Spring
2014

KindaRight

CLIENT: BLACKSBURG WEB DEVELOPER

AUSTIN LÓPEZ-GÓMEZ & DIVIT SINGH

CS 4624

Table of Contents

Executive Summary

Concept

KindaRight is a collaborative, open network for artists of all disciplines. We emphasize collaboration between artists of all disciplines because we firmly believe that art comes from inspiration, and inspiration comes from people. The more you know and the more you see will help artists produce better, more beautiful pieces of art. We believe that the best people qualified to critique art are artists themselves, which is why KindaRight also revolves around a merit system which shows your status as an artist weighted for both how many people have liked you, and the respective merit of those people. Finally, we are first and foremost a network. One network for connecting those creating art to those buying art. One network to discover new art and new talent. One network where the entire art community can work together.

Current Status

As of the end of this semester we have implemented a full user experience for uploading and sharing photographs. We plan to continue this project and implement a design that is more closely related to our vision. We have included various milestone markers including our midterm and final presentations that detail our status at those points in time respectively. We also included our poster from VTURCS which gives a good overall description of our project and where our future works will be focused. Lastly we have included our final report which is a comprehensive documentation of everything that we have built this semester.

Contents

This report covers the uses development history to date of KindaRight. The first section is concerned with design and functionality of our website, introducing you to the core pillars of our site and the goals we set out to accomplish. The next section is concerned with the possibility of someone joining our project and getting up to speed of where we were when this semester ended. The final section is concerned with the thoughts that we have had this semester and what we have learned by constructing this project.

User's Manual

This Section is concerned with the design and guiding principles behind our project. Since we made this project ourselves a large portion of the semester was attributed to designing the concept. The website is not live today, but at a later date will be live at http://www.kindaright.com.

Notes

http://www.behance.com and http://www.dribbble.com are websites common in certain creative fields. Both have similar concepts to KindaRight but we differ in specific ways that we will show you during this report. More information can be found in our references section.

PATRON and ARTIST are monikers I will use to specify the specific ARTIST, PATRON use case. For that reason you may see PATRONs as a plural of that use case.

PILLARS

I. **COLLABORATION:** Enable ARTISTS to collaborate with other ARTISTS from any discipline and allow ARTISTS to be found by geographic location.
II. **NETWORKING:** Enable ARTISTS to showcase their portfolio and expand their professional network.
III. **PATRONAGE:** Enable anyone looking to work with or hire an ARTIST to find ARTISTS of any discipline.

COLLABORATION

PROBLEM: Currently there is no platform that encourages ARTISTS to learn and grow from each other. Most platforms restrict interactions to professional encounters, such as hiring an individual, and some, behance.com, allow for others to comment on an individual's work.

SOLUTION: KindaRight creates an environment where ARTISTS can opt to create a collaborative portfolio. Under the creative commons license the ARTIST gives up any rights or claims to the work produced in this specific portfolio in exchange for the ability to work with other individuals collaboratively on a project

KEY FEATURES:

- Any ARTIST participating in the project can modify, remove, or add content to this portfolio.
- Changes will be tracked.

NETWORKING

PROBLEM: Working in a creative discipline means that much of your professional network is related to people who you work closely with and know. These also happen to be

people who work on projects similar to you. Online portfolios and networking sites, specifically those for creative fields, are specialized to a particular industry and do not let you explore the creative world abroad.

SOLUTION: KindaRight is a website that will allow ARTISTS of any discipline to explore the work of other ARTISTS in any field from one location and allow communication and networking with any of these individuals.

KEY FEATURES:

- Masked email allows anyone to email any ARTIST without users disclosing their personal information. This information however can be made available if a CONNECTION is formed.
- Every ARTIST will have a public portfolio that can be accessed by anyone. This allows an ARTIST to be seen by as many people as possible.

PATRONAGE

PROBLEM: Given the issue of effectively networking, it is increasingly difficult for young ARTISTS to get hired. Most individuals are hired by people who they know personally or who have been referred by a past client. Several online portfolio sites (Behance.com, dribbble.com[note this is not a typo, it's the website]) allow for ARTISTS to be hired online. However:

I. These sites are scattered, and often times very specific to one creative profession.
II. These sites do not allow for filtering by location in the event that you would want to work with this person face to face.

SOLUTION: KindaRight allows for two user categories: ARTISTS and PATRONS. PATRONS are similar to the ARTIST user types, but provides a much more focused environment for one purpose, hiring ARTISTS.

KEY FEATURES:

- See PATRON below

TERMINOLOGY

USER: All users are either an ARTIST or a PATRON. However, both include the following features.

- Search for ARTISTs
- Contact ARTISTs
- Contact PATRONs

ARTIST: User account that allows for the full benefits of using KindaRight. This includes displaying portfolios, networking, and collaborating with other ARTISTS.

- Creating and displaying a Professional Portfolio is available to all ARTIST disciplines. Their work will be represented in either an Audiovisual, photographic, or audio representation. Work displayed in this portfolio will **always** be owned by the ARTIST protected under copyright.
- Creating and displaying a Collaborative Portfolio, will **always** be under the Creative Commons license and ARTISTs will not own any work created under these Portfolios.
- **ARTIST POST:** ARTISTs can post availability to work certain jobs that they are interested in. These posts can then be searched by PATRONs
    - Notifications when a PATRON POST is similar to an ARTIST POST

MERIT: Merit is based on merit. Other users are allowed to vote on how much they like the work. Everyone starts out with a basic amount of space. If you get enough votes you are upgraded to more space to show off your work.

PATRON: User account that allows for a limited, but more focused use of KindaRight. Meant for individuals who have no desire to participate in the creative process, but instead wish to hire creative talent. KindaRight provides a platform for searching all creative fields and managing CONNECTIONs and interactions from one place.

- PATRONS can post a classified ad on KindaRight when looking for work and FILTER it by location, if desired. Answers to these ads will send a notification to the PATRON's email, if desired. Or this notification can be viewed within the site itself.
- PATRONS can browse ARTISTS work, and determine whether or not they wish to extend an OFFER, at which point, an ARTIST may choose to inquire further or decline, ceasing contact with PATRON.
    - FILTERS: location, work-type, discipline
- **PATRON POST:** PATRONs can post the availability of a job position that they are looking to fill.

SPECTATOR: Term used for someone who is not logged into an account on KindaRight. Certain features are still available and this use form is meant for those who may want to use KindaRight quickly and sporadically.

- View ARTISTs portfolio.
- Contact ARTISTs

CONNECTION: When two users mutually agree to interact and work together, a connection can be made. Making a CONNECTION involves the following

- Sharing personal contact info that you have provided (email, phone, etc.)
- Activity Notifications.

JOB POST: Parent term for PATRON POST and ARTIST POST. Two ways of connecting PATRONs with ATRISTs and allowing ARTISTs to get hired. Both include the following information.

- **JOB DESCRIPTION:** What kind of job will you be doing? What is involved and what skills are required.

MONETIZATION

COMMISSION: On KindaRight, artists will be able to sell their work as they wish. Every sale that is made through our site will cost them a commission.

JOB LISTING: Patrons will be able to post job listings for works they are commissioning, events they need staffed, or positions they need filled. There will be a flat fee for posting a job listing.

Live Screenshots

This section gives screenshots of our web application as of May 4, 2014. Its purpose is to give the reader an idea of how our website functions as a whole.

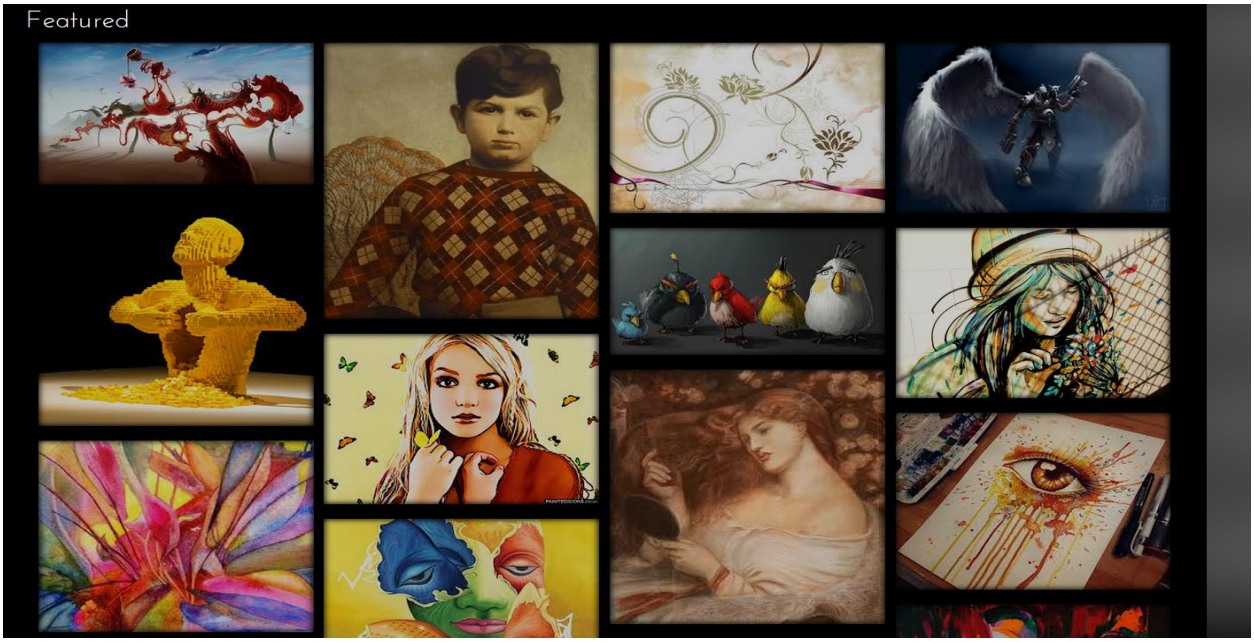


Figure 11 shows the main page of our application.

Figure 11 shows our main page. This can be thought of as the community page where everyone can see the work of all the artists that have submitted something to our web application.

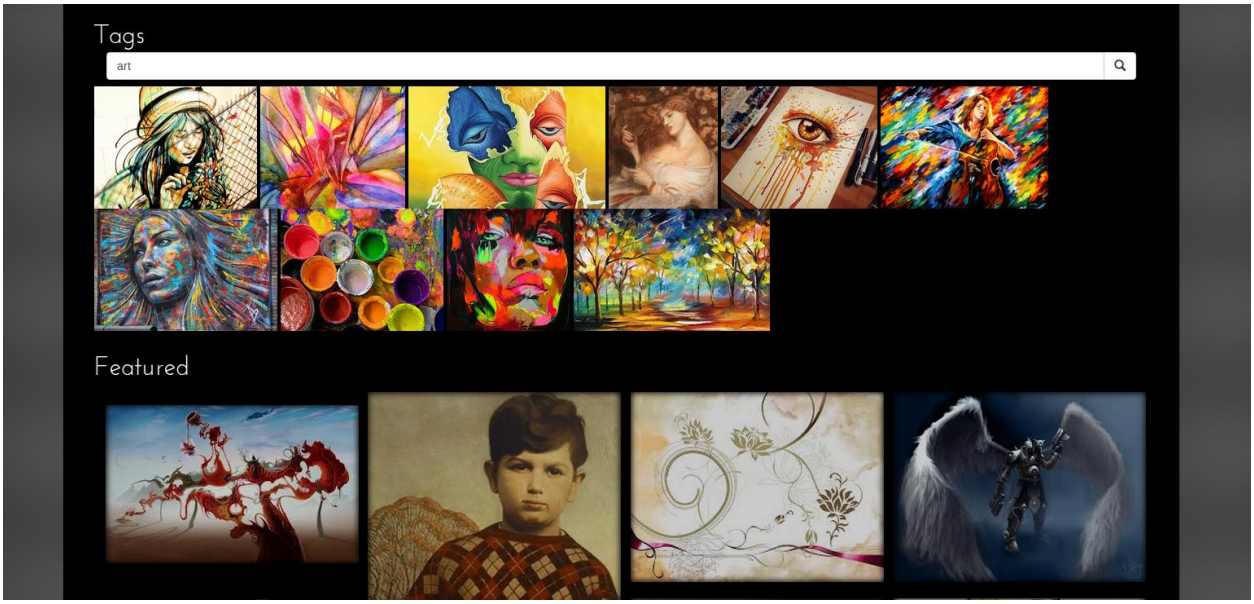


Figure 12 shows the tag functionality of our application.

We wanted to make sure that all images were easy to find. As our database started being filled with many pictures, we decided that tag functionalities were important as it would help filter the content to exactly what the user was looking for. Figure12 shows an example of searching for the tag "art". All images returned above the Featured section are the pictures that have the tag "art" associated with them.

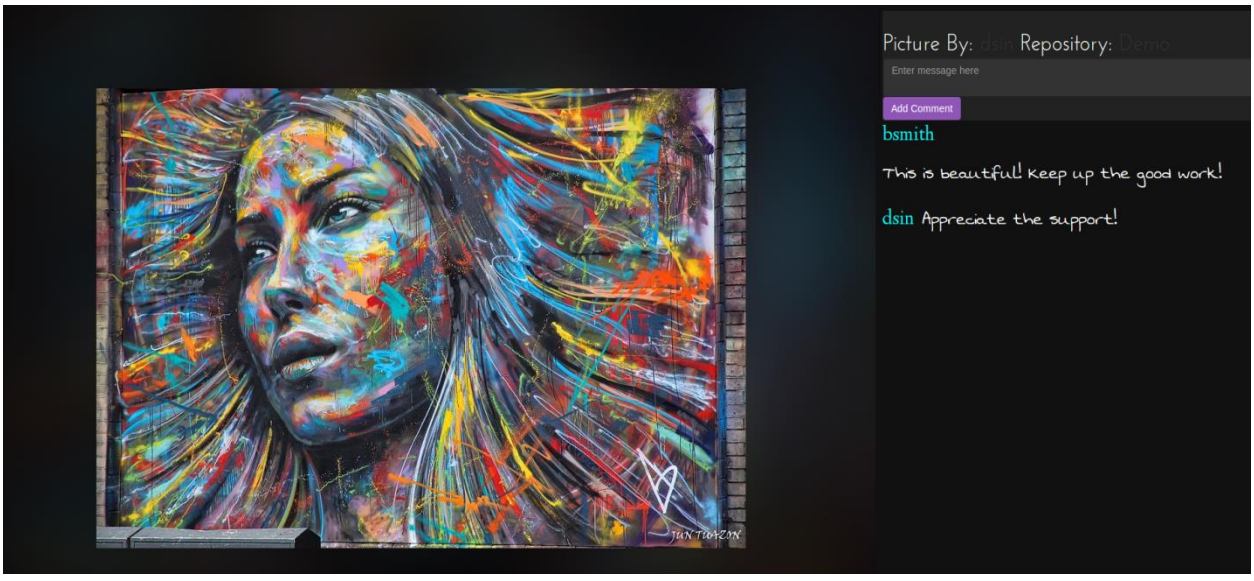


Figure 13 shows commenting functionality of our web application.

We wanted to encourage user interaction and feedback on all content that is uploaded to our web application. As such, we created a commenting system that shows the picture, the person who created it, as well as the repository it resides in. It allows for any individual to

comment on the picture itself. It keeps track of all comments, as well as which user actually commented on them.

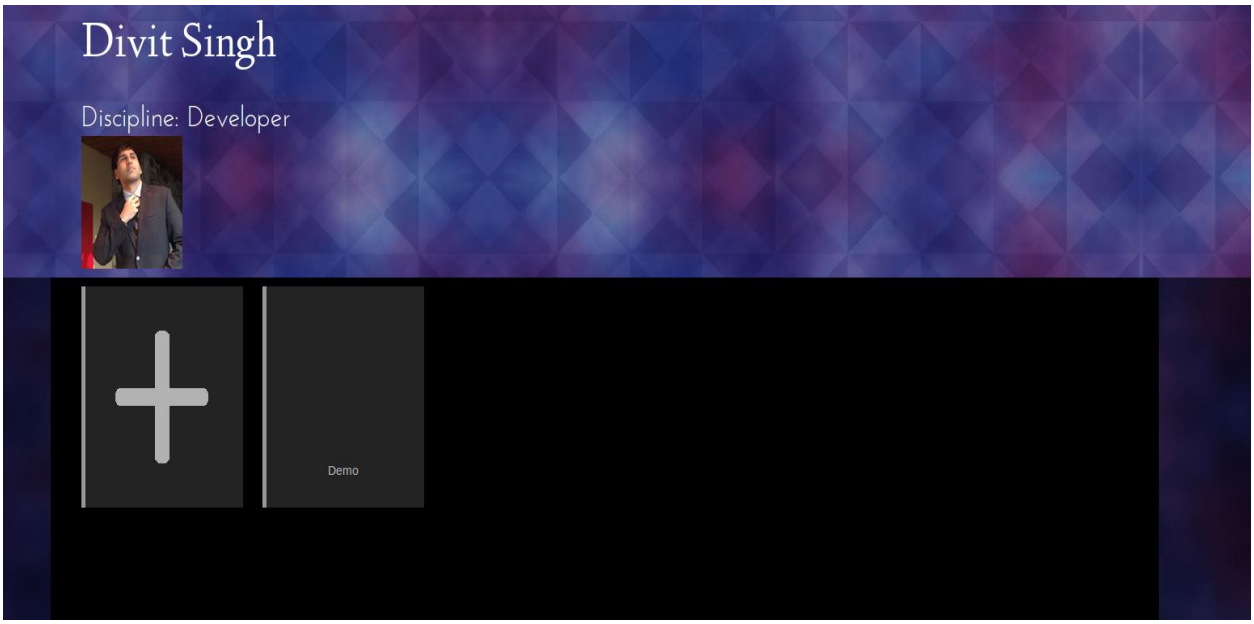


Figure 14 shows an example profile page of a User on our application.

Since we wanted to create a community, it was imperative that we create User functionality and management for our application. Figure 14 shows a sample user's profile page as well as his repository "Demo". This is where people that are interested in an artist's work will come to explore content. A user can create as many repositories as needed. Any user that searches for this user can see the repositories that they have created. When a repository is clicked, the user will be directed to another page that shows all content associated with the repository that is clicked. Figure 15 shows the result of clicked the repository "Demo" that is displayed on the profile page.

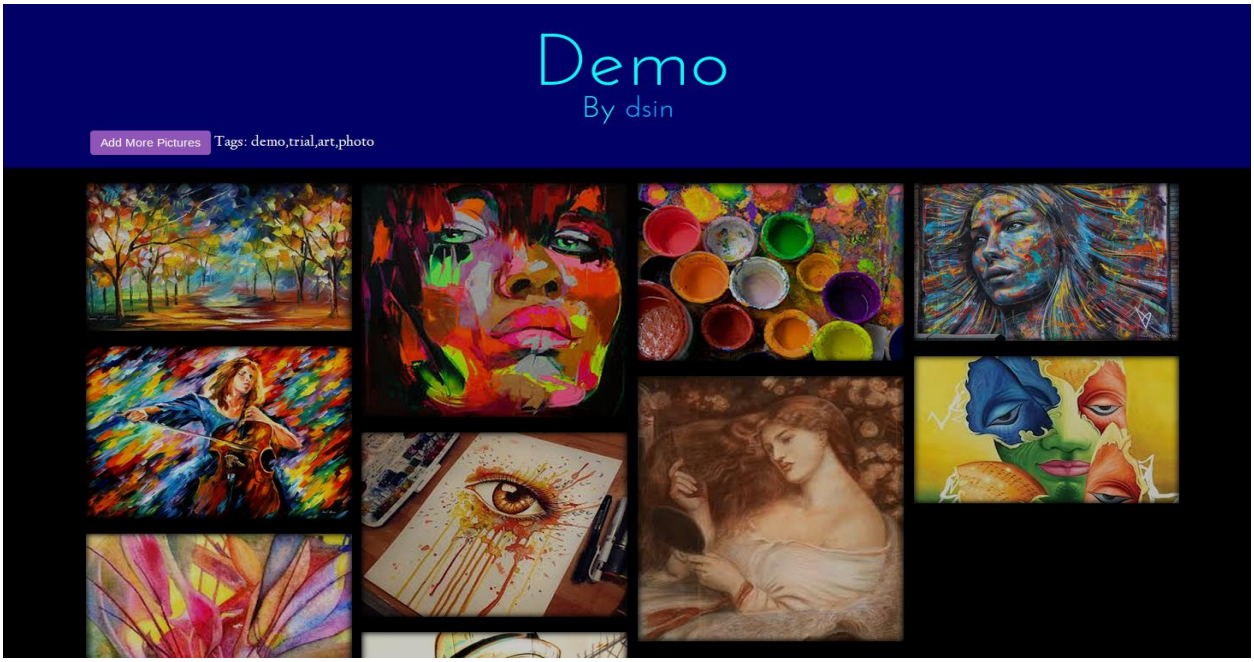


Figure 15 shows a sample repository of a user.

Developer's Manual

This section is created for any future developer working on this project to understand our code base and be able to contribute code as fast as possible. The code base that this document is intended for is current as of May 4, 2014.

App Structure and Installation

The backend framework we are using for this project is Node.js. Specifically we are using Expressjs, a popular web application framework (http://expressjs.com). We decided to use a NoSQL database for our storage. The specific NoSQL database we chose was MongoDB (https://www.mongodb.org). If one does not have prior experience with Node.js, consult online resources. Node.js (http://nodejs.org/) is a functional programming framework that utilizes callbacks. As such, if one does not completely understand Node.js semantics, consider learning the framework before continuing with this manual.

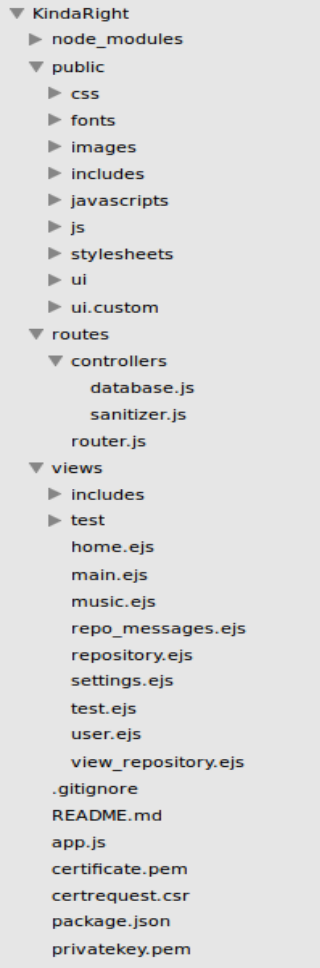
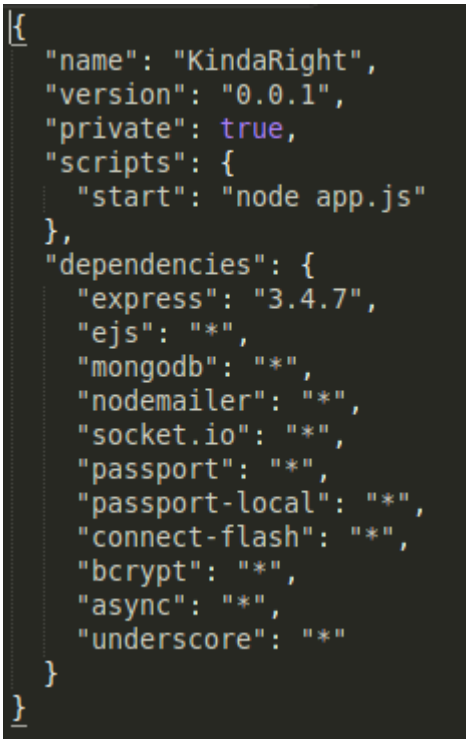


Figure 1 above shows the entire application structure.

In order to run the application on a local machine, simply enter the directory to which this application resides and in the run the command: *node app.js*. At this point, if the application

does not run, it may be because your computer does not have all the dependencies to run this application. Luckily, Node.js contains a script that contains all the dependencies that one needs in order to run this application. To run this script and install the required dependencies, run the command: *sudo npm install*. After entering your password, the installer should iterate through all the directories, installing them as needed. After this is done, your application still may not run. If this is the case, it means that you do not have MongoDB installed. Recall that MongoDB is the sole database of this entire web application.



```
{
  "name": "KindaRight",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.4.7",
    "ejs": "*",
    "mongodb": "*",
    "nodemailer": "*",
    "socket.io": "*",
    "passport": "*",
    "passport-local": "*",
    "connect-flash": "*",
    "bcrypt": "*",
    "async": "*",
    "underscore": "*"
  }
}
```

Figure 2 shows the script that contains all dependencies in KindaRight

In order to install MongoDB, follow the instructions on how to install MongoDB on their website: https://www.mongodb.org. If the instructions are followed correctly, you are ready to run this application on your machine! Simply run the command *node app.js*, this time everything should run correctly. Navigate to *http://locahost:3000* to see the login page.

Notice that app.js is the file that contains the code to actually run the server. Inside of app.js, it contains code that ties together all of the other code in the app structure. We tried to follow MVC as much as possible with our design. Most, if not all of the database code is contained in database.js within the routes/controllers directory. This code contains all communication with MongoDB. If you notice the bottom of each file within the routes directory, you will notice "module.exports" lines such as that shown below.

```
module.exports.insertIntoDB = insertIntoDB;
module.exports.checkExists = checkExists;
module.exports.find = find;
module.exports.getUser = getUser;
module.exports.search = search;
// module.exports.updateUser = updateUser;
module.exports.findAndModifyRepo = findAndModifyRepo;
module.exports.useGridFS = useGridFS;
module.exports.addMessage = addMessage;
module.exports.getMessages = getMessages;
module.exports.findAllPictures = findAllPictures;
module.exports.findAllPicturesForTag = findAllPicturesForTag;
```

Figure 3 shows how methods are passed between files in Node.js

Module methods are passed so that other files can use the methods in certain files. It is similar to "importing" in Java or "include" in C. In order to acquire instances of these methods in other files, a variable specifying the path to this file is necessary.

```
var router = require("./routes/router.js");
var db = require("./routes/controllers/database.js");
var passport = require("passport");
var LocalStrategy = require('passport-local').Strategy;
var sanitizer = require("./routes/controllers/sanitizer.js");
```

Figure 5 shows how to obtain instances of other files for modularized code.

If one wishes to create more endpoints, examine the app.js file carefully. Node.js follows HTTP functionalities like any other web framework; however, the syntax is simply that of JavaScript.

```
app.post("/logout", function (req, res) {
    req.logOut();
    res.clearCookie("loggedIn");
    res.redirect("/");
});
```

Figure 6 shows a simple HTTP endpoint in JavaScript syntax.

Using this information, you now have enough information to understand the code structure that KindaRight is coded in. In order for one to obtain more understanding of the code base, consider changing simple snippets of code and seeing the change that is caused in certain functionalities.

Database Specifics

Since this application is heavily dependent on the database, it is imperative that one understands the specific functionalities in order to modify it in the future. In order to run

MongoDB, simply type mongo in the console. This takes you to a child process within the terminal. The specific database name we are using is KindaRight. Type the following command: *use KindaRight*. This tells MongoDB that we are using the named database. Within a database are collections. We have multiple collections being used in our database. Most, if not all of these collections can be found by simply examining database.js or app.js.

```
function search(db, query, callback) {
    var projection = {
        "First_Name": true,
        "Last_Name": true,
        "Email": true,
        "Username": true,
        "_id": false
        };
    db.collection("Users").find(query, projection).toArray(callback);
}
```

Figure 7 shows an example of a collection that is being used in MongoDB.

If one wanted to see a collection in MongoDB (for debugging purposes), simply type: *db.Users.find()*. This finds all information that is under the "Users" collection within the KindaRight database. If one wants to explore more sophisticated queries, consult the MongoDB website for their Node.js API.

```
> db.Users.find().pretty()
{
        "Discipline" : "Artist",
        "Email" : "bobsmith@gmail.com",
        "First_Name" : "Bob",
        "Last_Name" : "Smith",
        "Password" : "$2a$04$MMqOhUd8MhI/eoFIMsb3S.n0XaAnMCRrErnAbpWvYvQPUxyJw2d
La",
        "Profile_Picture" : ObjectId("5361b76271b369bf184d1bd3"),
        "Repositories" : {
                "Art" : {
                        "Pictures" : [
                                ObjectId("5361b7bc47f58d3c19517352"),
                                ObjectId("5361b7bc47f58d3c19517355"),
                                ObjectId("5361b7bc47f58d3c19517358"),
                                ObjectId("5361b7bc47f58d3c1951735b"),
                                ObjectId("5361b7bc47f58d3c1951735e"),
                                ObjectId("5361b7bc47f58d3c19517361"),
                                ObjectId("5361b7bc47f58d3c19517364")
                        ],
                        "tags" : [
                                [
                                        "weird",
                                        "colors",
                                        "bright",
                                        "abstract"
                                ]
                        ]
                }
        },
        "Username" : "bsmith",
        "_id" : ObjectId("5361b76271b369bf184d1bd5")
}
```

Figure 8 shows the output of a sample user within the MongoDB database.

Lessons Learned

Front End Development

The front end development was concerned with a few key issues throughout the course of the development of KindaRight. At each part we learned a valuable lesson about the development of this website.

I. Easy navigation and management of a portfolio.
II. Effectively displaying large amounts of data.
III. Seamless user interactions
IV. Development Collaboration
V. Work Environment
VI. Defining Strict API
VII. Database Management

Easy navigation and management of a portfolio

In order for a website to work effectively we need a way to manage portfolios well. This led to a whole ton of designs that were not great in allowing people to filter through their data quickly. Or group things the way that they wanted to. In the end we scrapped everything and decided to look back at what other people have done. Often times management systems for other websites can be used analogously for any type of data. Our end design worked out much better. The lesson here: do no reinvent the wheel.

Effectively displaying large amounts of data

We wanted this site to be interactive, so that artists are able to collaborate with each other. A necessary step for that to happen effectively is to be able to sort and display large amounts of data. We needed a good system that allows for this to occur without the user thinking about it. The solution we came up with was tagging the picture, not the album. This allows us to immediately sort through large amounts of information. The lesson here: learn how to divvy up your information.

Seamless User Interaction

A big problem that we experienced earlier on was that the website looked “hacky” as our testers put it. In other words the website looked unprofessional and disjoint, like each page was its own website. This prevented people from getting into the flow of the website, which I suppose meant there was a lack of flow in our website. However this all changed when we added a unified theme and worked to make sure the website had a solid user interaction. This immediately improved the way people reacted and interacted with our site. The lesson here: uniformity lessens distractions and focuses the mind.

Project Lessons

Development Collaboration

Since both developers never worked together, we needed to establish a method so that one person can primarily focus on the backend, whereas the other developer can be primarily focused on the frontend portion. At first, we both had a considerable amount of merge conflicts due to lack of communication of the piece that is being worked on. In order to circumvent this problem, we agreed to use branching techniques. When one developer was satisfied with their work, he would notify the other developer to check his/her code so that it is merged to the pristine master branch.

Work Environment

When starting off this project, one developer was on Linux, the other on Windows. This created a problem as some dependencies of Node.js required a Linux distribution in order to function properly. After playing around with different configurations, we eventually settled on using a guest Linux installation running on VirtualBox. This increased our productivity immensely as both developers did not have to worry about if the error was produced by the OS.

Defining Strict API

As we progressed with our project, API management became increasingly complex. We started to expand our idea faster than we could maintain our code base. Eventually, we reached a point where a mass refactoring was needed in order to keep the API from getting out of control. This cost us a lot of time, and we made sure to keep the API as simple as possible so we don't waste time in the future. The API is simply the endpoints that are listed in app.js. We wanted to create a RESTful API. This became complex as we started expanding. There were endpoints that would have to be created carefully so as not to tamper with already existing functionality. Had we been more cautious with establishing a strict RESTful API, we would not have run into these problems.

Database Management

When we started off this project, we did not think we needed too much querying power. We were convinced that we only needed simple queries because there would not be too many parameters to store all user activity. This was one of our biggest mistakes as our database decision (MongoDB) is getting increasingly difficult to manage and understand. We have multiple collections and complex JSON object storage. All of this could have been avoided had we envisioned our idea better. We are planning to change our entire database dependency to a SQL database in the near future. The reason we are planning on doing this monumental task is so we can get greater query power. MongoDB is excellent for our original use case, however, the more we expand, the more it became evident that our database was going to hold complex structures. In order to reach these structures in MongoDB, it required constant refactoring so that all the data could be easily referenced. In addition, maintaining many collections became increasingly complex as if one reference was changed, the collections would be rendered useless.

Back End Development

GET/POST Decisions

One of the hardest concepts to grasp was the type of HTTP request that was necessary in order to accomplish a task. As a beginner into this area, the more work that was being done on the backend, the more refactoring that was needed in order to avoid security holes. A lot of requests turned into POST requests in order to validate information. As the API grew, refactoring was needed in order to transfer the required information between web pages. We did not know the advantage of our decision until weeks later, when we implemented cookies.

Framework Decision

Node.js is a relatively young framework. As such, there are very few good practices that exist. Some of them are Expressjs (http://expressjs.com), Sailsjs (http://sailsjs.org). We first started implementing our own framework, following tutorials online. However, as our knowledge grew, we discovered an excellent framework, Expressjs that handled web requests elegantly and modularly. This was our most crucial find as this framework is the backbone of our application. In addition, there was plenty amount of open source code available to help us implement our solution.

Asynchronous Coding

One of the hardest things to grasp was the asynchronous nature of Node.js. Its use of callbacks was hard to implement at first. This was especially apparent when iterating through loops. The big question was: "How would one iterate on code if a callback was called at the end of the first iteration?" The answer was the async module (https://github.com/caolan/async). This module allowed for elegant syntax as well as powerful asynchronous control handling.

```
async.eachSeries(acct.Pictures, function (item, next) {
    db.collection("Pictures").findOne({"id": item}, function (err, act) {
        imgIds.push("http://localhost:3000/repository/content/" + act.Creator  + "/" + item);
        msgIds.push("http://localhost:3000/repository/" + act.Creator + "/" + act.Repository + "/" + act.id);
        next();
    });
}
, function (err) {
    return callback(null, imgIds, msgIds);
});
```

Figure 9 shows an example of one of the powerful async module functionalities.

Middleware Injection

There were multiple design flaws that were encountered as we started expanding on our web application. The biggest flaw that we saw was how to pass a database instance to each request that needed it. The answer to this question was injecting middleware. The middleware injection is a method to insert any code that is needed to a request before it is actually processed. This functionality allowed us to insert a database connection to every request that needed it. Not only did this allow for cleaner code, but it also helped us to avoid considerable amount of coupled code.

```
app.use(function (req, res, next) {
    MongoClient.connect("mongodb://127.0.0.1:27017/KindaRight", function (err, db) {
            if (err) {
                throw err;
            }
            _db = db;
            next();
        });
});
```

Figure 10 shows middleware injection of a database to a request to our web application.

Works Cited

All Interviews were conducted with faculty and students at Virginia Tech

Abel, Troy. Personal Interview. 4 March 2014.

Fant, Dylan. Personal Interview. 24 February 2014.

Lassiter, Holly. Personal Interview 24 February 2014.

Weigert, Jonas. Personal Interview. 18 March 2014.

References

Behance: http://www.behance.com

Behance is a website for digital artists, mainly, to show their work and sell it to buyers.

Dribbble: http://www.dribbble.com

Dribbble is a website for graphic designers to show off their work and receive feedback.

MongoDB: https://www.mongodb.org

This is a non-relational database that we are using for our entire backend for KindaRight. This is used to store JSON documents directly in the database.

ExpressJS: http://expressjs.com

This is one of the most used web frameworks in Node.js. It provides very useful features in terms of maintaining and creating endpoints.

Node.js: https://nodejs.org

Node.js is an asynchronous server-side programming language. The entire back end for this project is built in Node.js.