CSE 490H 08au
Assignment 4

Cloud-Based Web Application Hosting

Administrative Details

Assigned: Thu 11/20/08

Part A Due: Tue 11/25/08 by 4:30pm
Part B Due: Tue 12/9/08 by 4:30 pm

Partners: You are required to use the same partner(s) as from assignment 3.

Submit: Part A via an email to Aaron and Slava. Part B via turnin on attu.

Starter Source Code: http://www.cs.washington.edu/education/courses/cse490h/08au/projects/ec2source.zip

Introduction

In assignment 3 you:

- Generated map tiles comprising a map of the the United States, computed at several zoom levels
- Computed an index from every street address range in the United States to its (lat, lon) coordinate

In this assignment, you will connect these data structures to a web-site frontend application which displays map tiles for a particular range of the map, and can be relocated to point at any address the user requests via a form submission.

We will host these web applications "in the cloud" using computers provided by Amazon's EC2 (Elastic Compute Cloud) framework. We will upload our data into S3 (the Simple Storage Service) where our EC2 instances will retrieve the information.

This assignment is to be performed in two stages. The first of these stages (Part A) involves configuring your user account so that you can log in to EC2. The second stage (Part B) is the "meat" of the assignment itself. Assuming there are no hiccups, part A should take you no more than two hours. Of course, the reason we want you to do this early is because there are always hiccups. You will need to get your EC2 access issues squared away, so we're including a "mini assignment" to make sure you can turn EC2 on.

Part A - Getting Started with EC2

In Part A, you must do the following:

- Initialize your compute environment
- Create an SSH keypair registered with EC2
- Start an EC2 instance and read back some information

- Prove to Aaron and Slava that you have done so by emailing this information to them before the part A deadline

This is not complicated;the instructions here are lengthy because they are a step-by-step formula to getting access to the machines.

If you have not already read the EC2 Getting Started Guide, stop and do so now. http://docs.amazonwebservices.com/AWSEC2/2008-05-05/GettingStartedGuide/

If you are stuck, refer back to this document, and the notes from the course lecture on Amazon Web Services.

Download the EC2 API tools from http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
These tools require that Java is installed, and that the JAVA_HOME environment variable points to their installation root.

Unzip the API tools to a directory somewhere. Set the environment variable EC2_HOME to point to this directory.

Important information about our shared account

We are all sharing one big account for our Amazon Web Services usage. Much like our shared Hadoop cluster, this requires that we all tread lightly and be polite to one another. As with the Hadoop cluster, security and isolation is minimal. Don't cheat or copy one another's data; this is strictly forbidden. (Obviously.) Furthermore, please don't terminate one another's instances, etc. There are no system checks that will prevent you from doing so -- you just need to be careful.

The rules for our shared account:

Aaron has distributed to you a set of files that identify our shared account. There is one account for all of us.

- Do **not** forward the account credentials files to anyone else.
- Do not run more than one EC2 instance per group. If you bork an instance, shut it down with ec2-terminate-instances, and then start a new one.
- Only run m1.small instances; do not use any 64-bit instances.
- Do not create multiple S3 buckets -- only one per group.
- Do not write to any S3 bucket besides your own.
- Always launch your instances in a security group identified by your netid
- Always launch your instances with a keypair identified by your netid

Setting up the rest of your environment:

The EC2 credentials come in two files: one named pk-XXXXX.pem, which is a private key that identifies our account, and another named cert-XXXXX.pem which is our public certificate.

Set the environment variable EC2_CERT to the full path to the cert file. Set the environment variable EC2_PRIVATE_KEY to the full path to the private key file.

You'll probably want to put $EC2_HOME/bin in your PATH variable, so that you can run

the ec2-* commands without typing out the full path to them every time.

Testing that your environment is correctly set:

You should now have four environment variables set (JAVA_HOME, EC2_HOME, EC2_CERT, EC2_PRIVATE_KEY). You may have optionally updated PATH. Restart any cygwin shells that were started before you set your environment up.

Then type "ec2-describe-images"; you should get the following output:

```
~$ ec2-describe-images
IMAGE ami-a86783c1 490h-data/490h-server-i386.manifest.xml 540060130083 available private i386 machine aki-a71cf9ce ari-a51cf9cc
```

This identifies the Amazon Machine Image (AMI) our class will use. It's name is the "ami-a86...." in the second field. There are several other AMIs available (type ec2-describe-images -a to see all images you have access to), but this one has all the software neccessary for hosting your service already installed.

SSH Keys:

ssh, as you know, is the tool of choice to remotely log in to a machine somewhere. ssh will attempt to log in with whatever username you provide; the remote machine may ask for a password which you type in -- the password is then encrypted for transmission and sent to the server -- and the server either grants or denies access.

An alternate ssh authentication mechanism is called *public key exchange*. In this system, you create two files: a "public key" and a "private key." Each of these files contains a very long number (encoded in ASCII). These two numbers are chosen in a special way so that you cannot deduce the contents of one number from another, but any "messages" encrypted by the public key can only be read by the person holding the private key.

SSH can use these files to authenticate a user. You install the public key file on the machine you want to log in to. You keep your private key secret -- it only lives on your own machine. SSH can prove that you are who you say you are, because you are the only person with the private key that matches the public key the server already knows.

This is how you can log in to your EC2 instance. You **create a public/private keypair** on your computer, and send the public key to a database maintained by EC2. When EC2 starts an instance for you, it will inject a copy of your public key file into the new instance. You can then log in with the ssh private key. Since you are the only one with the private key, this is secure.

To create a keypair, run:

$ ec2-add-keypair *keyname*

The *keyname* is how Amazon refers to your key, to distinguish it from mine, a classmate's, etc. Use your net id for your keyname. This will emit several lines of text to the console -- this is the private key. Save all of this output in a file named "id_rsa-*keyname*". This is the traditional name for an RSA-encryption private key file. You may share this key with your partner so you can both log in to the same instance conveniently, but don't post your private key to the world -- treat it like a password.

ssh wants you to keep this file secure, too. So it will refuse to use the file as your private key unless you set its permissions to be private to you only. You'll have to change the file permissions with chmod:

$ chmod 0600 id_rsa-*keyname*

If you prefer to use the mnemonics, equivalent commands are:

$ chmod go-rwx id_rsa-*keyname*

Security Groups:

EC2 instances run in a datacenter shared with dozens (hundreds? thousands?) of other users. To keep you isolated from them, and vice versa, Amazon provides a *firewall service* to prevent unauthorized traffic getting to your machine. You must enable the "ports" on your computer that define the intra-machine addresses for the various services you want to host.

Your server will need to allow connections to ssh so you can log in. It must also allow connections to the web server. These run on ports 22 and 80, respectively.

EC2 firewall settings are applied to an object called a "security group." When you start an instance, you chose which security group it is created in. You must create your own security group, and enable the ports we need.

To create the security group, run:

$ ec2-add-group *groupname* -d "*Description of the group*"

Enter your netid for both the groupname and description. When it succeeds, it will give you back a line something like:
GROUP 540060130083 aaron Aaron

Now you must enable the ports you want:
$ ec2-authorize *groupname* -P tcp -p 22
$ ec2-authorize *groupname* -P tcp -p 80

If you then read back the information about your security group, you should see:

$ ec2-describe-group *groupname*

with output something like:
GROUP 540060130083 aaron Aaron
PERMISSION 540060130083 aaron ALLOWS tcp 22 22 FROM CIDR 0.0.0.0/0
PERMISSION 540060130083 aaron ALLOWS tcp 80 80 FROM CIDR 0.0.0.0/0

The "FROM CIDR 0.0.0.0/0" means that the entire Internet can access those ports. (Good thing ssh has the private key file check.) If you wanted to, you could restrict it so that only certain subnet addresses could access the machine.

Launching an instance:

You are now ready to launch your instance! First, run ec2-describe-images to get the name of our image ("ami-XXXXXXX").

Then run an instance of our image with the following command:

$ ec2-run-instances -t m1.small -n 1 -k *keypairname* -g *groupname* ami-*XXXXXX*

This will return information about your new instance, including the word "pending"; something like the following:

```
RESERVATION r-f0e54399 540060130083 aaron
INSTANCE i-96eb5bff ami-a86783c1 pending aaron 0m1.small
2008-11-20T03:23:40+0000 us-east-1b aki-a71cf9ce ari-a51cf9cc
```

The instance is not yet started; you have simply asked the EC2 provisioning service to run it, and it immediately returns. The provisioninng service will now find space for your instance, download a copy of our AMI to that machine, and then power it on for you. This takes about a minute or two.

After a couple minutes, see if your instance is ready. Run:

```
$ ec2-describe-instances
```

and it will give you a list of all our instances which are pending, running, or recently terminated. This gives you information about not only your instances, but all instances associated with our account. Fortunately, they are sorted by security group. Each "RESERVATION" line begins the list of instances associated with another security group. Find your reservation in the list, and then look at the instance following it. (You should only ever have one instance in your reservation at a time.) If your instance is still "pending," wait a bit. If for some reason it is marked "shutting-down" or "terminated", you'll need to launch another one.

Hopefully, though, you'll see a line that looks something like this:

```
$ ec2-describe-instances
RESERVATION r-f6e4429f 540060130083 aaron
INSTANCE i-98ea5af1 ami-2b5fba42
ec2-75-101-220-49.compute-1.amazonaws.com
domU-12-31-39-00-79-82.compute-1.internal running aaron 0m1.small
2008-11-20T02:55:51+0000 us-east-1b aki-a71cf9ce ari-a51cf9cc
```

The word "running" means that our instance has powered on. (It may not be 100% booted, but this takes less than a minute, so if you still have trouble, just wait a minute.)

The "ec2-75-...amazonaws.com" entry is the machine's external DNS name. You can ssh to your instance now by running:

```
$ ssh -i /path/to/id_rsa-keyname root@ec2-75-whatever-dns-you-have.amazonaws.com
```

Hopefully your machine is now running; you should be able to log in and get a welcome message and a terminal. You can now look around (cd, ls, and all the other usual linux commands are there). Point your web browser at the DNS address for your machine. You should see a web page with the magic word on it.

What to Turn in for Part A:

Email aaron and slava with:

- The magic word from the web page at http://ec2-your-address-here.amazonaws.com/
- The rev number from /etc/ec2/release-notes (type "cat /etc/ec2/release-notes" to read the file)

Shutting Down your Instance:

It is important not to just walk away and leave instances running indefinitely. We can only run a fixed number at a time, and they cost money (against our credit cap) to run, even when idle. To shut down an instance, log in as root and type "/sbin/shutdown -h now" without the quotes. This will shut down your instance.

You may notice that you're doing all of your commands on this instance as root. This means that you have unchecked access to all of the system's configuration. If you type a powerful delete command, for instance, it won't stop you from blowing away the entire machine's file system (or possibly locking you out, etc). If you get locked out of your instance, or otherwise screw it up, you will need to kill it "the hard way."

Type:

$ ec2-describe-instances

To get a list of all the instances. Find your instance in the list. It will have an instance id: "i-XXXXXXXX" in the second field. Terminate it with the command:

$ ec2-terminate-instances i-*XXXXXXXX*

This will terminate the instance. Remember: we all share an account. Make sure that you only do this to your own instances; other students will be very unhappy if you accidentally shut down theirs!

Important: If students report that they can't create an instance because other students have created too many, I'll just look through the list for whomever is running multiple instances and start terminating them until you're left with one per security group. Hopefully I don't shut down the one you are using :) Please monitor what your usage is.

Part B - Web Application Development

We will be using a *servlet engine* to allow you to write programs people can access through the web. This system is designed to connect small Java programs called "servlets" to URLs. The servlet is comprised of two things: Java code to run, and an XML file describing which Java classes are bound to which URLs. When a request is given via HTTP to a URL on your server, the servlet engine checks if it is a URL associated with a servlet. If so, it then runs your servlet program. The servlet program is given access to a *response* object which it can fill with a status code (e.g., "404 not found") and text (the "document body"). This response is then sent back to the originator of the HTTP request.

The particular servlet engine we will be using is called Tomcat 5.5. It is itself written in Java. You will need to download this from http://tomcat.apache.org/download-55.cgi to compile your source code. You do not need to configure this on your own system -- you just need to have the .jar files so that you can compile against them (much like with Hadoop).

Your servlet must:

- accept a 4-tuple of (address, street name, zipcode, zoomlevel)
- return a 4-tuple of (lat, lon, tile_x, tile_y)

The format for each of these is described in more detail below. In general, this project should be significantly less work than project 3. There is considerably less code to write. That having been said, you should still start early because:

- This project relies on you being a bit more of an expert user of a linux system than you might already be. You're going to need to get comfortable with these tools with unfamiliar syntax.
- You may need to go back to project 3 and modify your geocode index, depending on whether or not you set up your index in a way that makes it efficiently scannable.

Download the starter source code for this project if you have not already done so, and unzip it. We will call this directory $BASE.

This contains a few directories and files:

- **html/** - Contains the web front end (e.g., the map viewer); this is very similar to the one you had for project 3, but it has been programmed to use JavaScript to send requests to your servlet
- **WEB-INF/** - Contains the XML description of the servlet bindings
- **WEB-INF/lib/** - Contains Java libraries you need to include in your package
- **src/** - Contains starter source code
- **build.xml** - A script that can be parsed by ant to compile and package up your servlet. If you are doing this on a lab machine, ant is already installed. If you are working on your own machine, download and install ant from http://ant.apache.org/ now.
- **bootstrap** - A placeholder for your bootstrap script

Using ant:

Open the build.xml file in a text editor. Near the top is a property named "tomcat-home". You should edit the value="..." to point to the directory where you unzipped tomcat 5.5 to. This will allow compilation against Tomcat. You won't be able to compile otherwise.

To compile all your .java files in the $BASE/src/ directory, go to the $BASE directory and run "ant". This compiles .java into .class files.

To remove any intermediate and output files, run: ant clean
To build a file named map.tar.gz which contains everything you should upload to your server, run: ant tar

This requires cygwin on windows.

In the src/ dir...:

You must edit edu/washington/cse490h/geo/LookupServlet.java so that it contains your servlet body. An instance of your servlet will be constructed, and the doGet() method will be invoked when your URL is visited (each time an address is looked up).

The doGet() method must:

- Return status code SC_NOT_FOUND (404) if it cannot look up an address
- Write each of the latitude, longitude, tileId.x, and tileId.y to the 'out' writer if it can look up an address. Each of these must be on its own line. They must be in the same order listed here.

You are free to create your own helper methods, helper classes, etc, as necessary to make your implementation efficient. You may copy in some or all of your sources from project 3 to support this implementation. (Several classes such as your TileSetDivider implementation and record data types may prove *very* useful.)

The doGet() method receives an HttpServletRequest object from the servlet engine. This describes the entire request that was used to invoke the servlet. This is how you receive "arguments" that describe, for example, what address you want to look up.

The method request.getQueryString() will return a string that lists all the arguments to your servlet in the form "key1=val1&key2=val2&key3=val3&..." You should pull these arguments apart into individual entities. The keys and values themselves will be "URL encoded" so they can be transmitted in a URL. (For example, URLs cannot contain spaces, so they use a special escape sequence to encode this fact.) Use the java.net.URLDecoder.decode() method to turn a URL-encoded string into a "plain" string. The character encoding that you should assume is "UTF-8" (see the Java API reference for this method for more information if you need it).

The index.html page will send a request with four keys to your servlet. The keys are:

- **address** -- the number to lookup (e.g., "600")
- **street** -- the street the address is on (e.g., "apple st")
- **zip** -- the zip code (e.g., "98105")
- **zoom** -- the map's current zoom level (e.g., "6")

It is important that your lookup system be efficient. It should not need to do a linear scan of the entire index to find an address. Using hashing and sorted indices, you must be able to support O(log(n)) lookup time.

Uploading your system:

So you've written source code locally. Time to run it on EC2. Run 'ant tar' to compile your code and make it into a big tarball to upload. This will be in $BASE/map.tar.gz

You can upload the package with the 'scp' program. Use ec2-describe-instances to get the DNS name of your server, then do:

$ scp -i /path/to/id_rsa-keyname map.tar.gz root@your-instance-dns.amazonaws.com:

This will upload the map.tar.gz file to /root/ on your instance. If you specify a directory after the ...amazonaws.com:/some/dir/goes/here, it will upload the file there instead.

You should then log in to your instance and unzip this into the directory where tomcat looks for web applications:

root@yourinstance# tar vzxf path/to/map.tar.gz -C /usr/share/tomcat5/webapps

Tomcat caches your servlet in memory for higher performance. If you change your servlet, you may need to restart tomcat. Run:

root@yourinstance# /etc/init.d/tomcat5 restart

...to do this. Tomcat takes about 30 seconds or so to start up even after it returns "[OK]"; be patient.

To access the servlet itself, you can send a web browser to:

http://your-instance-dns.amazonaws.com:/map/
LookupAddress?key=val&key2=val2&key3=val3

If given the keys "address", "street", "zip", and "zoom" and good values for all the above, it should print four lines of text when you return lat, lon, tile_x, and tile_y. If it returns an error, time to debug! When it looks like this works, then try it through the web site interface. Navigate to http://your-instance-dns.amazonaws.com:/map/ (note:

the final '/' is important) and type an address in to the text boxes and click the lookup button. Hopefully this is successful :) If not, go back up to the previous step.

Getting data off our Hadoop cluster and into S3:

The astute reader will note that completing all of the above is pretty much doomed to failure on its own. How can this display your map, or look up addresses in your index when it's all trapped in our Hadoop cluster? So the first thing we have to do is get the data from Hadoop into EC2.

The server includes a hadoop-0.18.2 installation, which is itself configured to talk to our Hadoop cluster. This will require you to log in to the gateway node from the EC2 instance. So ssh to your EC2 instance, then SSH to *yourusername*@hadoop.cs.washington.edu with "-D 2600":

$ ssh -D 2600 *username*@hadoop.cs.washington.edu

Now you have a tunnel open to our hadoop cluster. Minimize this ssh session and forget about it for the time being. Use ssh to log in to your EC2 instance a second time.

Hadoop is installed in /usr/local/hadoop. So you should now be able to run:

$ /usr/local/hadoop/bin/hadoop fs -ls /

And see the contents of HDFS :)
More importantly, you will be able to run commands of the form:

$ /usr/local/hadoop/bin/hadoop fs -copyToLocal /some/hdfs/path /some/local/path

Using this, you can copy your data from HDFS onto your instance. You should put this data in the /mnt/ directory on your instance. Your instance has two disks associated with it. '/' has a small system disk mounted on it that is only a few GB. Just enough to boot the system. '/mnt/', however, is a 160 GB volume that is completely empty. All of your tile and geocode index data should be downloaded to here from HDFS.

Remember that your instance is ephemeral. When you shut it down, all the data on it disappears. (So if you modify source code on the instance itself, make sure you copy it off before you shut it down!) It is **a very bad idea** to always acquire data from our hadoop cluster every time you need it. Our hadoop cluster is in Seattle; EC2 servers are on the east coast. This is time consuming, puts a heavy load on our cluster, and incurs bandwidth charges ($$/GB) that we want to minimize. So we **only want to copy data from our Hadoop cluster once**.

After you run the copyToLocal commands necessary to get your data from HDFS into your instance, you should then **copy this data to S3** -- the Amazon simple storage service. This service lives in the same datacenters as EC2. Bandwidth between S3 and EC2 is much faster, and it's free.

There are two programs which allow you to access S3: **s3cmd.rb** and **s3sync.rb**. These commands allow you to copy local data to S3 and vice versa. s3cmd.rb is good for copying individual files. s3sync.rb is good for copying entire directories.

In the same email where you were given the EC2 credentials, you were also given a set of environment variables to use with S3. We will use these here. These environment variables are named AWS_ACCESS_KEY_ID, AWS_ACCOUNT_ID, and AWS_SECRET_ACCESS_KEY.

On your EC2 instance in your ssh terminal, run commands of the form "export VARIABLE=value" to set these. For example "export AWS_ACCESS_KEY_ID=0123456789". Do this for all the above. You can then use the s3sync.rb and s3cmd.rb programs.

(If you log out and log in with another ssh terminal, you'll need to repeat the above process.)

The first thing you should do with S3 is **create a bucket** to hold your data. Bucket names must be globally unique -- across our account, and all others -- so it is unlikely that you'll get a very basic bucket name like "bob". Try to name it something involving your netid:

$ s3cmd.rb createbucket *bucketname*

You can list all our buckets with:

$ s3cmd.rb listbuckets

You can then copy an individual file to S3 using the command:

$ s3cmd.rb put *bucketname*:*filenameInBucket local/path/to/file*

If you want to copy an entire directory to S3, you have a couple options:

- Create a .tar.gz containing all the files in the directory and then put that file with s3cmd.rb
- Use s3sync

In general, getting and putting many small objects is a very bad idea. Each get or put has a high overhead to start. So putting 10,000 files will take a very long time. Putting them in a compressed tar file and uploading that 1 file will take a very small fraction of that time. But copying 50 or 100 files with s3sync to S3 is ok.

To create a tarball, let's assume we have a directory named /foo/bar, and we want the contents of 'bar' and all its children to be in the tarball.

Do:

```
you:~$ cd /foo
you:foo$ ls
bar/
you:foo$ tar cvzf bar.tar.gz bar/
... all files in bar/ are listed..
you:foo$ ls
bar.tar.gz bar/
```

Then
you:foo$ s3cmd.rb put *mybucket*:*myFileInSpace* bar.tar.gz

To use s3sync, run:

```
you:~$ cd /foo
you:foo$ s3sync.rb -v --make-dirs bar/ myBucket:bar/
```

This will upload bar/ and all its contents to S3. s3sync.rb is picky about how you use trailing '/' characters; it may or may not store the directory name itself, depending on

this.

You will need to get all your tiles (For all zoom levels) and your entire Geocode index into S3.

Getting data out of S3 and in place on your server:

Assuming that you've uploaded your files into S3 and then shut down your instance.. you will eventually want to continue working on this project. Start a new instance -- no map tiles are there. This time, let's retrieve them from S3 -- much faster than transferring from UW!

First, set the AWS_... environment variables as described in the previous section.

Now, get individual files by running:

$ s3cmd.rb get *mybucket*:*myfilename* path/to/local/destination

Note that s3cmd.rb uses (s3obj, localobj) as its argument order; not necessarily (dest, src) or (src, dest) in particular.

To run s3sync.rb in reverse, run:

$ s3sync.rb -v --make-dirs *myBucket*:bar/ bar/

This does use (src, dest) as its argument order.

To uncompress a tarball, run:

$ tar vzxf tarballFileName.tar.gz

If you add the argument "-C some/dir", it will change directory to some/dir before unzipping.

The geo program from project 3 generated compressed sequence files as its output format for all your tiles. You need to "unpack" these files before you can display them on the web site. Run the extraction tool in edu.washington.cse490h.geo.TileExtractor to unpack the SequenceFiles. This will generate a directory called tiles/ with several numbered subdirectories, one per zoom level. Move the tiles/ directory into /mnt/html/ (so you have /mnt/html/tiles/). This is where the web page is programmed to look for the tile data.

You should put your geocode index into a directory under /mnt; where you put these files is up to you, as the code that must read this index is entirely in your control.

Bootstrap scripts:

All of the steps that you had to perform in the preceeding section must be done every time the instance starts. Ideally, you will eventually have a system that can turn on, and perform these steps automatically. This is done with a bootstrap script. The AMI is pre-programmed to download a file named *myBucket*:bootstrap from S3, and run it. You should write a shell script that runs all of the above unpacking commands, and use s3cmd.rb to put this in your bucket. There is a file named "bootstrap" in your starter source code; you should start from here.

Eventually, you should put your map.tar.gz file in S3. Your bootstrap script should

download this from S3 and unpack it to /usr/share/tomcat5/webapps. We will launch a copy of our AMI, with the metadata pointing at your bucket. Your bootstrap script should completely initialize your server so that it can be "cold started" with no human intervention.

Sending metadata to your instance:

When the system is bootstrapping, how does it know which bucket to read the file from? How does it know what to set the AWS_ACCOUNT_ID, etc, environment variables to?

This is accomplished with instance-specific metadata. Your bootstrap script is run by yet another script, which is already in the AMI. This is called start-bootstrap. You can set the environment variables that are present in start-bootstrap with a specially formatted metadata string, of the form "VAR=value:VAR2=value2:VAR3=value3:..." This string can be passed to ec2-run-instances with the -d argument.

e.g.,

$ ec2-run-instances ami-*XXXXX* -g *group* -k *keypair* -d "FOO=fval:BAR=bval"

The environment variables that are needed are:
AWS_ACCOUNT_ID
AWS_SECRET_ACCESS_KEY
AWS_ACCESS_KEY_ID
BUCKET

The first three are their usual names; the final one of these identifies your bucket, from which it looks for a file named 'bootstrap'.

Extensions:

Totally not necessary, but if you're ambitious, here are some random ideas...

- Do more clever things with the map; add "pins" that float over addresses you look up; allow more interaction with the mouse, etc. This requires knowledge of JavaScript/AJAX and modifying the index.html file we gave you
- Create another servlet that can parse an address out of a single line, rather than requiring users to break up the fields themselves.
- Do some mashup with the real google map API widget side-by-side with yours, or with data from both overlaid on a single surface (?!). Sky's the limit.
- Driving directions. Satelite coverage. Telephone number lookup by address. Totally awesome and ridiculous mappy things, etc.

Hints and Tips:

- Write a non-web-based lookup system first with its own main() method and test this first
- When this works, directly send URLs to http://your-address.aws.com/map/LookupAddress?... to test the servlet interface
- Then and only then should you be testing with the AJAX interface in /map/index.html
- When writing your bootstrap script, test individual commands on the command line as you add them to the script. Then run the script start-to-finish on your own... then upload it to S3 as the bootstrap script and see if it powers your instance on properly.
- When writing and debugging a bash script, replacing "#!/bin/bash" with "#!/bin/bash -x" at the top yields useful information.

- Putting the command "set -e" in your bash script makes it halt if a sub-command returns an error. "set +e" undoes this.
- Important: we get charged by the hour for each instance we run. This is fine, as we have many hours available to us. But if you run an instance for even a minute, shut it down, and start another one -- that 1 minute instance also costs an hour. So don't start and stop instances 50 times an hour to test your bootstrap system, as that's the same as running 50 instances for an hour/1 instance for 50 hours. Run your script on the command line on one instance yourself, tweaking things as needed, and only run it "from boot" after you're pretty sure it works.
- Leverage as much functionality from your project 3 code as possible.
- If you don't know how a linux command works, type "man *somecommand*" to read its manual page.
- Start early, naturally :)

What to Turn In:

- All source code for part B
- A writeup answering the following questions:

1. The names / netids of everyone in your group
2. What is your bucket name?
3. Describe your servlet system design.
4. What steps does your bootstrap script do?
5. What parts of project 3, if any, did you reuse? For what purposes?
6. Did you need to change how you computed the geocode index from what you turned in for project 3? What changes did you need to make? How did these help?
7. Describe how you tested your system
8. We are going to launch your server by running our AMI with ec2-run-instances. This must completely boot all the way to a functioning web app with no more intervention on our part. What is the metadata string neccessary to run your server like this?
9. As a purely fun and voluntary contest, we are going to hold a "demo contest" to show off everyone's successful implementations. We will email the mailing list with a list of servers running your map implementations. Then you can all look at these and vote on who has the niftiest map. If you would like to be included in the contest, please say "yes" here.

References and Links

EC2 Getting Started Guide: http://docs.amazonwebservices.com/AWSEC2/2008-05-05/GettingStartedGuide/
EC2 Tools: http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
Tomcat 5.5 download: http://tomcat.apache.org/download-55.cgi
Tomcat 5.5 reference manual: http://tomcat.apache.org/tomcat-5.5-doc/index.html
Starter source code: http://www.cs.washington.edu/education/courses/cse490h/08au/projects/ec2source.zip
Ant: http://ant.apache.org/

The preceding material is from the University of Washington Computer Science & Engineering senior undergraduate course:

CSE 490H
Scalable Systems: Design, Implementation and Use of Large Scale Clusters
Autumn 2008

For further information (including all lecture material) see:

http://www.cs.washington.edu/education/courses/490h/08au/