Cloud Computing (INGI2145) - Lab Session 7

Thanh Nguyen, Marco Canini

Today, we will continue our exploration of Javascript web programming. In particular, you will be introduced to Node.js and Express (a simple web application framework that runs on top of Node.js). You will also be introduced to client-side Javascript programming. Finally, we introduce the way in which a Web application interacts with Cassandra database

Setup

We have activated a port forwarding in the Vagrant config file. When you access http://localhost:3002 on your host machine, the connection is forwarded automatically to the INGI2145-vm.

This lab session makes use of Cassandra. To install Cassandra in your VM, please run `vagrant up --provision`. If you need to do it manually inside the VM, then run the following commands:

```
echo "deb http://debian.datastax.com/community stable main"
    | sudo tee -a /etc/apt/sources.list.d/cassandra.sources.list
curl -L http://debian.datastax.com/debian/repo_key | sudo apt-key add -
sudo apt-get update
sudo apt-get install dsc20=2.0.11-1 cassandra=2.0.11
sudo service cassandra stop
sudo rm -rf /var/lib/cassandra/data/system/*
sudo service cassandra start
```

Javascript

If you need to learn some basics of Javascript, we refer you to this handout from a lab session on Javascript.

Node.js and Express

To use express, enter the `/labs/7_web/app` folder under `INGI2145-vm`, and have a look at the file called `package.json`. This file contains a description of your Node.js application including its dependencies. Then install the dependencies with:

```
npm install
```

Now, you can use express locally by writing the following into a Javascript file (name it `test.js`, that you can run with `nodejs test.js`:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello, World!');
});

var server = app.listen(3002, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('Example app listening at http://%s:%s', host, port);
});
```

If you now point your browser to `http://localhost:3002/`, you should see the "Hello World!" message.

What this simple example does is that it defines a "route". The `app.get` call says that for HTTP `GET` requests on the root address, the server should reply with "Hello, World!".

What Express does is essentially map the HTTP requests it receives to a callback which will determine the answer to the request. The callback is a Javascript function that receives two parameters: a request object, and a response object. Convenience methods are defined on these objects to help with the usual actions: getting the request's parameters, sending data as output, etc...

Express' role is to map each request to multiple callbacks, and to specify for each callback the class of requests it should apply to. Express calls these callbacks "middleware". The `app.get` call in the examples installs a middleware that applies only to `GET` requests to the root. The more general method to add middleware is `app.use`. It does not allow to discriminate by HTTP verb however, but you could do this filtering inside the middleware itself.

Middleware are executed in the order in which they are specified in the file.

```
app.use(function (req, res, next) {
    console.log('Time:', Date.now());
    next();
});

app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

In this example, the same request as before would now invoke the middleware passed to `app.use`, before passing control to the next middleware via the `next()` call. Note that the callback in `app.get` could also take `next` as a third parameter. It is ommitted here only because it is unnecessary (as this is the last middleware in the chain).

Why this emphasis on "middleware", and why not simply call them "callbacks" or "functions"? The idea is that those callbacks can hide some fairly complex (and potentially stateful) processing. Moreover, since middleware is the unit of processing, they are easy to bundle and share. See this list to get a sense of what middleware are used for.

Node.js and Express Exercises

For the exercises, we'll start with a series of exercises from nodeschool.io, like we did last week.

```
npm install -g expressworks
expressworks
```

These exercises will walk you through a series of simple scenarios using Express and some external middlewares. You will notably learn how to serve static files, render template files and serve them.

If at some point you feel out of depth, don't hesitate to consult the guides and the API reference on the official website.

Extra Practice

If you feel you need some more practice with Express, or if you're just interested, you can follow this tutorial on how to setup a simple blog with Express. It uses handlebars for templating (yes, there are **a lot** of Javascript templating engines).

EJS

We'll now practice how to use the EJS (Embedded Javascript) templating engine. First, read this introduction.

Can you get Express to render a list of supplies like in the example?

Client-Side Javascript

So far, we've mostly talked about Javascript on the server side. But Javascript is famous mostly for being *the* client-side programming language.

When using Javascript on the client side, what we seek to do is to manipulate the DOM (Document Object Model – in essence, the HTML markup of the page) in response to user actions, without needing to navigate to a new page. Another key feature of client-side Javascript is the ability to download data in the background. This is sometimes called AJAX (Asynchronous Javascript And XML – although nowadays it has little to do with XML). The data obtained this way will often be displayed via DOM manipulation.

There is a standardized API to manipulate the DOM, but it used to be more limited, and to be plagued by browser incompatibilities. For this reason, most people ended up using the jQuery library to manipulate the DOM and handle events.

If you want to learn about jQuery, the documentation is rather good. You can also check the API reference. If you want some hands-on practice, check this codecademy course. If you want to learn about the vanilla way, read this introduction to the DOM and this introduction to events.

AngularJS

Going up a layer, there are a lot of client-side MVC (Model-View-Controller) and MVVM(Model-View-ViewModel) Javascript frameworks. These help build what is called Single Page Applications (SPA): basically a webpage that acts and feels like a real application - this is made possible by AJAX and DOM manipulation: everything happens on a single page without the need to navigate to other pages.

These apps tend to have a lot more Javascript code than your typical webpage, hence the need for frameworks to organize this code.

AngularJS is a framework, created by Google, that provides an effective way to organize the code in client side. In terms of concepts, a typical AngularJS application consists primarily of a view, model, and controller, but there are other important components, such as services, directives, and filters. Therefore, AngularJS actually adopts Model-View-Whatever (MVW) model. Regardless of the name, the most important benefit is that the framework provides a clear separation of the concerns between the application layers, providing modularity, extensibility, and testability

There are multiple ways of organizing the client code with AngularJS. In this project, we create the skeleton in which one folder (app/public/js/angular) contains all controllers in separate files. Another folder (app/views) consists of all views as EJS template files. `template.js` is an important file that declares the HTML frame and the links to all the partial views.

- Each `.js` file (controller) in app/public/js/angular defines the way of binding data from data object (loaded from database) to the view.
- `$injector`: AngularJS is powered by a dependency injection mechanism that manages the life cycle of each controller. This mechanism is responsible for creating and distributing the controllers within the application.
- `$scope` is a special object that acts as a shared context between the view and the controller that allows these layers to exchange information related to the application model

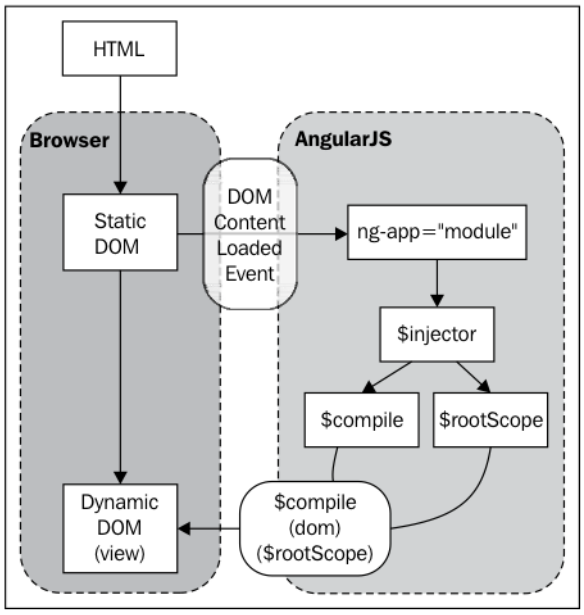

Figure 1: alt text

Cassandra

The Apache Cassandra database is a sufficient choice when you need scalability and high availability without compromising performance. It has linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure.

Cassandra's data model offers column indexes with the performance of log-structured updates, strong support for denormalization and materialized views, and powerful built-in caching.

Cassandra uses `CQL` to perform queries to database. CQL is almost similar to SQL statement.

There are two ways to view data in Cassandra:

- `cqlsh`: to query and view data with style of a relational database management system (RDBMS). Data are shown in table style.
- `cassandra-cli`: to query and view data with key-value and column-family style.

Tutorial

In this part, you will learn how to connect to Cassandra from a Web application written with Node.js and AngularJS.

Before doing this part, you need to run `cqlsh` in the terminal, then run the following CQL scripts to `create` the table Users in `keyspace` twitter and `insert` user into users table:

```
CREATE KEYSPACE IF NOT EXISTS twitter WITH
    replication = {'class': 'SimpleStrategy', 'replication_factor': '1' };
CREATE TABLE IF NOT EXISTS twitter.Users (username varchar PRIMARY KEY, name text, pass text);
INSERT INTO twitter.Users (username, name, pass) VALUES('test', 'test',
    'MhEPf7MRZuf2f3aa2b2f29386278c614d79683c28a');
```

Connect to Cassandra

Node.js uses `cassandra-driver` to connect to Cassandra database. An example of database connect is given in the link below. You can also find in the `app.js` file of the lab session skeleton.

```
app.db = new cassandra.Client( { contactPoints : [ '127.0.0.1' ] } );
app.db.connect(function(err, result) {
    console.log('Connected.');

    [...]
});
```

Execute and query

On the server side, you can use the following pattern to interact with Cassandra:

```
var getUser = "SELECT * FROM twitter.Users WHERE username=?";
exports.getOneUser = function(user, callback) {
    app.db.execute(getUser, [ user ], function(e, result) {
        if (result.rows.length > 0) {
            // HERE YOU GOT DATA
        }
        else {
            // NO DATA :-(
        }
    });
}
```

Exercise

Log-in Authentication and maintaining sessions

Before starting to work on this task, you should check out the following files: - `routes.js`: This file is used by the Node.js application to maintain mappings between routes and server functionality (e.g., it discerns which view will get rendered when we access our `/home` page). - `account-manager.js`: This file contains auxiliary functions that allow authenticating users.

Session is a set of variables in server side. Node.js stores those variables in **Express session**. Add `express-session` to the list of dependencies in package.json and install it with `npm install`.

An example of session is given in this **example**.

You are asked to first modify the `account-manager.js` in order to implement the manualLogin function. This involves querying Cassandra for the username, and if a match is found, checking for equality of passwords. Then you should check the `/validate` route in `routes.js`. This route receives two parameters: username and password. It should use the manualLogin function to try to authenticate the user. If successful, it stores the user information as session state in a cookie (via express's req.session object).