NETS 212: Scalable & Cloud Computing

Fall 2014

Assignment 1: Image Search "Software as a Service"

Milestone 1 due September 18th at 10:00pm EST

Milestone 2 due September 25th at 10:00pm EST

For your first homework assignment, you will focus on **key-value stores** and **AJAX web interfaces**. Over the course of two milestones you will be building an **image search portal** based on the images in Wikipedia (or, specifically, DBPedia, which is an extracted version of Wikipedia). You can find out more about DBpedia at:

- **dbpedia.org**
- Auer, S.; Bizer, C.; Lehmann, J.; Kobilarov, G.; Cyganiak, R.; Ives, Z.: DBpedia: A Nucleus for a Web of Open Data. 6th International Semantic Web Conference.

You will be using **Node.js** and **Express** to write the "Software as a Service" component, and Java to write the indexer component.

Milestone 1: Web image-search front-end (Due Sept. 18)

The first stage in this assignment involves completing an existing Node.js front-end we have provided for you. This front-end will connect to **Amazon DynamoDB** for its storage system.

Code Checkout and Migration to Your Repository

Start by clicking on the Eclipse icon on the left side of the screen. Then, as with Homework 1:

- Right-click in the **Package Explorer** or **Project Explorer** (whichever you see) and choose **Import…**
- Select **SVN | Checkout Projects from SVN** (this is probably already highlighted)
- Choose Use existing repository location: and highlight (or create) the `svn+ssh://nets212@eniac.seas.upenn.edu/home1/n/nets212` entry. Hit **Next**.
- Enter your password to unlock the private key (`nets212`).
- Choose **HW1-webapp**, then **Next**.
- Change the **Project Name** to `HW1-webapp`. Click **Finish**.

Next, as before, we want to switch the code to your own Subversion repository:

- Now right-click on **HW1-webapp**. Choose **Team | Disconnect…** Select **Also delete the SVN meta information** and hit **Yes**.
- Right-click **HW1-webapp**. Choose **Team | Share Project…** and choose **SVN**. Click **Next**.

- Choose **Create a new repository location** and enter the URL to your own subversion repository on eniac:
  `svn+ssh://{your userid}@eniac.seas.upenn.edu/home1/{first initial}/{your userid}/nets212`
- Hit **Next**, then be sure to choose **Use project name as folder name**.
- Eventually you'll be prompted to **Confirm Open Perspective** with the **Synchronize View**. You can say **No**.

Setting up Amazon Web Services

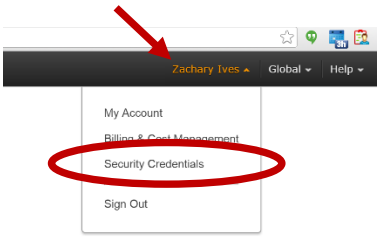
Next, you will need to sign up for Amazon Web Services, from any browser.

1. Go to `aws.amazon.com`, click **Sign Up** and enter your user information and a credit card.
2. Then go to `aws.amazon.com/awscredits` and enter your AWS credit code (given out by email).

This should give you $100 in credits towards AWS resources. Beyond this your credit card will be charged. **Note that only one code will work per account**.

Now let's set up the software.

Log in to http://aws.amazon.com/ ***from your virtual machine***, and find the menu in the upper right corner with your name in it. Click to drop that down.



Select **Security Credentials.** If you see a dialog box recommending that you create IAM Users, instead click on **Continue to Security Credentials**.

Click the plus sign to expand **Access Keys** and hit **Create a new Access Key**. A dialog box will open that tells you your key has been created successfully. Click **Show Access Key**, then copy, paste, and save your **Access Key ID** and the **Secret Access Key** as follows...

Open the **Terminal** on your Virtual Machine. Enter **`mkdir ~/.aws`** and **`mkdir ~/.ec2`**, and then **`gedit ~/.aws/credentials`**. Enter the following into the **gedit** window, then save and exit:

```
[default]
aws_access_key_id={your access key, no braces}
aws_secret_access_key={your secret access key, no braces}
```

Next switch back to your Eclipse project (from the above instructions) and open **config.json**. Change **accessKeyId** to have the value of your access key (in quotes), and **secretAccessKey** to have the value of your access key (in quotes).

Also **Download Key File** and save **rootkey.csv** in `~/.aws`.

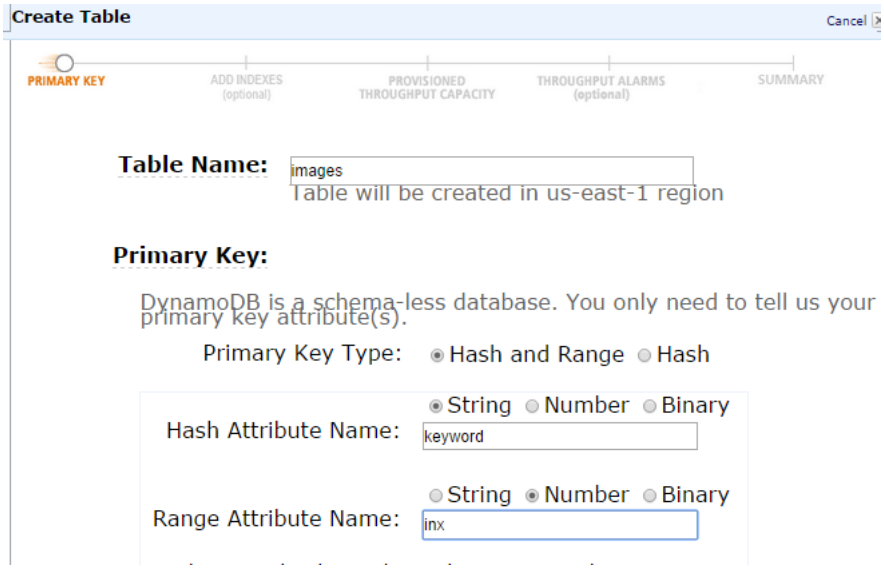
Click the plus sign to expand **X.509 Certificates** and create a new certificate. Save the Private Key file as `~/.ec2/access.pem` and the X.509 Certificate as `~/.ec2/cert.pem`.

Configuring Your Cloud-Based Storage

Next we will set up **DynamoDB**, one of Amazon's storage systems. From **aws.amazon.com**, go into **AWS Management Console**, select **DynamoDB** (under **Database**).

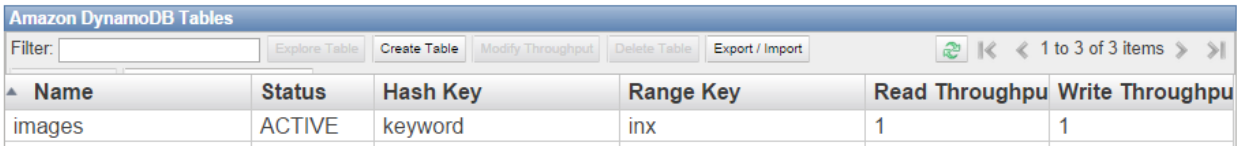
It will tell you that you need to create a table to get started. Click on **Create Table**. Now let's create a table called **images**:

- To let DynamoDB know how to manage data layout, first setting up a **Primary Key** for "sharding" and looking up data.
- For the key, you'll be adding two columns to the table for storing keywords and an integer index.
    - The first is the keyword (this is a hash attribute), and the second is the integer index. Fill out the screen as follows…



- Now click **Continue**. For the next two screens, **Add Indexes (optional)** and **Provisioned Throughput Capacity**, just leave them alone and hit **Continue**.
- For **Throughput Alarms**, add your email address in "**Send notification to**". Then hit **Continue**.
- Finally hit **Create**.

Now you should be able to see your table listed:



From here you can click on the row, then click "**Explore Table**" to see the contents of the table (it's initially empty).

Next, repeat the above to create a second table called **titles**.

Loading Test Data

Since we are building the Web search engine before we have built the crawler, we first need to create some sample data in **images** to enable testing. We can do that from **HW1-webapp** in Eclipse.

First, we need to make sure your **Node.js** setup is ready. Let's begin by going to **Terminal** and typing in:

```
sudo apt-get install nodejs-legacy
```

and typing in the root password (nets212). Next, in Eclipse, first click on the **Node** perspective in the upper right corner. Then right-click on **package.json** and choose **Run As | npm install** to download all necessary libraries.

Right-click on "**loader.js**" and choose **Run As | Node Application**. The program should print a few lines to the console and then say "**Loading complete**".

To confirm that this worked, you can **Explore** the **Table** from the Amazon DynamoDB console on the Web.

Now you can test the application itself. Right-click on **app.js** and choose **Run As | Node Application**. The console should say that the server is running. Start Firefox and open the URL that is shown in the console; you should see some sample images and an input search field. Enter "butterfly" or "dog" into this field and click on "Search"; you should see a set of search results.

Running, Stopping, and Updating Your Node.js Application

You can now stop the application by clicking on the red Stop icon right above the console.

When you make changes to the .js files, you'll generally need to remember to Stop your existing Node.js server with the red button, before you run again. Otherwise you will typically get a message saying "Error: listen EADDRINUSE," meaning that the TCP port 8080 is already taken by another copy of the server.

If you end up in this situation, here's how to recover. First, click on the gray "X" icon above (or the "XX" icon to its right). This will remove all consoles of terminated applications, such as the one that just gave you the EADDRINUSE message. This might put you back to the *previous* instance of your server; if not, click on the "Display selected console" icon (2nd from right and gray in the figure above, but likely with a color icon when you click it) until see the console of your running server. You can now click the Stop button.

When you make changes to .ejs files, you can generally just save the updated .ejs file and refresh your browser. But stopping and restarting will also work.

Understanding the Implementation

Your first task is to familiarize yourself with the code. There are several main aspects:

- **loader.js** is the sample data loader.
- **public/sample-data.json** is the sample data to be loaded.

- **package.json** specifies the dependencies on Node.js libraries.
- **app.js** is the main application for the Web client.
- **views** holds the Express EJS files for displaying HTML content. See "Understanding Express / EJS" below.
- **routes/index.js** holds the JavaScript code called when browser requests are made. It in turn renders the views. See "Routing and Express" below.
- **public/styles.css** is the stylesheet that associates formatting with the different HTML elements from the views.
- **config.json** is the configuration info for AWS.
- **keyvaluestore.js** is a simple Javascript implementation of a key/value store over Amazon DynamoDB. Its API roughly mirrors the **IKeyValueStore** interface in Java for Milestone 2, except that it uses callbacks in Node.js fashion.
- **test.sh** is a simple (non-exhaustive!) test suite to make sure your application is generating plausible results.

Understanding Express / EJS

There are many different tools for combining HTML "templates" with Javascript, including JADE and EJS. We are using EJS here. Let's open up **index.ejs** in the **views** folder to take a look.

You'll see that the document roughly looks like an HTML document, with some embedded Javascript. But you'll see some additional aspects:

- Special commands are delimited by <% {something} %>, which trigger Javascript logic. An example is the **<% if (images && images.length > 0) { %>** … **<% } %>** block, which creates the results table *only if* there are images in the array.

  Another special command is <% layout('layout') %> which embeds the contents of **layout.ejs**. In turn this file **includes header.ejs** and **footer.ejs**.
- Variable outputs are delimited by **<%= *{variable}* %>**, e.g. **<%= title %>** in **header.ejs**.

Understanding Routing and Express

Next let's open **routes/index.js**. This is the file that was loaded when app.js included the command **require('./routes')**. (As with HTTP, by default if you give a directory, the index page is loaded.)

You'll see that index.js **exports** two functions, **index** and **query**. In turn, if you refer back to **app.js** you'll see that HTTP GET requests to "/" and "/index.html" call **index**, and HTTP form POST requests to "**/query**" call **query**.

Milestone 1 Tasks

You'll also need to answer the questions at:

- https://docs.google.com/a/seas.upenn.edu/forms/d/1TZU9POwT73cprFWzNWzwJqzP_EHglnWKF-SqiQ-v-8s/viewform?usp=send_form
- You'll need to use your SEAS account to register. If you use an alternate account from Google / Gmail, you'll probably want to open the above URL from "Incognito" or "Private" mode in the browser – so that you don't need to switch from your existing account to the SEAS one.

You need to add the following:

1. The appropriate HTML to produce, for each image, a hyperlink to the original image.
2. Code to support a **Previous** and **Next** button, and use them to support *pagination*, i.e., the ability to display the data one screenful at a time.
3. Code to display "x – y of z results" at the bottom of the list, e.g., "1 - 4 of 32 results". (Please be sure to take into account the spaces around the hyphen.)

When you think it's right, you should open the Terminal (while your Node app is running) and run the simple automated test suite we've provided: enter **bash ~/workspace/HW1-webapp/test.sh**. You'll see it run a suite of queries against the application, and you can validate that you are getting the right numeric values. ***Note that we will be using a more comprehensive test suite for grading – do not assume that this validates everything!***

Milestone 1 Submission

Ensure you have answered the questions on the Web form. Also commit your code to subversion. Finally, via Firefox on the Virtual Machine, submit your updated **index.ejs** file (in **workspace/HW1-webapp/views**) via Canvas.

Milestone 2: Image Indexer (Due Sept. September 25th)

Code Checkout, Migration to Your Repository, and Data Download

Following the procedure documented for Milestone 1, use Eclipse to check out the project **HW1-indexer** and switch it to your own svn repository.

Next, download (from the Assignments page) the files `images_en.nt.bz2` and `labels_en.nt.bz2`. Using the Terminal, run **bunzip2** on each to decompress it. Move these into your **HW1-indexer** in the **data** subdirectory. From Eclipse, right-click on **HW1-indexer** and select **Refresh**.

The `images_en.nt` file associates Wikipedia categories with images, whereas the `labels_en.nt` file associates Wikipedia categories with labels. For instance, the category "Cloud" might be associated with an image of a cloud, as well as the label "cloud" (and perhaps other labels). In combination, we can use these files to search for images using search terms (i.e., text). Both files consist of triples `<A> <B> <C>` that describe various aspects of Wikipedia categories, not just images and labels. The triples that are of interest to us here are the ones in `images_en.nt` where B is `http://xmlns.com/foaf/0.1/ depiction` (in this case, A is the category and C is an image URL), and the ones in `labels_en.nt` where B is `http://www.w3.org/2000/01/rdf-schema#label` (in this case, A is the category and C is the

label). You can use the `less` command to have a look at the file. However, you do not need to write code for reading and parsing these files directly; we have provided some code for you.

Rough Road Map

You are given a variety of classes / interfaces to "put together", all in the …`nets212` package:

1. Storage is in package **keyvalue**. You will use the **IKeyValueStorage** interface to initialize, read, and write key/value-set pairs. You should not directly instantiate any of the implementation classes, **BdbStorage** and **DynamoDBStorage**. Instead use **KeyValueStoreFactory**'s **getKeyValueStore(dbName)**.

   The parameter DEFAULT_TYPE will specify which kind of storage subsystem is used by the factory. By default this is BERKELEY – a local BerkeleyDB instance.

2. The package **parser**, and specifically class **ParseTriples**, takes a name for one of the DBPedia files (`images_en.nt` or `labels_en.nt`) and opens it.

   Repeated calls to **ParseTriples**' **getNextTriple()** return **Triples** representing subject/relationship/object associations from DBPedia.

3. The package **querier** contains the basic **IQuery** interface for a query system, plus a **QueryFactory** and two implementations of the query system. **StubQuery** is a simple test driver and the class **QueryKVS** (for key-value store) is a skeleton file you're expected to modify (see below).

4. The package **indexer** contains the basic **IIndexer** interface for indexing triples, plus an **IndexerFactory**. The abstract base class **BasicIndexer** gives some default methods, and the two implementations are **StubIndexer** (for indexing) and **IndexKVS** (a skeleton file you're expected to modify, as described below). There are several useful "helper" functions: the **TermFilter** interface and the **TermPrefixFilter** class are useful for allowing you to easily filter terms while indexing; the **PorterStemmer** class is useful for "stemming" keywords (see below).

Task 1: Creating the DBpedia Triple Indexer

The first task will be to *index* the images embedded within the DBpedia data, by keywords from the description. This is sometimes called an "inverted index" and is the basis of search engines, such as the one from Milestone 1. This involves filling out **IndexKVS**. This class should invoke the parser used in its constructor on the appropriate source files, fetch the appropriately named key/value store from the factory, and then load the store.

To ensure you can efficiently answer queries with approximate matches, you should (a) regularize the case for the words in the title, and (b) use a *stemming* algorithm to remove suffixes before storing the words. We are including the "helper" class **PorterStemmer**, which stems words. You will need to learn how to use it from looking at the function definition for **stem**.

Now create a command-line **IndexImages** that invokes the **IndexKVS** over two different source files with two different target key/value stores. The first store, **images**, should be loaded with appropriate subject/object pairs representing Wikipedia *categories* and related *images*. A second store, **titles**, should

have title keywords paired with Wikipedia categories. There are some restrictions on how you should do this:

1. You should *first* index the images from `images_en.nt`, and *only* index the ones where the relationship (accessible via **getPredicate()**) is `http://xmlns.com/foaf/0.1/depiction` This represents an image depicting a specific Wikipedia topic, as opposed to other related topics. All images should be stored in a key/value store called `images`, where the key is the Wikipedia category (via **getSubject()**) and the value is the image URL (via **getObject()**).

   To be able to index **subsets of the data**, you should take a **TermFilter** and when it is not null, ensure that the Wikipedia topic passes. The topic is the part after the last '/' in the path, e.g., the string `Archimedes` in `http://dbpedia.org/resource/Archimedes`. This topic should be indexed, for example, if `filter` is set to **TermPrefixFilter("Ar")**, but not, for example, if the filter is set to `XY`. For initial testing, you should set the prefix filter to "`A`".

2. You should *next* create an "inverted index" from `labels_en.nt`. Here the idea is to index, for each label, the Wikipedia category or categories that correspond to that label. All labels should be stored in a key/value store called `titles` where the key is a word from the label and the value is the Wikipedia category. You should *only* add an entry to this store *if the category exists in the images key/value store*, i.e., if we have an image.

   If a label contains multiple words, you should **create separate entries** for each word.

Example: The `images` database might contain an entry like the following:

key: `http://dbpedia.org/resource/American_National_Standards_Institute`
value: `http://upload.wikimedia.org/wikipedia/commons/8/8f/ANSI_logo.GIF`

and the `titles` database might include the following entries:

key: `american`

value: `http://dbpedia.org/resource/American_National_Standards_Institute`

key: `nate`

value: `http://dbpedia.org/resource/American_National_Standards_Institute`

key: `standard`

value: `http://dbpedia.org/resource/American_National_Standards_Institute`

key: `institut`

value: `http://dbpedia.org/resource/American_National_Standards_Institute`

Note that the label ("American National Standards Institute") has been broken into separate words, and that case regularization and stemming have been applied. Note also that 1) the output of the stemmer is not necessarily an English word, and 2) the input to the stemmer must consist of letters only (otherwise the stemmer will return "Invalid term"). If you encounter a word that contains other characters, you should not apply the stemmer to it, and just use the word as it is.

Task 2: Creating the Querier System

Now, you will fill out the query module in **QueryKVS**., It will both the `titles` and `images` key/value stores via the factory. For each keyword, it should;

1. Retrieve the Wikipedia category from the `titles` store, then
2. Retrieve all URL matches to those categories from the `images` key/value store. This latter value is the set of URLs to be returned.

Note that if you used stemming and case regularization for the loader, you should apply the same transformations to keywords given to the querier before doing the lookup.

Next, you should implement the command-line query system. **edu.upenn.cis.nets212.dbpedia.QueryImages** should use the **QueryFactory** to create an **IQuerier**, then return, for each keyword on the command line, the set of matching results. For each keyword from the command line, it should simply print each keyword, then the list of matches, to the console.

Testing the Querier and Indexer

Ensure that you can run **IndexImages** followed by **QueryImages** (with a keyword you know is indexed, e.g., American) to get the appropriate results.

You might also want to adapt some of the test cases in **edu.upenn.cis.nets212.dbpedia**.

Cloud-Based Data Loader

Now modify your loader application as follows. Change the **KeyValueStoreFactory**.**DEFAULT_TYPE** to **STORETYPE**.**DYNAMODB**. Re-run **IndexImages**.

The total Wikpedia dataset will result in about 1.5GB worth of data, which will take a long time to create (and, worse, a lot of Amazon cycles, which will reduce your credits). You should only index Wikipedia topics whose term (the item after the last "/" in the path, as in "Anarchism" in "http://dbpedia.org/resource/Anarchism") starts with the letter "A".

After (quite) some time, the data should be posted on DynamoDB. You should be able to use the **AWS Console DynamoDB Explore Table** functionality to see the contents.

Milestone 2 Submission

Commit your code to subversion. Next open the Terminal and **cd** into **~/workspace/HW1-indexer**. Run **bash package.sh** to create a zipfile that can be submitted; verify that it adds **IndexImages**, **QueryImages**, **IndexKVS**, and **QueryKVS .java**. Now using Firefox, submit **~/workspace/HW1-indexer/hw2m2/hw2m2.zip** to Canvas.

Extra Credit

We will offer the following extra credit items in this assignment:

- **Milestone 1:** Remove image links that point to dead images. [+5%]

  (Do this within index.ejs and submit as part of Milestone 1.)

- **Milestone 2:** Submit an additional implementation of the **Querier** in JavaScript [+10%]

  (Add your updated .js files, with their correct paths, into **hw2m2.zip** before submitting.)

These points will *only* be awarded if the main portions of the assignment work correctly. Any extra credit solutions should be submitted with the relevant milestone, and they should be described in the README file.