project memoriae

A Living History Application

Instructor: Dr. Suzan Üsküdarlı
Team: Band of Brothers
Members:

Arda Oğulcan Pektaş
Bahattin Ünal
Emre Koray Karpuz
Fatih Aracı
Mustafa Gönül

Github:
https://github.com/574BandOfBrothers/memoriae

Contents

A. ABOUT MEMORIAE

Memoriae is an application that can run both on Android and IOS. This application lets you create a profile and share your memories (which we will name them *story*). You can view your stories through the timeline. You can select your stories and annotate them. Location, time, and description are allowed to describe your story content. We will explain *story* and *annotation* later in this document.

B. REQUIREMENTS

1. Functional Requirements

1.1 Registration

1. Users shall register to system with their email, username and a password.
2. Users shall request for their usernames with their e-mails when they cannot remember their usernames. The system shall send e-mail regarding forgotten usernames.
3. Users shall reset their passwords. The system shall send a link via their e-mail.

1.2 Profile

1. Users shall change their profile password.

2. Users shall change e-mail information attached to their profile.

3. Users shall edit and save their personal information.

1.3 Story

1. Users shall create annotation bodies as stories from scratch. If Users sign the website in, they shall see "New Story" Button on the website.
2. Users shall set titles to their stories. By default, every story (annotation body) is annotated by their titles. The annotation is textual.
3. Users shall create other annotations besides their titles. (default annotation)
4. Stories shall embed external or internal resources.
5. User shall upload media to embed to their stories.

1.4 Annotations

1. Users shall select text from the stories and create annotations.
2. Users shall add story for their annotations.
3. The requirements under **Story** section is also valid for creating annotation body for the annotation created with selecting text from stories.

1.5 Sort & Filter

1. The users shall sort and filter the stories they can access according to:

- The creator
- The story creation and modification time
- The annotation types
- The annotation count

2. The sorting shall be ascending or descending.

1.6 Default Sort & Filter

1. Default sort and filtering services shall be accessible for users via homepage. The services are:

- Latest
- Top
- Today
- Map
- Time

2. Latest stories shall be shown at homepage.

1.7 Search

1. Users shall search story titles with text.
2. Users shall search story context with text.

1. Non-Functional Requirements

1. Back-end and application programming interface of the system shall be implemented using Node.js.

2. Applications of the system shall be implemented using React Native.

3. Javascript shall be programming language.

4. MongoDB shall be used as database management system.

C. SYSTEM ARCHITECTURE

Memoriae is a client-server application therefore it can be divided into two main components.

1. BACK-END SERVER

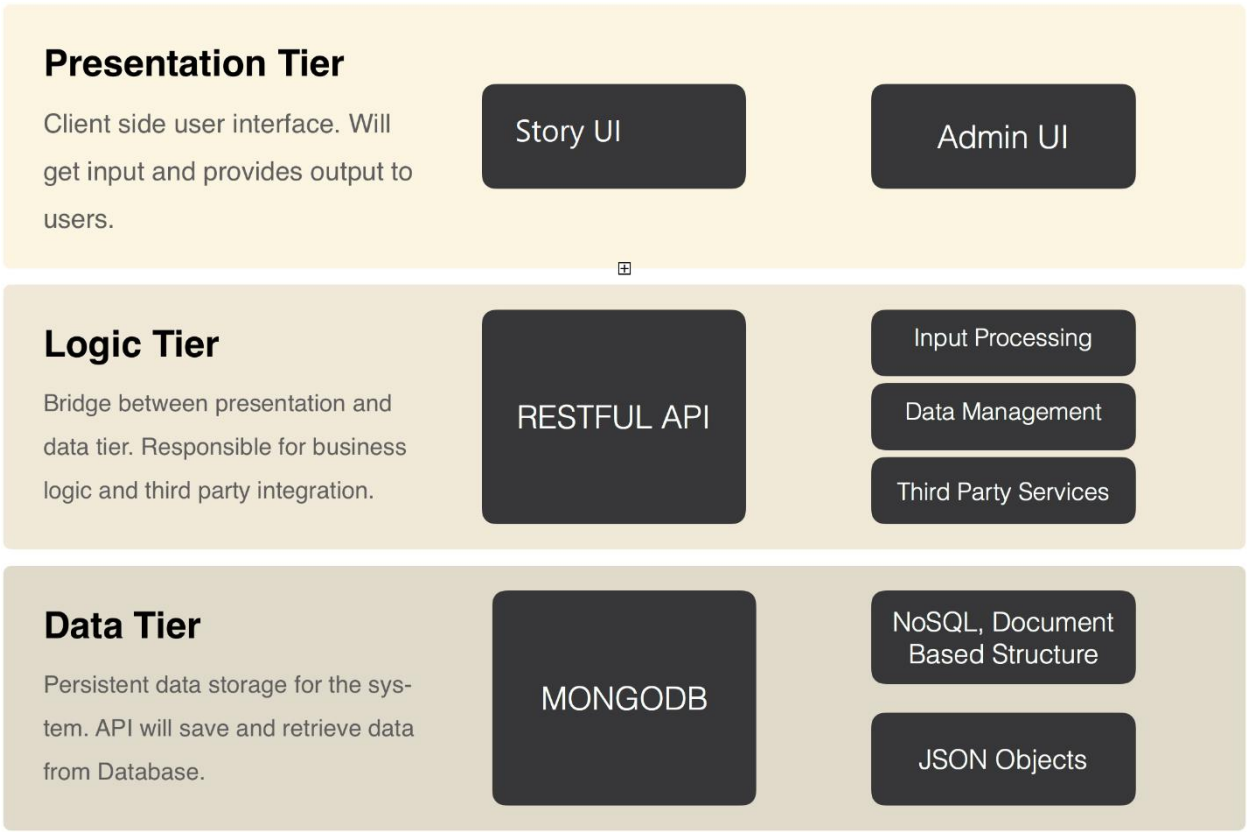
All the logical functions of system will be performed by a RESTFUL API. Which will act as a bridge between Persistent Data Storage and client. Clients will access API with authorization token to request operations. For the login required endpoints access token should be provided by client. Access token will be generated and issued to the client in form of JWT by using auth endpoint. Endpoints will receive and output information in form of JSON.

To provide maximum security and scalability back end server will be running on Amazon Cloud Service (AWS).

RESTFUL API will be written in Express Framework on Node.js to benefit from high throughput capacity.

2. CLIENT SIDE

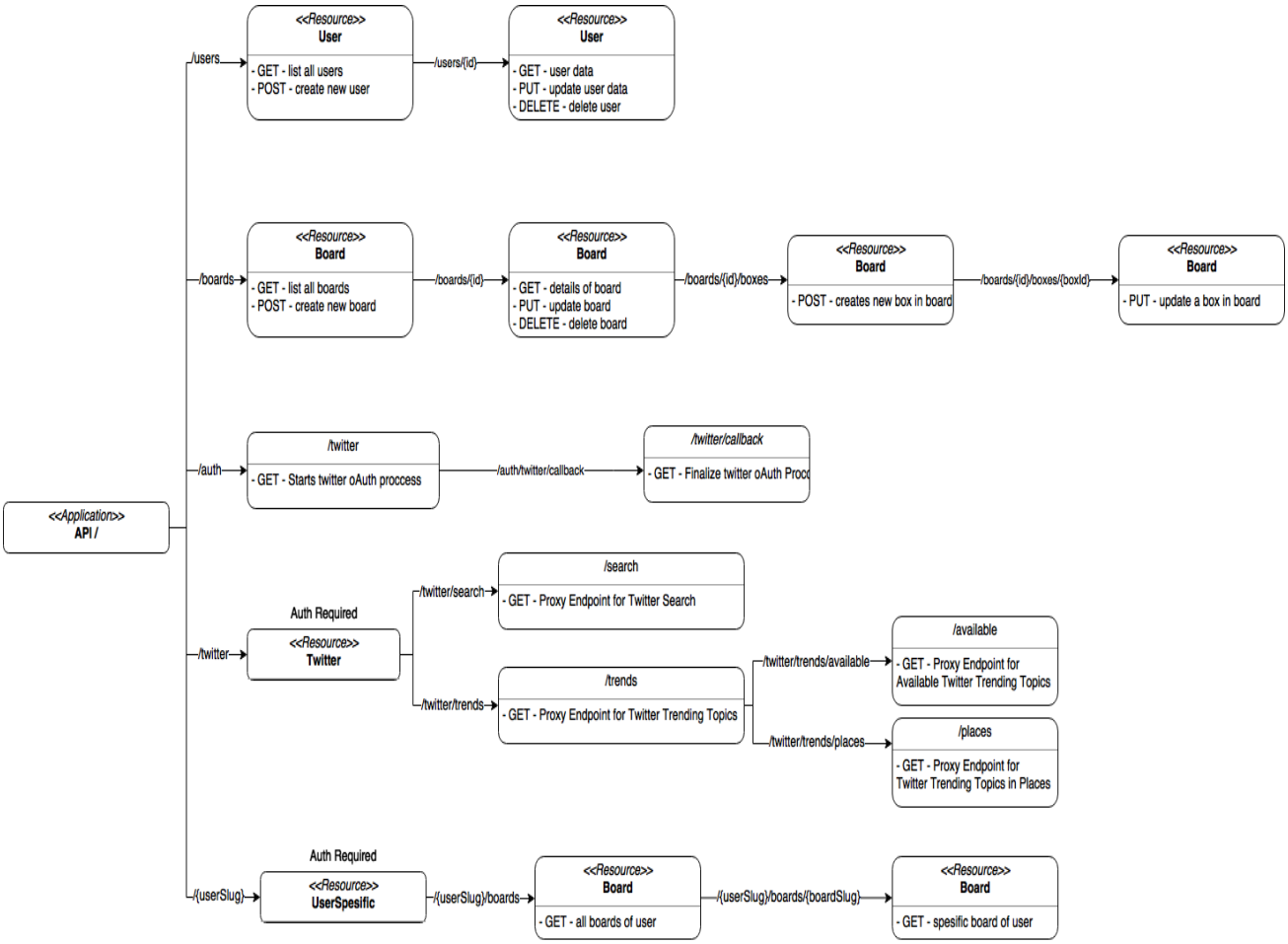
Client side which provides User Interface will be written in React Native to benefit from reusable and easy to use components. *Redux* and *immutable* should be used for data management on client side. As described, system will use Three-tier architecture pattern, which is illustrated below.



3. RESTFUL API ARCHITECTURE

Back-End of Memoriae will be exposed as a restful API to the client applications. MongoDB will be used for database and Node.js will be used for API. JWT tokens will be used for API authentication. JWT token will be generated for user upon sign-in and it should be stored in client-side for future requests.

4. API ENDPOINTS



General Properties for Endpoints

- API uses semantic versioning v1/, v2/
- API output as JSON and able to get requests with JSON objects.
- API responses follow HTTP Response Code standards.

- Routes configured by RESTFUL approach.
  - GET resource/ -> List resources
  - GET resource/:id -> Output specific resource
  - POST resource -> Create resource
  - PUT resource/:id -> Update specific resource
  - DELETE resource/:id -> Delete specific resource

User Endpoint

GET /users
- Lists all user records
- Password field should be excluded

GET /users/:id
- Gets user with given id

POST /users
- Creates a user record
- Upon creation user slug will be created automatically from user's name (first name, last name) if same slug exists it will be created with increment like john-doe-2

PUT /users/:id
- Updates user with given id with given data
- Update user slug if user name is changed and stores old slug.

DELETE /users/:id
- Deletes user with given id
- User record will be softly deleted by setting the deletion time.

Memoriae API Endpoints

No	Endpoint	Link	Method
1	Login a User	http://api.memoriae.online/authenticate	POST
2	Register a User	http://api.memoriae.online/users	POST
3	Get User Information	http://api.memoriae.online/me/	GET

4	Get List of Users	http://api.memoriae.online/users	GET
5	Get a user's Stories	http://api.memoriae.online/me/stories	GET
6	Get a Signed Url for S3 File Upload	http://api.memoriae.online/uploads/request	GET
7	Upload Files with signedURL	signed URL	xhr, PUT
8	Create Story	http://api.memoriae.online/stories	POST
9	Search Stories	http://api.memoriae.online/search?query=${query}	GET

Annotation API Endpoints

No	Endpoint	Link	Methods
1	Get Annotations	http://api.memoriae.online/annotations	GET

Authenticate Endpoint

- POST /authenticate

Stories Endpoint

GET /stories

- List all stories recorded to the system

GET /stories/:storyId

- Gets story with given id

POST /stories

- Creates a story

PUT /stories/:storyId

- Updates the story with given id with given data

DELETE /stories/:storyId

- Deletes the story with given id
- The story record will be softly deleted by setting the deletion time.

Me Endpoint

GET /me

- Lists the authenticated user info
- Passwords should be excluded

GET /me/stories

- Lists the stories added by the authenticated user

Uploads Endpoint

GET /uploads/request

- Get a signed Url for S3 file upload

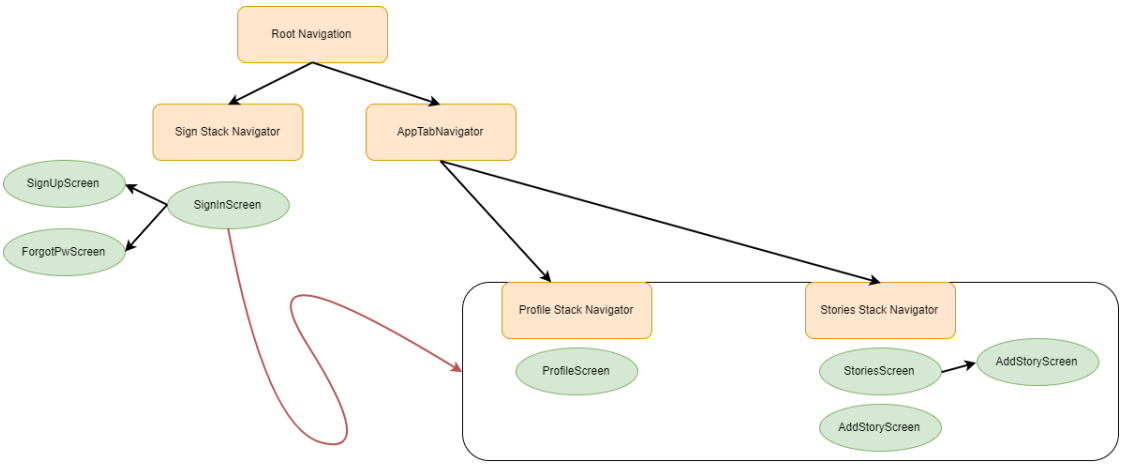
D. CLIENT-SIDE APPLICATION ARCHITECTURE

Modern Front-end technologies are used in client side web application. Application developed on top of React and Redux with EcmaScript 6 and Webpack is used for transpiling, linting and creating optimized build scripts.

1. Technology Overview

React Native

React Native lets you build mobile apps using only JavaScript. It uses the same design as React, letting you compose a rich mobile UI from declarative components. With React Native, you don't build a "mobile web app", an "HTML5 app", or a "hybrid app". You build a real mobile app that's indistinguishable from an app built using Objective-C or Java. React Native uses the same fundamental UI building blocks as regular IOS and Android apps. You just put those building blocks together using JavaScript and React. Our front-end structure which built with React-Native:



Redux

Redux is a predictable state container for JavaScript apps. It helps to write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger (Available as Chrome extension). Redux creates a global singleton State store for react application on top of Functional Programming Practices that are actions and reducers.

Immutable

Immutable data cannot be changed once created, leading to much simpler application development, no defensive copying, and enabling advanced memoization and change detection techniques with simple logic. Persistent data presents a mutative API which does not update the data in-place, but instead always yields new updated data.

Immutable.js provides many Persistent Immutable data structures including: List, Stack, Map, OrderedMap, Set, OrderedSet and Record.

These data structures are highly efficient on modern JavaScript VMs by using structural sharing via hash maps tries and vector tries as popularized by Clojure and Scala, minimizing the need to copy or cache data. By using immutable with React and Redux we will achieve strong and persistent Application state management.

Webpack

Webpack is a powerful module bundler for javascript which supports hot module reloading, lazy loading, bundle splitting, hashing and source maps. It takes all kinds of assets which are used for modern web development such as source code, images, css, etc. and turns that into an optimized and compressed bundle which can be used by client browsers.

2. Application Structure

Client web application is a Single Page React Application.

Routes

Responsible for application routing. Redirects incoming routes to the related container pass required parameters to container and manages browsers history state.

Containers

Containers are the controllers for the route. Containers are responsible for requesting data by dispatching actions and transfers data to the dump components within page. They are directly connected to the Redux store and contains dump components. App container is the root container of the system and all other containers are living inside App container.

Components

Components are dump and reusable views. They are not connected to the redux store they receive props from its parent and process according to their props and they delegate actions via props.

State Shaping and Redux

Requests to the API are called according to the dispatched actions from containers. Dispatched actions may be seen on the figure above on

the left hand-side. Actions make required API calls by API helper and dispatch results to the Reducers.

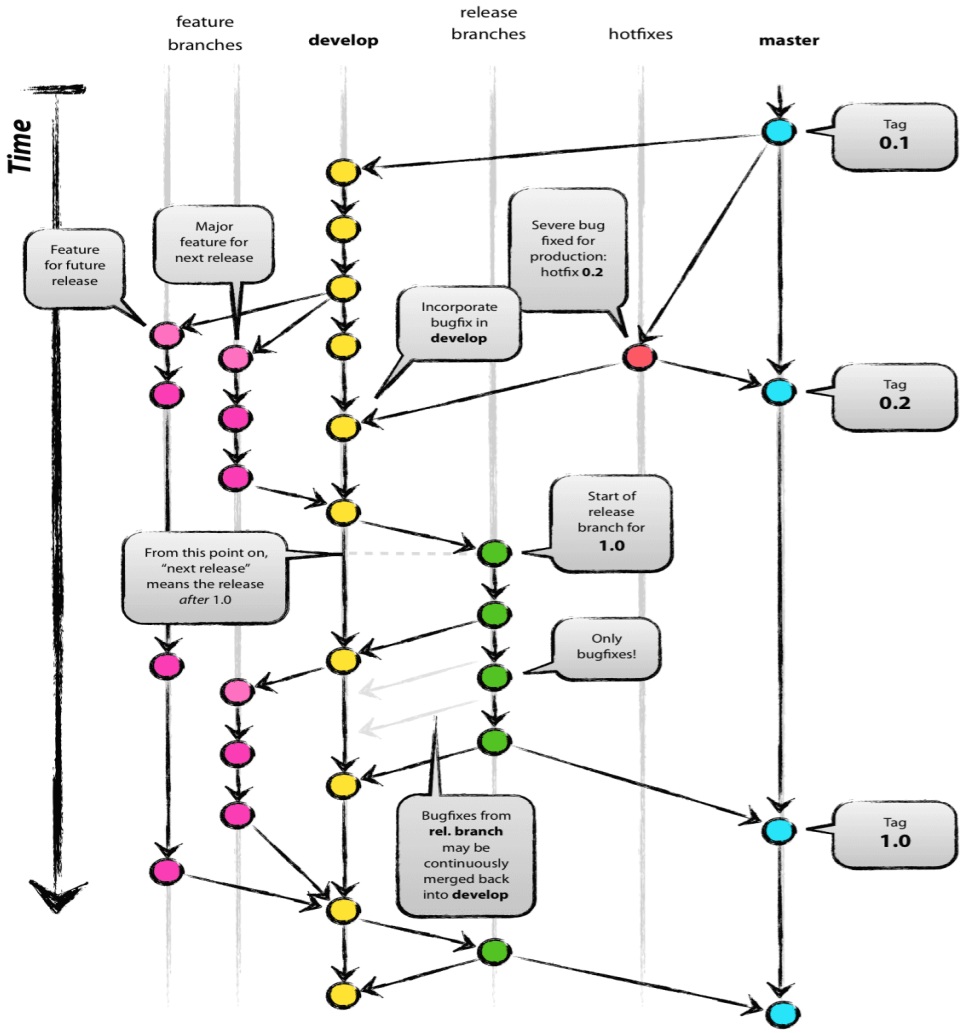
Reducers are the objects for the global redux store and they map and filter response data and store it on global state. Global state is changed after actions. On the figure above, final state of the application can be observed on the right-hand side.

It should be noted that data is stored as normalized on the store instead of arrays. Which means instead storing users as array, they have stored as key value objects according to their slugs. Also, boards of users stored in same manner. By normalizing data, it is easier to access data when necessary by just using the key, instead of searching through the array.

E. DEVELOPMENT ENVIRONMENT

For both backend and client, Git was used for code versioning and Github was used for centralized repository. For development environment and release management we have used Git flow branching model which is proven to be very effective for multiple developer environments and safe for release management.

Git Flow



As illustrated above, Git flow branching basically depends on separated branches. There are two main branches, one is master which represents production code and one is develop which is development code. Developers should create branches from develop to make their changes to the code and after their work finalize they should open pull requests to the develop branch.

1. Repos and Environment Variables

There are separate Github repositories for Backend Code and Client Code. Both repo should have configured to have develop branch as origin. And both repos configured to read config variables from JSON files by environment type. By using separate config files for different environments, it is easy to configure and safer to split production and development environment from each other without changing the code.

Preparing the Development Environment for Backend API

Prerequisites

- *Node*
- *MongoDB*

Installing Node with NVM on Ubuntu

First, we need to install Nodejs to Ubuntu server. We will use nvm (node version manager) to manage nodejs installations.

Note: Sudoer account will be necessary to install nodejs on Ubuntu
We will install required dependencies to build nvm by typing below to the terminal.

```
$ sudo apt-get update
$ sudo apt-get install build-essential libssl-dev
```

After the dependencies successfully installed we may install nvm by following;

```
$ curl -sL https://raw.githubusercontent.com/creationix/nvm/v0.31.0/install.sh -o
install_nvm.sh

$ bash install_nvm.sh

$ source ~/.profile
```

Once nvm installed, we can install all versions of nodejs by using it. For our application we will be using latest stable nodejs version.

```
$ nvm install stable
```

Following Node.JS libraries will be used:

2. Backend Tools

Package	Link	Explanation
bcryptjs	https://www.npmjs.com/package/bcryptjs	Optimized bcrypt in JavaScript with zero dependencies.
body-parser	https://www.npmjs.com/package/body-parser	Node.js body parsing middleware.
compression	https://www.npmjs.com/package/compression	Node.js compression middleware.
cors	https://www.npmjs.com/package/cors	CORS is a node.js package for providing a Connect/Express middleware that can be used to enable CORS with various options.
express	https://www.npmjs.com/package/express	Fast, unopinionated, minimalist web framework for node.
express-boom	https://www.npmjs.com/package/express-boom	Boom response objects in Express.
helmet	https://www.npmjs.com/package/helmet	Helmet helps you secure your Express apps by setting various HTTP headers.
jsonld	https://www.npmjs.com/package/jsonld	This library is an implementation of the JSON-LD specification in JavaScript.
jsonwebtoken	https://www.npmjs.com/package/jsonwebtoken	An implementation of JSON Web Tokens.
mongoose	http://mongoosejs.com/	elegant mongodb object modeling for node.js
chai-as-promised	https://github.com/domenic/chai-as-promised	Chai as Promised extends Chai with a fluent language for asserting facts about promises.
eslint	https://eslint.org/	The pluggable linting utility for JavaScript and JSX

Package	Link	Explanation
nodemon	https://nodemon.io/	Nodemon is a utility that will monitor for any changes in your source and automatically restart your server
nyc	https://www.npmjs.com/package/nyc	Istanbul's state of the art command line interface, with support for: applications that spawn subprocesses

Installing Mongodb on Ubuntu

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 0C49F3730359A14518585931BC711F9BA15703C6

$ echo "deb [ arch=amd64,arm64 ] http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.4.list

$ sudo apt-get update

$ sudo apt-get install -y mongodb-org
```

After installing mongodb, mongo service should be started;

```
$ sudo service mongod start
```

Configuring Mongodb

Memoriae API needs two Databases, one for application, one for tests. Also, Memoriae API uses attended users to connect mongodb. Again two users should be created on mongodb, one for application, one for test application.

To Create Database user and test user;

```
# Creating Collections
mongo memoriae     --eval 'db.createCollection("User");'
mongo memoriae     --eval 'db.createCollection("Story");'
mongo memoriae     --eval 'db.createCollection("Annotation");'
mongo memoriae-dev  --eval 'db.createCollection("User");'
mongo memoriae-dev  --eval 'db.createCollection("Story");'
mongo memoriae-dev  --eval 'db.createCollection("Annotation");'
mongo memoriae-test --eval 'db.createCollection("User");'
mongo memoriae-test --eval 'db.createCollection("Story");'
mongo memoriae-test --eval 'db.createCollection("Annotation");'
# Creating users
# No users should be added for production database by now.
mongo memoriae-dev  --eval 'db.createUser({user:"memoriae-dev",pwd:"123456",
roles:[{role:"readWrite",db:"memoriae-dev"}]});'
mongo memoriae-test --eval 'db.createUser({user:"memoriae-test",pwd:"123456",
roles:[{role:"readWrite",db:"memoriae-test"}]});'
```

Getting Code and Configuring for Development

Code for Backend API is stored on Github repository. To clone repository;
Note: Git should be installed.

```
$ git clone https://github.com/574BandOfBrothers/memoriae-api.git

$ git clone https://github.com/574BandOfBrothers/memoriae-app.git
```

After cloning the repository, environment configuration should be created;

```
$ cp src/config/envirpments/enviroments.json.example development.json
$ cp src/config/envirpments/enviroments.json.example  test.json
$ cp src/config/envirpments/enviroments.json.example  production.json
```

After creating json files, DB configuration should be filled with respect to previous steps. Also twitter consumer key and consumer secret should be filled.
After above requirements fulfilled, to run development backend application;
To install dependencies;

```
$ npm install
```

To start project;

```
$ npm start
```

Our application's QR code will be generated automatically and users can scan with EXPO app to scan. Note: Client application should be opened with the same netwrok with the API, otherwise CORS errors may occur.
To run linter;

```
$ npm run lint
```

To run tests, jest is used for tests;

```
$ npm run test
```

To create optimized production build;

```
$ npm run build
```

Build will be placed under build directory.

3. Test Tools

Package	Link	Explanation
chai	http://chaijs.com/	Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any javascript testing framework.
sinon	http://sinonjs.org/	Standalone test spies, stubs and mocks for JavaScript.
mocha	https://mochajs.org/	Mocha is a feature-rich JavaScript test framework

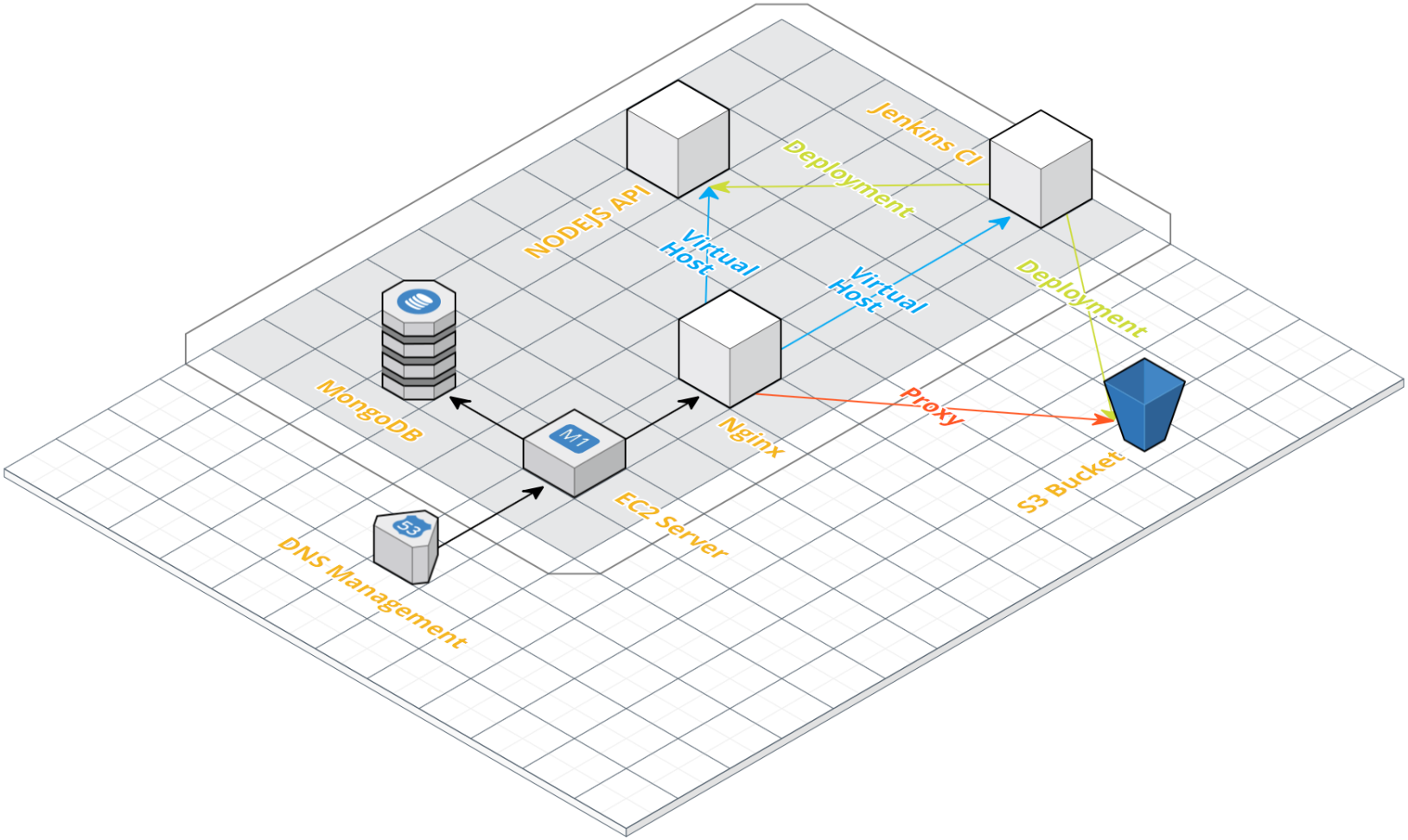
Package	Link	Explanation
chai-HTTP	https://github.com/chaijs/chai-http	HTTP integration testing with Chai assertions.

F. PRODUCTION ENVIRONMENT & CONTINUOUS INTEGRATION

Memoriae application is deployed and have running production environment. By saying production, it should be noted that the application is on sandbox environment right now.

Memoriae is powered by AWS Cloud. On the sandbox environment, due to the cost cutting reasons, there is only one EC2 instance is running for both MongoDB, and Nodejs API. Also, there is an automation server running on the same instance to support continuous integration (CI) for development environment.

Single page web application is serving under Amazon S3, as it is a combination of static assets. S3 was chosen for serving static assets, to decrease load on API Server and to benefit from dynamic CDN features provided by S3 buckets. There is a graph below to illustrate cloud formation of sandbox environment.


Jenkins CI
Deployment
NODEJS API
Virtual Host
Virtual Host
Deployment
MongoDB
M1
Nginx
Proxy
S3 Bucket
EC2 Server
53
DNS Management

Sandbox application web site lives on http://memoriae.online/ and CI server lives on http://ci.memoriae.online

DNS management is configured on Amazon Route 53 which is transferring all requests for ci.ardaogulcan.com and memoriae.online to Micro 1 EC2 instance running on Frankfurt (eu-central-1). EC2 instance has MongoDB server running on 27017 port and Nginx server which is listening all incoming requests to 80 port.

Nginx server is configured to proxy requests to underlying local services. There 2 virtual host configurations one for Automation Server, which welcomes requests from http://ci.memoriae.online, and one for Nodejs API.

CI Configuration

To provide continuous integration, Jenkins automation server was used. Jenkins is an open source automation server with rich plugin support for build deploying and automating projects.

To provide Github hook support a Team account was generated under Github for repos. All developers and automated jenkins-ci account added to the team. Also, jenkins-ci account configured to have write access to both repos.

Our Github repos are listed below:

For technical documentation:
https://github.com/574BandOfBrothers/memoriae-utils

For project management:
https://github.com/574BandOfBrothers/memoriae

For application development:
https://github.com/574BandOfBrothers/memoriae-app

For API development:
https://github.com/574BandOfBrothers/memoriae-api

API Pull Request Checker Job

Pull request checker is listening Github hooks for new Pull requests to the develop branch on Memoriae-API repository. Once a new pull requests have made, PR checker Job starts and check if the codebase on PR is passing Unit tests and conforms to the Code Linter. If this checks fails, PR cannot be merged and reason for failure can be investigated on Jenkins console output.

This job runs following commands to automate checks;

```
$ npm-cache clean --cacheDirectory $WORKSPACE/npm-cache/
$ npm-cache install --cacheDirectory $WORKSPACE/npm-cache/
$ npm run lint
$ npm run test
```

API Deployment Job

Deployment job is listening Github hooks for merges on develop branch. Once a PR is merged to the repository, Deployment Job runs linter and tests again and if all checks passes it automatically build new codebase and restart Nodejs application with new codebase;

This job runs following commands to automate deployment;

```
$ npm install
$ npm run lint
$ npm run test
$ ssh ubuntu@localhost -t -t <<'ENDSSH'
  cd ~/repos/Memoriae-API
  rm -rf dist
  git pull
  npm install --production
  pm2 reload memoriae
  exit
ENDSSH
```

Client Pull Request Checker Job

Pull request checker is listening Github hooks for new Pull requests to the develop branch on Memoriae-Client repository. Once a new pull requests have made, PR checker Job starts and check if the codebase on PR is passing Unit tests and conforms to the Code Linter. If this checks fails, PR cannot be merged and reason for failure can be investigated on Jenkins console output.

This job runs following commands to automate checks;

```
$ npm-cache clean --cacheDirectory $WORKSPACE/npm-cache/
$ npm-cache install --cacheDirectory $WORKSPACE/npm-cache/
$ npm run lint
$ npm run test
```

Client Deployment Job

Deployment job is listening GitHub hooks for merges on develop branch. Once a PR is merged to the repository, Deployment Job runs linter and tests again and if all checks pass it automatically build optimized production scripts and upload those files to a S3 bucket for serving.

This job runs following commands to automate deployment;

```
$ npm install
$ npm run lint
$ npm run test
$ ssh ubuntu@localhost -t -t <<'ENDSSH'
  cd ~/repos/memoriae-app
  rm -rf build
  git pull
  npm install --production
  npm run build
  aws s3 rm s3://memoriae.online --recursive
  aws s3 cp build/ s3:// memoriae.online --recursive
  exit
ENDSSH
```

Note: Required AWS IAM Role Keys are preconfigured on deployment machine.

G. DEPLOYMENT

Memoriae consist of 2 main components:

- BACKEND : Memoriae API
- FRONTEND: Memoriae APP

Memoriae can be install to your local directory automatically on linux based distributions by using following scripts, which can be found in project files under Scripts directory:

- api.sh
- app.sh

Also there is an automated DB clean up script for local deployment. PLEASE note that using docker prevents you doing that:

- resetdb.sh

Please refer this wiki page and readme.md for more information:

https://github.com/574BandOfBrothers/memoriae/wiki/Auto-Deployment

https://github.com/574BandOfBrothers/memoriae-api/blob/develop/README.md

https://github.com/574BandOfBrothers/memoriae-app/blob/develop/README.md