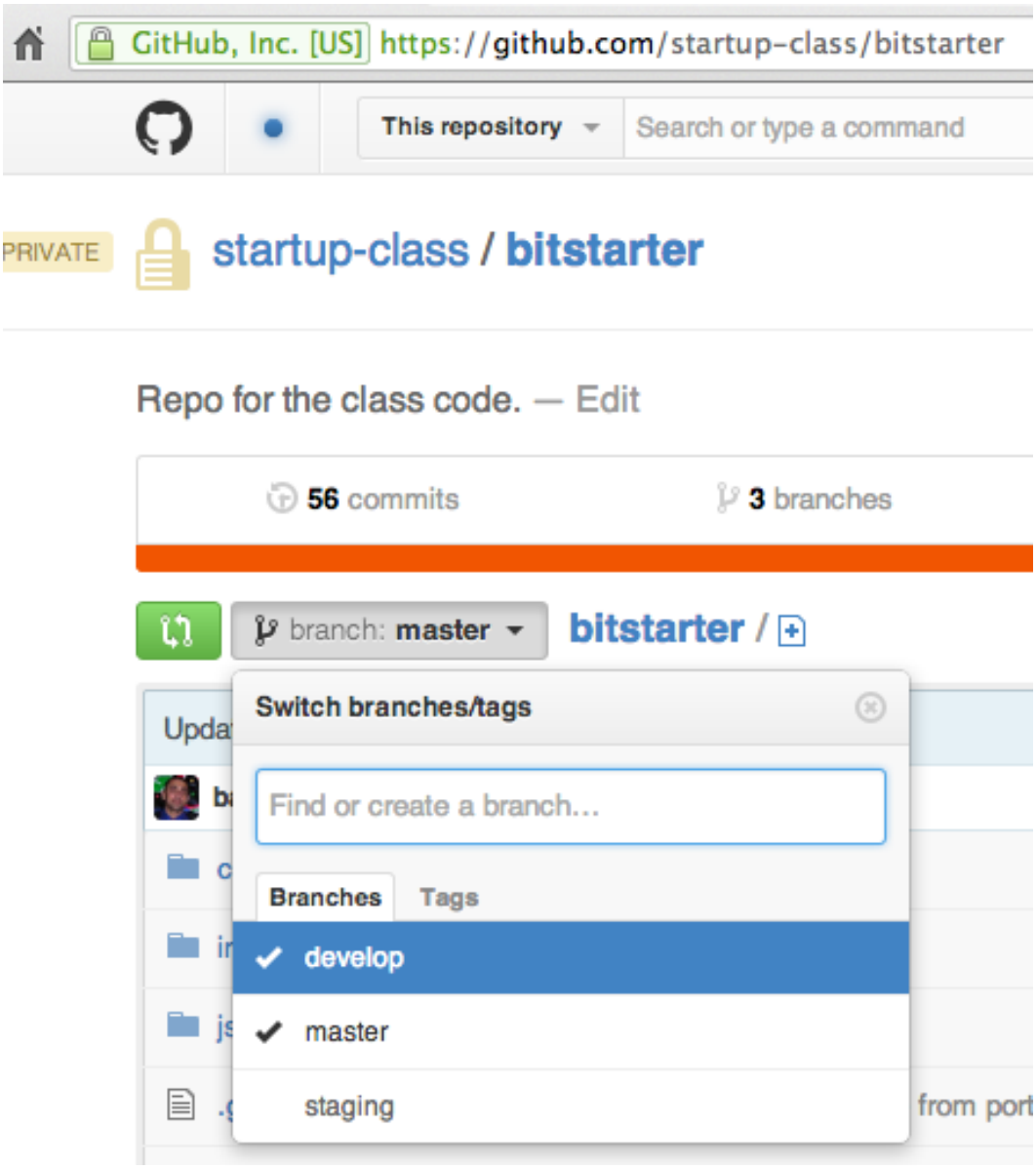

Figure 22: *After you have created these branches, go to your github repo online at github.com/USERNAME/bitstarter and you should be able to see the branches, as shown.*

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku login
Enter your Heroku credentials.
Email: balajis@stanford.edu
Password (typing will be hidden):
Authentication successful.
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku keys:add
Found existing public key: /home/ubuntu/.ssh/id_rsa.pub
Uploading SSH public key /home/ubuntu/.ssh/id_rsa.pub... done
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku apps:create balajis-bitstarter-s-mooc --remote staging-heroku
Creating balajis-bitstarter-s-mooc... done, region is us
http://balajis-bitstarter-s-mooc.herokuapp.com/ | git@heroku.com:balajis-bitstarter-s-mooc.git
Git remote staging-heroku added
[ubuntu@ip-172-31-34-196:~/bitstarter]$heroku apps:create balajis-bitstarter-mooc --remote production-heroku
Creating balajis-bitstarter-mooc... done, region is us
http://balajis-bitstarter-mooc.herokuapp.com/ | git@heroku.com:balajis-bitstarter-mooc.git
Git remote production-heroku added
```

Figure 23: *Now return to the command line, log into heroku and add your SSH key. This is the same key you generated and pasted into Github (see previous commands in the interactive session). Next, create a staging and production app as shown.*

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout develop
Switched to branch 'develop'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
* develop
  master
  staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$sed -i 's/Product/Foo/g' index.html
[ubuntu@ip-172-31-34-196:~/bitstarter]$git diff
diff --git a/index.html b/index.html
index bcd823a..81ccc0e 100644
--- a/index.html
+++ b/index.html
@@ -133,7 +133,7 @@
       <!-- www.google.com/fonts/specimen/Ubuntu#pairings -->
       <div class="row-fluid heading">
         <div class="span12">
-          <h1>Product: Development Version</h1>
+          <h1>Foo: Development Version</h1>
         </div>
       </div>
       <div class="row-fluid subheading">
[ubuntu@ip-172-31-34-196:~/bitstarter]$git commit -a -m "Updated product description."
[develop d20a72f] Updated product description.
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push origin develop
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 300 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
   4257b98..d20a72f  develop -> develop
```

Figure 24: *Let's now simulate the development process. We make an edit to a file, commit that edit, and then push these changes to Github.*

Figure 25: *Start up your node server in one screen tab...*

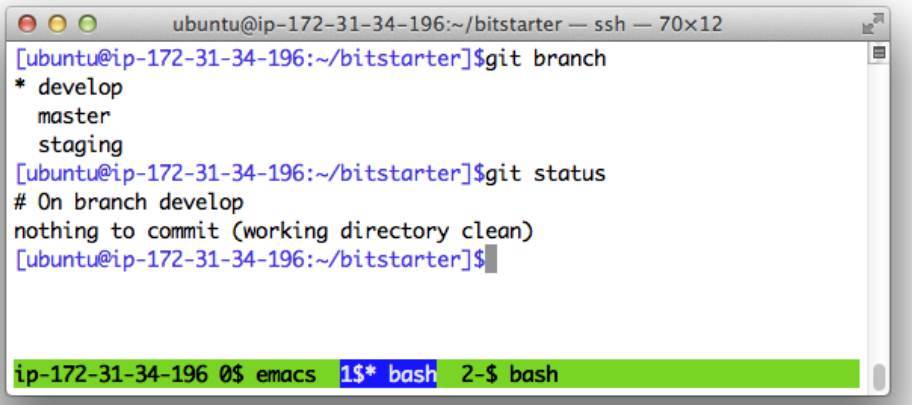

Figure 26: *... and continue editing in another.*

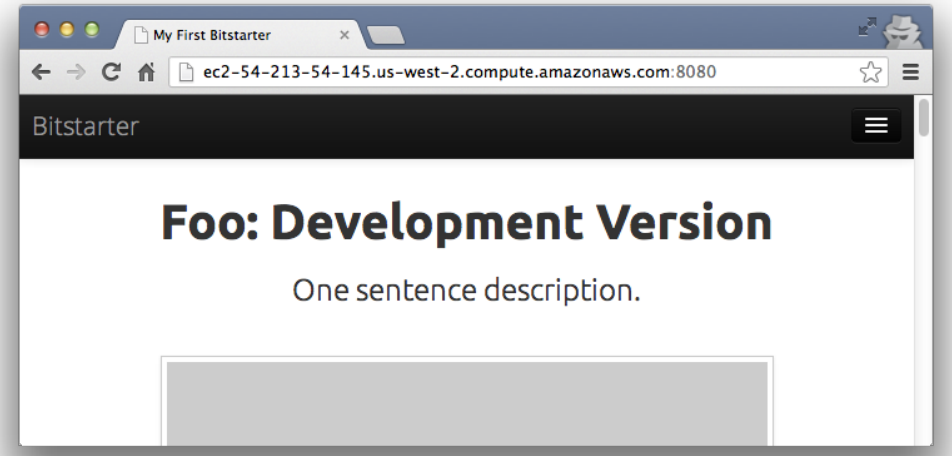

Figure 27: *Now you can preview your* `develop` *branch edits in the browser.*

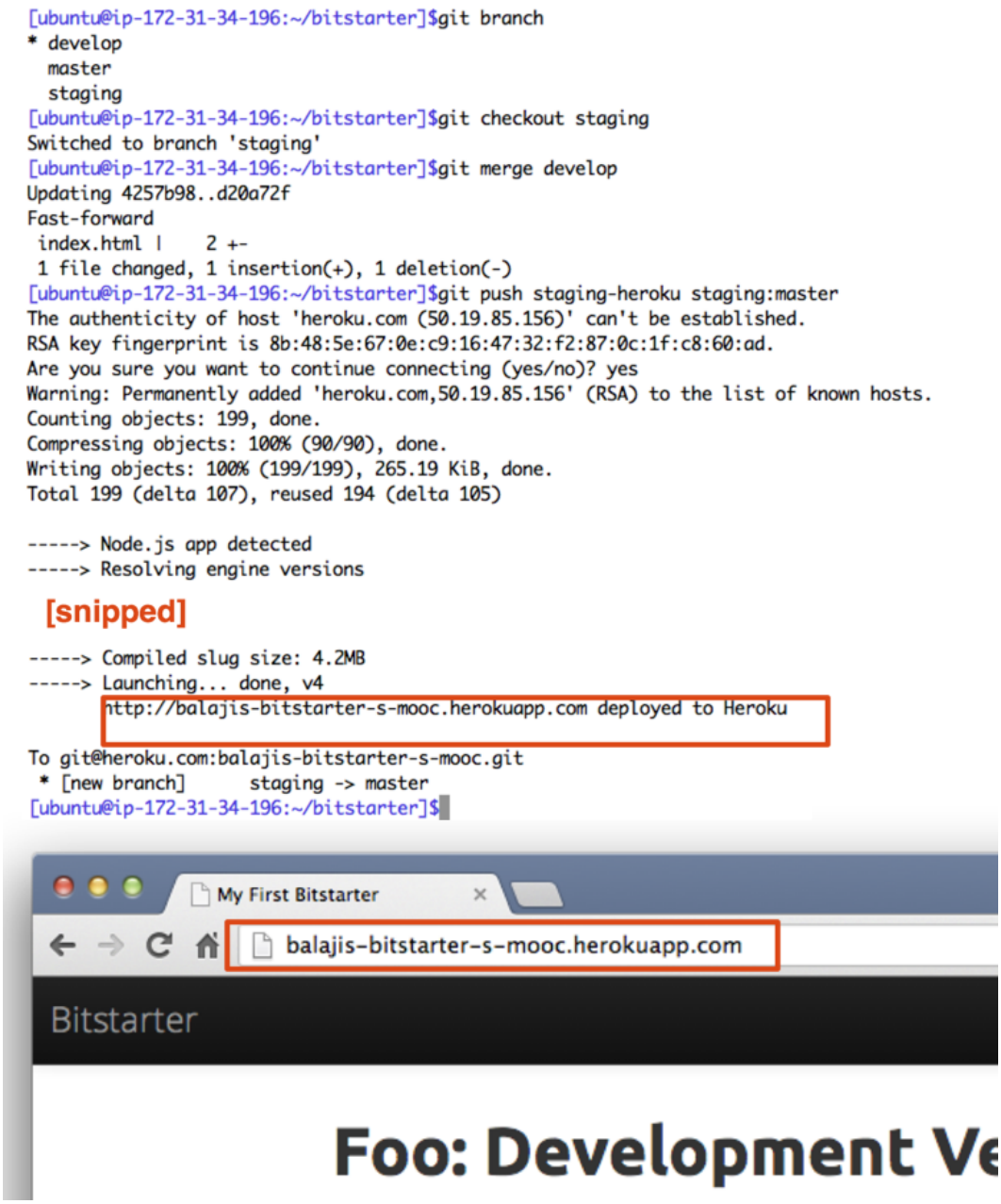

Figure 28: *When you feel you have a release candidate, you* `git checkout` *the* `staging` *branch and merge in the commits from* `develop`*, as shown. Then you push this* `staging` *branch to the* `master` *branch of the* `staging-heroku` *remote, which we set up with the* `heroku create` *command earlier. The syntax here is admittedly weird; see here and here for details.*

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout develop
Switched to branch 'develop'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
* develop
  master
  staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$sed -i 's/Foo:/Bar:/g' index.html
[ubuntu@ip-172-31-34-196:~/bitstarter]$git diff
diff --git a/index.html b/index.html
index 81ccc0e..9ef6f5d 100644
--- a/index.html
+++ b/index.html
@@ -133,7 +133,7 @@
       <!-- www.google.com/fonts/specimen/Ubuntu#pairings -->
       <div class="row-fluid heading">
         <div class="span12">
-          <h1>Foo: Development Version</h1>
+          <h1>Bar: Development Version</h1>
         </div>
       </div>
       <div class="row-fluid subheading">
[ubuntu@ip-172-31-34-196:~/bitstarter]$git commit -a -m "Fixed bug found in staging. Committing to develop branch."
[develop eddbb40] Fixed bug found in staging. Committing to develop branch.
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push origin develop # push to github
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 324 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
   d20a72f..eddbb40  develop -> develop
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout staging
Switched to branch 'staging'
Your branch is ahead of 'origin/staging' by 1 commit.
[ubuntu@ip-172-31-34-196:~/bitstarter]$git merge develop
Updating d20a72f..eddbb40
Fast-forward
 index.html |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push origin staging
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/bitstarter.git
   4257b98..eddbb40  staging -> staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push staging-heroku staging:master
```

[snipped]

```
-----> Compiled slug size: 4.2MB
-----> Launching... done, v5
       http://balajis-bitstarter-s-mooc.herokuapp.com deployed to Heroku

To git@heroku.com:balajis-bitstarter-s-mooc.git
   d20a72f..eddbb40  staging -> master
```

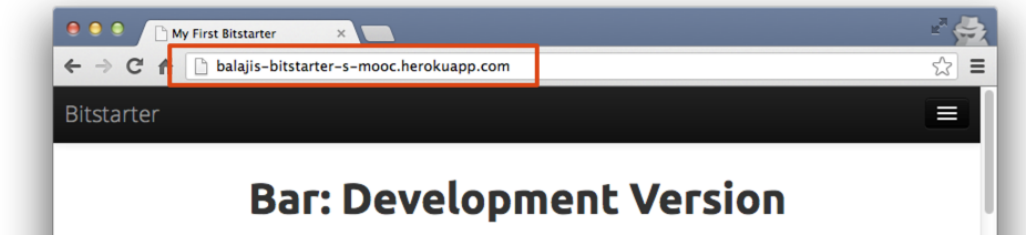

Figure 29: *Now let's say upon viewing staging we determine there is a bug - the text Foo should have been Bar. We go back to develop, make an edit and then push these commits to github with* `git push origin develop` *as before. When we are done fixing the bug, we again merge* `develop` *into* `staging` *and push to* `staging-heroku`*. And now we can see that the bug is fixed at the staging URL. The important thing here is that we are using staging as a "dumb" branch which only merges in commits from develop rather than incorporating any edits of its own. This is generally good practice: commits should flow from* `develop` *to* `staging` *to* `master` *in a one-way flow (though you can make this model more sophisticated to account for e.g. hotfixes and the like).*

```
[ubuntu@ip-172-31-34-196:~/bitstarter]$git branch
  develop
  master
* staging
[ubuntu@ip-172-31-34-196:~/bitstarter]$git checkout master
Switched to branch 'master'
[ubuntu@ip-172-31-34-196:~/bitstarter]$git merge staging
Updating 4257b98..eddbb40
Fast-forward
 index.html |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-34-196:~/bitstarter]$git push production-heroku master:master
```

[snipped]

```
-----> Launching... done, v4
       http://balajis-bitstarter-mooc.herokuapp.com deployed to Heroku

To git@heroku.com:balajis-bitstarter-mooc.git
 * [new branch]      master -> master
[ubuntu@ip-172-31-34-196:~/bitstarter]$
```

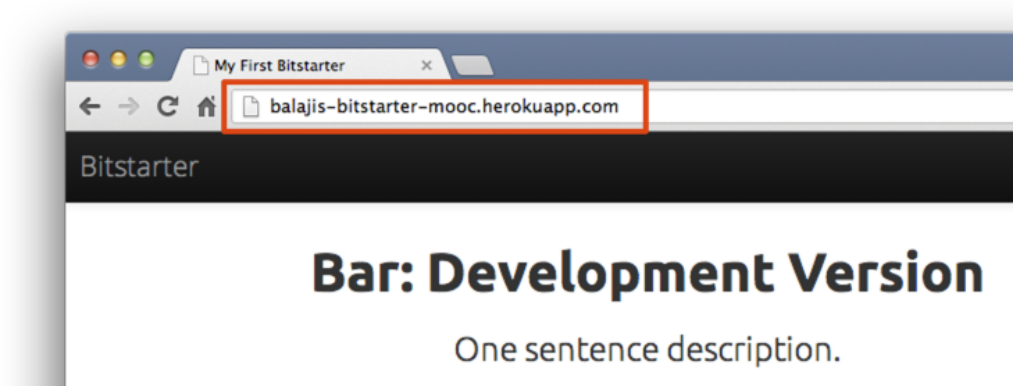

Figure 30: *Once we determine that staging is ready to ship, we simply* `git checkout master; git merge staging; git push production-heroku master:master` *to change to the master branch, merge in the commits from staging, and push to the production-heroku remote. At this point the site is deployed at the production Heroku URL.*

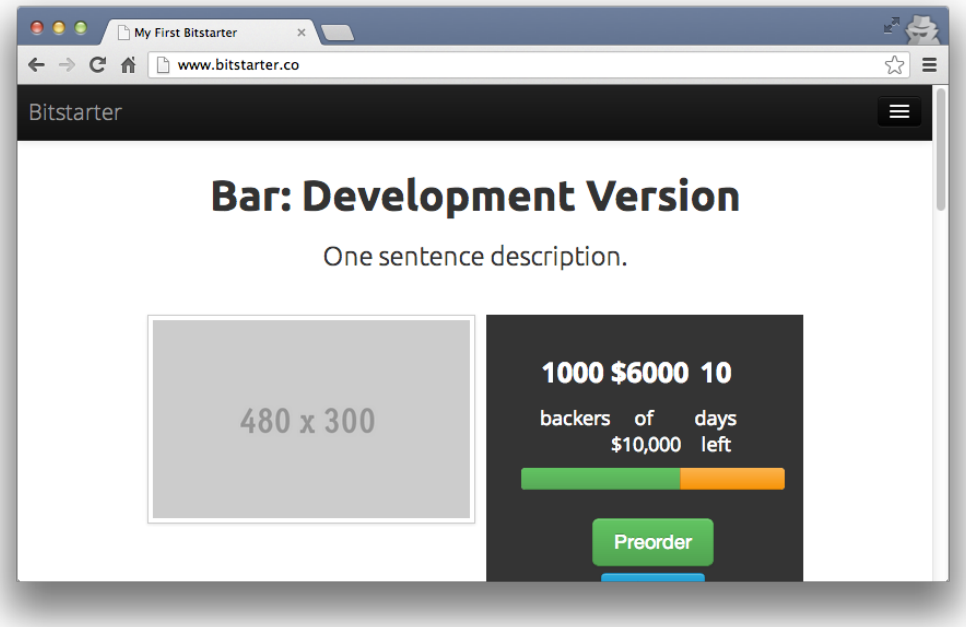

Figure 31: *Once we have deployed to production, the very last step is viewing the same content via our custom domain name. To learn how to set this up, read the next section.*

DNS and Custom Domains

This section is optional but highly recommended. It is optional as registering a domain will cost a little bit of money (<$20-$30), but will allow you to use your own custom domain (`example.com`) rather than a `example.herokuapp.com` domain. But this is recommended as a good domain will certainly help in raising money for your crowdfunder.

DNS Basics

Before registering our custom domain, let's talk briefly about the Domain Name System (DNS). When you type a domain name like `example.com` into the URL bar of a browser and hit enter, the browser first checks the local DNS cache for the corresponding IP address. If missing, it sends a request to a remote machine called a *DNS server* (Figure 32), which either gives back the IP address (e.g. 171.61.42.47) currently set for the domain name or else asks another DNS server nearby (Figure 33) if it knows the IP address. Given an IP address, your browser can then attempt to establish a connection to that IP address using the TCP/IP protocol, with the HTTP protocol then being a layer on top of that.

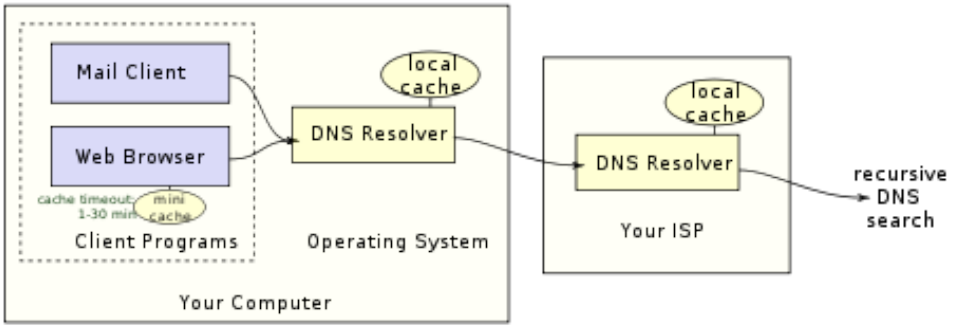

Figure 32: *When you input a domain name into your browser's URL bar, your computer first tries to look up the information locally, then remotely at the ISP, and then through recursive DNS search (Figure credit: Wikipedia).*

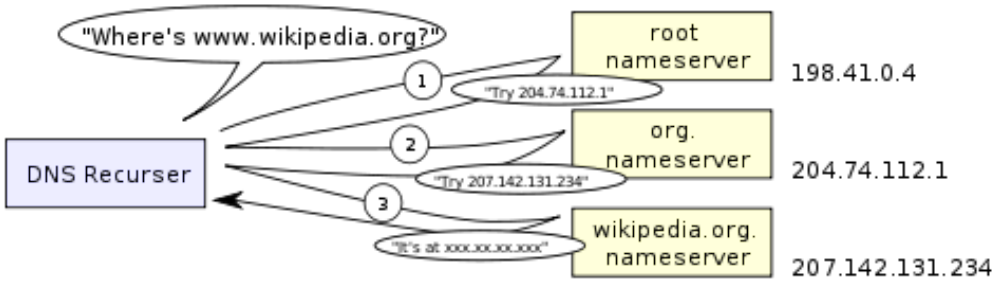

Figure 33: *This is what the recursive resolution mechanism for DNS looks like. The end result is the IP address of the requested URL. (Figure credit: Wikipedia).*

So there are at least three protocols at play here (DNS, TCP/IP, and HTTP), which seems surprisingly complicated. You might reasonably ask why we even need DNS at all, and why we don't just type in IP addresses directly into the browser.This actually does work (Figure 34), but has two caveats. First, humans find it much easier to remember words than numbers, which is why we use domain names rather than IP addresses for lookup. Second, engineers have utilized this "bug" of having to do a DNS lookup and turned it into a "feature"; some sites will have multiple servers with different IPs behind a given domain name (e.g. for load balancing), and on these different IPs may get repurposed for other tasks. The point is that the IP address behind a domain name can change without notice, especially for larger sites, so the domain name is really the canonical address of a site rather than the IP address.

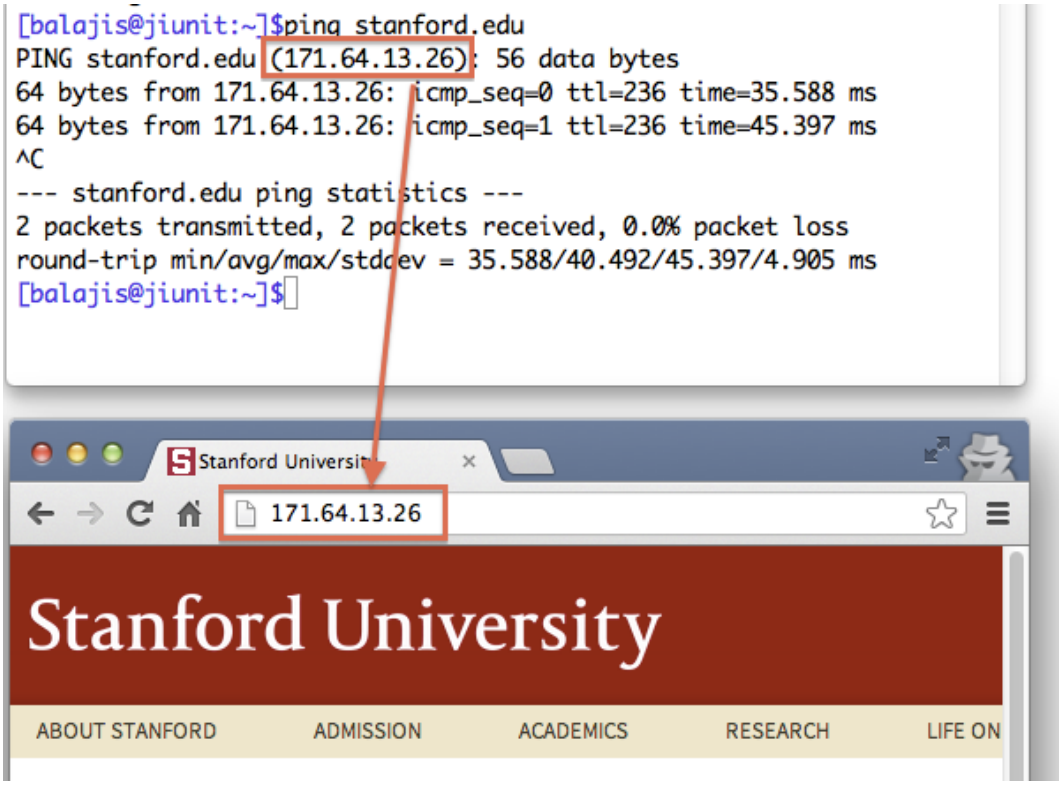

Figure 34: *You can use ping or another command to find the IP address associated with a domain and navigate to it directly. However, the exact IP address for the remote server is subject to change, while the domain will likely stay constant.*

While DNS is a distributed database in the sense that many nodes have a copy (or partial copy) of the domain-name-to-IP lookup table, it is highly nondistributed in one crucial sense: who is allowed to write new domain names to the database. For the last twenty years[2] the domain name registration system has been centralized, such that one has to pay one of many licensed registrars to get a domain name.

[2]With the development of Bitcoin it is now possible to distribute many protocols that were previously thought to be fundamentally reliant upon a central authority. In particular, the Namecoin project is a first stab at implementing a distributed DNS; it may not end up being the actual winner, but it has reawakened research in this area.

Finding a good domain: `domize.com`

When choosing a domain name, keep the following points in mind:

1. It should be easily pronounceable over the phone or in person.
2. If it's a product, consider having a unique string that distinguishes it; this will make it easier to find when just starting out (e.g. `lockitron.com` or `airbnb.com` were fairly unique).
3. You should consider SEO/SEM, using the Google Keyword Planner and Facebook Census tools from the market research lecture to determine the likely traffic to the domain once it can rank highly in Google search. Exact matches for search queries are highly privileged by Google, in that `ketodiet.com` will rank much more highly than `ketosis-dietary.com` for the search query [keto diet].
4. It should be as short as possible. If the .com is not available, get the .co or the (more expensive) .io. That is, get `example.io` over a spammy sounding name like `my-awesome-example.com`. The expense of the `io` domain here is a feature rather than a bug, in that it disincentivizes domain squatters.
5. If you use a non .com domain, consider using one that Google has whitelisted as a Generic TLD, unless you are genuinely international and want to rank for non-English searches (in which case, say, an `.fr` domain may be appropriate).
6. See whether you can get the corresponding Facebook, Twitter, and Github URLs (e.g. `facebook.com/ketodiet`).
7. For the US: see whether you can get the trademark using trademark search. A tool like this may be available in your country as well.
8. Much more difficult: see whether your name has a negative connotation in another language. It's almost impossible to check this without knowing many languages right now (though you could imagine creating a naming webservice that helped with this), but we mention it for completeness.
9. Don't spend too much time on a name, or too much money on a premium domain name from a place like `sedo.com` unless it's absolutely crucial for your business. Dropbox used `getdropbox.com` for many years before getting the `dropbox.com` URL, and FB used `thefacebook.com` before getting `facebook.com` and eventually `fb.com`. Though these did become nontrivial inconveniences it certainly didn't kill them. The only difference: rather than `getexample.com` or `theexample.com`, today one might do `example.io` first, and then get the `.com` later.

The `domize.com` site is a very useful tool for going through many combinations of domain names (Figure 35-36).

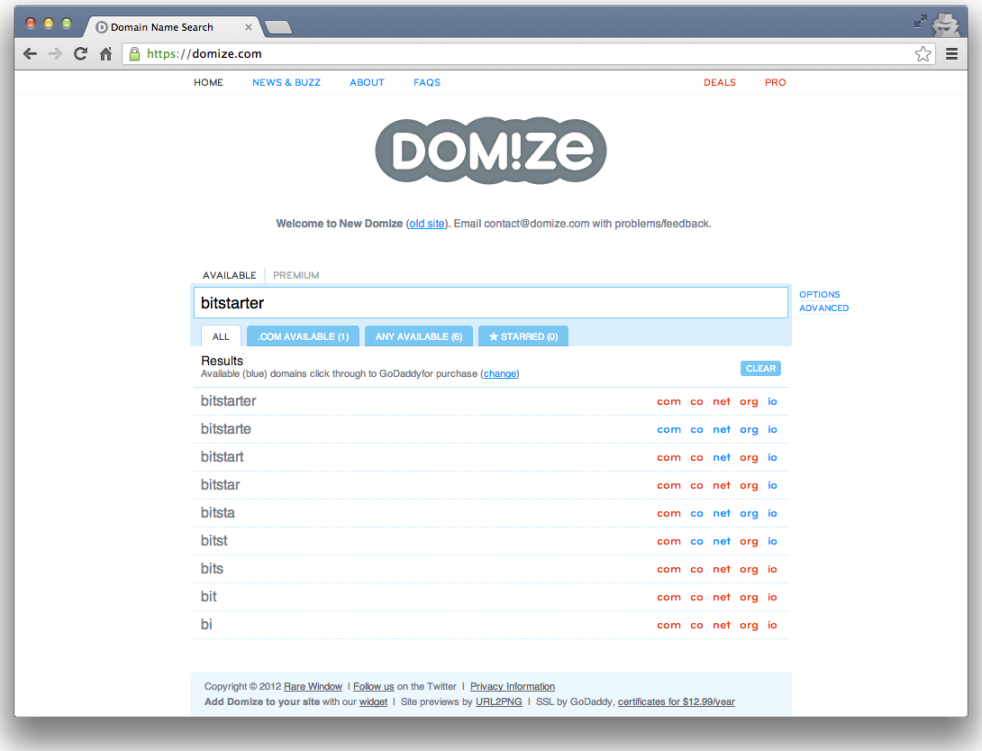

Figure 35: *Use domize to find an available custom domain name.*

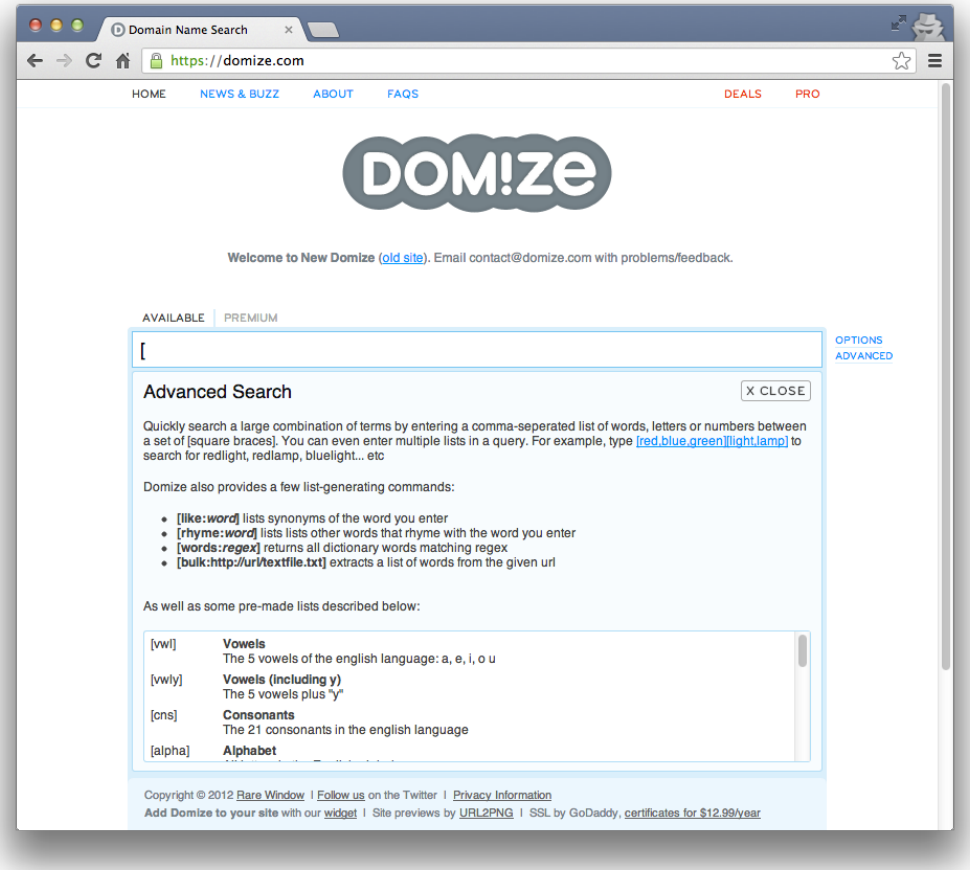

Figure 36: *Domize supports many advanced features for finding domains. Play around with these.*

Registering a domain: `dnsimple.com`

Once you've found an available domain through domize.com you will want to register it. Though it is more expensive than pure registrars like GoDaddy or NameCheap (say $20-$30 vs. $6), DNSimple is a good choice for a single domain which you plan to register and utilize *immediately*. This is because it gives you an extremely well designed UI and API for working with the DNS records for your domain. This is particularly helpful for beginners as DNS can be quite confusing. In Figures 37-39 we show how to sign up for DNSimple, register a domain, and edit DNS entries.

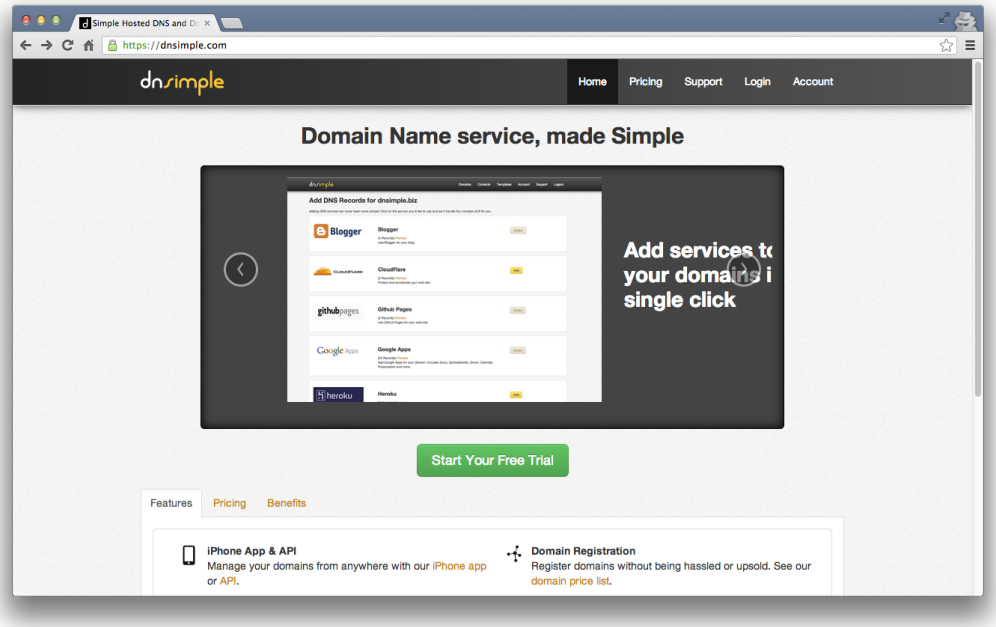

Figure 37: *Sign up for dnsimple.com.*

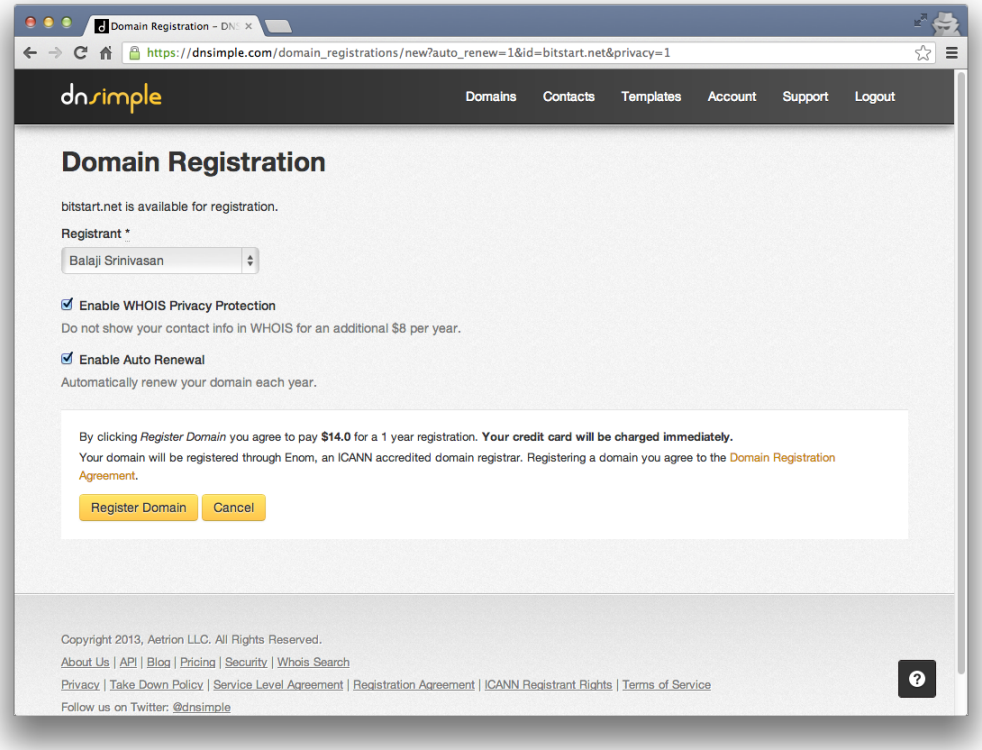

Figure 38: *At dnsimple.com/domains/new, you can register a domain. NOTE: you WILL be charged if you do this, so only do it if you actually want a custom domain. We do recommend enabling WHOIS protection to prevent your address being emailed by spammers.*

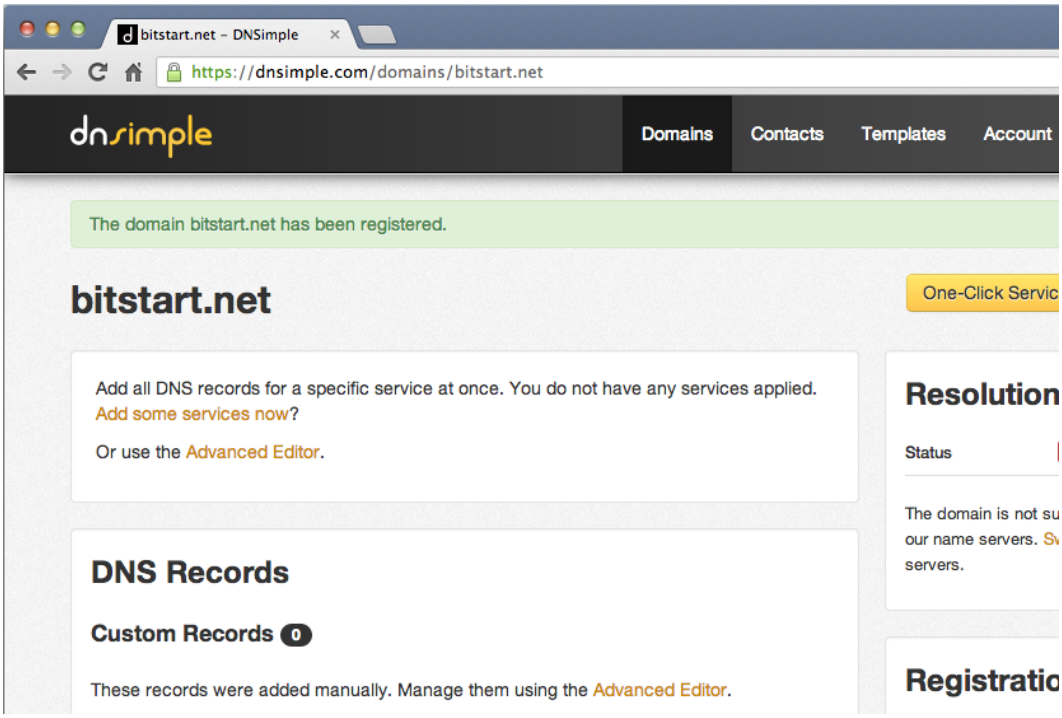

Figure 39: *Now you have a web interface for managing your domain.*

Configuring your DNS to work with Heroku

So, at this point you used domize.com to pick out a custom domain `example.com` and dnsimple.com to register it. You also have[3] `example-site-staging.herokuapp.com` and `example-site.herokuapp.com` live. But how do we make the `.com` point to `example-site.herokuapp.com`? We need to do two things. First, we need to configure things on the DNS side; with DNSimple this is very easy. Next, we need to tell Heroku to accept requests from this new domain. Finally, we confirm that it works by viewing things in the browser. Read the instructions here and here. Then look at the screenshots below.

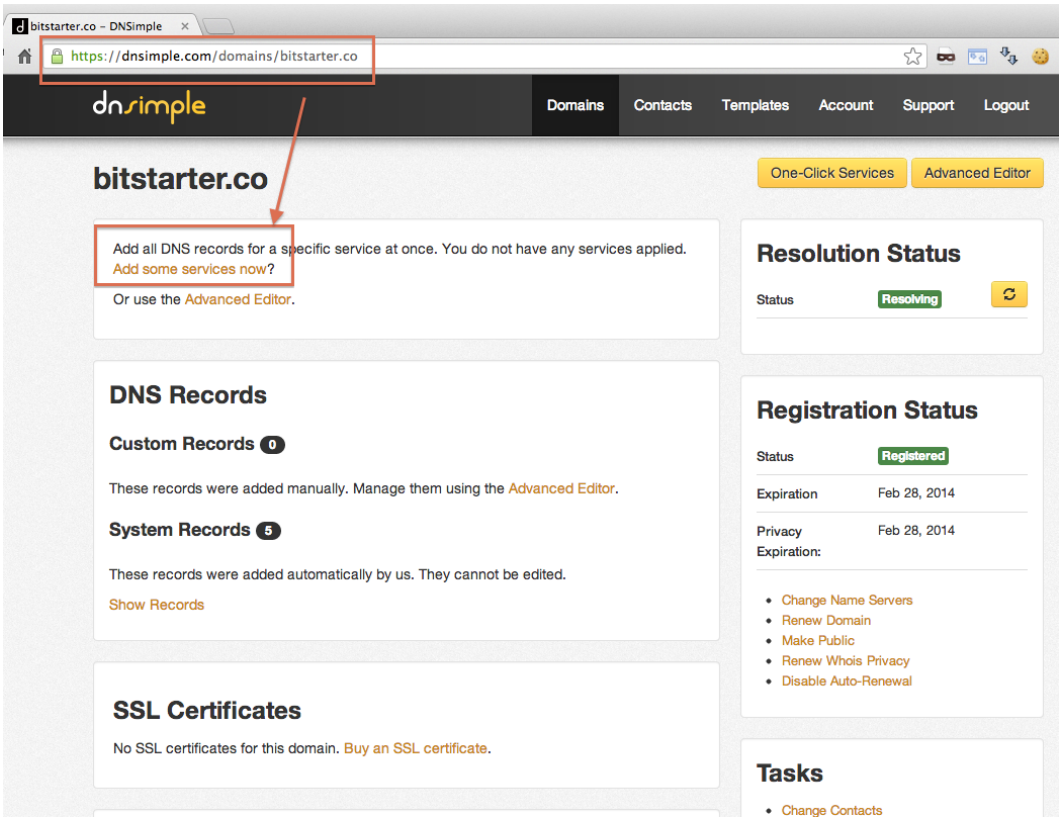

Figure 40: *Your goal is to have end users see a custom domain like bitstarter.co rather than bitstarter.herokuapp.com. To do this, first go to dnsimple.com/domains/example.com, plugging in your own registered domain for example.com, and then click Add some services now as shown.*

[3]Note that in this case the custom domain (`example.com`) has a similar name to the Heroku site(s) (`example-site-staging.herokuapp.com`, `example-site.herokuapp.com`). But this is by no means necessary, and can be useful to have staging URLs which do not include the custom domain's name (e.g. `foo-bar-baz.herokuapp.com`).

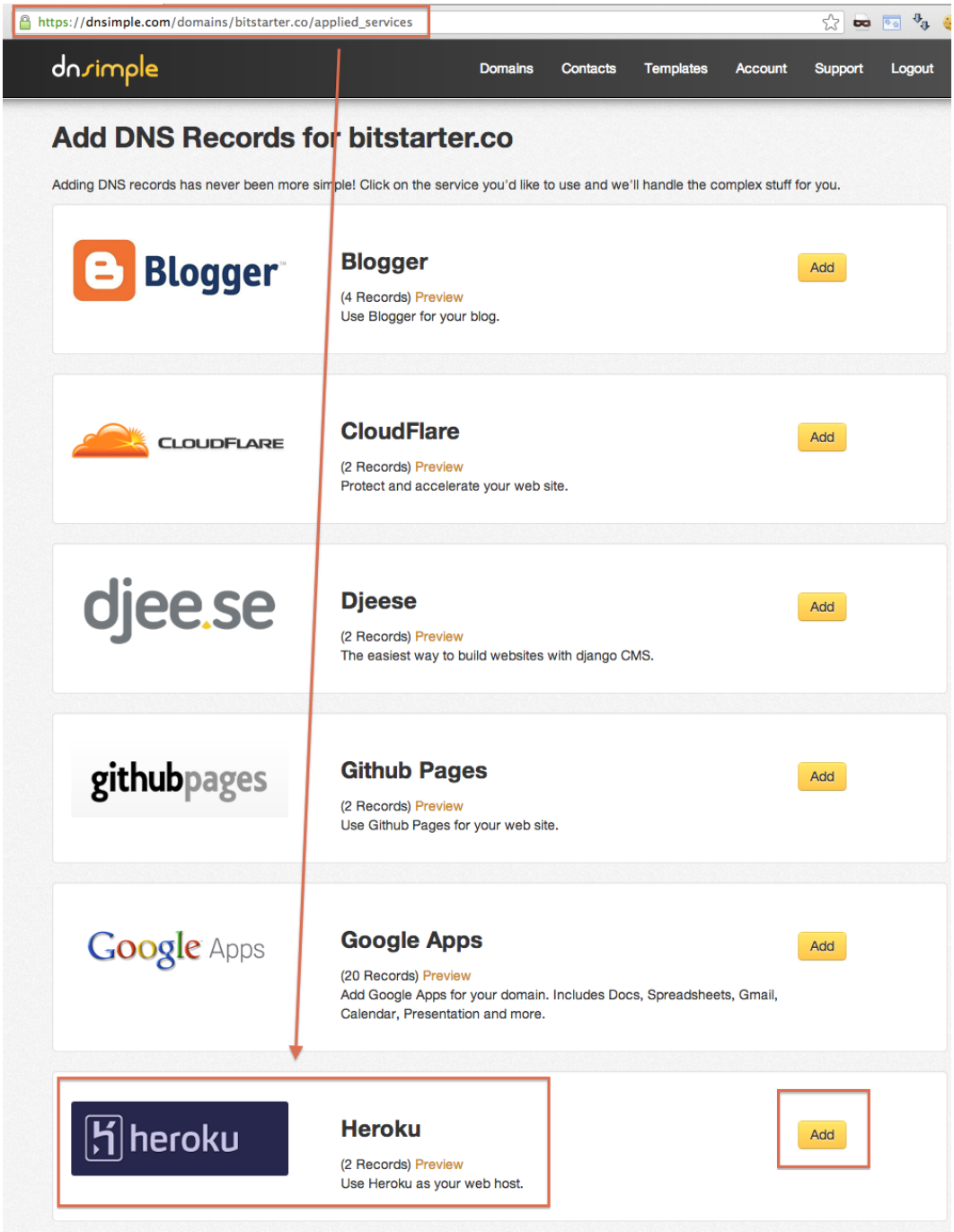

Figure 41: *Click Add as shown.*

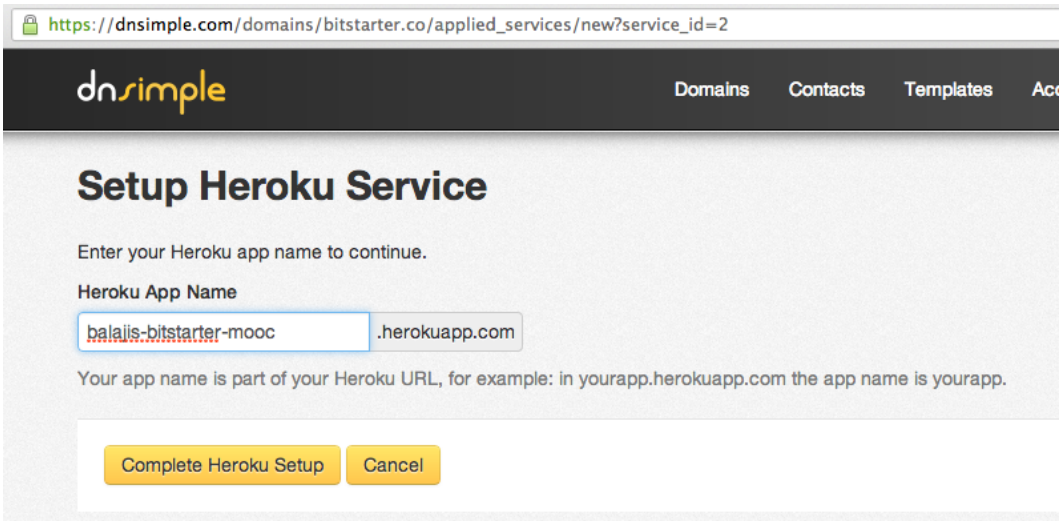

Figure 42: *Enter your app name. Note that this is your PRODUCTION app name (e.g. balajis-bitstarter-mooc.herokuapp.com) not your STAGING app name (e.g. balajis-bitstarter-s-mooc.herokuapp.com). This is because an external domain is only important for end users; the staging URL is viewed only internally by devs.*

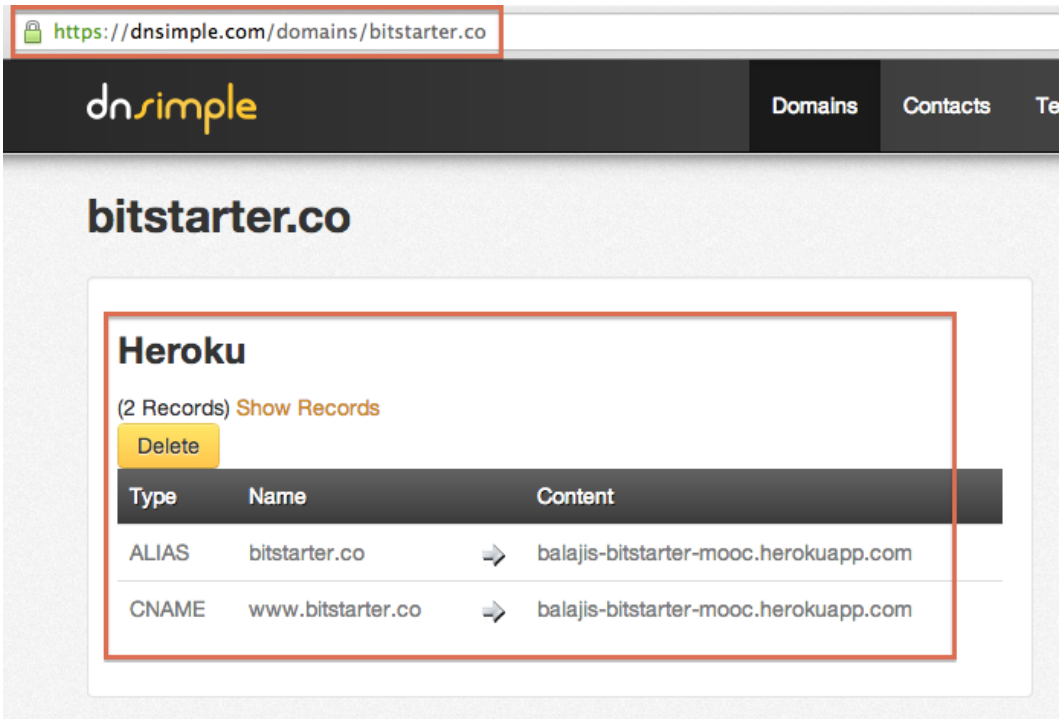

Figure 43: *Once things are set up, then you should see the following on your DNSimple homepage for this domain.*

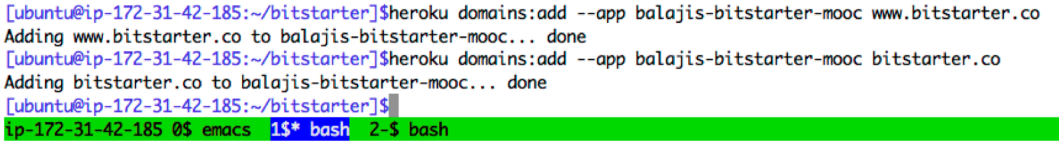

Figure 44: *Your final and key step is to authorize Heroku at the command line to accept connections to that app from this domain, for both the www.example.com and example.com variants.*

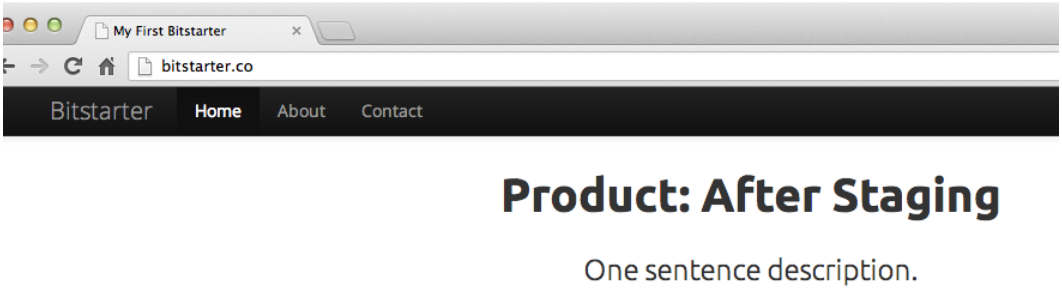

Figure 45: *If all goes well, this works in the browser.*

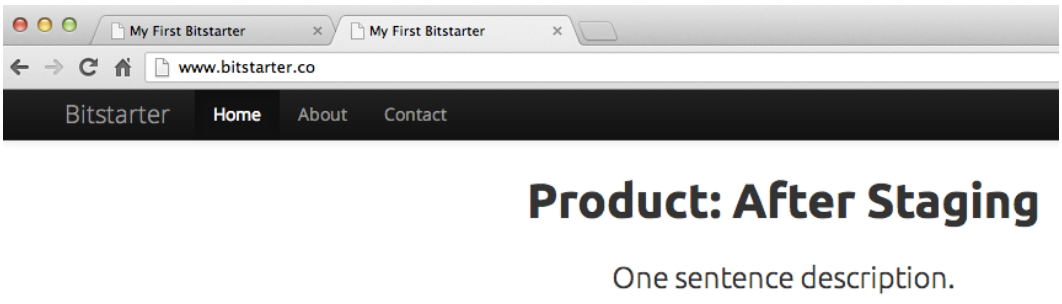

Figure 46: *If you have set it up right, both the www.example.com subdomain and example.com should route to the same place.*

Congratulations. You just set up a custom domain.

Setting up HTTPS and Google Apps

These final two sections are also optional, but recommended if you really want to build out a site.

1. *Get an SSL certificate.* This is a bit expensive (up to $100 if you get a wildcard certificate), but required to permit logins or other authenticated HTTPS connections to your website. To do this you can buy an SSL certificate through DNSimple at https://dnsimple.com/domains/example.com/certificates/new (substituting your own domain name). Then configure your SSL setup with Heroku, as detailed in these instructions: 1, 2, 3. Unless cost is a very pressing issue, get a wildcard certificate as it'll make your life easier in the long run (more details: 1, 2).

2. *Set up Google Apps for your domain.* You will probably want to be able to receive email at your domain to process feedback during the crowdfunder. One way to do this is by setting up Google Apps for your domain. Note that you might be able to get a free account through this link, but otherwise Google Apps costs about $50 per user per year (quite inexpensive as business software goes). If you want to continue, first sign up for Google Apps, configuring any desired aliases (e.g. `admin@example.com` and `press@example.com`), and then set things up within DNSimple to support Google Apps mail, chat, and other services. Then wait a little while for DNS propagation and then test it out. This process will be fastest on a new domain; with domains bought from others it can take up to one week for all the DNS records to propagate over.

If you do both of these steps, you should be able to both support HTTPS connections and be able to set up arbitrary email addresses at your new custom domain.