2019

METADATA-BASED IMAGE COLLECTING AND DATABASING FOR SHARING AND ANALYSIS

Xi Wu
University of Kentucky, xnwu222@g.uky.edu
Author ORCID Identifier:
https://orcid.org/0000-0001-8590-0750
Digital Object Identifier: https://doi.org/10.13023/etd.2019.164

Xi Wu, Student

Dr. Guo-Qiang Zhang, Major Professor

Dr. Mirek Truszczynski, Director of Graduate Studies

METADATA-BASED IMAGE COLLECTING AND DATABASING FOR SHARING AND ANALYSIS

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in the College of Engineering at the University of Kentucky

By

Xi Wu

Lexington, Kentucky

Co- Directors: Dr. Guo-Qiang Zhang, Professor of Biomedical Informatics

and Dr. Jin Chen, Professor of Biomedical Informatics

Lexington, Kentucky

2019

ABSTRACT OF THESIS

METADATA-BASED IMAGE COLLECTING AND DATABASING FOR SHARING AND ANALYSIS

Data collecting and preparing is generally considered a crucial process in data science projects. Especially for image data, adding semantic attributes when preparing image data provides much more insights for data scientists. In this project, we aim to implement a general-purpose central image data repository that allows image researchers to collect data with semantic properties as well as data query. One of our researchers has come up with the specific challenge of collecting images with weight data of infants in least developed countries with limited internet access. The rationale is to predict infant weights based on image data by applying Machine Learning techniques. To address the data collecting issue, I implemented a mobile application which features online and offline image and annotation upload and a web application which features image query functionality. This work is derived and partly decoupled from the previous project – ImageSfERe (Image Sharing for Epilepsy Research), which is a web-based platform to collect and share epilepsy patient imaging.

KEYWORDS: image data collecting, text-based image query, mobile development, web development, APIs.

Xi Wu
(Name of Student)

03/01/2019
Date

METADATA-BASED IMAGE COLLECTING AND DATABASING FOR SHARING AND ANALYSIS

By
Xi Wu

Dr. Guo-Qiang Zhang
Co-Director of Thesis

Dr. Jin Chen
Co-Director of Thesis

Dr. Mirek Truszczynski
Director of Graduate Studies

03/01/2019
Date

ACKNOWLEDGMENTS

First of all, I am sincerely indebted to my former advisor, Dr. GQ Zhang, for his consistent guidance, encouragements, advice, and financial support along the journey of my master's studies both at Case Western Reserve University and University of Kentucky. He is one of the greatest examples of being a good researcher in Informatics field. He showed immense knowledge, enthusiasm, and motivations. In all my research and projects, I can focus on solving the problems and apply the solutions without the pressure of publications. "Publications are always the byproducts in research, our researchers focus on the problem itself." That is his attitude towards all works done by his students. Without such motivation, I would not have done my projects with such dedication, and this thesis would not have been possible without him.

I would also like to express my sincere gratitude to Dr. Jin Chen for being my co-advisor and thesis committee chair and Dr. Jinze Liu and Dr. Tingting Yu for joining my thesis committee.

My sincere thanks also go to Steve Roggenkamp and Dr. Shiqiang Tao who has worked with me in modeling the system and provided technical assistance whenever I need it.

I would like to thank my entire lab in Institute for Biomedical informatics at University of Kentucky for their help in my research who have provided valuable insights and technical advice.

Last but not the least, I would also like to thank my wife, Mengyan Wang and my parents for supporting my life. Great thanks to all of them.

TABLE OF CONTENTS

LIST OF FIGURES

[redacted] INTRODUCTION

1.1 MOTIVATION

This work is inspired by one of my previous projects – ImageSfERe (Image sharing for epilepsy research). ImageSfERe aims to provide an online imaging repository for imaging sharing among epilepsy researchers. [1] There are still a large number of hospitals and healthcare systems intensively use CDs/DVDs to store and share medical imaging. It harms the data security since the imaging stored in CDs is not encrypted. It also makes the data transfer a lot more difficult. Not to mention these disks could be shared back and forth among a large health system and other provider organizations. [2] There could be data losses or damages due to the process of constant media transport.

In this situation, an online portal is crucial for the share of imaging. By using the web-based image repository, it enables secure, consistent and quick patient data storage. It also allows faster retrieval of patient record which includes patient de-identified background information, imaging and associated annotations.

In addition, the goal of ImageSfERe is to share the de-identified epilepsy patient imaging data to as many researchers as possible. Especially, the query interface should be accessible to everyone who is interested in epilepsy field to apply their own data analytics on a specific dataset.

The data resources are collected from epilepsy patient imaging along with the metadata and annotations made by radiologists and physicians. To improve the use of the image data, I implemented a query module with the user interface and the backend query logic. However, the query module is more or less imaging-focused or patient-focused which indicates the limitation of being reused in other image-related projects. To address this challenge, I have removed patient and clinical context and related components and implemented a REST (Representational State Transfer) API (Application Programming Interface) to improve the adaptivity and extensibility of the system being reused in another context. In fact, the goal for the project is to build a data collecting and query platform for institutions or enterprises to collect file-based and text-based data for analysis.

1.2 RELATED WORK

Extensible Neuroimaging Archive Toolkit (XNAT) is the most widely-used imaging informatics software platform built for imaging-based research. Its main functionality include imaging importing, storing, processing and retrieving. XNAT can be applied to a wide range of use cases such as user defined single studies, multi-center trials, institutional full of studies, HIPAA (Health Insurance Portability and Accountability Act) -compliant clinical translation services, or public store of public datasets. [3] With the increasing prevalence of data sharing, XNAT has been extended to serve in this context. It provides a harmonization module for investigators to replace their scan labeling scheme with commonly used concepts.

Researchers from the Department of Radiology at Stanford University proposed BIMM (Biomedical Image Metadata Manager), a system that uses the imaging metadata to drive imaging queries. This system was mainly built to interface with PACS (picture archiving and communication systems). As a complement, it addresses the needs of image query for the data stored in PACS. [4] An interesting idea introduced in this system is that users can retrieve not only images based on the specified metadata but similar metadata. It is crucial for this project because the database is starting with small datasets. In order to retrieve enough research data, offering more data can be effective. More importantly, it has been proved by previous work which has shown that using metadata to query medical image database and retrieving similar images improve diagnosis. [5-10]

1.3 ORGANIZATION OF THESIS

This thesis demonstrates a system that can be reused as a module or a standalone application to help organizations to collect text, image or file data and make it accessible to users with the need to acquire specific datasets. The structure of the thesis is as follows:

Chapter 2 covers the background of the project which includes the in-depth introduction of ImageSfERe - the previous project that the thesis derived from as well as more detailed purposes of initiating and implementing the application.

Chapter 3 emphasizes the methods I applied, the optimizations I implemented in order to provide better system performance, and the technologies I adopted to build various components.

Chapter 4 demonstrates the implementation details, the complete workflow of the system with screenshots for comprehension.

Chapter 5 illustrates the preliminary results of the sample data being imported into the system.

Chapter 6 provides the conclusion of what we have been implemented, what features we can provide as well as the current limitations in terms of annotation quality and corresponding potential improvements.

[redacted] BACKGROUND

My previous work – ImageSfERe is funded by the CSR (Center for SUDEP Research). It is a National Institute for Neurological Disorders and Stroke funded “Center Without Walls” for prospective collaborative research in the Epilepsies. [11] The ultimate goal of ImageSfERe is to identify patients at risk of SUDEP by studying the features of imaging from SUDEP patients. [1] One significant distinction of ImageSfERe’s approach from traditional ones is that imaging annotations take place which integrates NIH CDE (National Institute of Health Common Data Elements) as the semantic reference to facilitate data capture and integration process.

ImageSfERe supports DICOM (Digital Imaging and Communications in Medicine) which is the global standard for medical imaging and associated metadata. This format is adopted by widely used radiology, cardiology imaging and devices. [12]

However, annotations of images are also important in other fields. Associated metadata is crucial in a lot of other contexts. One researcher at University of Kentucky has raised the potential application to predict infant weights by applying Machine Learning methods on infant image and annotation data. With images of infants and their weights being annotated, a naïve linear regression can be applied to predict the weights of incoming infants based on images.

The researcher has mentioned that this project focuses on African infants due to the lack of health instruments including weighing scales. Doctors in Africa often tell the weight of an infant based on its appearance. The mismeasurement can lead to overdose or underdose of certain medicine. The researcher will be using his mobile device to take pictures of infants and record their weight and other important text-based data. One issue of data collecting is that it is difficult to access internet in least developed countries. Thus, offline image and metadata upload must be a feature of the application.

To better achieve the goal, I integrated some key components from ImageSfERe into the new application and applied the REST practice to build a web client, a mobile client and a server. In the next chapter, I will introduce the methods and methodologies I adopted.

[redacted] METHODS

3.1 DATA MODELS

In ImageSfERe, there are relatively complicated associations between different models. The blow Figure 1 describes various models and their relations.

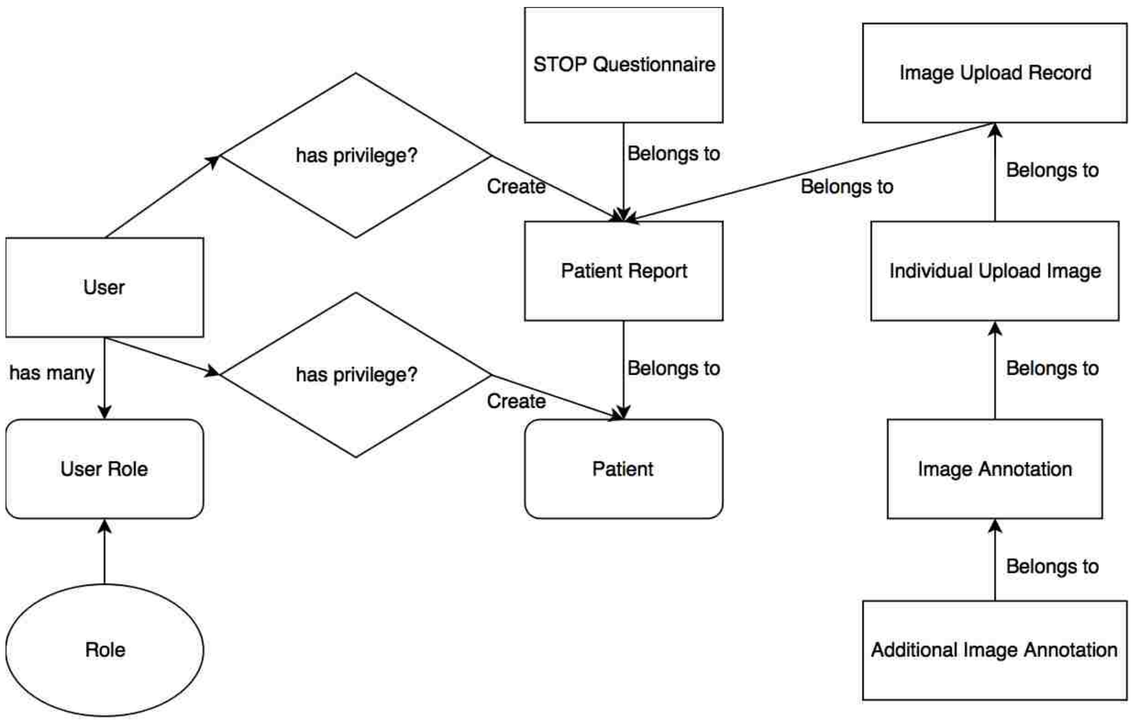


FIGURE 1. DATA SCHEMA OF IMAGESFERE

We used the above figure as the schema to build the MySQL database. In this schema, some of the items are application-specific. For example, the three models in the middle: STOP Questionnaire, Patient Report, and Patient.

The nine-sectioned STOP Questionnaire is a strictly defined EHR-like set of patient data collection that aims to record information like "Was subject's mom ill during pregnancy?" The questionnaire consists of four other tables: sections, questions, options, and user answers. Sections have many questions. Questions have many options.

The Patient Report includes patient demographics, STOP Questionnaire, and autopsy reports. One patient can only have one patient report.

In the patient table, data like referring site, institution identifier, age at death, etc. are stored.

Images in this context are combined with individual patients. However, in order to reuse the schema, images should be the central data model which means images are independent to any other entity except the user that uploads them. Thus, the database is able to host images from different sources and be accessible through a unified query interface.

Figure 2. is the proposed and revised schema for this application.

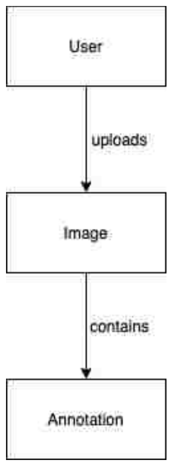


FIGURE 2. REVISED DATA SCHEMA FOR ADAPTIVITY

After the data model is defined, it can be applied and used as the object type for data traveling around the application.

3.2 TECHOLOGIES AND FRAMEWORKS

[redacted] RUBY ON RAILS

Ruby on Rails is an open-source widely-used web application development framework written in Ruby language. [13] Its goal is to reduce the burden of programming which allows developers to accomplish more functionalities with less code.

ImageSfERe was implemented using Ruby on Rails for two main reasons:

1. DRY (Don't Repeat Yourself): It is a simple principle which advocates code reuse, every piece of information on both the backend and frontend should have a unified and unambiguous representation. This improves the ease of maintaining the website content and extensibility.

2. Convention Over Configuration: Rails has already optimized and set conventions for many configurations in the web application, instead of making developers to go through every configuration. Developers are able to focus more on distinct functionalities to provide desirable features. This specific feature allows us to quickly setup the application.

Ruby on Rails follows the MVC (Model-View-Controller) design pattern. The Model is the core of the pattern providing the logic and rules of data going in and out. [14] The View is the component that generates the pages that directly interface with users. It can be any representation of information, e.g. a html page or a table. The Controller is the central component that communicates with both the Model and the View. It contains code to fetch necessary data by talking to Model and the database, it also contains the code that pass data to the View which will be accessible to users.

Since the image upload component is already completed in ImageSfERe, it is straightforward to reuse the interface with minimal changes in order to facilitate general image upload applications.

However, the query interface of ImageSfERe contains excessive query logic and code to retrieve medical imaging under certain criteria. A major cleaning or rewriting is needed.

In summary, the image uploading component can be extracted as an API that enables users to upload images, annotations and all types of metadata via web. The query interface will be rewritten by a more general architecture and also be wrapped up as an API for taking requests from all sources.

[redacted] NODE.JS

The app is using Node.js as the backbone to handle network communications between different platforms and interfaces. Node.js is an asynchronous event driven JavaScript runtime that provides scalability.

Node.js is in contrast to concurrency model where threads are employed. Thread-based networking always contains chunks of code to handle dead-locking situations, thus, it is relatively inefficient and difficult to developers. No process in Node.js blocks because Node does not have function that directly performs I/O. No blocks means that the system will be more scalable. [15]

Another reason of using Node.js is that we will be building a mobile application based on React Native which will be introduced in the next section. React Native is a JavaScript framework. Thus, they share bunches of JavaScript libraries in nature.

[redacted] REACT AND REACT NATIVE

React.js is a JavaScript library for building UIs (User Interfaces) developed by Facebook. Facebook Ads app became difficult to manage in 2012. Finally, they came up with React.js to keep up with their cascading updates. Previously, when data flows into the app, it causes some small changes deep in the tree which ends up re-rendering the entire page by looking for places where changes need to be made. This drastically slows down the whole application and destroys the user experience.

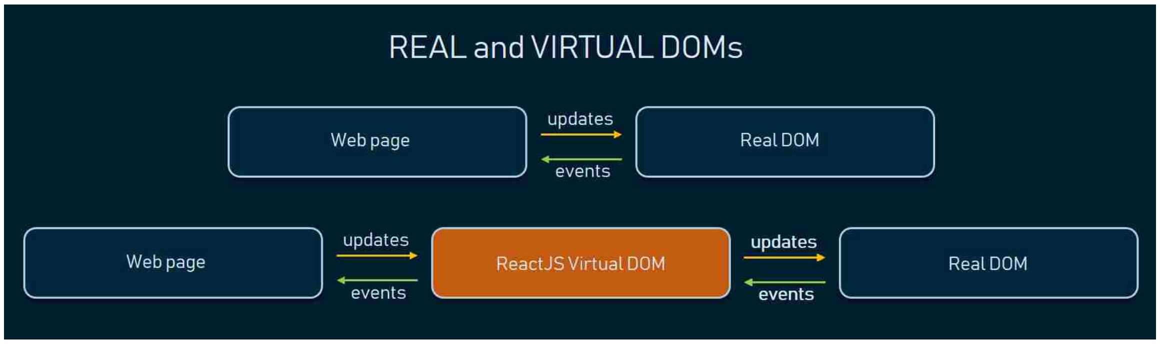


FIGURE 3. REAL AND VIRTUAL DOMS [16]

React.js provides a new way to render pages, which is highly responsive to user inputs. The magic behind the scene is the virtual DOM which will be re-rendered every time there is something changed in the browser, see figure 3. Next, React calculates the difference between the previous state of virtual DOM and current state and only applies the difference to the actual DOM which speeds up the re-rendering process without refreshing the unchanged content.

React Native was released by the same group of Facebook engineers. It is a mobile application development framework that is platform-independent in terms of mobile operating systems. The high compatibility originates from its native nature. Unlike other cross-platform tools that use mobile engines to wrap code, it controls native mobile components and thus, provides faster UIs. [16] And that is where the "Native" comes from.

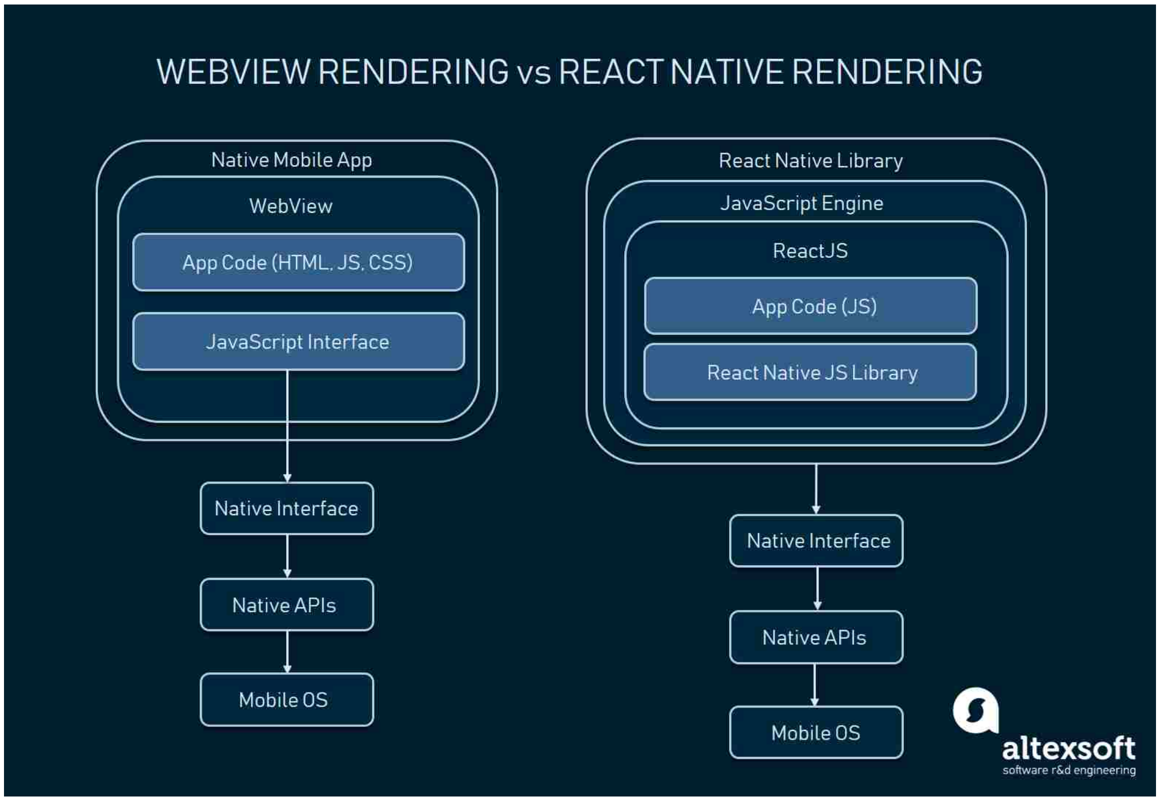


FIGURE 4. WEBVIEW RENDERING VS REACT NATIVE RENDERING [16]

Figure 4 shows the difference on the backend between the traditional WebView Rendering and React Native Rendering. Unlike other cross-platform mobile frameworks which render code via WebView, React Native communicates with specific targeted components in iOS and Android and applies different configurations to these two platforms to achieve this.

[redacted] DATABASE-RELATED TECHNOLOGIES

3.2.4.1 MYSQL

MySQL is an open source RDBMS (Relational Database Management System). [17] It is free and popular that being used by applications of all platforms. MySQL has a huge developer community, makes it a mature and stable database management to use. It has high compatibility and provides connectors to Node.js, Ruby, C#, C++, Java, etc. Therefore, it can be connected by all the components in our app.

In this application, images will be queried by two columns in annotations table: annotation label and annotation value. In order to speed up query, adding composite indexes on these two columns is a reasonable way to allow query optimizer to use for queries.

However, the values for annotations can be either numerical or textual. Thus, the index for annotation value column is not sufficient to provide the performance to both sort the numbers and strings (i.e. If one index sorts the column using alphabetical order it cannot be in numerical order). The approach is to create composite index consisting of annotation label and annotation numerical value columns, and another composite index consisting of annotation label and annotation string value columns.

Here, an interesting problem is that how do we know that user types in a number or a text. My solution is adding another column called units. Once we see a non-NULL value in this column, it is guaranteed a numerical value. If it is NULL, then the value depends on the input. If it does not contain any character outside of 0 to 9, it is a numerical value. Otherwise, it is a text value.

Essentially, queries we execute most will be in the form of below MySQL statement:

```sql
SELECT
    *
FROM
    annotations
WHERE
    label=user_input_label AND
    string_value=user_input_string_value;
```

FIGURE 5. CODE TO RETRIEVE ANNOTATIONS THAT MATCH LABEL AND STRING VALUE

The above SQL query is for retrieving the annotations with label equals to the user defined label and string value equals to user input string, users can also query annotations with numerical value or a value range as follows:

```sql
SELECT
    *
FROM
    annotations
WHERE
    label=user_input_label AND
    numerical_value < user_input_numerical_value_lower_bound AND
```

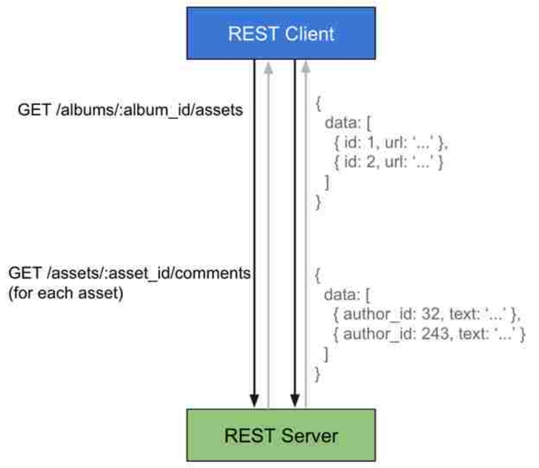
```
        numerical_value > user_input_numerical_value_upper_bound;
```

FIGURE 6. CODE TO RETRIEVE ANNOTATIONS THAT MATCH LABEL AND A VALUE RANGE

3.2.4.2 GRAPHQL

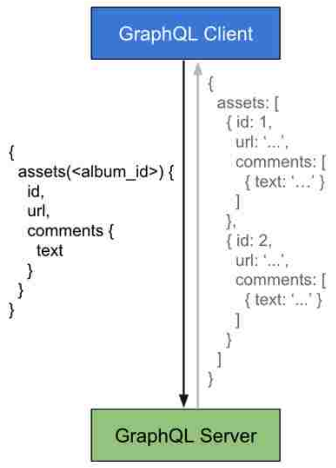
GraphQL is a query language for APIs and a runtime for query data. It gives clients the capabilities to ask for a specific structured data and no redundant data. [18] This approach allows clients to focus on what precise data is needed for the products or applications without worrying about the server-side query logic.

Compared to resource-oriented REST approaches, GraphQL fetches data more efficiently. For example, in order to fetch a list of resources, what REST might do is sending the request to the server asking for a list of resources. Then the server responds with a list of references of how to get those resources. Then the client has to send request to ask for each individual resource. In terms of time complexity of the network traffic, it is linear time. However, GraphQL finishes fetching the list of resources all in one round trip.



```
{
  data: [
    { id: 1, url: '...' },
    { id: 2, url: '...' }
  ]
}
```

```
{
  data: [
    { author_id: 32, text: '...' },
    { author_id: 243, text: '...' }
  ]
}
```



```
{
  assets(<album_id>) {
    id,
    url,
    comments {
      text
    }
  }
}
```

```
{
  assets: [
    { id: 1,
      url: '...',
      comments: [
        { text: '...' }
      ]
    },
    { id: 2,
      url: '...',
      comments: [
        { text: '...' }
      ]
    }
  ]
}
```

FIGURE 7. REST VS. GRAPHQL IN TERMS OF GET REQUEST [19]

Figure 7 is another example of comparison between REST and GraphQL. The user is trying to request two lists of resources from the server: albums and assets. Albums have many assets. In REST environment of GET request, there is no possibility of fetching these two lists at once. Because these two are different resources, having them in one server endpoint breaks the set of rules of REST. Thus, there are two individual round trips for acquiring the desirable data. In contrary, GraphQL client sends a structured GET request that specifies what the data is actually needed. The response is exactly structured like the request but contains all the data the client requested. One more important feature is that, each field of a resource is an independent component means that it is only contained in the response if the client has requested. For example, if "url" in Figure 7 is not what the client needs, data in the response will look like below:

```
{
    assets: [
        {
            id: 1,
            comments: [
                { text: '...' }
            ]
        },
        {
            id: 2,
            comments: [
                { text: '...' }
            ]
        },
    ]
}
```

FIGURE 8. GRAPHQL DATA RESPONSE

By leveraging GraphQL, our app is capable of retrieving only the user needed set of data for one image and nothing more. Also, it will optimize everything to reduce the burden of network traffic. Thus, GraphQL is desirable in this scenario as the server query logic to respond to user requests.

3.2.4.3 SEQUELIZE

Sequelize is a promise-based ORM (Object-Relational Mapping) for Node.js v4 and up. [20] It serves as connectors to bunches of different database management systems like PostgreSQL, MySQL and Microsoft SQL.

It is the most actively maintained and has an active community support. [21] One library called "sequelize-cli" is extremely useful to setup the MySQL schema for JavaScript, to load the initial set of data in the seeders and rebuild the database if necessary.

[redacted] APPLICATIONS

In this chapter, I will go through the workflow of the data capture and data query and how components work with each other seamlessly. First of all, the server-side data model schema is crucial to facilitate the data flow in-between the client and server. Next, I will demonstrate the main endpoints or APIs on the server side for listening to client requests. Finally, two front-end clients will be introduced: Web client written in React.js and Mobile client written in React Native.

4.1 DATA WORKFLOW

Data will be collected from mobile devices. Thus, image data will be uploaded through mobile applications or web applications. Users can take pictures or select images stored on mobile devices and leverage the APIs on the server to upload. If users have no internet access, we will store the image paths with annotations in the mobile local storage and whenever users gain access to the internet, they can sync the images to the image database.

The server APIs contain logic to structure the data sent from users and save it to the database tables. Usually, one upload includes one image and several annotation terms. Therefore, one record will be appended to the image table and some records will be appended to the annotation table.

From another perspective, researchers that are looking for image data can take advantage of the query interface to find certain dataset. By filling in the form to define what text features the desirable images should have and send query requests, the server is able to communicate with the database to ask for certain annotations and retrieve associated images back to the users.

The entire dataflow is depicted below in Figure 9.

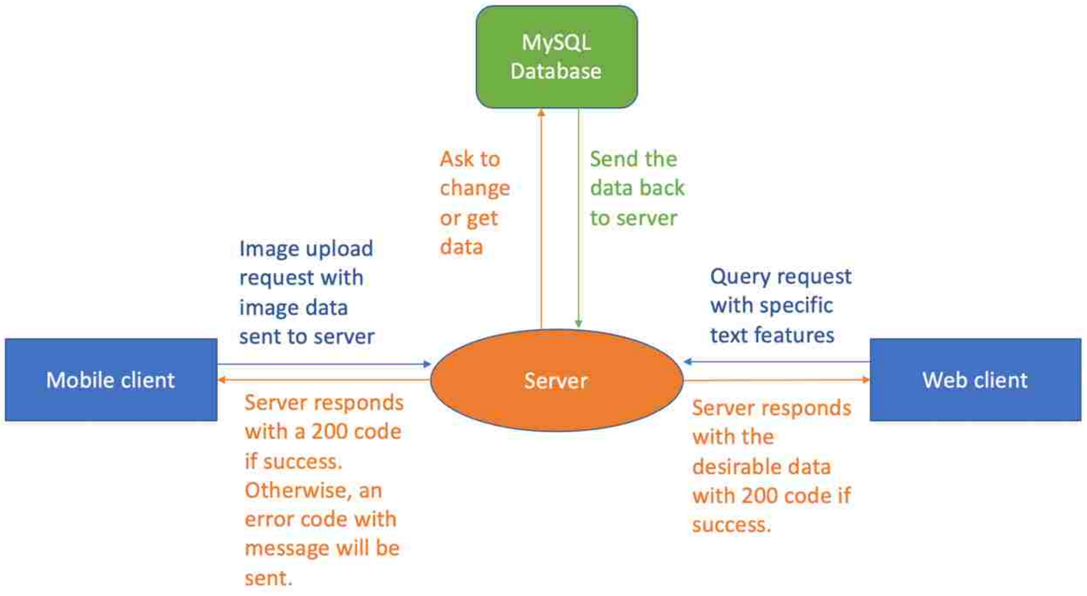

MySQL Database
Ask to change or get data
Send the data back to server
Image upload request with image data sent to server
Query request with specific text features
Mobile client
Server
Web client
Server responds with a 200 code if success. Otherwise, an error code with message will be sent.
Server responds with the desirable data with 200 code if success.

FIGURE 9. SYSTEM DATAFLOW

4.2 MODEL DEFINITIONS

Model definitions in this app is important because the data flow relies on the schema. And invalid data input can be ruled out by setting up restrictions on the server APIs.

In Sequelize.js, models are defined as Sequelize objects in JavaScript files. Below are the three key models that all our interfaces will be referring to.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {
    email: DataTypes.STRING,
    password_token: DataTypes.STRING
  }, {});
  User.associate = function(models) {
    // associations can be defined here
    User.hasMany(models.Image);
  };
  return User;
};
```

FIGURE 10. USER DEFINITION IN SEQUELIZE

In above code snippet, the User model is defined. This is the minimal setup to provide authentication and authorization features. “email” is the user ID that used to represent a valid user. “password_token” is the composite string that contains the hash value of a combination of salt and original password.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Image = sequelize.define('Image', {
    path: DataTypes.STRING,
    filename: DataTypes.STRING,
    userId: DataTypes.INTEGER,
    title: DataTypes.STRING
  }, {});
  Image.associate = function(models) {
    // associations can be defined here
    Image.belongsTo(models.User);
    Image.hasMany(models.Annotation);
  };
  return Image;
};
```

FIGURE 11. IMAGE DEFINITION IN SEQUELIZE

Figure 11 shows the Image model which contains “path”, “filename”, “userId”, and “title” fields.

“path” represents where the image can be found in the repository on the server so that front-end interface can refer to and render such image.

“filename” is the filename of the image being uploaded.

“userId” is the ID of the user that uploads the image.

“title” is the image title that user types in.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Annotation = sequelize.define('Annotation', {
    label: DataTypes.STRING,
    numerical_value: DataTypes.FLOAT,
    string_value: DataTypes.STRING,
    units: DataTypes.STRING,
    imageId: DataTypes.INTEGER
  }, {});
  Annotation.associate = function(models) {
    // associations can be defined here
    Annotation.belongsTo(models.Image);
  };
  return Annotation;
};
```

FIGURE 12. ANNOTATION DEFINITION IN SEQUELIZE

The most important model that relates to the image query is the annotation model defined in Figure 12. It defines “label”, “numerical_value”, “string_value”, “units”, and “imageId”.

“label” stands for the annotation term. For example, “height”, “weight”, “texture”, “color”, etc.

“numerical_value” stores the number that user types in through the mobile app. It is a float-type number with 4-byte precision.

“units” is the unit for the “numerical_value”.

“string_value” is the textual value that user types in.

Once the models are set up, other APIs will be using the same or silimar data structure that defined in the schema and pass objects around to each other.

4.3 SERVER API SETUP

Images are uploaded by users, so we should grant users all the rights about the images. Therefore, link images to users is a must and users must have a way to be authenticated and authorized to upload data via the app.

Routes that allows users to register and log in are configured in the following ways:

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const User = require('../models').User;

exports.register = (req, res) => {
    bcrypt.hash(req.body.token, saltRounds, function(err, token) {
        const user = User.build({
            'email': req.body.email,
            'password_token': token
        });
        user.save().then(u => {
            console.log('created: ' + u);
            const resUser = {
                _id: u.id,
                pwdToken: u.password_token
            };
            res.send({
                code: 200,
                success: 'User successfully registered!',
                user: resUser
            });
        });
    });
}

```

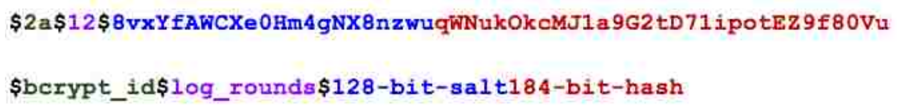
```
exports.login = (req, res) => {
    console.log('login');
    const email = req.body.email;
    const password = req.body.token;
    // sequelize login user
    User.findOne({
        attributes: ['id', 'password_token'],
        where: {
            email
        }
    }).then(u => {
      if (u) {
        bcrypt.compare(password, u.password_token, (error, compare_result) => {

          if (compare_result) {
              var resUser = {
                  _id: u.id.toString(),
                  pwdToken: u.password_token
              }
              res.send({
                  code: 200,
                  success: 'Login successfully',
                  user: resUser
              });
          } else {
              res.send({
                  code: 204,
                  success: 'Email and password does not match'
              });
          }
        });

      } else {
        res.send({
            code: 204,
            success: 'Email does not exist.'
        })
      }
    })
}
```

FIGURE 13. USER REGISTER AND LOGIN

In the code of Figure 13. We adopted the encryption method provided by "bcrypt". It generates a hash using a generated user-based salt hash and combine with "bcrypt_id", "log_rounds", and original password. We only need one column to store password related strings as shown in Figure 14. "2a" stands for the bcrypt id. "12" stands for the log round which is how many iterations that hash function runs to slow down the decryption process. While this makes an unnoticeable difference in user experience, it makes a huge difference in increasing the difficulty of password being guessed by hackers. [22]


$2a$12$8vxYfAWCXe0Hm4gNX8nzwuqWNukOkcMJ1a9G2tD71ipotEZ9f80Vu

$bcrypt_id$log_rounds$128-bit-salt184-bit-hash

FIGURE 14. HASHED USER PASSWORD [22]

4.4 MOBILE CLIENT FOR DATA CAPTURE

[redacted] REACT REDUX

To better manage the React.js states, I used Redux as the state management for our application.

A component in React has a state. Sometimes, we need to share states among various components and these components can be far from each other. Without a state management library, the state has to be lifted up and travel along the long state tree and gets to the targeted component. When application starts to scale, states can be unpredictable and difficult to maintain. [23]

Redux has one central store that holds the global truth of the application data. Components can access the central state without having to traverse the state tree.

Redux has three main building blocks: actions, store and reducers.

Actions are events triggered by user interactions. The only way to send data to Redux store is through actions. The current state will be sent to reducers to make an action and generate a new state. The store contains the application state, actions performed always return a new state to make the state predictable, maintainable, and debuggable.

Let us use user login as the example to show how Redux store and states work.

First of all, there is this login form when user launches the app. The code from line 12 to line 15 in Figure 15 will render a button with an event function triggered by user pressing the button.

```
1.  onButtonPress() {
2.      const { email, password } = this.props;
3.      this.props.loginUser({ email, password });
4.  }
5.
6.
7.  renderButton() {
8.      if (this.props.loading) {
9.          return <Spinner size='large' />;
10.     }
11.     return (
12.         <Button
13.             onPress={this.onButtonPress.bind(this)}>
14.             Login
15.         </Button>
16.     );
17. }
```

FIGURE 15. USER LOGIN BUTTON

In this "Button" component, there is an onButtonPress function defined on line 1 which calls the loginUser() function originated from the Authentication Action because the AuthAction provides the potential access to update the store.

In AuthAction, the loginUser() was defined as follows:

```
1.  export const loginUser = ({ email, password }) => {
2.      return (dispatch) => {
3.          dispatch({ type: LOGIN_USER });
4.          var url = mySQLBaseURL + '/login';
5.          axios.post(url, {
6.              email: email,
7.              token: password
8.          })
9.          .then(function (response) {
10.             console.log(response.data);
11.             if (response.data.code !== 200) {
12.                 loginUserFail(dispatch);
13.             } else {
14.                 loginUserSuccess(dispatch, response.data.user);
15.                 AsyncStorage.multiSet([['weight_teller_pwd_token', response.data
    .user.pwdToken], ['weight_teller_user_id', response.data.user._id]]);
16.             }
17.         })
18.         .catch(function (error) {
19.             console.log(error);
20.         });
21.     };
22. };
```

FIGURE 16. LOGINUSER() IN THE AUTHACTION

On line 3, the dispatch() function will carry a field called "type" which guides the reducer to update the state accordingly and return a new state. "LOGIN_USER" here represents the state that the system is currently running authentication process and at this phase, there will be a spinner animation that tells the user to wait. The corresponding code that changed the spinner status is in Figure 17.

```
1.  case LOGIN_USER:
```

```
2.              return { ...state, loading: true, error: '' };
```

FIGURE 17. LOGIN_USER CASE IN THE AUTHREDUCER

The object the function returned on line 2 in Figure 17 is the new state that inherits all the fields except overriding the “loading” field and the “error” field. When the “loading” set to true, the spinner will be displayed as an indication of system processing. In figure 18, we demonstrate how the new state reflects to the change on the user interface.

```
renderButton() {
    if (this.props.loading) {
        return <Spinner size='large' />;
    }
    return (
        <Button
            onPress={this.onButtonPress.bind(this)}>
            Login
        </Button>
    );
}
```

FIGURE 18. THE RENDER FUNCTION IN LOGIN COMPONENT THAT EITHER SHOWS A LOGIN BUTTON OR A SPINNER

Since the loading has been changed to “true”, the spinner component will be displayed to replace the original login button. Once the login process is finished everything in the state of store will be restored to initial state.

The Redux is everywhere in the app where there is an event being triggered. And each event will produce a round trip that travels through an action, a reducer, and a store.

[redacted] ONLINE MODE

In our current implementation, only Wi-Fi connection will be treated as valid internet connection to make sure our image upload will not cost users anything. I used a library named “react-native-image-picker” which supports select photos or upload pictures that are taken for both iOS and Android phones.

Later on, the uploaded images will be taken care of by the server if it received requests sent from mobile devices. On the client side, the logic to decide the status of internet is handled in Figure 19.

```
getImage(){
```

```
2.    ImagePicker.showImagePicker(options, (response) => {
3.      if (response.didCancel) {
4.        console.log('User cancelled image picker');
5.      }
6.      else if (response.error) {
7.        console.log('ImagePicker Error: ', response.error);
8.      }
9.      else if (response.customButton) {
10.       console.log('User tapped custom button: ', response.customButton);
11.     }
12.     else {
13.       NetInfo.getConnectionInfo().then((connectionInfo) => {
14.         if (connectionInfo.type === 'wifi') {
15.           // with Wifi connection
16.           this.props.imageUpload(response);
17.         } else if (connectionInfo.type === 'none') {
18.           // save locally
19.           this.props.saveImageLocally(response);
20.         } else {
21.           // with mobile data, ask if user wants to proceed
22.           // var retVal = confirm("Are you sure to upload with data?");
23.           // if( retVal == true ){
24.           //   this.props.imageUpload(response);
25.           // }
26.         }
27.         console.log('Initial, type: ' + connectionInfo.type + ', effectiveType:
    ' + connectionInfo.effectiveType);
28.       });
29.     }
30.   });
31. }
```

FIGURE 19. THE LOGIC TO DECIDE THE FUNCTION CALL FROM THE STATUS OF INTERNET CONNECTION

Notice that on line 20 in Figure 19, the “else” represents that the user is connecting via its data provided by the phone service providers. It is straightforward enough to comment it back if we set up the user agreement to use data when needed.

On line 16 in Figure 19, the imageUpload() function is called when a user has valid Wi-Fi connection, and “response” represents the image pixel data and some basic metadata not annotated by users.

We have mentioned that the user interaction triggers one action. Here, the imageUpload() function is defined in the ImageAction file. Figure 20 shows how the action structures the image data and wrap it as a request, send to the server, receive the response with an image ID so that the later annotations can be linked to the image uploaded.

```
1.  export const imageUpload = (response) => {
2.      return (dispatch) => {
        AsyncStorage.multiGet(['weight_teller_pwd_token', 'weight_teller_user_id']).then((data) => {
            if (data[0][1]) {
                // valid user
                const formData = new FormData();
                formData.append('name', 'avatar');
                formData.append('userID', data[1][1]);
                formData.append('imageData', {
                    uri : response.uri,
                    type: response.type,
                    name: response.fileName
                });
                const config = {
                    method: 'POST',
                    headers: {
                        'Accept': 'application/json',
                        'Content-Type': 'multipart/form-data'
                    },
                    body: formData,
                };
                var url = mySQLBaseURL + '/createImage';
                fetch(url,
                    config
                )
                .then((res) => {
                    if (res.status !== 200) {
                        console.log("Upload Error!");
                    } else {
                        console.log(res);
                        const { imageId } = JSON.parse(res._bodyText);
                        console.log("image upload result: " + imageId);
                        dispatch({
                            type: IMAGE_UPLOAD_SUCCESS,
                            payload: imageId
                        });
                        Alert.alert(
                            'Message',
                            'Image uploaded.',
                            [
                                {text: 'OK', onPress: () => console.log('OK Pressed')}
                            ],
                            { cancelable: false }
                        );
                        Actions.imageCreate();
                    }
                }).catch((err) => {
                    console.log(err);
                })
            }
        });
    };
};
```

FIGURE 20. IMAGE BEING UPLOADED TO THE SERVER AND STATE WILL BE UPDATED

On line 33 in Figure 20, the dispatch() function was called and the type is specified as “IMAGE_UPLOAD_SUCCESS”, and the imageID will also be passed as a parameter that is important for annotations.

In ImageReducer file, it takes the type and the imageID, create a new object with imageID. It is shown in Figure 21.

```
case IMAGE_UPLOAD_SUCCESS:
    const imageID = action.payload;
    return { ...state, imageID };
```

FIGURE 21. IMAGE IS SUCCESSFULLY UPLOADED AND IMAGEID WILL BE PASSED TO THE NEXT STEP WHICH IS IMAGE ANNOTATION

Users will be immediately led to image annotation once the image has been uploaded. The annotation form consists of as many annotations as users want. There is a “+” button at the bottom which allows users to add more annotations.

Each annotation is formed by three text inputs: label, value, and unit. The state for the annotation form component will store annotations as an array that consists of a list of label, value and unit triples. In the annotation form component, there is also a function for specifying whether to store the data locally or upload to the server. The details are shown in Figure 22.

```
onButtonPress() {
    const { properties, imageID, title } = this.props;
    NetInfo.getConnectionInfo().then((connectionInfo) => {
        if (connectionInfo.type === 'wifi') {
          // with Wifi connection
          this.props.imageCreate({ properties, imageID, title });
        } else if (connectionInfo.type === 'none') {
          // save locally
          this.props.saveAnnotationLocally({ properties, imageID, title });
        } else {
          // with mobile data, ask if user wants to proceed
        //    var retVal = confirm("Are you sure to upload with data?");
        //    if( retVal == true ){
        //      this.props.imageCreate({ properties, imageID, title });
        //    }
        }
        console.log('Initial, type: ' + connectionInfo.type + ', effectiveType: ' + connectionInfo.effectiveType);
    });
}
```

FIGURE 22. ANNOTATION UPLOAD

Again, the annotations will not be uploaded to the server but saved locally if the user does not have Wi-Fi access. The imageCreate() function takes the list of triples which is the “properties”, an imageID so that the database knows which image should the annotations be linked to, and the image title which was defined by the user.

The imageCreate() function in the ImageAction file is sending the annotation request to the server using http protocol. When the server sends back the success code, the upload process is completed, and the user will be taken back to the image upload page.

[redacted] OFFLINE MODE

The offline mode does not vary too much from the online mode except the data will be stored locally in what is called the AsyncStorage which is a data structure provided by the React Native library.

The technique here is for each image upload that being saved locally, what will be stored is the image path, instead of the actual image pixel data. And of course, the annotations and the image title will be saved with the image path to form a triple.

Thus, the AsyncStorage stores the same data structure as the “properties” parameter on line 6 in Figure 22. The reason for that is saving the image data when it is online is no different with the image data stored locally. In this way, the server API that handles saving the data can worry less about whether the internet is connected or not.

4.5 WEB CLIENT FOR DATA QUERY

The most important backend query logic is defined in GraphQL component. In order to perform a GraphQL query, a “graphqlHTTP” API is called with the query schema that defines what data clients are able to acquire.

In the schema definition, there always should be a root query that is specified for GraphQL. The root query can be defined as many levels of resources as the developer wishes to. For example, in order to retrieve images in the image library, there should be an image resource right below the root query, which returns all the images.

The code that defines images is described below in Figure 23.

```
images: {
    type: new GraphQLList(Image),
    resolve(root, args) {
        return DbImage.findAll({});
    }
}
```

FIGURE 23. GRAPHQL QUERY FOR IMAGE LIBRARY

This is where the Sequelize goes in, the DbImage defined in Sequelize object type can directly serve as the data type returned by GraphQL. With the root query, GraphQL is able to return a list of Image object which is also defined in schema showed in Figure 24.

```
const Image = new GraphQLObjectType({
    name: 'Image',
    description: 'This represents an Image',
    fields: () => {
        return {
            id: {
                type: GraphQLInt,
                resolve(image) {
                    return image.id;
                }
            },
            path: {
                type: GraphQLString,
                resolve(image) {
                    return image.path;
                }
            },
            userId: {
                type: GraphQLString,
                resolve(image) {
                    return image.userId
                }
            },
            title: {
                type: GraphQLString,
                resolve(image) {
                    return image.title
                }
            },
            user: {
                type: User,
                resolve(image) {
                    return image.getUser();
                }
            },
            annotations: {
                type: GraphQLList(Annotation),
                resolve(image) {
                    return image.getAnnotations();
                }
            }
        }
    }
});
```

FIGURE 24. IMAGE OBJECT DEFINITION IN GRAPHQL

This definition is so powerful that it allows almost infinite possible parameters that users from client side can define. We will discuss that later. But with the most basic fields like “id”, “path”, “userId”, “title”, and “user”, clients are able to do queries like the following one in Figure 25.

```
{
  images {
    id
  }
}
```

FIGURE 25. AN EXAMPLE OF A USER-CUSTOMIZED QUERY

And the server is sending the following data in Figure 26 back to the client.

```
{
  "data": {
    "images": [
      {
        "id": 1
      },
      {
        "id": 2
      }
    ]
  }
}
```

FIGURE 26. SERVER RESPONDED WITH DATA IN SPECIFIED STRUCTURE

Of course, clients are able to add more fields in the query to get the image library, and the server is responding with the data in the exact format as defined in the client, nothing more or less.

We have mentioned previously that a user can almost have infinite number of customized data structure. That is because the associated model annotation is defined as a field in Image object. That field is not simply a number or a string, it is another object. Thus, users are able to retrieve annotations for every single image in one round trip of network traffic with a query like Figure 27.

```
{
  images {
    annotations {
      id
    }
  }
}
```

FIGURE 27. QUERY WITH NESTED OBJECTS

The response is shown below in Figure 28.

```
{
  "data": {
    "images": [
      {
        "annotations": [
          {
            "id": 1
          },
          {
            "id": 2
          }
        ]
      }
    ]
  }
}
```

FIGURE 28. DATA IN THE RESPONSE WITH NESTED STRUCTURE

As we can imaging that if we define Image as a field in Annotation object, it forms a cycle. In fact, the annotations have an Image field.

```
const Annotation = new GraphQLObjectType({
    name: 'Annotation',
    description: 'This represents an Annotation',
    fields: ()=> {
        return {
            ...
            image: {
                type: Image,
                resolve(annotation) {
                    return annotation.getImage();
                }
            }
        }
    }
});
```

FIGURE 29. ANNTATION OBJECT DEFINITION IN GRAPHQL

With the Annotation definition in Figure 29, users are able to define multi-level nested data type in client queries like the one in Figure 30.

```
{
  images {
    annotations {
      image {
        annotations {
          image {
            annotations {
              ...
```

FIGURE 30. INFINITE-LEVEL NESTED DATA AS A QUERY

The above setup is the backend query logic, and of course we need a query interface for users to query. The implementation details will be introduced in the Result section. Here, I would like to illustrate the main design for query workflow.

First of all, the system would like to know which annotations the user cares most. Thus, A list of recommended annotation labels are displayed to let users select at the first step of the query. After the labels being selected, the server is now able to ask the database for annotations with those labels. For example, the user has selected the “location” label in the first step. All the locations will be wrapped in a menu for the user to later specify in the second step. For numerical annotations such as “height”, the maximum and the minimum records will be retrieved from the database. And users are able to select a height range as the specification for the image query. Finally, after all the annotations are specified with a certain value or a value range, the desirable images can be retrieved by the association with the annotation table.

Figure 31 shows the entire query process.

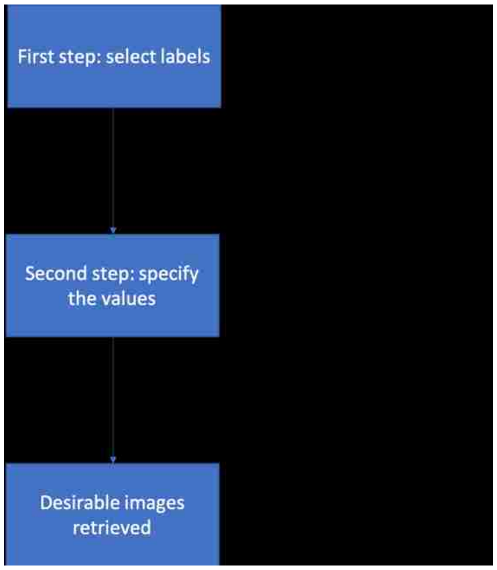

First step: select labels
Second step: specify the values
Desirable images retrieved

FIGURE 31. QUERY PROCESS

[redacted] RESULTS

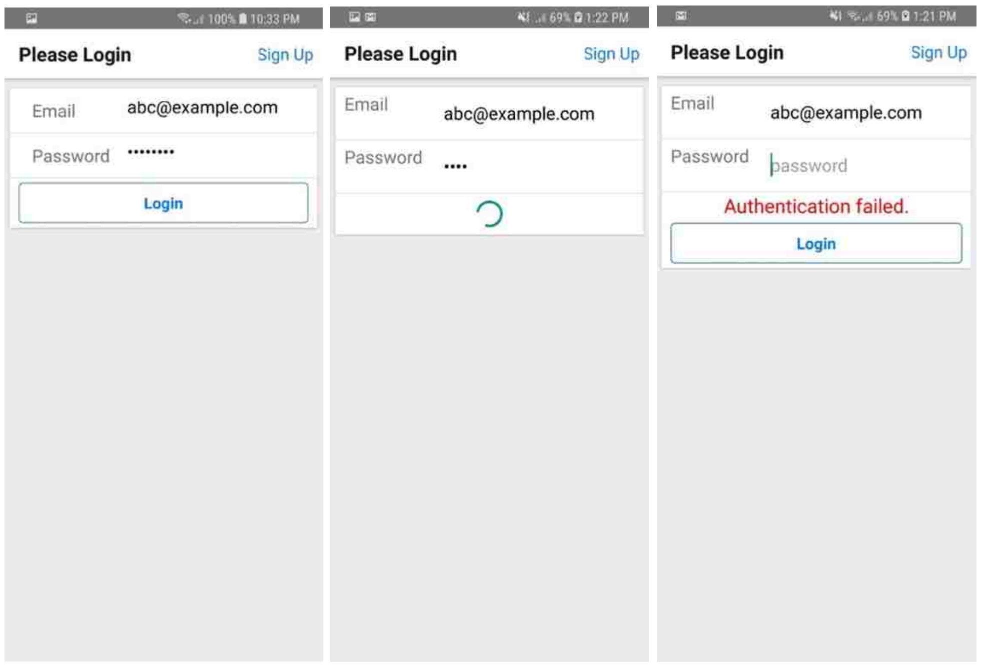
In this section, the interface design will be covered with descriptive screenshots. The mobile application includes user authentication, image upload, and annotation functions will be visualized as screenshots with descriptions. The web application mainly focuses on the image query.

5.1 MOBILE APPLICATION

The home page for this application is simply the login page because we assume that users are well aware of the app they are using, and no other information should be displayed at this moment.

By the way, the app is running in development mode that is provided by React native. It allows developers to test the application on both PCs or actual mobile devices. It is a crucial feature to test the application before launch.

Figure 32 shows the different login status on the interface. The leftmost screenshot shows the form with login credentials typed in. The middle one is when user press the "Login" button and the system starts to process the credentials to see if they match one of our records in the database. It turned out the user has either typed in the wrong email address or the wrong password.



Email abc@example.com
Password ••••••••

Email abc@example.com
Password ••••

Email abc@example.com
Password password
Authentication failed.

FIGURE 32. USER LOGIN PAGE: BEFORE LOGIN (LEFT), WAITING FOR AUTHENTICATION (MIDDLE), LOGIN FAILED (RIGHT)

Of course, on the top right corner, users can sign up if they do not have account yet. The form for users to sign up is similar to the form for the user to log in. In fact, we use the same form from backend for both actions by distinguish them only in the actions file for simplicity.

Currently, the user will be taken to the image upload page right after being successfully logged in. The CRUD (Create, Read, Update, Delete) functions are not complete. We have Create and Read implemented, Update and Delete should also be granted to users who upload the images. Moreover, some images users may not consider share them at all. An attribute in the image table that indicates the access is needed.

Figure 33 shows the image upload page under both online and offline modes. Basically, a warning message is the only difference that tells the user there is no internet connection, all image data will be saved offline.

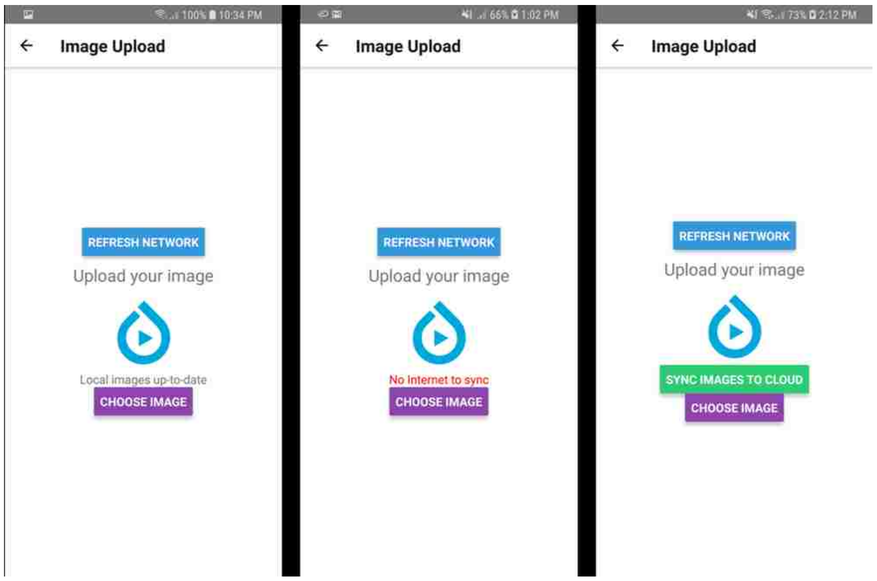


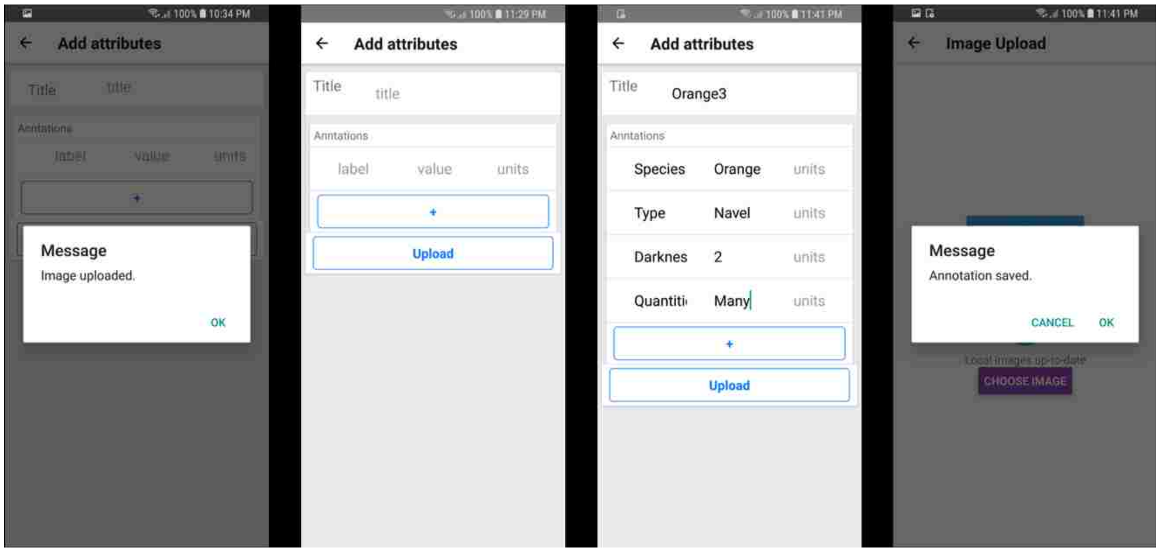
FIGURE 33. DIFFERENT STATUS OF IMAGE UPLOAD: ONLINE WITHOUT IMAGES IN THE LOCAL STORAGE (LEFT), NO WI-FI ACCESS (MIDDLE), ONLINE WITH IMAGES STORED LOCALLY (RIGHT)

To the left in Figure 33 is the online upload mode. It shows that all local images are pushed to the server. The screenshot in the middle shows that there is no Wi-Fi connection at this moment, but users can still save images and annotations to the local storage. Notice that, it is not necessary to differentiate if there is data or not in the local storage because image data will be appended to the queue of image data even if there is data previously in the local storage. To the right of Figure 33 is the online mode with image data in the storage waiting to be synced. In this situation, users can click the sync button to push images with attached annotations to the server. When the upload is finished, it loops back to the leftmost screen which is the online image upload mode without an empty image data queue in the local storage.

Users have multiple options to select images to be uploaded. They are able to upload from their phone gallery, from their Facebook account, or they can start taking a picture to upload.

In the application of weighing infants, taking a picture to upload might be the mode that is being most frequently used.

In Figure 34, a success message shows up after the image upload is complete. Users are able to add additional attributes to the image they have just uploaded. One thing to notice is that the “+” button will allow users to add any number of annotation triples they like in order to make the image data complete. The second to the right screen in Figure 34 is an example of uploading an orange image with attributes such as “Species”, “Type”, “Darkness” which means the darkness of the color, and “Quantities”.



Message
Image uploaded.

Title Orange3

Species	Orange	units
Type	Navel	units
Darknes	2	units
Quantiti	Many	units

Message
Annotation saved.

FIGURE 34. ANNOTATION PAGE

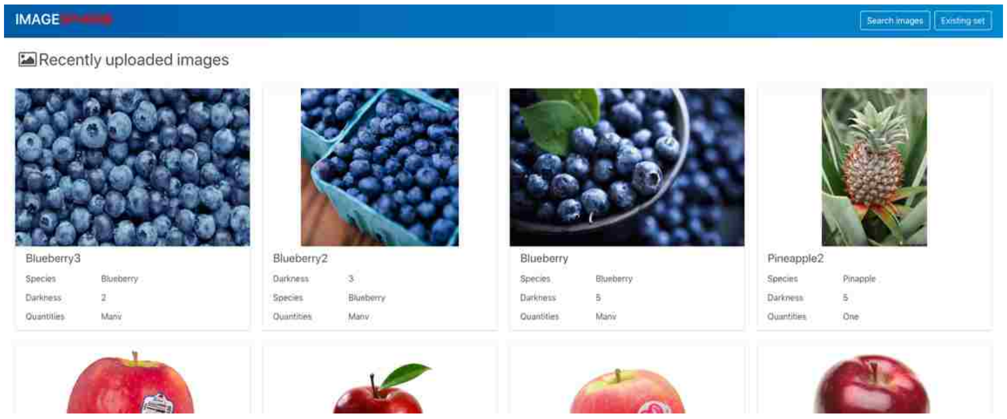
A message will show to inform users the annotations have been successfully saved. At this moment, the user just finished an upload of an image of an orange.

One can imagine the application being used in another data-related application. In the data capturing component, developers can specify a set of fields that require data uploaders to fill in to improve the quality of the image data.

5.2 WEB APPLATION

To facilitate the mobile app, I decoupled the query interface from ImageSfERe and re-implemented the main logic in Node.js and React.js.

To begin with, the homepage for the query interface in the web application is designed to show all current images in reverse chronological order so users can see the most recent images and have an idea whether some new image data has been added in.



Blueberry3

Species	Blueberry
Darkness	2
Quantities	Manv

Blueberry2

Darkness	3
Species	Blueberry
Quantities	Manv

Blueberry

Species	Blueberry
Darkness	5
Quantities	Manv

Pineapple2

Species	Pinapple
Darkness	5
Quantities	One

FIGURE 35. THE HOMEPAGE FOR THE WEB APPLICATION

For elaboration purposes, I have imported fruit images with annotations as the dummy data. Figure 35 shows the homepage with recently uploaded fruit image data. On the top right corner, users can navigate to the query page by pressing the “Search images” button. The “Existing set” feature is not available at this moment and will be implemented after we have adopted the role-based access control system from ImageSfERe.

Figure 36 is what users will visualize after pressing the “Search images” button. The database generates annotations data uploaders have used and saved and provide them as the query attributes.

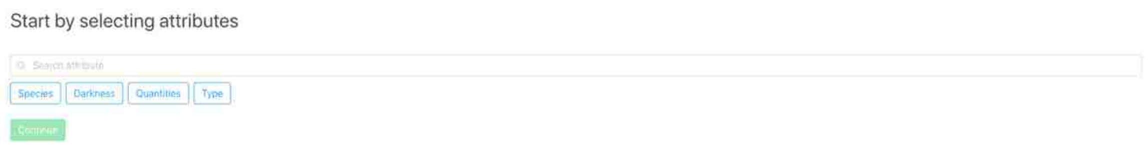
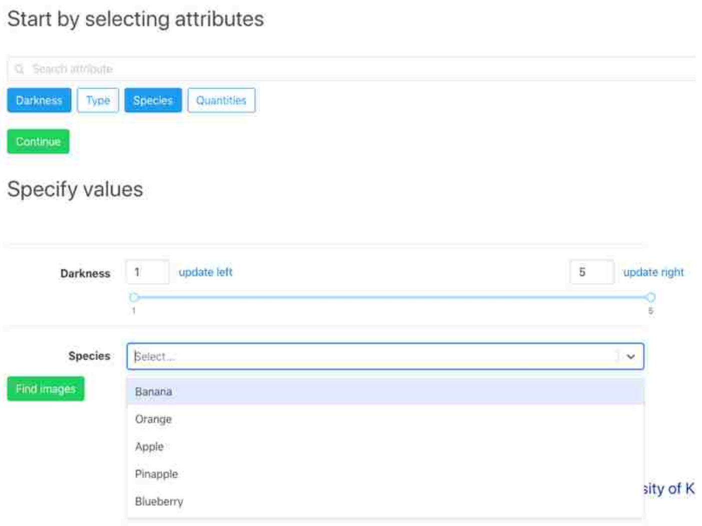


FIGURE 36. SELECTING QUERY ATTRIBUTES ON THE QUERY INTERFACE

Data is growing once users upload more image data, and more attributes will be used to tag images. Therefore, I implemented a search function so that users can type in keywords to find attributes they would like to query. The mostly related terms will be re-ordered to the leftmost for users to select.

In the following example shown in Figure 37, “Darkness” and “Species” attributes have been selected.



Start by selecting attributes

Search attribute

Darkness | Type | Species | Quantities

Continue

Specify values

Darkness: 1 update left — 5 update right

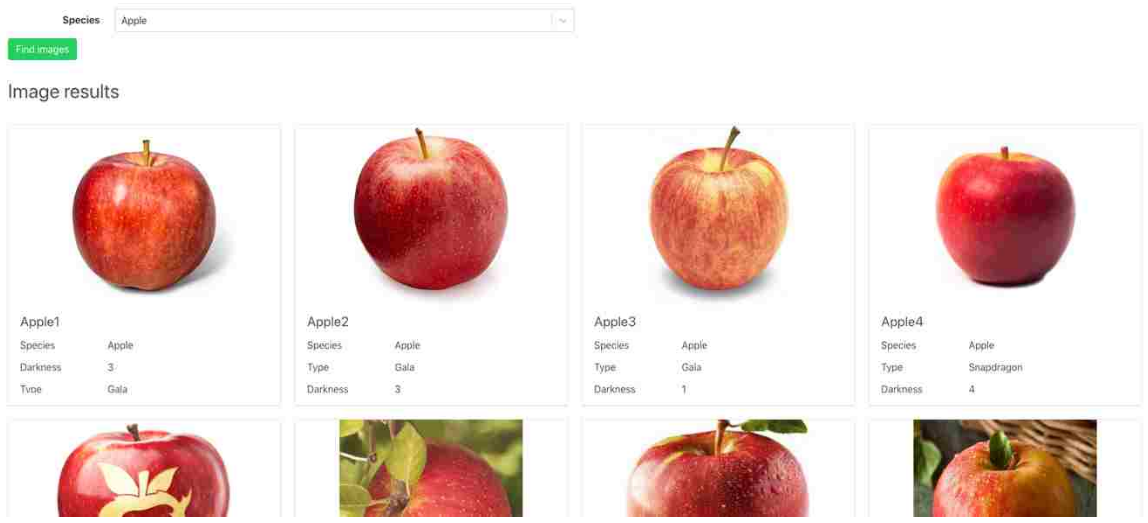
Species: Select...

Find images

Banana
Orange
Apple
Pinapple
Blueberry

FIGURE 37. SPECIFYING VALUES FOR THE ATTRIBUTES

After users have done selecting the attributes and pressed the “Continue” button in Figure 37, the server database will return the values or value range for each attribute for users to specify in the bottom section of Figure 37. The “Darkness” is a value ranges from 1 to 5, the higher, the darker. The “Species” is a textual attribute that users need to specify a species from the dropdown menu showed up when clicking on the select box.



Species: Apple

Find images

Image results

Apple1

Species	Apple
Darkness	3
Type	Gala

Apple2

Species	Apple
Type	Gala
Darkness	3

Apple3

Species	Apple
Type	Gala
Darkness	1

Apple4

Species	Apple
Type	Snapdragon
Darkness	4

FIGURE 38. IMAGE DATA AS THE QUERY RESULTS

That is all it takes to set up the query. By pressing the “Find images” button, the image results will be displayed at the bottom of the query page displayed in Figure 38. And we managed to have query attributes and query results on one single page. If the user figures out there are changes on the query attributes or values, they can update at any time even after result images have been rendered. Figure 38 displays the final results of the above query in Figure 37.

The query logic implemented here is logical conjunction “AND” means images retrieved should match all query attributes and values. With this setup, the selective results can provide high-quality data to users that demands images with specific set of attributes.

5.3 TESTING

In order to test different APIs, I used Mocha as the library to build unit tests and Chai to build integration tests.

Unit tests in this application include attribute verifications, association verifications, Javascript functions, and individual models.

Integration tests include server endpoints which consists of query APIs, user authentication APIs, and GraphQL APIs.

Sequelize has automatically generated a seeders folder so that it is straightforward to pre-define a good amount of dummy data for testing. The environment separation is also handled by Sequelize, I have “development”, “test”, and “production” configurations so that different databases do not affect each other.

The following models are tested individually:

- User
- Image
- Annotation

The following integrations are tested as well:

- User registration and login
- CRUD (Create, Read, Update and Delete) for User, Image, and Annotation models
- Image query

[redacted] DISCUSSION

6.1 CONCLUSION

This work is highly inspired by XNAT and ImageSfERe. XNAT provides the insights of sharing data with or without restrictions for research. And ImageSfERe, on the other hand, is being reused as the image upload web API server and its query interface has been decoupled to query image-based data or even file-based data.

This data capturing and data sharing application is developed to fulfill the requirements of collect image and weight data from infants in order to conduct a data analysis to predict infants from their images. The application features the online and offline data upload, online image dataset query for analysis. The offline upload feature is highly desirable due to the limitation of the internet resources in least developed countries. With image and weight data collected, one is able to use Maching Learning techniques to predict infant weight from the images.

And this work is highly extensible to be used in many other contexts. For example, it can be used as a photo gallery with advanced search functionalities. Or it can be used as an image database that hosts different kinds of data since the category of image can be specified in the annotation.

6.2 LIMITATIONS AND IMPROVEMENTS

Although we implemented the query interface as a metadata-based engine, it requires data uploaders to be cautious enough not to give any typological errors in annotations. Otherwise, the system is not able to recognize the annotation label and query such images can be impossible.

Another issue is that we have not provided any mechanisms to harmonize annotation terms. For example, one tends to use “color” while another uses “colour”. These two will be different terms in the database.

One possible way to resolve above issues is that we can provide a list of existing labels that are currently in the database to ask user if those are the terms they want to type in, but that certainly adds a system overhead such that whenever user types in a character, it triggers excessive computation for the backend to generate the list of possible terms. Of course, caching can be an effective way to provide a quick access to frequently used terms that are similar to the user inputs. Thus, implementing this feature can be an interesting part in my future work.

On the other hand, collecting enough data is always a big challenge. By making use of our application, in order to collect desirable high-quality data, it needs more and more users to be involved to upload the data. One can consider leveraging the online crowdsourcing marketplace services such that participants are rewarded in some way.

In terms of improvements that can be made to the application, there are several ideas worth considering. First of all, it is convenient to have a query library feature for users to reuse their previous image query. Because often times, one query can be quite complicated but useful that it would be time-consuming to reproduce. By reusing queries, users can retrieve a list of updated images without re-typing all those configurations. The second consideration is the CRUD feature on the mobile app. It is crucial to allow users to review their uploaded images and annotations in case there are errors that need adjustments and updates. The third point is that the current database setup is open to public. There is no user log in needed to query. It is beneficial for public researchers to quick query images they want but it is difficult for one to save the query without a user identity. Although it is feasible to allow them to save the query anonymously, it is again difficult to find the query they previous used.

In this scenario, I consider implementing the Role-based access control system that we have previous integrated in ImageSfERe. Users will not only have a way to save queries, but a way to share useful queries to others. From the mobile application's perspective, users will be able to configure the image access, whether share it to all people, share it with couple of others, or not share it at all.

BIBLIOGRAPHY

[1] Xi Wu, Steve K. Roggenkamp, Shiqiang Tao and Guo-Qiang Zhang, "ImageSfERe: Image sharing for epilepsy research," 2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), Kansas City, MO, 2017, pp. 817-823. doi: 10.1109/BIBM.2017.8217760

[2] Dave Pearson, Rethinking Image Sharing: Cleveland Trailblazers Link 20+ Hospitals and Provider Sites (2018). https://www.radiologybusiness.com/sponsored/1068/topics/health-it/rethinking-image-sharing-cleveland-trailblazers-link-20-hospitals

[3] Marcus DS, Olsen TR, Ramaratnam M, Buckner RL. The Extensible Neuroimaging A rchive Toolkit (XNAT): an informatics platform for managing, exploring, and sharing neuroimaging data. 2007 Spring;5(1):11-34.

[4] Daniel K, Daniel R, Sandy N, Cesar R, and Chris B. "Managing Biomedical Image M etadata for Search and Retrieval of Similar Images". 2011 Aug.

[5] Dina D-F, Sameer A, Mohammad-Reza S, Hamid S-Z, Farshad F, Kost E: Automatic ally Finding Images for Clinical Decision Support: IEEE Computer Society, 2007

[6] Hai J, et al: Content and semantic context based image retrieval for medical image gri d: IEEE Computer Society, 2007

[7] Oria V, et al: Modeling Images for Content-Based Queries: The DISIMA Approach, 1 997

[8] Solomon A, Richard C, Lionel B: Content-Based and Metadata Retrieval in Medical I mage Database: IEEE Computer Society, 2002

[9] Warren R, et al. MammoGrid—a prototype distributed mammographic database for E urope. Clin Radiol. 2007;62:1044–1051. doi: 10.1016/j.crad.2006.09.032.

[10] Wesley WC, Chih-Cheng H, Alfonso FC, Cardenas AF, Ricky KT. Knowledge-B ased Image Retrieval with Spatial and Temporal Constructs. IEEE Trans on Knowl an d Data Eng. 1998;10:872–888. doi: 10.1109/69.738355.

[11] Zhang GQ, Tao S, Xing G, Mozes J, Zonjy B, Lhatoo SD, Cui L. "NHash: Random- ized N-Gram Hashing for Distributed Generation of Validatable Unique Study Identi- fiers in Multicenter Research". JMIR Med Inform. 2015 Nov 10;3(4):e35.

[12] NEMA (01/26/2017). Retrieved from http://dicom.nema.org/Dicom/about-DICOM.html

[13] Ruby on Rails. Retrieved from http://guides.rubyonrails.org/

[14] Model-View-Controller. Retrieved from https://en.wikipedia.org/wiki/Model-view-controller

[15] Node.js. Retrieved from https://nodejs.org/en/about/

[16] React and React Native. Retrieved from https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-reactjs-and-react-native/

[17] MySQL Wikipedia. Retrieved from https://en.wikipedia.org/wiki/MySQL

[18] GraphQL. Retrieved from https://graphql.org/
[19] GraphQL vs. REST – A comparison for fetching data from GitHub's GraphQL and REST APIs. Retrieved from https://www.andrewrobertmcburney.com/blog/graphql-vs-rest
[20] Sequelize.js. Retrieved from http://docs.sequelizejs.com/
[21] Tom Lagier, "Scaffolding a NodeJS GraphQL API server". 2017 Nov. Retrieved from https://medium.com/@tomlagier/scaffolding-a-rock-solid-graphql-api-b651c2a36438
[22] Dustin Boswell - Storing User Passwords Securely: hashing, salting, and Bcrypt. 2012 Jun.
[23] Neo Ighodaro. "Why use Redux? Reasons with clear examples". Retrieved from https://blog.logrocket.com/why-use-redux-reasons-with-clear-examples-d21bffd5835

VITA

Xi Wu

Education

BSc Physics, University of Science and Technology of China, Hefei, China, 2013

MSc Computer Engineering, Case Western Reserve University, Cleveland, Ohio, 2017

Professional Experience

Graduate Research Assistant, Institute for Biomedical Informatics, University of Kentucky, Lexington, Kentucky. Jan 2016 – Dec 2018.

Professional publications

X. Wu, S. K. Roggenkamp, S. Tao and G. Zhang, "ImageSfERe: Image sharing for epilepsy research," *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Kansas City, MO, 2017, pp. 817-823.
doi: 10.1109/BIBM.2017.8217760