Technical Specification

For

Shark Tagging Game

|  |  |
| --- | --- |
| **Author** | **Student ID** |
| Povilas Auskalnis | 13346476 |
| Jerzy Baran | 13307086 |

Date: 09/03/2016

Table of Contents

[1. Introduction 3](#_Toc445817441)

[1.1. Overview 3](#_Toc445817442)

[1.2. Glossary 3](#_Toc445817443)

[2. System Architecture 4](#_Toc445817444)

[2.1. The game 5](#_Toc445817445)

[2.2. The back-end server 5](#_Toc445817446)

[2.3. The frame extractor 5](#_Toc445817447)

[2.4. The database 6](#_Toc445817448)

[3. High-Level Design 7](#_Toc445817449)

[3.1. Database schema 7](#_Toc445817450)

[3.2. Players relation 7](#_Toc445817451)

[3.3. Images relation 8](#_Toc445817452)

[3.4. Tagged Images relation 8](#_Toc445817453)

[3.5. Tags relation 8](#_Toc445817454)

[3.6. Live Config relation 9](#_Toc445817455)

[3.7. REST API 9](#_Toc445817456)

[5. Installation Guide 11](#_Toc445817457)

[5.1. Installing the server 11](#_Toc445817458)

[5.2. Installing the application 11](#_Toc445817459)

# 1. Introduction

## Overview

Shark Tagging Game is an educational application for children and adults. The main goal of this project is to correctly tag thousands of images in preparation for a neural network training. It will be available on multiple platforms including desktop (HTML5), mobile phones (Android/iOS) and tablets (Android/iOS).

It was developed using a client/server model by building a game (i.e. the client) and a back-end (i.e. the server). The game talks to the server and in turn the server responds to client requests. The server handles communication with the database and file I/O.

The server was built using NodeJS which is an event-driven and non-blocking I/O technology built using JavaScript. The NodeJS backend uses MongoDB, Express.js and Mongoose to build a REST API with which the game can easily communicate using HTTP requests.

The game was built using the LibGDX library which allows to build games using Java for multiple platforms like desktop, HTML5, Android and iOS. The LibGDX library is perfect for our solution because it provides built-in libraries to convert between Java and JSON objects easily.

## Glossary

**NodeJS** - Event-driven I/O server-side JavaScript environment based on V8.

**MongoDB** - Cross-platform document-oriented database classified as a NoSQL database.

**Express.js** - Node.js web application server framework for building web applications.

**Mongoose** - Elegant MongoDB object modelling framework for NodeJS

**REST API** - representational state transfer API which allows communication through requests using the HTTP protocol.

**LibGDX** - game development framework written in Java with some C/C++ components that allows game development of desktop and mobile games using the same code base.

**JSON** - JavaScript Object Notation is a lightweight data-interchange format.

# System Architecture



There are five components in the system and each handle a set of functionality: the game (client), back-end server (NodeJS), database (MongoDB), disk storage and the frame extractor tool for processing video footage.

The game which is running on the user-side communicates with and only with the back-end server. Database queries/transactions and file I/O is handled by the back-end server.

## 2.1. The game

The game is built using a 3rd party game library called LibGDX. The main game code is written using the Java programming language and the code compiles directly to a Java desktop application, Android application, iOS application and HTML5.

Communication to the server is done using a REST API. The game uses LibGDX’s networking libraries to create HTTP requests to the server and send JSON-encoded requests which are converted from Java objects (such as LoginRequest).

The game is directly responsible for handling client side gameplay and no back-end work is done on the client side.

## 2.2. The back-end server

The back-end server handles HTTP requests for the REST API and in turn accesses the database/file system when it’s relevant to do so.

Authentication is done using a login procedure. When the username/email and password are deemed to be correct the server generates a random string called “token” and this token is sent back to the client. The client must use the token for all further interaction in order to identify the user.

The NodeJS back-end utilises various 3rd party tools:

* Express.js - a powerful web application framework which allows to define routes for different HTTP requests.
* MongoDB client - a connector tool to talk with the MongoDB database
* Body-parser - a library for parsing request bodies to JSON objects
* Bcrypt - a library for hashing player passwords
* Mongoose - a modelling framework for defining the structure of database models
* Morgan - a library for logging HTTP requests
* NodeMailer - a library for sending emails

## 2.3. The frame extractor

The frame extractor is a command line tool for exporting video frames to JPEG images to be used in the game. It is written in Python and uses the OpenCV framework for reading video files and exporting JPEG files.

The tool currently takes a list of video files and outputs high quality JPEG images into a specified output directory. These images can then be indexed for use in-game using the NodeJS back-end server.

## 2.4. The database

The project uses MongoDB as the database system. MongoDB was chosen because our server communication involves sending/receiving JSON-encoded objects. MongoDB is perfect because all the database contents are already stored in JSON format so there is very little conversion to be done.

The database is used to store player data, image indexing, tagged images and the tags themselves.

# 3. High-Level Design

## 3.1. Database schema



## 3.2. Players relation

The players’ relation represents a player account. It stores basic details about the player like their name, email address and so forth.

The **password** is stored as a hash using the bcrypt function which uses the Blowfish algorithm. This ensures that player passwords cannot be compromised and are stored securely. Bcrypt generates password hashes using salts for further protection.

The **token** is a combination of the \_id (which is an internal string generated by MongoDB to identify tuples) and a random string to create a string of length 48 characters for communication. When a player logs in this token is stored as an attribute and it can be used for all further communication such as image requests, tag submission and so on.

The **recoveryCode** attributes stores the recovery code sent to an account owner’s email when they attempt to recover a password. When there is no password recovery attempted or password recovery is complete, this attribute is stored as a null value.

The **activationCode** attribute stores an activation code which was sent to the account owner’s email when they register. This is used to verify that players use correct email addresses. The account cannot be used until the account is activated. This attribute is a null value once activation is completed.

## 3.3. Images relation

This relation stores the indexing data for image used in-game. Each tuple points to a single image on the file system.

The **chunk** attribute defines the chunk number that the image belongs to. Each chunk number represents 100 images and this number can be configured in the server configuration file.

The **folder** attribute stores the directory in which the image is being held.

The **imageFile** attribute stores the filename of the image including the file extension.

The **testImage** attribute decides whether an image is a test image which we already know and have identified the shark for testing the user.

## 3.4. Tagged Images relation

This relation stores images that have been tagged by players. Each tuple points to the imageId key of the tagged image, the playerId key of the player and an IP address to be able to track where the player is located in the world.

## 3.5. Tags relation

Each tagged image may contain more than one shark (or none at all), a tags relation is required. Each tuple represents a tagged shark on a certain tagged image.

The **taggedImageId** attribute specifies which tagged image the tag belongs to.

The **sharkId** attribute specifies which specie of shark was found.

The **posX** and **posY** attributes define the location of the box and the **sizeX** and **sizeY** attributes define the sizing of the box. The tags can be represented like below:



## 3.6. Live Config relation

This relation is used for storing information about the game status. It stores the chunking data and this is required because this chunking data is used by the image indexing/extraction tool parallel to the game server.

The **currentChunk** attribute specifies on which chunk the game is currently based on. New players should not tag images which have been already tagged by many others.

The **currentInsertChunk** attribute specifies which chunk the next indexed image will belong to.

The **currentInsertCount** attribute specifies the number of images belonging to the latest chunk.

## 3.7. REST API

The REST API operates on the following routes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Route** | **Description** | **Request Data** | **Response Data** |
| /register | Allows a player to create a new account | **“username”** - player’s chosen name  **“email”** - player’s email address  **“password”** - player’s chosen password | **“success”** - integer result status  **“message”** - registration result message for client |
| /login | Allows a player to log in | **“username”** - player’s username or email address  **“password”** - player’s password | **“success”** - integer result status  **“message”** - login result message  **“token”** - session key for communication  **“username”** - player’s username  **“tutorialFinished”** - Boolean flag specifying whether a player has finished the tutorial  **“score”** - current player’s score |
| /reqimage | Returns the metadata for the next image to be tagged by the player | **“token”** - player’s session key | **“success”** - integer result status  **“imageId”** - returns the ID of the image to be tagged  **“url”** - URL to the image download |
| /getimage/{imageId} | Returns JPEG image | None - URL defines imageId | None - returns image with content type of JPEG |
| /submittags | Submits all the tags the user came up with for an image | **“token”** - player’s session key  **“imageId”** - the imageId of the image the player is tagging  **“tags”** - an array of tags containing **“sharkId”** - the ID of found shark, **“posX”** and **“posY”** - the position of the box and **“sizeX”** and **“sizeY”** - the size of the box | **“success”** - integer result status  **“message”** - submission result  **“score”** - the player’s updated score |
| /finishtutorial | Updates the tutorial flag when player finished tutorial | **“token”** - player’s session key | **“success”** - integer result status  **“message”** - tutorial completion message |
| /autologin | Attempts to re-login with a saved session token | **“token”** - player’s session key | **“success”** - integer result status  **“message”** - login result message |
| /logout | Logs a player out of the game and disables their token session key | **“token”** - player’s session token | **“success”** - integer result status  **“message”** - logout result message |
| /recoverpassword | Forgotten password recovery | **“username”** - player’s username or email | “**success”** - integer result status  **“message”** - recovery result message  **“username”** - the player’s password |
| /recoverpasswordchange | Password recovery step 2: changing password | **“username”** - player’s username  **“code”** - recovery code received by email  **“password”**  - new password | **“success”** - integer result status  **“message”** - password recovery result message |
| /activate/{username}/{code} | Activates a new player account | None - URL includes username and code | None - shows a message in the web browser for player |
| /leaderboard | Returns the current high scores | **“token”** - player’s session key | **“success”** - integer result status  **“leaderboard”** - an array of top 20 leaders containing **“username”** - player’s username and **“score”** - corresponding player’s score |

# 4. Problems and Resolution

## 4.1. Native HTTP requests

We discovered Java HTTP requests are not supported by all platforms like HTML5. As a result we had to rewrite our networking code to use the LibGDX networking classes which are wrappers for native code on each platform.

This was a major step-back because the LibGDX HTTP requests worked completely differently to Java HTTP requests and it required us to re-learn the networking interfacing.

## 4.2. Case-sensitivity in MongoDB

MongoDB by default is case-sensitive. This means that when searching for specific attributes in a tuple, the value of the attribute needs to be in the exact same case as it is stored in the database.

This resulted in a major problem for logging in. If a player typed their username or email address in the wrong case then their account was not found.

We resolved this issue by using regular expressions. A regular expression is passed during queries which ignores case instead of searching for the exact name. This means that the player doesn’t need to make sure the username or email address in the exact case they used when they registered their account.

## 4.3. Image storage

We had a really tough decision with images - should we store them in the database encoded as base64 text or use the database for indexing.

Initially we tried to store the images as text in the database but this proved to be a problem. The conversion from a byte array to a base64 string which increased by 33% in size was then sent over the network to the game and then converted back to a byte array. This process took some time and made the gameplay slow.

We resolved this issue by using the image database for image indexing. The images remain stored in the file system and the database only stores metadata for that image such as the location of the file and the filename. This created more file management overheads but it made the gameplay faster.

# 5. Installation Guide

## 5.1. Installing the server

Installation steps performed on a Debian based Linux distribution such as Ubuntu

1. Install MongoDB (v2.4.0) - **apt-get install mongodb**
2. Install Node Version Manager - **git clone** [**https://github.com/creationix/nvm.git**](https://github.com/creationix/nvm.git)
3. Install NodeJS (v4.3.0) - **nvm install 4.3**
4. Copy the server files to the Linux-based server
5. Install NodeJS dependencies using Node Package Manager which comes with NodeJS - **npm install** (inside game server directory). This automatically gets the required Node packages with their versions which are listed in the **package.json** file.
6. Initialise database using the initialiser script - **node initDb.js**
7. Copy **stgserver.service** to the SystemD start-up script directory (**/etc/systemd/system**)
8. Start game server using SystemD - **systemctl start stgserver**
9. Enable game server to start on boot - **systemctl enable stgserver**

The server should now be installed and running on the server.

## 5.2. Installing the application

**Desktop (Windows, Linux, MacOS)**

The desktop application does not need to be installed. It is a single JAR file which can be run from anywhere.

**Android**

Export the APK file from Eclipse. You can install it in two ways.

1. Copy the APK onto the phone and install it using any file manager. For this to work, the phone’s setting need to be set to “Allow installation of untrusted packages”
2. Upload the APK to Google Play Store. Users will download and install the application through the app store.

**iOS**

Upload the game to the App Store from where the users will be able to download and play the game.