International Journal For Technological Research In Engineering
Volume 4, Issue 11, July-2017 *ISSN (Online): 2347 - 4718*

CLOUD COMPUTING WITH NETWORK SYSTEM

Tanvi[1], Anjali Namdev[2]
[1]PG Student, [2]Assistant Professor, CSE Department, S(PG)ITM Rewari, Haryana, India

I. INTRODUCTION

With the increasing popularity of CLOUD COMPUTING WITH NETWORK SYSTEM services, more and more businesses are moving their infrastructure to the cloud. Developers can benefit from it by reducing the amount of code they need to write to have a fully running service, while end-users get the ability to access their data from virtually any device that has a network connection. There are many possibilities on how to incorporate CLOUD COMPUTING WITH NETWORK SYSTEM computing into your service. CLOUD COMPUTING WITH NETWORK SYSTEM service providers are offering a wide range of services to satisfy different needs of modern applications by providing network infrastructure, platform or even database services on demand. Developers can also use different techniques that transform their software into services with automatic data synchronization across all of the users' devices. But because these technologies and techniques are relatively new, there are still many uncertainties about how to incorporate them into the development pipe to gain most benefits. This is especially true today when fascinating new server-side technologies like Node.js are changing software design patterns. In my Bachelor Thesis I'm going to explore these possibilities and formulate a methodology for developing and designing software as a service application, using the newest open source technologies, using CLOUD COMPUTING WITH NETWORK SYSTEM service providers, and demonstrating methodology in work by developing an application from sketch to deployment.

II. METHODOLOGY

In this chapter I will present a methodology for developing CLOUD COMPUTING WITH NETWORK SYSTEM services. First, I will introduce important technologies that I recommend for building SaaS applications. Then I will discuss important topics that need to be considered, while developing the server-side of the service, including authentication methods, service requests, routing requests and security in general. For modeling purposes, I'm going to use UML

TECHNOLOGIES

Hosting Services

One of the first and most important decision while building CLOUD COMPUTING WITH NETWORK SYSTEM application is how are you going to deploy and host your application. I recommend using Heroku PaaS for these purposes as mentioned previously. To start using Heroku, you first need to sign up for the service and install the Heroku toolbelt that is available from their official website.

After that, you need to log in your Heroku client and download SSH public key, to access your Heroku profile and applications:

$ heroku login

Enter your Heroku credentials. Email: john@example.com Password:

Could not find an existing public key. Would you like to generate one? [Yn] Generating new SSH public key.

Uploading ssh public key /Users/john/.ssh/id_rsa.pub

After that you can start using the Heroku toolbelt to deploy your application. To do that you need to set up a Git repository, and then you can deploy your application to the Heroku right from the Terminal:

$ git init $ git add .

$ git commit -m 'init'

$ heroku apps:rename appname $ heroku create
$ git push heroku master

Later, when you will need to push a newer version of your application and deploy it, you will just need to type one line:

$ git push heroku master
Heroku will update your application's data and deploy it automatically. At any time, you can see how your application is doing, by checking processes:

$ heroku ps

=== web: 'node appname.js' web.1: up for 10s

But Heroku doesn't offer MongoDB [7] solution, instead there is an add-on from MongoLab, which offers MongoDB in a form of DaaS (Database as a Server). The free basic version of MongoLab [8] provides 500 MB of storage and web-based management tools, which is enough for testing purposes or even small applications. You can add MongoLab to your project and get connection URI[6] for your database from the Heroku website or right from the Terminal:

$ heroku addons:add mongolab

$ heroku config | grep MONGOLAB_URI

MONGOLAB_URI=>

mongodb://heroku_app1234:random_password@ds029017...

This is enough configuration to start using Heroku PaaS and MongoLab DaaS in your project, however, to make your application fully functional, you need to make sure that you're using a special port. This port is provided to your application by Heroku and without it you won't be able to receive any of HTTP requests.

If everything is set up properly, the application will be available from the URI provided by the Heroku, or alternatively, you can set up your own domain and add the domain to your Heroku settings.

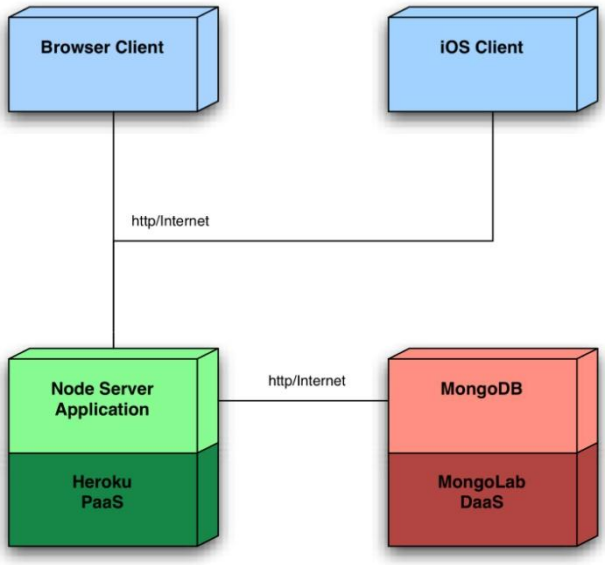

Figure . Application's Infrastructure

Uniform Resource Identifier (URI) is a string of characters used to identify a name or a web resource.

Server Core Technologies

Successful SaaS applications should support different clients, including web browsers, mobile applications and desktop applications. While there may be many different patterns on how to implement this, the easiest way is to build a HTTP server that will serve requests for all clients, returning JSON objects. This can be easily done with Node.js and supporting technologies.

Node.js is a platform for building network applications based on Google's V8 JavaScript engine . It uses an event-driven, asynchronous I/O model, which makes it ideal for frequently accessed service. Another nice quality of Node.js is that there is no separation between your application and the web server, so you can create the server, customize it, and deliver content all at once, using JavaScript. It's a great solution for server-side because it has a large community of developers that frequently contribute to the project, it's open source and it's modular. Due to its big community and modular principles, you can easily extend your application, including some of the available modules or developing your own module.

The core modules of Node.js lets you create HTTP or HTTPS server, and handle its requests.

MongoDB

MongoDB is a document-oriented NoSQL database that stores structured data in a form close to JSON. Unlike classic relational databases, it uses dynamic schemas which makes integration of data easier, so you can change your schemas on the go without affecting already existing objects. One of the biggest advantages is that MongoDB is schema-free, which means there is no schema migrations. The only schema that's going to be defined is in your code.

Express.js

Express.js is a module for Node.js. It is a very minimal web application framework that provides tools for routing, template and view rendering, session management, serving static files, application settings and more. Although you can handle everything directly in Node.js, eventually, you will end up reinventing many of the capabilities that Express.js already has.

Mongoose.js

Mongoose.js is another module for Node.js, which provides object modeling tools on top of MongoDB. It is rather difficult to use MongoDB queries directly, especially if you'll end up with a complicated data structure. Mongoose let's you create objects that will represent your data and call object methods instead of manually querying the database.

Passport.js

Passport.js is authentication middleware for Express.js. It provides certain methods for using third-party authentication (Facebook, Twitter, Google+), as well as classic username/password solutions. If for any reason you don't want to store password in your database, it's a great idea to let users authenticate with their already existing Facebook account. Facebook provides its developers with SDKs, but unfortunately, there is no official supported SDK for Node.js. With Passport.js, using Facebook authentication is easy.

Socket.IO

Socket.IO is a socket solution for Node.js application. It consists of a nearly identical APIs for both server-side and client side browser applications. Socket.IO primarily uses WebSockets to provide connectivity, but when it's not available, it's automatically switched to another transportation method, like Adobe Flash sockets, AJAX long polling, AJAX multipart streaming, JSONP poling or another.

The advantage of using Socket.IO is it provides cross-browser compatibility and allows you to support relatively old browsers with the same API. It also provides essential features for real-time web applications, like heartbeats, timeouts and more. Another great concept about Socket.IO is that you can force it to use specific protocol, which is very useful while using Heroku (where WebSocket is not

supported yet.) You can forcefully use long pulling, e.g. xhr-polling, and later on use WebSocket support when it is added. Your final step would be to change a few parameters.

Authentication
In order to know if a user is authorized to perform an action, he should be authenticated. There are many different authentication methods to achieve this. First of all, you can use your own login and registration forms with classic username and password fields. Then, you will need to store users' password in a secure way. You should consider hashing your password with salt using cryptographic hash function. Node.js has a module called "crypto", which can provide you tools for hashing your password. Another aspect is protecting your database against brute-force attacks. It might be a good idea to count the number of wrong log in attempts for each user, and when there will be 10 recursively wrong attempts you can disable the log in for several minutes or add a CAPTCHA field, to ensure that the request is performed by a human. If you want to avoid these complexities, you can use third-party services for authentication. For example, you can implement "Log in with Facebook" button, and let them worry about possible security issues. Passport.js, an open-source Node.js module, providing tools for most of the known authentication methods, including OAuth 2.0, Facebook, Twitter, Google+ and more. Your application just sends a request to Facebook, and if a user is authenticated within a browser (or device) and allows your application to use his profile data, you will receive user profile information.

Log in with Facebook
Facebook login is a great example of how modern social services cooperate to provide a better experience for both users and developers. Users usually don't like to spend their time on registration process, in fact, sometimes they even make a decision not to use a service because of the registration process that usually requires email or phone verification and, most importantly, delays users ability to use the service.

Developers also benefit from using Facebook log ins. Relying on Facebook means that developers don't have to worry about encrypting and storing passwords, providing registration form and brute force attacks against our application. Later, when they establish core functionality, they can add our custom registration if they need it.

Edit profile
User must be able to change his profile data, like name or surname. Again, only basic profile information will be available, but it can easily be extended later.

Add or remove friends
A user must be able to add friends. To accomplish this, a user must know his friends name and will be provided by a friends search feature. Users will send friend requests, and recipients will decide to add or reject the friend request.

Send and receive messages
After adding a friend, users will be able to communicate with each other in form of live chatting.

Use Case Diagram
So our users will be provided with tools for managing their friend list and communicating with their friends. This is the basic functionality that nearly every social service needs to have. Below is a use case diagram that represents this functionality.

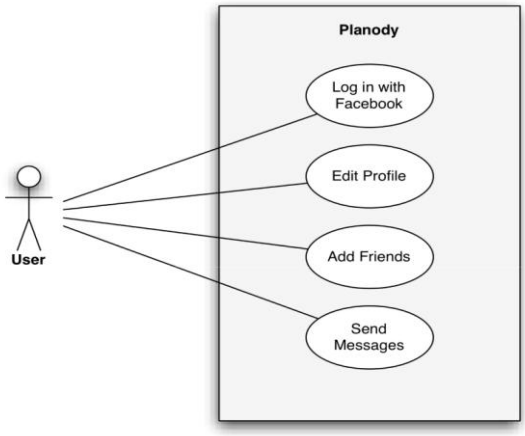

Figure Use Case Diagram in UML

Class Diagram
Our User model is going to be the core class of our application. It will be used for verifying and saving user data, storing access tokens that allow access to user's public profile data from corresponding Facebook friend lists and conversations. I could have just implemented messages, without having a conversation middleware class, but then we'd have to perform expensive queries in order to perform basic functionality, like queuing all messages between two friends, number of new messages or last message sent. Message classes are going to store message bodies and other essential attributes. I added sender and receiver fields, so we will be able to identify a message after the user deletes a message from a conversation and messages status, so we will know if the user has read a received message.

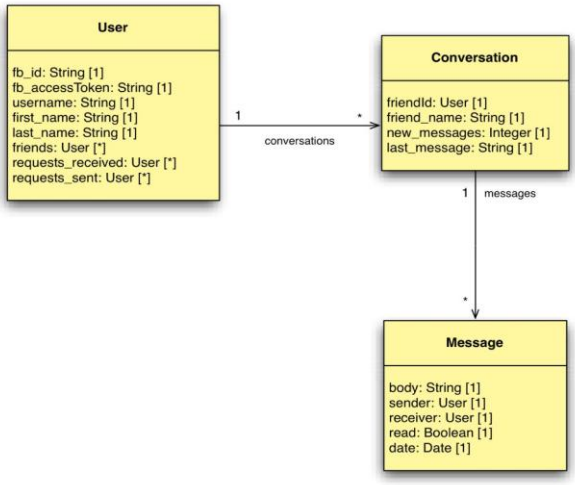

Figure Class Diagram in UML

Implementation

Each part of application was implemented and tested locally, and then deployed on Heroku PaaS. First, I implemented a HTTP server, to test its basic functionality. Then I implemented models, and tested them locally. After that, I implemented routing system, templates and client-side.

Folder Structure
The application was structured in such a way so it wouldn't be hard to add features later. It also helps organizing the code and making development easier.
Planody
node_modules
models
views
public
images
scripts
styles
app.js
router.js
tests.js
package.json
commit.sh Procfile
Folder node_modules contains all of the dependencies for Node that I decided to use in my project. They also need to be defined in the file package.json, so they could easily be installed later, on any machine by running npm install command in the terminal. This contains the package.json file with the application's dependencies:
{
"name": "planody", "author": "Vasily Zhovner", "version": "0.0.1", "dependencies": {
"express": "3.0.0", "mongoose": "3.6.9", "jade": "*", "passport": "*", "passport-facebook": "*"
}
}
Another file called Procfile is a declaration of a script that needs to be invoked to run our application. This is where Heroku looks when deploying our application. My Procfile contains
this: web: NODE_ENV=production node app.js.
I have created another shell helper script that will take care of Git versioning and deploying our application on Heroku. It's called commit.sh:
git add .
git commit -m "init" git push heroku master
After pushing your code to Heroku it will launch script in Procfile and application will be ready to use.

Main Application
This file contains the main Node application, starts the server and configures the application. It also loads the required modules of our application, defined before in package.json files.
var express = require('express')
app = express()
jade = require('jade')
mongoose = require('mongoose')
passport = require('passport')
Facebook Strategy = require('passport-facebook').Strategy
router = require('./router');
There are two different configurations: one with production settings and another with development settings. They differ in Facebook application data and the database they connect to:
// Development settings
app.configure ('development', function () {
mongoose.connect ('localhost', 'planody');

FB_CALLBACK_URL = "http://localhost:3000/auth/facebook/callback";
FB_CLIENT_ID = '480176905384895';
FB_CLIENT_SECRET = <hidden>;
});
// Production settings
app.configure ('production', function () { mongoose.connect ('mongodb://planody:<password>.mongolab.com:72345');
FB_CALLBACK_URL = "http://planody.herokuapp.com/auth/facebook/callback";
FB_CLIENT_ID = '168496716625349';
FB_CLIENT_SECRET = <hidden>;
});
Then I set up default application's settings, like a static folder, start the server on a specific port
and initialize the router:

// Default settings
app.configure(function () { app.use(express.static(__dirname + '/public')); app.set('view engine', 'jade');
... more here
});
// Start server
app.listen(PORT);
console.log('Application is listening on PORT ' + PORT);

// Start router
router.start(app, passport);
One more thing that I did in app.js was set up Passport.js authentication. First, I created a Facebook authentication strategy, told Passport.js to use that strategy and finally set up a method for serializing and deserializing the user:

// Passport settings
var User = require('./models/User'); // User Model, explained later
function facebook Login (accessToken, refreshToken, profile, done) {User.loginFacebook(profile, accessToken, function(err, user) {
done(err, user);
});
};
var facebook Strategy = new Facebook Strategy({ clientID: FB_CLIENT_ID,
clientSecret: FB_CLIENT_SECRET, callbackURL: FB_CALLBACK_URL
}, facebookLogin);


```
passport.use (facebook Strategy);
passport. serializeUser (function(user, done) {
done(null, user.id);
});
passport.deserializeUser (function(id, done) {User.findById
(id, function(err, user) {
done(err, user);
});
});
```

III. CONCLUSION

The main aim of my thesis was to formulate a methodology for developing and designing SaaS applications in the cloud. This task was successfully completed, and methodology can be used to develop and deploy applications in the CLOUD COMPUTING WITH NETWORK SYSTEM as shown in case study application (Chapter 5.) I studied and reviewed some of the most popular CLOUD COMPUTING WITH NETWORK SYSTEM service providers, and learned how they can be incorporated into deployment process. I found that PaaS services are generally more suited for developers with no prior CLOUD COMPUTING WITH NETWORK SYSTEM experience, because of its easy integration and deployment process. Among other PaaS, Heroku offers a very wide range of functionality that can be used in different projects, and can be used in both commercial and educational purposes for free. Formulated methodology can be used by developers with little to no prior CLOUD COMPUTING WITH NETWORK SYSTEM experience to study problem domain, learn how to develop SaaS applications and deploy their applications in the cloud, using technologies described in the methodology. The practical part of methodology resulted in development of a fully functional SaaS service, deployed on Heroku PaaS that can be used as a foundation for a more extended product. The methodology only covers how to design and deploy applications on PaaS, and can be further expanded to study possibilities of integrating with IaaS.

REFERENCES

[1] Christensen, Clayton M., The Innovator's Challenge: Understanding the Influence of Market Environment on Processes of Technology Development in the Rigid Disk Drive Industry. Unpublished doctoral dissertation, Harvard University Graduate School of Business Administration, Boston, Massachusetts, 1992
[2] B. Majumdar, Innovations, Product developments and Technology Transfer: An Empirical Study of Dynamic Competitive Advantage, The Case of Electronic Calculator, Ph. D, Diss. Case Western Reserve University, Cleveland, Ohio, 1977 Thesis: Cloud Computing Models Page 81
[3] http://en.community.dell.com/techcenter/b/techcenter/archive/2012/02/29/digging-deepinto-dell-boomi-how-does-it-work.aspx
[4] http://www.gartner.com/it/page.jsp?id=1903814
[5] http://en.community.dell.com/techcenter/b/techcenter/archive/2012/02/29/digging-deepinto-dell-boomi-how-does-it-work.aspx
[6] collab.net
[7] http://en.wikipedia.org/wiki/Agile_software_development
[8] http://en.wikipedia.org/wiki/Waterfall_model
[9] http://www.slideshare.net/rajdeep/introduction-to-google-app-engine-presentation
[10] http://www.gartner.com/it/page.jsp?id=1963815
[11] http://salesforce.com

